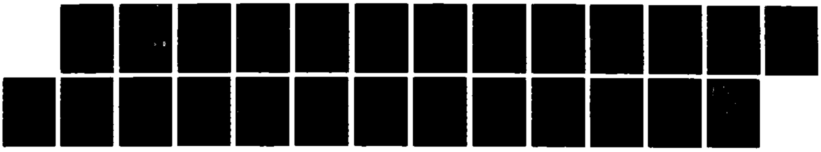
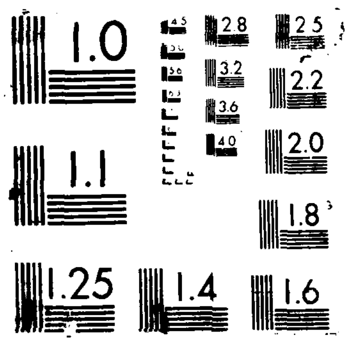
AD-A195 931 SCHEME86: A SYSTEM FOR INTERPRETING SCHEME(U) 1/1
MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL
INTELLIGENCE LAB A BERLIN ET AL. APR 88 AI-M-1040
UNCLASSIFIED N00014-86-K-0180 F/G 12/5 NL




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY

(4)

AD-A195 931

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI Memo 1040	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Scheme86: A System for Interpreting Scheme		5. TYPE OF REPORT & PERIOD COVERED memorandum
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Andrew Berlin and Henry Wu		8. CONTRACT OR GRANT NUMBER(s) N00014-86-K-0180
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Inteligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE April 1988
		13. NUMBER OF PAGES 23
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Distribution is unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES None		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Scheme Lisp computer architecture interpretive techniques		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Scheme86 is a computer system designed to interpret programs written in the Scheme dialect of Lisp. A specialized architecture, coupled with new techniques for optimizing register management in the interpreter, allow Scheme86 to execute interpreted Scheme at a speed comparable to that of compiled Lisp on conventional workstations.		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Scheme86
A System for Interpreting Scheme

Andrew A. Berlin and Henry M. Wu

Artificial Intelligence Laboratory
and
Department of Electrical Engineering and Computer Science

Massachusetts Institute of Technology

A.I. Memo No. 1040 April, 1988

Abstract

Scheme86 is a computer system designed to interpret programs written in the Scheme dialect of Lisp. A specialized architecture, coupled with new techniques for optimizing register management in the interpreter, allow Scheme86 to execute *interpreted* Scheme at a speed comparable to that of *compiled* Lisp on conventional workstations.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-86-K-0180.

Introduction

Scheme86 is a computer system designed to interpret programs written in the Scheme dialect of Lisp. Modern high-performance Lisp systems compile programs into the low-level instruction sets required by conventional processors. The semantics of these low-level machine languages correspond to motions of data within the computer, rather than to the high-level semantics of Lisp. This dichotomy between semantics tends to complicate compilers, and makes compiled code difficult to debug. Scheme86 directly executes a high-level representation language in which the semantics of Scheme are preserved. A specialized architecture, coupled with new techniques for register management in the interpreter, allow Scheme86 to *interpret* this high-level representation of Scheme at a speed comparable to that of *compiled* Lisp on conventional workstations.

The high-level representation language used in Scheme86 is known as S-code. The semantics of S-code correspond to those of Scheme; Scheme special forms correspond to S-code expression types. Tags annotate S-code with information that specifies the type of each expression. This improves performance by eliminating the need for the interpreter to parse list structure representations of Scheme programs. The annotated information allows the interpreter to utilize routines which are tailored to each class of expression.

The Scheme86 design is based on the observation that sequences of chronologically dependent memory references are often the factor limiting the execution speed of Lisp programs. To attack the dependent reference problem, the architecture is optimized to reduce the latency of *memory-processor-memory* operations. Parallel execution units allow multiple operations to occur during each machine cycle, making effective use of a low-latency memory system.

We present an architectural description, an overview of the optimized interpreter, details of the new register management techniques, and results from both architectural and digital circuit level simulations.

Architectural Design Goals

The memory system is often the bottleneck in the performance of modern computer systems. Computations are composed of both processor operations and memory transactions. Inputs to a computation must be read from memory, while computed results must be written back to the memory system. For many computations, the processor operations may be performed in parallel, causing the memory system to become the factor limiting performance. These computations may then be viewed as a sequence of memory transactions. To improve performance, we must reduce the time required to complete these transactions.

To achieve high-performance execution of memory intensive Lisp programs, a high-speed memory system is indispensable. In order to effectively utilize such a high-speed memory system, the processor must be sufficiently powerful to complete the operations required to support each memory transaction before the memory system becomes idle, waiting for requests from the processor.

The performance of a high-speed memory system is specified not only by its throughput, but also by its latency of operation. A low-latency memory system will be effective for sequences of chronologically dependent references, while a high-throughput memory system will be effective on sequences of independent memory references. The processor design must account for both the memory system latency and throughput.

