The Runtime Environment for Screme, a Scheme Implementation on the 88000

Steven R. Vegdahl
Tektronix Laboratories
P.O. Box 500, M/S 50-662
Beaverton, OR 97077
stevev@tekchips.crl.tek.com

Uwe F. Pleban
Applied Dynamics International
3800 Stone School Rd.
Ann Arbor, MI 48108
uwe%amara.uucp@umix.cc.umich.edu

Abstract

We are implementing a Scheme development system for the Motorola 88000. The core of the implementation is an optimizing native code compiler, together with a carefully designed runtime system. This paper describes our experiences with the 88000 as a target architecture. We focus on the design decisions concerning the runtime system, particularly with respect to data type representations, tag checking, procedure calling protocol, generic arithmetic, and the handling of continuations. We also discuss rejected design alternatives, and evaluate the strengths and weaknesses of the instruction set with respect to our constraints.

1 Introduction

1.1 An overview of Scheme

The Lisp dialect Scheme [Rees86] is based on the λ-calculus with assignment. It differs from algorithmic languages like Pascal or C in several important aspects. First, Scheme has *latent* as opposed to manifest *types*, which means that types are associated with values (also called objects) rather than with variables. Non-numeric data types, such as lists, symbols, and strings, are supported directly. In addition, *generic arithmetic* works on (usually arbitrary precision) integers, ratios, floating point and complex numbers. Second, all objects created in the course of a computation have *unlimited extent*. In particular, Scheme *procedures are first class objects*: they can be created dynamically, stored in data structures, and returned as results of procedures. *Continuations* [Clin88] *also have first class status*, which makes them useful for implementing a wide variety of advanced control constructs, including non-local exits, backtracking, and coroutines. Third, variables usually obtain values by *binding rather than assignment*. Finally, all implementations of Scheme are required to be *properly tail recursive*. This allows iteration to be expressed using the ordinary procedure-call mechanics, so that special iteration constructs are essentially only syntactic sweetener.

Unlike almost all other dialects of Lisp (with the exception of Common Lisp), Scheme is *lexically scoped*, which makes it more amenable to efficient compilation. In contrast with Common Lisp, the language was designed to have an exceptionally clear and simple semantics and few ways to form expressions. Indeed, the core of the language is defined in terms of only seven fundamental constructs: *procedural (lambda) abstraction, procedure (function) application, sequencing, conditional, assignment, variable reference*, and *literal*. These fundamental constructs are augmented with a rich set of predefined functions and macros.

On a pragmatic level, Scheme implementations are usually *incremental*, and always supply *automatic storage reclamation*.

1.2 Approaches to Scheme implementation

It is straightforward to construct an interpretive implementation for Scheme. The simplest approach trivially maps a source program on its syntax tree representation and interprets the resulting internal structure. A more complex approach first compiles programs into a low level intermediate code (usually called byte code), and then executes the resulting program with the help of a byte code interpreter. The most difficult type of implementation is a compiler which generates efficient native code.

Previous research has demonstrated that certain compilation techniques (e.g., closure and assignment analysis) can significantly increase the performance of a Scheme implementation [Stee78, Broo82, Kran86, Kran88]. In addition to compiler optimizations, however, the design of the runtime environment is equally important for good system performance [Sheb87].

It should be apparent from the preceding description of Scheme that the efficient implementation of the following features is crucial to the overall success of a Scheme system:

- Procedure call and return.
- Primitive operations.
- Dynamic type-tag checks.
- The integer aspect of generic arithmetic.
- Environment references.
- Automatic storage management (garbage collection).
- The handling of continuations.

We elaborate on this list later.

1.3 The 88000 architecture

The Motorola 88000 [Moto88a] is a load/store "RISC style" architecture with pipelined instruction execution, thirty-two 32-bit registers, and instruction and data caches. Most instructions occupy the instruction pipeline for only a single cycle. Every instruction occupies 4 bytes.

Register R0 contains the constant zero, and is not writable. The other 31 registers are homogeneous, except that subroutine call instructions implicitly store the return address into R1. The architecture uses a register scoreboard to stop the instruction pipeline whenever it is necessary to wait for a memory reference to complete; this allows multiple memory reads to occur simultaneously.

The 88000 has five general classes of instructions: load/store, integer arithmetic/logical, transfer of control, bit manipulation, and floating point. The load/store, integer, and bit manipulation instructions generally read either two registers or a register and an unsigned constant, and combine them to form a result. The branch instructions generally read a single register, from which a condition code or target address is determined. The floating point instructions (together with integer multiplication and division) are directed to a separate function unit, which may execute in parallel with other instructions.

The *load and store instructions* compute their address using either the sum of two registers (with one optionally scaled), or the sum of a register and a 16-bit unsigned literal. Byte, halfword, fullword, and doubleword versions are available for each; byte and halfword loads can optionally be sign extended. Assuming a cache hit, *memory references incur a two-cycle latency* [Moto88b]. Misaligned memory references either cause a trap, or produce undefined results, depending on a user-settable status bit.

The usual complement of *arithmetic, relational, and logical instructions* is available. The result computed by the *compare* instruction is a bitvector of condition code bits for all 10 signed and unsigned comparisons. There is no hardware support for detecting arithmetic overflow without taking a trap.

All *transfer of control instructions* come in two flavors. The ".n" version executes the instruction in the delay slot regardless of whether the branch is taken or not. The "standard" version suppresses the execution of the delay-slot instruction if the branch is taken[1]. The `jmp` (jump) and `jsr` (jump to subroutine) instructions use as a target address a value in a register. The branch instructions, including `bsr` (branch to subroutine), `bb0` and `bb1` (branch on a specific bit in a register being 0 or 1), and `bcnd` (branch on condition) use a branch offset to specify the target.

Instructions for manipulating bit fields include field extraction, masking, setting, clearing, and rotation. The extraction and masking instructions are generalizations of shift instructions found in other architectures. There is also a *find first bit* instruction which computes the bit position of the most significant zero/one in a word.

1.4 The Screme system

The components of our Scheme system for the 88000 (dubbed Screme) comprise a *native code optimizing compiler* with a built-in "machine independent" assembler, a *dynamic linker/loader*, *runtime support* (library, garbage collector, etc.), and a *runtime debugger*. With the exception of the garbage collector and a small portion of the runtime library, it is entirely written in Scheme itself. The Screme compiler is currently a *cross compiler* running on the Mac II

[1]Unlike some RISC architectures, the 88000 does *not* have a branch that *executes* the delay-slot instruction only if the branch is taken.

under MacScheme, but will be bootstrapped to the 88000. The initial target is the 88000 plug-in board for the Mac II.

One of our design goals has been to ease future efforts of porting the system across boards and/or operating systems. As a result, we have been intentionally conservative about making assumptions about the environment in which the implementation may run, other than that the processor is an 88000. More specifically, we certainly do not want to exclude running on a Unix-based workstation rather than as a Macintosh plug-in; these two systems make quite different assumptions about such things as virtual memory support and exception handling.

1.5 Overview

The remainder of this paper is a discussion of our Scheme implementation. Section 2 gives a brief overview of the compiler. Section 3 discusses the runtime environment, including an examination of the operations that need to be particularly fast, the representations selected for various Scheme data types, and the way the architecture's 32 registers are used; it concludes by giving several examples of code the compiler generates for common operations. Section 4 evaluates the 88000 as an architecture for running Scheme. Finally, section 5 discusses the status of the project and improvements that we would like to make.

2 The Screme Compiler

Although the compiler is probably the most important part of the system, we only sketch its structure here.

First, the Scheme *reader* performs lexical and syntactic analysis, and converts a program into an S-expression. The *macro expander* then "explains away" the syntactic sweetener, yielding a program in the core language.

The *analysis pass* builds an abstract syntax tree, distinguishes between global, local, and uplevel references, alphatizes all non-global variable references, and identifies calls to primitive procedures (primops). It also performs *escape and assignment analysis*. During escape analysis, procedures are classified into those which need to be represented as heap-allocated closures, and those which may simply be compiled into machine code. Assignment analysis inspects all "`set!`" forms (many of which are usually generated by the macro expander), and introduces "cells" for variables which are multiply assigned. As a consequence, all other variables can safely be assumed to be bound to exactly one value, which may be freely substituted (provided that side effects are not duplicated).

The *code generator* first transforms the syntax tree into a lower level representation, which distinguishes, among other things, between the various kinds of procedure call (unknown call, known call, inline lambda application, primop call), and whether the call is in tail position or not. The *recognition of local calls* allows the code generator to set up tailored calling sequences, to suppress runtime argument-count checking, and to use a shorter instruction sequence for the call. A subsequent pass then performs a simple kind of *type determination* by propagating type information along execution paths. This allows primop calls to be specialized if the type of an argument is known at code generation time. In addition, primop calls are categorized as being in control flow or value yielding context. Another pass over the intermediate representation annotates variable references with *last use information*. Finally, storage is assigned and instructions are generated. Specifically, *variables are allocated in registers*, and spilled into stack frames when necessary. At the present time, no instruction scheduling for minimizing memory fetch latencies is performed.

3 The Screme Runtime Environment

3.1 General considerations

We now discuss the interesting aspects of the design of the Screme runtime environment.

Conceptually, a runtime environment consists of the following:

- A representational mapping from code and data structures to machine structures.
- A set of system-wide invariants and conventions that all code can assume to hold, and is required to maintain.
- A set of support routines which compiled code can invoke. Some of these are written in Scheme, and are available to the user; others, which we call *millicode*, are written in machine language, have tailored calling sequences, and typically destroy very few registers. Millicode routines, which are termed "fastcall" by Brooks [Broo86], are used to implement low-level operations such as storage-allocation.