In symbolic languages such as Lisp, both user data structures and interpreter control structures rely heavily upon sequences of chronologically dependent memory references. In user programs, dependent references frequently take the form of CAR-CDR chaining, including tree walks and table lookup. In the Scheme interpreter, they include following environment parent pointers during lexical variable lookup, and walking the tree-structured S-code instructions. In chronologically dependent reference chains, generating a memory request requires that the processor first receive the result of the previous request. Thus, dependent reference chains cannot benefit from a pipelined memory system. To combat this problem, Scheme86 utilizes a low-latency memory. Consequently, the Scheme86 architecture is designed to minimize the latency of memory-processor-memory operations.

In order to execute a high-performance Scheme interpreter, Scheme86 must make effective use of its low-latency memory system. The Scheme interpreter must often perform multiple computations to support *each* memory transaction. These include address generation, bounds checking, type-code checking, result generation, and conditional branching. A very powerful processor is required to perform all of these operations while still keeping the memory system busy. This processor power may be achieved by using an extremely fast processing unit multiple times per memory cycle. Alternatively, the processor may contain multiple processing units which operate in parallel. Scheme86 takes the latter approach, incorporating multiple execution units into the processor.

Architecture

The Scheme86 architecture is influenced by the implementation technology available in the mid 1980's. Machine words are 32 bits wide, divided into 8 bits of tag and 24 bits of datum. Addresses are 24 bits wide, with each address corresponding to a full 32-bit word. This provides a total address space of 64 megabytes.

Scheme86 contains three execution units, as shown in Figure 1. The first execution unit is capable of performing 24-bit wide pointer and integer arithmetic, as well as type-code generation and modification. The second execution unit has similar capabilities, but also contains a barrel-shifter. A third execution unit contains both a 32-bit wide equality comparator and a register transfer path. The execution units are connected through a register bank that contains three write-ports and eight read-ports. Two of the read-ports feed a type-code unit that performs two simultaneous tag comparisons. The results of the tag comparisons may be combined to generate exceptions and branch conditions.

The operation of the memory system is concurrent with that of the processor. To support low-latency memory-processor-memory operations, the system timing is arranged so that memory-read requests are issued at the

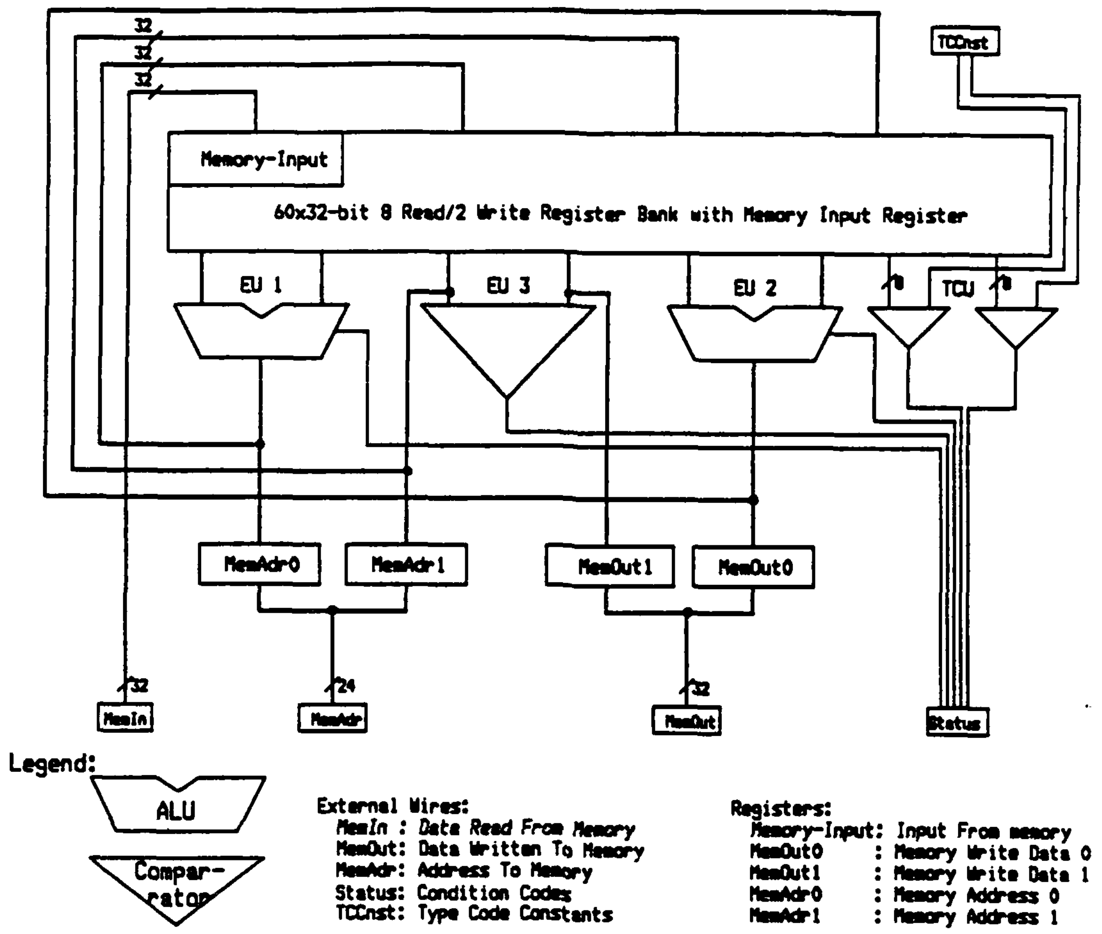

32
32
32
TCCnst
Memory-Input
60x32-bit 8 Read/2 Write Register Bank with Memory Input Register
EU 1
EU 3
EU 2
8
TCU
8
MemAdr0
MemAdr1
MemOut1
MemOut0
32
24
32
MemIn
MemAdr
MemOut
Status
Legend:
ALU
Comparator
External Wires:
MemIn : Data Read From Memory
MemOut: Data Written To Memory
MemAdr: Address To Memory
Status: Condition Codes
TCCnst: Type Code Constants
Registers:
Memory-Input: Input From memory
MemOut0 : Memory Write Data 0
MemOut1 : Memory Write Data 1
MemAdr0 : Memory Address 0
MemAdr1 : Memory Address 1

Figure 1: Parallel Data Paths: The execution and type-code units are fed by an eight-port register bank. Three results are written back into the register bank during every cycle.

completion of the ALU operations of each cycle. The data returned from memory is then available for use in the ALU operations of the next cycle. While the memory operation is in progress, the processor performs register operations, instruction fetches, and dispatching. Overlapping the instruction dispatch with the operation of the memory system dramatically reduces the overhead of interpreting Scheme.

The Scheme86 microcode engine contains a writable microcode memory. The control structure allows branching and dispatching to take place concurrently with other processor operations during any instruction cycle without time penalty. This transfer of control may take the form of either a two-way conditional branch, or a 256-way dispatch into either of two dispatch tables. The zero-cost branching feature proved to greatly improve software performance and ease of micro-instruction placement. A block diagram of the micro-controller is shown in Figure 2.

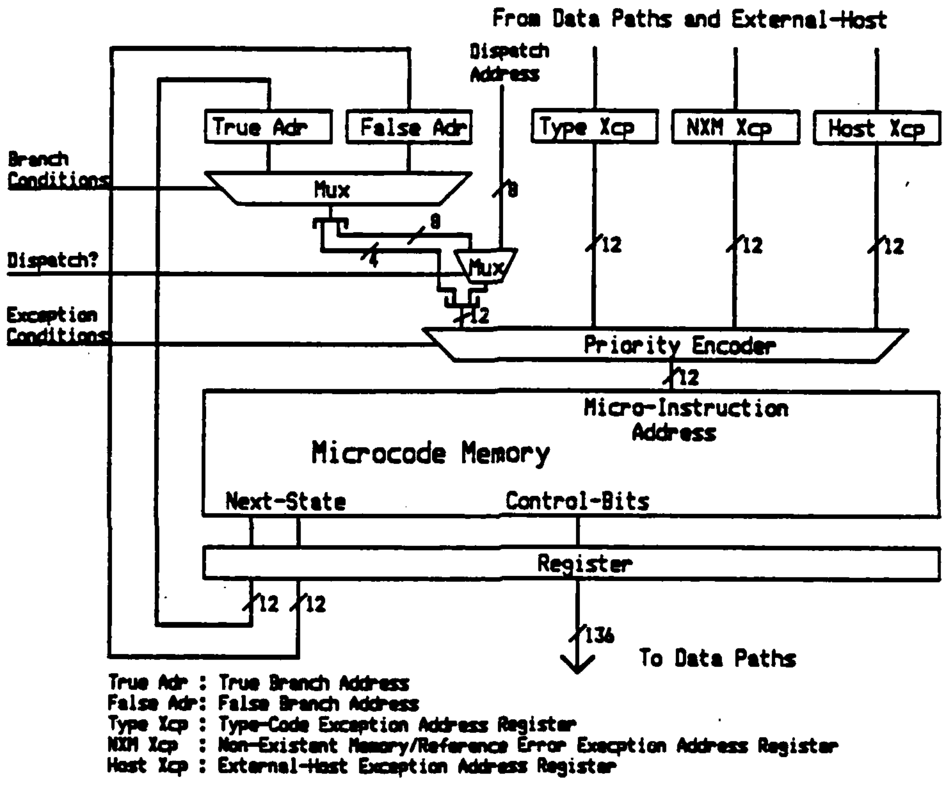

From Data Paths and External-Host
Dispatch Address
True Adr
False Adr
Type Xcp
NXM Xcp
Host Xcp
Branch Conditions
Mux
8
8
4
Dispatch?
Mux
12
12
12
12
Exception Conditions
Priority Encoder
12
Micro-Instruction Address
Microcode Memory
Next-State
Control-Bits
Register
12
12
136
To Data Paths
True Adr : True Branch Address
False Adr: False Branch Address
Type Xcp : Type-Code Exception Address Register
NXM Xcp : Non-Existent Memory/Reference Error Exception Address Register
Host Xcp : External-Host Exception Address Register

Figure 2: Micro-Controller: A 2-way conditional branch or 256-way conditional dispatch may occur on any instruction cycle without time penalty.