The design of the runtime environment for a language with runtime type tag checking is considerably more complex than for a language with manifest types

[Sheb87, Stee87]. First, none of the Scheme data types maps directly on a machine data type. Second, since type tags are inspected frequently [Stee86], the overhead for tag testing by the most common primitive operations (procedure call, fixnum arithmetic, and basic list processing) must be minimized.

The ubiquity of procedure calls in Scheme programs makes an efficient call/return sequence absolutely mandatory. In addition, those primop calls which are expanded inline by the compiler should minimize the number of instruction cycles for the common cases (fixnum arithmetic and list processing), possibly at the expense of static code size.

The potentially unlimited extent of every Scheme object mandates the presence of a garbage collector. This imposes fundamental invariants on the code and data structures, and allows for various time/space trade-offs.

The first class status of procedures and continuations warrants additional design considerations. Most importantly, programs which do not make use of these advanced features should not be unduly penalized by implementation overhead due to their potential presence. On the other hand, if such features are used, performance should not degrade drastically.

3.2 Partitioning of operations

Based on experience gained during several years of using, implementing, and studying Scheme, we have divided all runtime operations into three categories. The first group of operations, whose execution time efficiency we felt had to be maximized "at all costs", includes the following:

- Procedure call/return
  - Setting up the arguments to a procedure.
  - Testing whether the object being called is a procedure.
  - Argument count checking.
  - Polling for interrupts[2].
  - Allocating a stack frame.
  - Return object manipulation.
  - Deallocating a stack frame.
  - Returning from a procedure.
- Fixnum arithmetic
  - Testing whether an object is a fixnum.
  - Converting fixnums to machine integers for hardware operations, and vice versa.
  - Addition, subtraction, and comparison of fixnums.
  - Testing whether an integer overflow has occurred.
- Basic list processing
  - Testing whether an object is a pair.
  - Accessing the *car* and *cdr* fields of a pair.
  - Testing for the empty list.
- Iteration
- Garbage collection

The second group of operations includes those which should be relatively fast: all storage allocation (for pairs, strings, vectors, and bytevectors), conditional branching on false, especially when the outcome of the test is determined by a primop call (both the boolean value `#f` and the empty list count as false), character, string, vector and bytevector operations, uplevel variable references, and the use of continuations for loop exits.

The third group includes generic arithmetic, general uses of continuations, procedure calls with "rest" arguments, and operations with side-effects. Specifically, assignments may interact with the garbage collector.

The following subsections discuss our design solutions and alternatives for implementing the operations in the first category. Since the implementation of iteration and uplevel variable references depends on the analysis phase of the compiler, these topics are excluded from further discussion.

3.3 Registers

Each of the 32 general purpose registers is dedicated to a particular use. At the highest level, the registers are divided into two categories, *rooted* and *unrooted.* The rooted registers are the only ones examined by the garbage collector. Therefore, the generated code must *ensure that during any period within which a garbage collection might occur, all rooted registers contain valid objects, and no live pointer is contained in an unrooted register.* If either of these invariants were not to hold, system integrity would be

[2]If an interrupt could be serviced at an arbitrary time, the code generator would have to maintain garbage-collector invariants between arbitrary instructions; this would result in substantially poorer code. In our system, hardware interrupts only register themselves (e.g., by setting a bit in an interrupt-polling (`INTPEND`) register, but without doing anything that might cause a garbage collection). The code generator must guarantee that no infinite loop/recursion will occur without interrupts being polled. We enforce this by requiring all (external) procedure entries and loops to poll the `INTPEND` register.

violated, resulting in an almost certain crash at some later time.

Eleven of the 32 registers are unrooted: R0 (ZERO) and R1 (CODE), due to the hardware restrictions mentioned earlier; four scratch registers (S0–S3), which are available to the code generator for storing intermediate results of open-coded primop calls; the INTPEND register for allowing polling for interrupts; the INARGS register, which is used for passing the argument count on procedure calls; and registers R29–R31, which by convention are reserved for operating system use, and are completely ignored by our implementation.

The remaining 21 rooted registers are partitioned into the following sets: eleven argument registers (A0–A10) used for passing arguments to procedures[3]; six registers (M0–M5) for interfacing with millicode routines; the RETURN register for caching the current "return object" (a relocatable encoding of the return address); a continuation frame pointer (CONT); a stack limit register (LIMIT); and the RUNTIME register, which points to a vector of "global" information, such as the values for all the global variables. All but the last three can be used as rooted scratch registers when not being used for procedure and millicode calls.

3.4 Objects

Each object in the system is represented as a 32-bit quantity. The value of an *unboxed object* can be completely described in 32 bits. A *boxed object* is represented indirectly as a pointer to a heap-allocated structure.