The memory system is tightly integrated with the processor, so that memory operations can be initiated during every machine cycle. Memory addresses may come from either of two *memory-address* registers. Data for memory-write operations may come from either of two *memory-output* registers. The results of memory-read operations are placed into the *memory-input* register. This register is mapped into the general register address space, where it is accessible by any of the execution units.

Implementation Details

An implementation of the Scheme86 architecture has been completely designed. The design primarily uses Advanced Schottky TTL technology. Based upon detailed timing simulations at the logic level, we determined that the worst-case cycle time is 165 nanoseconds.

Using 1986 technology, building an eight read-port, three write-port register bank required the use of four smaller register files. Each register file has two read-ports, and two write-ports. During register-write operations, identical copies of the data are simultaneously written into all of the register files. During register-read operations, the register files operate independently, thereby providing a total of eight read-ports.

The Scheme86 register bank supports three register-write operations per cycle. The register bank is implemented using register files containing only two write-ports, one of which is used twice per cycle. The first register-write overlaps with the operation of the ALU's. The other two occur in parallel with the operation of memory, and with the fetch of the next instruction, as illustrated in Figure 3.

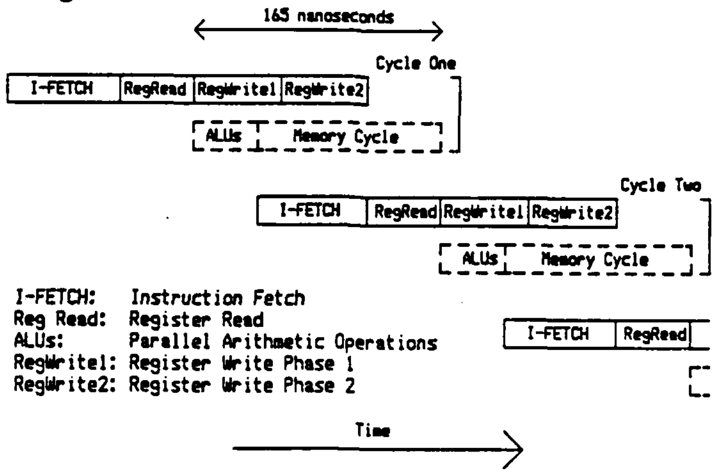


Figure 3: Scheme86 TTL Implementation Timing Diagram: The operation of the processor overlaps with the operation of the memory system. Memory references are separated only by the operation of the ALU's.

The Interpreter

The Scheme86 interpreter executes a high-level representation of Scheme programs known as S-code. S-code is a tree-structured, typed-pointer instruction set. The structure and class of each Scheme expression is encoded in the type field of its S-code representation. The S-code used in Scheme86 has been extended to make very fine distinctions between different categories of expressions. The interpreter uses the type field as a dispatch address to select a routine which is specifically tailored to the type of the expression being evaluated. The Scheme86 interpreter and primitives were hand-coded to take full advantage of the multiple execution units and low memory latency of the architecture. In most cases, we were able to avoid idle memory cycles, and the computation was entirely memory-bound.

Scheme programs are translated into S-code by a process known as *syntaxing.* The syntaxer categorizes Scheme expressions based on their structure and computational requirements, annotating this information in the tag of the S-code form it produces. The analysis performed by the syntaxer is a simple and straightforward categorization that does not require complex data flow analysis or block compilation techniques. A typical expression category detected by the syntaxer is the case in which the operator of a procedure application (also known as a *combination*) is known to be a *primitive procedure.* The tag of the S-code representation informs the interpreter that the application involves a primitive operation. This allows the interpreter to pass the arguments to the primitive directly in processor registers, avoiding the creation of an environment frame in memory for the application.

The microcoded evaluator executes an S-code form by dispatching on its 8-bit tag field, which encodes information about the class of the Scheme expression it represents. A 24-bit datum field points to the components of the expression, which themselves are encoded in S-code. The evaluator retrieves the components of the expression and invokes itself recursively, until it reaches a primitive application, a variable, or a self-evaluating object.

The state of the computation is maintained by a small number of registers. For example, the ENV register maintains the current lexical environment, while the EXP register maintains the current S-code expression, and the

CONT register maintains the continuation. When the evaluator recursively executes subexpressions, these registers are saved and later restored from a stack allocated region of memory.

Variable References

Variable lookups, which contribute a large amount to the execution time of Scheme programs, must be efficiently handled in a high-performance system. The S-code representation of a variable is a pointer to a structure which contains the name of the variable. This structure also contains slots for keeping the lexical depth and frame offset relative to the current environment, where the value of the variable can be found. By storing the lexical address in these slots, the value of the variable can be accessed without searching through nested environment frames. Because Scheme is statically scoped, lexical addresses for bound variables can be determined at syntax-time. However, lexical addresses for free variables must be determined by the interpreter when they are first encountered. In the Scheme86 system, all lexical address calculations are performed by the interpreter, in order to simplify the syntaxer.

Two variable lookup optimizations are performed in the Scheme86 system. Variables bound in the local environment frame are encoded using a special S-code form. The datum part of this form contains the offset into the current frame in which the variable value is kept. This saves space and avoids two extra memory references to fetch the lexical address. The syntaxer can also be instructed to assume that all free references refer to global variables, which are stored in value cells associated with the variable symbols. The S-code instruction can then directly point to the value cell, again eliminating the fetch of the lexical address. Since most parameters are local, and most procedures are globally bound, these two optimizations account for a sizable speed up when executing Scheme programs.

```
;;; Scheme ASSQ primitive. Register Arg1 contains the
;;; key to match for, and Arg2 contains the association
;;; list (a-list). The result is returned in the VAL
;;; register.

(define-dispatch-handler (primitive assq)
  (define-register (reg key) (reg arg1))
  (define-register (reg a-list) (reg arg2))
  (state
   ;; Compute address and start fetching the first pair
   (eu1 (memory-fetch (offset (reg a-list) CAR)))
   ;; Setup answer for case where a-list is empty
   (eu2 (assign (reg val) (fetch (reg nil))))
   ;; Simultaneously test to see if the list is empty
   (if (tcu (type=? (reg memory-input) (type null)))
       (goto ASSQ-RETURN-WITH-ANSWER)))
  (state ASSQ-LOOP
   ;; Fetch the key field in the association pair,
   ;; which is just being returned from memory
   (eu1 (memory-fetch (offset (reg memory-input) CAR)))
   ;; Put this pair in VAL assuming it's the answer
   (eu2 (assign (reg val) (fetch (reg memory-input)))))
  (state
   ;; Prefetch the rest of the a-list
   (eu1 (memory-fetch (offset (reg a-list) CDR)))
   ;; Simultaneously test to see if current pair
   ;; matches key. If so, return with answer in VAL
   (if (eu3 (eq? (reg memory-input) (reg key)))
       (goto ASSQ-RETURN-WITH-ANSWER)))
  (state
   ;; Re-enter ASSQ. This is essentially the same as
   ;; the first instruction, only here the current
   ;; a-list is returning from memory.
   (eu1 (memory-fetch (offset (reg memory-input) CAR)))
   ;; Prepare for answer being NIL
   (eu2 (assign (reg val) (fetch (reg nil))))
   ;; Make memory input the current a-list
   (eu3 (assign (reg a-list) (fetch (reg memory-input))))
   ;; Test for a-list being NIL. If so, terminate
   (if (tcu (not (type=? (reg memory-input) (type null))))
       (goto ASSQ-LOOP)))
  (state ASSQ-RETURN-WITH-ANSWER
   ;; Return to interpreter by dispatching through address
   ;; in the continuation register.
   (dispatch return-addresses (eu3 (fetch (reg cont))))))
```

Figure 4: Microcode for the primitive procedure ASSQ. Each *state* describes operations happening in one machine cycle. In typical applications, two or more execution units are utilized in more than 64% of the cycles, with memory transactions taking place 80% of the time.

Primitive Procedures

Scheme86 supports a large number of microcoded primitive procedures. These procedures are hand-coded to take full advantage of the parallel features of the architecture. For example, the code for `ASSQ`, as shown in Figure 4, makes effective use of the multiple execution sites and low-latency memory system.

Garbage Collection

Scheme86 uses a microcoded two-space stop-and-copy style garbage collector.[3] A straightforward implementation of this relatively simple algorithm provides high speed storage reclamation, obviating the need for schemes which reduce the frequency of garbage collection. The Scheme86 dispatch mechanism proved to be well suited to the needs of the garbage collector. By using the parallel features of the architecture, the garbage collector was able to utilize the high-speed memory system effectively.

Register Management Techniques

Programs frequently use processor registers for data storage. When a subroutine call occurs, the newly invoked routine may reuse some of the processor registers, overwriting data stored by the caller. A register management discipline must be employed to ensure that the contents of these registers are restored to the appropriate values at the completion of the subroutine call. Typically, this involves saving, and later restoring the contents of the registers from a stack allocated region of memory. Register management tends to introduce inefficiency by unnecessarily saving and restoring register contents. The Scheme86 interpreter uses a new register management technique which reduces the unnecessary saving and restoring of the environment (ENV) register.

Background

Register management mechanisms typically establish a calling convention between the *caller* and *callee*. Either the callee or the caller will preserve the register contents by saving, and later restoring the contents from a stack. The *caller-saves* convention is inefficient; the caller must preserve all registers which it needs, even though the callee may not overwrite their contents. Similarly, the *callee-saves* convention is inefficient; the callee must preserve the contents of any register which it overwrites, even though the caller may not need the contents.

A hardware optimization known as a **rack**[8] was used in the Scheme-81 chips to implement a more efficient register management mechanism. A rack is a combination of a register and a stack. When executing programs which use the caller-saves convention, the rack hardware does not actually save the contents of any register until the callee attempts to assign a new value to the register. If the callee never attempts to use the register, then it will not be saved.