Our implementation differentiates between three kinds of objects:

- *Exact fixnums* (unboxed), representing integers between -2^{28} and $2^{28}-1$.
- *Immediates* (unboxed), which include booleans, characters, small points, the empty list, and the special values #!unspecified and #!undefined.
- *Pointers* (boxed), which refer to objects such as pairs, vectors, strings, symbols, bytevectors, bignums, ratnums, flonums, complexnums, procedures, ports, and continuation frames.

The two least significant bits in an object constitute a *tag*, and indicate to which of the above categories a given 32-bit quantity belongs. A tag of #b00 denotes a fixnum, #b10 an immediate, and #b11 a pointer. A tag value of #b01 is illegal.

The choice of #b00 as the fixnum tag allows many arithmetic operations such as addition to be performed *without any tag manipulation* [Stee87]. Moreover, by disallowing a tag value of #b01, we obtain a one-instruction "branch on bit" test to determine if an object is a fixnum or a pointer. As both of these tests are extremely common, we feel that the performance improvement justifies the "wasting" of the additional tag value. Furthermore, fixnums can be directly used as vector indices, as vector elements are aligned on word boundaries.

Another benefit is that by using the low two bits as a tag, a number of dynamic type-checks can be caught "for free" using the *misaligned access trap.* This is detailed in the next section.

There are additional restrictions on object representations. First, fixnums have a *guard bit* adjacent to the sign bit, which allows a two-instruction overflow check for addition and subtraction without causing a trap. (See section 3.5.3 for an example.)

Several other constraints apply to the encoding of immediate values. The representations of the empty list '() and the boolean value #f have been chosen so that they differ in only a few bits; this makes the "if" test—which treats both of these values as *false*—inexpensive. Although the current Scheme standard [Rees86] allows their representations to be identical, we chose distinct representations in anticipation of compatibility with future versions of Scheme.

Finally, the representation of characters was chosen so that fixnum/character conversions are inexpensive, and so that extended character sets can be handled.

A pointer refers to a structure in the heap, which contains *two header words* in addition to the object's data. The first header word usually contains an encoding of the object's class. However, for procedure objects, it points to the object's compiled code (see section 3.5.1). The second header word contains a 23-bit length field, an internal object bit[4], and an 8-bit class tag field that (redundantly) encodes the object's class if the class is "known to the system". The latter allows many primitives to be implemented more efficiently; it is somewhat horizontally encoded to increase the performance of the garbage collector. Although it would have been possible to encode the header information in a single word, we have chosen the less compact representation in order to increase performance.

[3]Syntactic forms such as let introduce local variables that are transformed into lambda-applications by the macro preprocessing phase of the compiler. Local variables, therefore, are treated like arguments, and are typically stored in the argument registers.

[4]If the *internal object bit* is set in the second header word, it is an indication that the structure is internal to some other structure. This effectively allows a pointer to refer to a location inside an object. In the current implementation this is used primarily for return addresses and entry points within code objects and for continuation frames within the frame cache.

3.5 Implementation of common operations

3.5.1 Procedure call and return

We have adopted a "caller saves" convention. The standard procedure calling sequence requires that the caller perform the following actions:

- Save any live registers in the current continuation frame. This frame is allocated upon entry to the caller (see below). Also, variable values in live registers are spilled on the stack only once.
- Place the arguments in the argument registers A1 through A10. If there are 10 or more arguments, the overflow arguments are placed in a vector, which itself is passed in A10.
- *OR* together the argument count and the contents of the INTPEND register, and place the result in the INARGS register.
- Load register A0 with the procedure object. This will allow the procedure to access its referencing environment.
- Place the return object (which encodes the return address) in the RETURN register.
- Compute the procedure's "code object" and invoke it.

An explicit check that the object in A0 is a procedure is avoided using a well-known technique of putting a stub in the "code" slot of all boxed, non-procedure objects that immediately traps to the debugger. (The "code" slot doubles as the "class" slot for non-procedure objects.) Thus an attempt to invoke any non-procedure object in our system either raises an address alignment trap (if the object is unboxed), or traps to the debugger directly (if the object is boxed).

The called procedure checks the number of arguments and the interrupt bit. If it makes further procedure calls in non-tail position from within its body, it stores the return object in the continuation frame, and allocates its own frame(s). However, if it only calls primops, or calls other procedures tail recursively, then the return address remains cached in the RETURN register, which elides both a store and a load instruction. This savings occurs for all procedures at the leaves of the execution tree, and is made possible because of the large number of registers in the 88000.

To simplify this discussion, we have ignored the complications introduced by "rest arguments". However, the additional expense is incurred only upon entry of procedures that expect such arguments.

Upon return, the frame allocated on entry is deallocated, the return object is placed in the RETURN register, converted to a machine address (see section 3.7), and a jump to that address is executed. At the call site, all live registers (possibly including A0) are restored, and execution proceeds.

The following is the code sequence for the most general case: a non-tail call to an unknown procedure, with n arguments being passed. (Note: names containing the string "$$" are integer offsets; the addu and subu instructions perform addition and subtraction, respectively, without trapping on overflow.)

```
;save A0, but only if live
[st    A0,CONT,frame$$slot0]
 <save all live registers in the frame>
 <load argument registers with n arguments>
 <load A0 with the procedure object>
;load procedure's code object
 ld    CODE,A0,object$$class
;set # args to n, include interrupt bit
 or    INARGS,INTPEND,n
  ;NOTE: memory latency causes 1-cycle delay
  ; here unless it can be filled by
  ; reorganizing instructions
;convert code object to entry address
 addu  CODE,CODE,class$$instructions
;jump to procedure, after converting
;return address to return object
 jsr.n CODE
 subu  RETURN,CODE,1
<<three data words of return chunk information>>
;;;
;;;at call site, after call has completed
;;;
;reload A0, if necessary
[ld    A0,CONT,frame$$slot0]
 <restore all live registers>
```

Thus, the most general call takes 5 instructions, and a minimum of 6 cycles (more if the instruction which loads the CODE register generates an address not in the cache).

Upon entry to the called procedure, the following is executed, assuming that the procedure expects m arguments, does not have "rest" arguments, calls other procedures, and needs to allocate one continuation frame.

```
;;;
;;;argument and interrupt check
;;;
;check argument count and interrupt bit,
; if good, to $noCheck
 cmp   S0,INARGS,m
 bb1   eq,S0,$noCheck
;otherwise investigate by calling millicode
 addu  CODE,RUNTIME,runtime$$poll
 jsr   CODE
 <<data word for polling routine>>
$noCheck:
;store return object in caller's frame
 st    RETURN,CONT,frame$$return-chunk
;;;
;;;frame allocation
;;;
;check for frame cache overflow
;if ok, to $ok, after allocating one frame
 cmp   S0,CONT,LIMIT
 bb1.n gt,S0,$ok
 subu  CONT,CONT,frame-delta
;otherwise, handle frame cache overflow
; in millicode
 addu  CODE,RUNTIME,runtime$$cache-overflow
 jsr   CODE
 <<data word for overflow routine>>
$ok:
 <<code for procedure body>>
```

If all checks succeed, this entry sequence consumes 7 cycles.

The code sequence for procedure exit is simple:

```
;deallocate frame
 addu  CONT,CONT,frame-delta
;reload return object
 ld    RETURN,CONT,frame$$return-chunk
  ;NOTE: potentially a 2 cycle memory delay here
;convert object to return address
 addu  CODE,RETURN,chunk$$instructions
;return to caller
 jmp   CODE
  ;NOTE: one-cycle memory delay here
```

Under the assumption that the reloading of the return object hits the cache, this return sequence also consumes 7 cycles.

A general procedure call therefore costs 13 cycles, and the return costs another 7 cycles. Three of these cycles could potentially be eliminated through instruction reordering. For realistic programs, we would expect that one of them can always be avoided. For purposes of comparison, the Common Lisp implementation group for the IBM RT PC [McDo87] reports that a simple function call of a symbol with no arguments takes 45 cycles, and that the returning of one value takes 35 cycles. (Note, however, that a single branch on the RT typically takes several cycles.)

3.5.2 Basic list processing

Accessing the **car** or **cdr** field of a list cell (pair) involves the following steps:

- Testing that the object is a pointer. This is done implicitly by relying on the address misalignment trap.
- Testing that the structure pointed at is a pair. Since the pair tag is encoded as zero, a comparison instruction can be elided, though the conditional branch is still necessary.
- Reading the data element from memory.

It is an error to take the **car** or **cdr** of the empty list.

The following code sequence implements the application **(car pair)**, where the value of **pair** is assumed to reside in register **Ai**, and the result is to be returned in register **Ak**.

```
;get type tag
 ld.bu S0,Ai,object$$tag
;get car (causes trap if not pointer)
 ld    S1,Ai,pair$$car
  ;potential one cycle memory delay
;trap if not a pair
 tcnd  ne0,S0,exception$not-a-pair
;copy car field to rooted register
 or    Ak,S1,ZERO
```

Thus, **(car pair)** is compiled into 4 instructions, which execute in 5 cycles (including one cycle due to memory delay), assuming cache hits on both load instructions. The extra instruction that copies the result from a non-rooted to a rooted register is necessary to ensure that no rooted register contains an illegal value in the event of a trap caused by the second load instruction. By complicating the trap handler, it may be possible to eliminate the extra instruction.

3.5.3 Fixnum arithmetic

Most language implementations set a bound on the range of integers for which arithmetic operations are implemented; this bound generally coincides with the size of a machine word. A Scheme implementation, on the other hand, is expected to provide arbitrary-precision integer arithmetic. Arithmetic operations are additionally complicated by the fact that most operators must work on all available numeric types, which usually include floating point numbers, and possibly rational and complex numbers.