The Scheme interpreter is recursive. During each invocation, processor registers may be changed. The interpreter must employ a calling convention to ensure that the contents of registers such as CONT and ENV are preserved. This requires memory operations, which frequently fall in the critical path of the computation. The Scheme interpreter must be tail-recursive, meaning that when a procedure has no more work remaining, it must relinquish all storage associated with it. Using the callee-saves convention, the callee saves the contents of the registers, even if the caller no longer needs these values. This unnecessary saving of values violates tail-recursion, prohibiting the use of the callee-saves convention in a Scheme interpreter.

Although the caller-saves convention satisfies the requirements of tail-recursion, it introduces serious inefficiencies, which are particularly harmful in the case of the environment register. The interpreter must save ENV before calling itself recursively to evaluate a subexpression, so that the environment will be available when evaluating remaining subexpressions. However, evaluation of this subexpression often results in further recursive calls, to evaluate its own subexpressions. These subsequent recursive calls must also save the ENV register. Thus ENV is written onto the stack multiple times, even though once would have sufficed. Indeed, the recursive invocations might never clobber ENV, in which case all of the saves of ENV were unnecessary. Use of the caller-saves convention for the environment register frequently results in repetitive saves, sometimes followed by register usage, followed by repetitive restores.

To optimize the repetitive save and restore operations associated with the environment register, Steele and Sussman[8] suggest a variation on the rack idea which we refer to as a *push-counter rack*. In this scheme, a counter is associated with each register. Each time a caller requests a register-save, the counter is incremented. If a save has been requested, i.e. the counter is non-zero, and a callee attempts to reuse the register, then both the register contents and the value of the counter are saved on a stack. Push-counter racks reduce the overhead of managing the environment register by optimizing long sequences of save and restore operations.

A New Technique: Save-If-Requested

A new register management mechanism, *save-if-requested,* has been developed. Similar in effect to a push-counter rack, this mechanism may be implemented in either hardware or software. The Scheme86 interpreter uses this new technique to reduce unnecessary saving and restoring of the ENV register.

The save-if-requested mechanism associates two flags with each register. The *contents-needed* flag informs the callee as to whether the caller needs the contents of the register preserved. The *restore-is-needed* flag is set by the callee, indicating that the caller's data has in fact been saved on the stack. The caller explicitly preserves the restore-is-needed flag before invoking the callee.

When the callee wants to reuse a register whose contents-needed flag is set, it saves the contents of the register on the stack, sets the restore-is-needed flag, and clears the contents-needed flag. When control is finally returned to the caller, the register contents will be restored from the stack only if the restore-is-needed flag is set.

In summary, the requirements of the Save-If-Requested calling convention are as follows:

- When performing a subroutine call, the **caller** must perform two tasks:
  - It must set the contents-needed flag on all registers whose contents will be needed later.
  - It must save the value of the restore-is-needed flag.
- When the **callee** reuses a register whose contents-needed flag is set, the callee
  - Saves the contents of the register on the stack.
  - Clears the contents-needed flag.
  - Sets the restore-is-needed flag.
- When control returns to the caller:
  - If the restore-is-needed flag is set, the register contents are restored from the stack.
  - The original value of the restore-is-needed flag is restored.

Implementing Save-If-Requested

The save-if-requested calling convention does not require explicit hardware to support the contents-needed and restore-is-needed flags. The trick is to maintain two logical copies of the interpreter.[7] One copy is for situations in which the caller requires that the contents of the ENV register be preserved, whereas the other copy is for situations in which the caller no longer needs the contents of ENV. This duplication of code represents the contents-needed flag implicitly in the program counter. The copy of the interpreter which corresponds to the contents-needed flag being lowered simply overwrites ENV without saving it, whereas the other copy arranges to save ENV before reusing it, and sets the restore-is-needed flag.

The restore-is-needed flag may be represented implicitly as part of the continuation. When the callee saves the ENV register, it can simply modify the continuation (for example, by subtracting one) to return to a point which will restore the ENV register from the stack, and then proceed normally. Merging the restore-is-needed flag with the continuation automatically preserves the flag during recursive calls, since the continuation is preserved.

It is interesting to note that while the save-if-requested optimization requires maintaining two logical copies of the interpreter, in practice much code can be shared between the two copies. This code sharing is further supported by the zero-cost branching feature of the Scheme86 architecture.

Register Shadowing

During recursive invocations, the interpreter state can be saved in private, quickly accessible *shadow* registers, *provided* that no region of the interpreter that uses the same shadow registers is entered. The Scheme86 interpreter uses this technique, allocating shadow registers from the general purpose register bank.

We sepa·.te expressions into two types. *Simple* expressions are those that can be evaluated without recursively invoking the interpreter. These include self-evaluating objects, variables, and lambda-expressions. *Compound* expressions are those that require recursive invocations of the interpreter.

Examples include combinations and conditionals. The syntaxer classifies the components of each compound expression as either simple or compound. Compound expressions containing only simple expressions as components may be evaluated using shadow registers.