Clearly, such *generic arithmetic* must generally be implemented by calls to out-of-line procedures. However, since arithmetic on small integers is extremely common, always dispatching on the generic arithmetic millicode would incur an unacceptable execution time penalty. Therefore, most implementations divide the space of integers into *fixnums*, which fit (along with tagging information) into a single machine word, and *bignums*, which are represented via pointers to heap-allocated structures containing a multiword representation of the value [Whit86]. Fixnums are assigned a unique tag value; most arithmetic on fixnums is implemented inline.

The presence of differing type tags for fixnums and other numeric quantities means that arithmetic operators in the language cannot be directly mapped onto the standard machine instructions unless there is special hardware support. Without such support, arithmetic operations necessarily comprise arithmetic instructions as well as instructions for checking fixnum tags and detecting overflow.

In Screme, generic arithmetic is fully implemented in millicode. However, important fixnum operations such as addition, subtraction, and comparison are open-coded for efficiency. The following code sequence implements binary addition, with the operands in registers **Ai** and **Ak**, and the result returned in a different register **Am**:

```
;perform addition
 addu    Am,Ai,Ak
;test 1st operand for fixnum
 bb1     tagbit$not-fixnum,Ai,$_1
;test 2nd operand for fixnum
 bb0.n   tagbit$not-fixnum,Ak,$_2
;prepare for overflow test
 clr     S0,Am,30<0>
$_1:
;branch to generic millicode
 bsr     $_segment_generic
 <<word to encode op. for generic routine>>
$_2:
;check for overflow
 bcnd    9,S0,$_1
$_3:
  ;continue with remainder of program
```

This code sequence occupies six instruction words and one data word. In the case of both operands and the result being fixnums, it executes in 5 cycles. An overflow has occurred if the sign bit and the guard bit differ. We test this by masking off all but these two bits, and use the generality of the **bcnd** instruction to test if the masked value is either the most negative integer, or is positive.

3.6 Continuation frames

A continuation is represented by a *continuation frame* (activation record), which is an internal structure within a frame cache. The frame cache contains a large number of contiguous continuation frames. Continuation frames must be more general than activation records for a traditional language, because it is possible that a pointer to a continuation (i.e., a frame and all its predecessors) might be captured by storing it in a variable [Clin88].

Although space limitations preclude a detailed discussion of our frame cache implementation, the following points briefly summarize its salient characteristics:

- When a frame cache overflow occurs, a new frame cache is allocated from the heap and linked to the old one; on underflow, the old frame cache is reinstated.
- Allocating a continuation frame requires three cycles: two for comparing the `LIMIT` and `CONT` registers, and one to bump the frame pointer `CONT`.
- Deallocating a continuation frame takes just one cycle, since underflow is detected by means of a special return address that is placed in a dummy frame at the bottom of each frame cache.
- When a continuation is captured, its frame caches are marked as shared; whenever an underflow into a shared cache occurs—or whenever a continuation is restarted—execution proceeds out of a copy of the frame cache.

Clearly, programs that do not use continuations are not penalized when they allocate or deallocate continuation frames upon procedure entry or exit, because the frame pointer could not be adjusted for free in any event. Programs using continuations in a nontrivial manner incur "only" the overhead of frame cache copying.

3.7 Garbage collection

Although we currently use a simple *stop and copy* garbage collector, the system has been designed with a fast *generation scavenging collector* [Unga84] in mind, in order to allow the programming of embedded applications. Aside from the requirement that the register set be partitioned into rooted and unrooted registers, the presence of a garbage collector manifests itself in the following ways:

- Since code objects may be relocated during a garbage collection, code pointers must generally be stored as a combination of a code segment (i.e., code object), and an offset within the object's structure.
- In order to increase the performance of the garbage collector, the low three bits of the tag field of a boxed object contain a horizontal encoding of whether the structure contains objects, binary data (e.g., strings), or a combination of the two.
- For some objects, the length field is redundant, but is always included in order to simplify the collector.

- Information must be deposited in each continuation frame concerning the number and position of root pointers within the frame.
- Because of the way a generation scavenging collector works, an assignment of a pointer value to a structure field must be registered with the collector if the structure belongs to an older generation than the pointer. However, since it is guaranteed that frame caches always belong to the youngest generation, pointers may be stored into stack frames without requiring notification of the garbage collector.

4 Evaluation of the 88000 as a Target Machine

4.1 Supporting features

Despite the high level of the Scheme programming language and the "low-level" RISC architecture, we found the 88000 to be a relatively nice architecture on which to build a Scheme implementation. The large, homogeneous register set leaves a generous supply of registers for arguments, local variables, and the millicode interface. We disagree with the conclusion by Steenkiste and Hennessy that 32 registers cannot effectively be used in a Lisp implementation [Stee86]. Although that number seems excessive if registers are used *only* for parameter-passing, there are a number of global values that we wanted in registers, but that were placed in memory because we ran out of register space. Examples are heap-allocation pointers and addresses of the most common millicode routines. Conversely, a previous attempt by members of our group at implementing Scheme on the 68000 found its 16 inhomogeneous registers to be inadequate.

Perhaps the most pleasant surprise was the usefulness of the misaligned memory exception in detecting runtime errors. This architectural feature allowed a number of important code sequences to be implemented more efficiently. Near the beginning of the project, we designed (on paper) a tagged version of the architecture, and sketched out non-tagged and tagged versions of code sequences for some benchmarks. To our surprise, the tagged versions were not significantly better; in some cases, the tag manipulation overhead made the tagged version more costly[5].

[5]We do not mean to imply by this that tagged hardware is a bad idea for Lisp implementations. Rather, the instruction set should be designed with tagging in mind from the very start [Hill86], instead of selecting an existing instruction set and adding tagging as an afterthought. Hardware tags are particularly important if floating point operations are to execute efficiently (i.e., without boxing all floating point objects).

Another useful architectural feature has been the availability of both delay-slot and non-delay-slot versions of conditional branch instructions. When the compiler is unable to fill in the delay slot, it is convenient to avoid the branch penalty when the branch is not taken. This is especially advantageous when a conditional branch is testing for an exceptional condition, where the speed of the taken branch is unimportant. On the other hand, it would have been nice to have had branches that executed the delay-slot instruction only when the branch *was* taken.

4.2 Problematic features

The architecture contains a number of "glitches" and non-orthogonalities that have occasionally been frustrating. These include the following:

- The two-instruction penalty for adding a literal exceeding 14 bits in magnitude to a register (e.g., for character literals).
- The lack of sign-extended literals and offsets.
- The lack of an indexed jump instruction which, if present, would have allowed single-instruction jumps through *return objects*.
- The lack of a "signed divide" instruction, which increases the cost of the *quotient* operation by nine instructions.
- The inability to directly obtain the remainder of a division. We were therefore required to synthesize it using division, multiplication, and subtraction instructions, which is not significantly faster than implementing the operation in Scheme source code.
- The lack of overflow detection without taking a trap.
- The problem that neither the non-commutative arithmetic operations (subtract and divide), nor the "complement second operand" versions of and and **or** allow a literal to be used as the first operand rather than the second.

We also note that (effectively) 14 of the possible 16 boolean operations in register-register mode were implemented.

5 Current Status and Future Work

5.1 Status

As of the time of this writing, the company (Tektronix) has put the project "on hold" for nontechnical reasons. Because we did not have working hardware at the time the project was active, we have used the 88000 simulator to debug the compiler and runtime system. We have not yet implemented either floating point or bignum arithmetic. However, the generic arithmetic structure is in place. Also, register spilling is not performed, limiting the compiler to toy programs. Finally, the debugging environment is not implemented.

We have yet to implement an instruction reorganizer that is compatible with our system. The "standard" reorganizer cannot be used because it violates assumptions made by the code generator concerning header words in code and garbage collector invariants. In addition, we suspect that we would be displeased with any post-processing instruction scheduler—as advocated by Gross [Gros83] for traditional languages—because the two-cycle memory latency behooves the merging of independent code sequences that contain delay slots, which our compiler generates with high frequency (e.g., *car*, *cdr*). This, in turn, requires scratch registers to be available to the scheduler for holding (additional) temporary results introduced by the reorganization. Ideally, it would seem that the register allocator and reorganizer should be implemented together [Bird87], although a strategy which reserves two registers (corresponding to the two-cycle memory delay) for the reorganizer might be adequate.

Due to the lack of working hardware, we can only offer a "pseudo" benchmark at this point. Consider the definition of the Fibonacci procedure below.

```
(define (fib x)
 (letrec ((fibo (lambda (n)
  (if (< n 2)
      n
      (+ (fibo (- n 1)) (fibo (- n 2)))))))
 (fibo x)))
```

We have computed that the 88000 code generated by our compiler would execute `(fib 20)` in roughly 492,500 cycles, or 24.65 milliseconds at 20MHz. In comparison, the native code produced by MacScheme on a Mac II for the same procedure takes 867 milliseconds to execute `(fib 20)`, which is about 35 times slower. In general, well informed sources have estimated that the 88000 runs between 15 and 20 times faster than the Mac II.

5.2 Improvements

In considering what might have been done differently in our implementation, three things come to mind. The first is that a significant execution penalty is paid to accommodate the possibility that code might be moved by the garbage collector. This leads to the representation of code addresses as tagged pointers, which requires a good deal of dynamic tag manipulation, as well as a significant amount of space for code object headers in the code itself. Because we expect the garbage collection of code to be a phenomenon that is (almost) exclusively utilized during program development, this amounts to a decision to slow down application programs in order to make it easier to implement an interactive programming environment. We would prefer either seeing this bookkeeping done in some other way, or having a compiler switch that would generate "application" code in which code addresses are represented as fixnums.