By using multiple sets of shadow registers, we can extend this idea to cover compound expressions containing compound components. In Scheme86, primitive applications whose components are all simple expressions are distinguished as *safe* primitive-combinations. The interpreter uses one set of shadow registers to evaluate safe primitive-combinations. Another set of registers are used to evaluate *less-safe* primitive-combinations, which may contain both simple expressions and safe primitive-combinations. Yet another set of shadow registers are used in evaluating *safe* procedure-combinations, whose subexpressions may in turn be simple expressions, safe primitive-combinations, or less-safe primitive-combinations.

The introduction of safe combinations into the S-code language, coupled with the use of shadow registers, reduced the saving and restoring of the contents of the EXP, CONT, argument, and function registers. The microcode required to process the new S-code forms required the addition of only 45 microinstructions, out of a total of over 700 microinstructions. This optimization, which does not require specialized hardware, is effective on conventional processor architectures as well.

Performance Analysis

A series of tests were conducted to determine the performance of the Scheme86 system, as well as the effectiveness of the optimizations incorporated into the S-code interpreter. The tests used a set of benchmarks, including the doubly recursive version of the Fibonacci function, the TAK and BOYER programs from Gabriel[4], and the simple query language from Abelson and Sussman[1]. These tests were chosen based on their intensive use of the arithmetic, list processing, and procedure call mechanisms of the interpreter; for their resemblance to the general class of large, well-written Scheme programs this system was designed to execute; and for their wide acceptance as standards for measuring Lisp systems.

Program (seconds)	Raw Time	Save-If-Requested	Global Variables	Safe Combinations
`(fib 20)`	0.26	0.24 (11%)	0.23 (1.6%)	0.19 (19%)
Tak	0.82	0.70 (15%)	0.69 (1.6%)	0.58 (16%)
Boyer		14.5	14.3 (1.3%)	12.4 (14%)
Query		0.052	0.040 (24%)	0.034 (15%)

Figure 5: Scheme86 Execution Times. The numbers in parentheses indicate the percentage of *incremental* speed-up as each additional optimization is enabled. The average *overall* speed-up is around 30%.

The execution time figures were computed based upon the worst-case cycle time of 165 nanoseconds derived from the simulated TTL design of Scheme86. Figure 5 shows the execution time and the *incremental* improvements[1] for the four benchmarks as each additional optimization was enabled. As shown, the save-if-needed environment register optimization sped up interpretation by a maximum of 15%. When free variables were assumed to be globally bound, a further improvement of as much as 24% was attained. When the system used safe combinations a further improvement of up to 19% was realized. On average, the combination of all of these optimizations

[1]The terms "savings", "speed up", and "improvements" are defined to be $(slowtime - fasttime)/slowtime$.

provided an improvement of 30% over an interpreter implemented without them.

Program (seconds)	Orbit HP9000s320	Orbit HP9000s350	Scheme86
`(fib 20)`	0.156	0.067	0.188
Tak	0.470	0.204	0.580
Boyer	52.18	23.4	12.4
Query	0.0966	0.0418	0.0337

Figure 6: Orbit/Scheme86 Benchmark Results. Scheme86 did extremely well in large programs where compiler optimizations are less effective.

As a comparison, the execution times of these four programs on conventional hardware was measured. The programs were compiled and run in the most recent version of the Orbit[5] Scheme system, with open-coding of integer arithmetic and list primitives enabled. The timing measurements were performed on two Hewlett-Packard workstations. The HP9000 Model 320 computer is a 68020 based workstation released in 1985. It has a clock speed of 16 Mhz and 7.5 Mbyte of storage. The HP9000 Model 350 workstation, which uses a 25 Mhz 68020 processor, contains 32 Mbyte of RAM. This model was first shipped in mid-1987, well after the hardware design and simulation of the TTL Scheme86 processor was done.

Sufficient heap space was allocated initially so that garbage collection did not occur during any of the tests. Figure 6 shows the execution times using Orbit, along with those attained by Scheme86, with all optimizations enabled in both cases. As is evident from this comparison, the performance of Scheme86 is competitive with that of compiled Scheme running on modern workstations, particularly in large programs where compiler optimizations are less effective.

Scheme86 made effective use of its hardware resources. While running our set of benchmarks, it kept two or more execution units busy more than 64% of the time. Memory transactions took place in 80% of the cycles. Dispatching took place about once every three cycles. The flow of control was conditionally determined in 60% of the cycles. These results indicate that the Scheme86 system was able to effectively exploit parallelism in achieving high performance.

These simulation results prove that the performance of Scheme86 is highly competitive with that of current Lisp systems. This performance level is realized without the use of complex compilation techniques or exotic hardware technology.

Related Work in Performance Analysis

Scheme86 used multiple execution units to improve the performance of the processor. An alternative approach is to reuse a single execution unit multiple times. These two approaches were compared in a separate study[11]. The comparison showed that as the latency of the memory system decreases, the cycle time of a single execution unit architecture becomes smaller than can be implemented using modern TTL or other comparable technology, whereas the parallel execution unit architecture was able to make up for its slower basic clock rate by performing multiple operations in each cycle. Although the single execution site architecture could be implemented using exotic technology, one can easily imagine an implementation of the parallel execution unit architecture which uses the same exotic technology, thereby producing an even faster processor.

Relation To Previous Work

The idea of building a machine which directly interprets an S-code representation of Scheme programs originated in the Scheme-79 chips. The Scheme-79 chips rivaled the performance of the compiled Lisp systems of the time, specifically MacLisp on the PDP-10. The S-code used in Scheme86 is more specialized than Scheme-79 S-code, supporting the multip e optimizations employed in the Scheme86 system.

These ideas were further pursued in the Scheme-81 chips, which were designed by automatically compiling the S-code interpreter into hardware. The data paths included special-purpose features which directly supported the requirements of the microcoded interpreter. For example, specialized processing units were associated with some of the registers. The Scheme-81 chips used racks to improve general register management, while a push-counter rack was used to optimize the saving and restoring of the environment register. Scheme-81 was designed to be part of a collection of chips, each specialized to perform a particular task such as `ASSQ`.

The architecture of Scheme86 differs from that of Scheme-81. Scheme86 is a hand-designed general purpose register machine with parallelism provided by multiple execution units. Scheme-81 specialized the private processing elements of some registers according to their function in the S-code interpreter. The general purpose nature of Scheme86 allows the primitive operations of Scheme to be microcoded, thereby using the same hardware which supports interpretation to implement high performance primitive operations. Implementing the Scheme86 processor using multiple integrated circuits, rather than on a single VLSI chip, allowed increased connectivity between the multiple execution units and the memory, thereby exploiting the benefits of a low-latency memory system.

Conclusions

Scheme86 is a high-performance implementation of a Scheme interpreter. A specialized architecture exploits the parallelism inherent in the interpreter, incorporating multiple execution units to effectively utilize a low-latency memory system. New register management optimizations, which are applicable to conventional architectures, provide significant performance improvements.

Historically, efficient execution of Lisp on conventional architectures has required the use of a fairly complicated compiler. We have shown that it is possible to design an architecture for executing interpreted code that performs competitively with compiled Lisp on conventional machines, but without the complexity associated with compiler technology. Interpreters have the further advantage that they can be easily changed, thus making Scheme86 a good vehicle for experimenting with language design and implementation.

Scheme86 executes a high-level representation of Scheme, whereas most modern Lisp systems execute low-level languages that describe data movement within the computer. The results from the Scheme86 project lead us to believe that it may be profitable to target future architecture efforts at executing an intermediate language which lies somewhere between high-level representation languages and low-level machine code.

Acknowledgments

We thank Gerald Sussman, Chris Hanson, and Bill Rozas for helping with the design of the Scheme86 architecture and interpreter. Yekta Gursel, David Espinosa, Oded Feingold, Mark Miller, and Steve Codell contributed significant time to the project. Special thanks is due to Thomas Simon, who wrote the final version of the Scheme interpreter, implemented most of the optimizations, and assisted in the performance analysis. We extend our gratitude to Hal Abelson and Jacob Katzenelson for reviewing drafts of this document.

References

[1] Abelson, Harold., Gerald Jay Sussman, with Julie Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, Mass., 1985.

[2] John Batali, Edmund Goodhue, Chris Hanson, Howie Shrobe, Richard M. Stallman, and Gerald Jay Sussman. "The SCHEME-81 Architecture - System and Chip". In *Proceedings of the MIT Conference on Advanced Research in VLSI,* edited by Paul Penfield, Jr. (Dedham, Mass.: Artech House).

[3] R. Fenichel, and J. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM* 12(11):611-612. 1969.

[4] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, Cambridge, Mass., 1985

[5] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. "Orbit: An optimizing compiler for Scheme". In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 219-233. ACM, June 1986. Proceedings published as *SIGPLAN Notices 21(7), July 1986.*

[6] J. Rees, and W. Clinger (editors), "Revised (3) Report on the Algorithmic Language Scheme", MIT AI Memo 848a, Cambridge, Mass., September, 1986.

[7] Thomas D. Simon. "An Alternative to Racks". Personal Communication, 1987.

[8] Guy Lewis Steele, Jr. and Gerald Jay Sussman. "The Dream of a Lifetime: A Lazy Scoping Mechanism", MIT AI Memo 527, Cambridge, Mass., November 1979.

[9] Gerald Jay Sussman, Jack Holloway, Guy Lewis Steele, Jr., and Alan Bell, "Scheme-79 – LISP on a Chip". In *Computer*, Vol. 14, No. 7, July 1981

[10] Henry M. Wu, "Scheme86 - An Architecture for Microcoding a Scheme Interpreter". S.B. thesis, Department of Electrical Engineering and Computer Science, MIT. May 1986.

[11] Henry M. Wu, "Performance Evaluation of the Scheme86 and HP Precision Architectures". S.M. thesis, Department of Electrical Engineering and Computer Science, MIT. May 1987.