Another decision that we may wish to rethink is the treatment of continuation frames as fixed-sized internal objects within a frame cache. The current design "wastes" perhaps half the space in a typical frame cache due to a combination of fragmentation and the space used for internal header information.

Finally, we might reexamine the storage needs for list cells (pairs). In our implementation, every list cell occupies 16 bytes, 8 of which are overhead. An alternate implementation technique would be to use a three-bit tag, with fixnums being `#b000`, immediates `#b100`, pairs `#b101`, and other pointers `#b110`; other tags would be illegal to allow one-instruction branches on fixnum, pointer, and pair. This would have advantages that include *car* and *cdr* being one-instruction operations, and pair-cells taking only 8 bytes of memory. These gains would seem to outweigh the disadvantages of requiring object-allocation on 8-byte boundaries and the slowing down of vector operations by one or two cycles.

6 Acknowledgements

In addition to the authors, Norman Adams, Will Clinger, and Ken Dickey contributed heavily to this system's design and implementation. We would like to express appreciation to Will and Ken for their helpful comments on this manuscript.

References

[Bird87] Bird, P.H.L., *Code Generation and Instruction Scheduling for Pipelined SISD Machines*, PhD thesis, University of Michigan (1987).

[Broo82] Brooks, R.A., Gabriel, R.P., and Steele, G.L., Jr., "An Optimizing Compiler for Lexically Scoped LISP," *Proc. SIGPLAN '82 Symposium on Compiler Construction*, pp. 261-275 (June 1982).

[Broo86] Brooks, R.A. et al., "Design of an Optimizing, Dynamically Retargetable Compiler for Common Lisp," *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pp. 67-85 (August 1986).

[Clin88] Clinger, W.D., Hartheimer, A.H., and Ost, E.M., "Implementation Strategies for Continuations," *Proc. 1988 ACM Conference on Lisp and Functional Programming*, pp. 124-131 (July 1988).

[Gros83] Gross, T., *Code Optimization of Pipeline Constraints*, Technical Report No. 83-255, Stanford University Computer Systems Laboratory (December 1983).

[Hill86] Hill, M. et al., "Design Decisions in SPUR," *IEEE Computer*, Vol. 19(10), pp. 8-22 (November 1986).

[Kran86] Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N., "ORBIT: An Optimizing Compiler for Scheme," *Proc. SIGPLAN '86 Symposium on Compiler Construction*, pp. 219-233 (June 1986).

[Kran88] Kranz, D.A., *ORBIT: An Optimizing Compiler for Scheme*, PhD thesis, Yale University (1988).

[McDo87] McDonald, D.B., Fahlman, S.E., and Spector, A.Z., *An Efficient Common Lisp for the IBM RT PC*, Technical Report CMU-CS-87-134, Carnegie-Mellon University (July 1987).

[Moto88a] Motorola, Inc., *MC88100 32-Bit Third-Generation Microprocessor Technical Summary*, Document No. BR-588/D, Motorola, Inc. (1988).

[Moto88b] Motorola, Inc., *MC88200 Cache Memory Management Unit Technical Summary*, Document No. BR-589/D, Motorola, Inc. (1988).

[Rees86] Rees, J. and Clinger, W. (Editors), "Revised Revised Revised Report on the Algorithmic Language Scheme," *ACM SIGPLAN Notices*, Vol. 21(12), pp. 37-79 (December 1986).

[Sheb87] Shebs, S. and Kessler, R., "Automatic Design and Implementation of Language Datatypes," *Proc. SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pp. 26-37 (June 1987).

[Stee78] Steele, G.L., Jr., *Rabbit: A Compiler for Scheme*" Technical Report 474, MIT Artificial Intelligence Laboratory (May 1978).

[Stee86] Steenkiste, P. and Hennessy, J., "LISP on a Reduced-Instruction-Set-Processor," *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pp. 192-201 (August 1986).

[Stee87] Steenkiste, P. and Hennessy, J., "Tags and Type-Checking in LISP: Hardware and Software Approaches," *Proc. ASPLOS II*, pp. 50-59 (October 1987).

[Unga84] Ungar, D., "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm," *Proc. 1986 ACM Symposium on Practical Software Development Environments*, pp. 157-167, (April 1984). Also published as *ACM SIGPLAN Notices*, Vol. 19(5), (May 1984), and as *ACM Software Engineering Notes*, Vol. 9(3), (May 1984).

[Whit86] White, J.L., "Reconfigurable, Retargetable, Bignums: A Case Study in Efficient, Portable Lisp System Building," *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pp. 174-191 (August 1986).