Compiling Higher-Order Languages into Fully Tail-Recursive Portable C

Marc Feeley James S. Miller Guillermo J. Rozas Jason A. Wilson

August 18, 1997

Note: This paper was written in 1993 and has not been modified since then. It is therefore out of sync with the current implementations of Gambit and MIT-Scheme.

Abstract

Two independently developed implementations of SCHEME have been extended to compile into portable C code that implements full tail-recursion. Like other compilers for higher-order languages that implement full tail-recursion, measurements of these systems indicate a performance degradation of a factor between two and three compared to the native code emitted by the same compilers. We describe the details of the compilation technique for a non-statically typed language (SCHEME) and show that the performance difficulty arises largely from the cost of C function calls. In theory, these expensive calls can be eliminated. In practice, however, they are required to avoid excessively long compilation times by modern C compilers, and to support separate compilation.

1 Introduction

Two independently-developed SCHEME systems (MIT SCHEME[7] and GAMBIT[4]) have been extended to produce portable C output code. The projects were undertaken completely independently with different design goals, yet have ultimately resorted to the same mechanisms. In both cases we find that the performance of our C code is between a factor of two and three slower than the native code generators for our compilers. In addition, recent work on compiling SML[12, 20] has yielded similar performance results, despite the fact that the SML compiler uses static typing information to overcome the overhead of encoding data type information at runtime. We find this coincidence to be compelling evidence that the observations are intrinsic to the problems of translating higher-order code into tail-recursive C. We describe a series of measurements which help illuminate the source of this difficulty.

To our knowledge, there is no generally accepted definition of "fully tail-recursive." For clarity, we adopt the definition from the SCHEME standard[9]:

> In a *tail-recursive* implementation, iterative processes can be expressed by means of procedure calls. (The process described by a program is iterative if and only if the order of its space growth is constant, aside from that used for the values of the program's variables.)

Note that, under this definition, merely compiling appropriate self-calls as jumps is not sufficient to achieve full tail-recursion. Instead, we syntactically divide all sub-expression positions in the source *language* into two classes: *tail* (or *reduction*) *position* and *subproblem position*. In the simple expression `(if` *predicate consequent alternative*`)`, the *predicate* is a subproblem position, while both *consequent* and *alternative* are in reduction positions. This syntactic notion can be easily extended to arbitrarily nested sub-expressions. Operationally, if we think in terms of a simple framed stack model for the control state of a program, procedure calls appearing in subproblem position must initiate a new frame by saving a return address before jumping to the destination; those appearing in tail position must remove their frame from the stack and then jump to the destination.

2 Compilation Issues

There have been compilers capable of converting a source language with higher-order procedures and requiring a tail-recursive implementation into another language lacking both of these features since at least 1976[15, 16]. Because of its ubiquity, the language `C` has become a popular target language for these compilers[13, 20, 1, 14, 19]. But there are difficulties that arise from the choice of `C` as a target language:

1. Fully tail-recursive languages (e.g. SCHEME, `ML`, and DYLAN[3]) consider procedure call to be "`goto` statements that happen to pass arguments,"[16] while `C` implementations are free to treat procedure call and `goto` completely separately. Worse yet, `C` implementations are encouraged to do so by restrictions on labels: they may be used *only* in a `goto` statement, and the statement must appear within the lexical scope of the label itself. Thus transfer of control via `goto` is syntactically restricted by `C` to occur only within a single lexical block (and hence only within a single procedure). Since our source languages require the ability to compute a destination address, the `C` label is not an adequate representation for our procedures. But, since our source languages also require fully tail-recursive behavior, the `C` procedure does not suffice either.

2. Both `SML` and SCHEME require a first-class procedure, `call-with-current-continuation`, which produces an object that (conceptually) saves the state of the continuation stack for later use. This object is treated in the source language indistinguishably from any other procedure, so the compiler is unable in the general case to determine whether a procedure call will result in the invocation of a primitive procedure, a procedure defined explicitly in the source code, or the procedure produced by a call to `call-with-current-continuation`.

3. The symmetry between procedure call and procedure return, which is highlighted by continuation passing style (CPS) conversion, is exploited by the native code generators for the SCHEME compilers we are using. Thus, from the code generator's point of view, the destination address of a procedure call may also be the location to which a function is expected to return its value.

4. When compiling to a native machine language a procedure can always be represented as a tagged pointer to machine instructions. For procedures with free references to non-trivial enclosing environments (i.e. closures) the compiler can sometimes arrange to place these instructions at a known distance from the captured environment. Free variable references can then be compiled to simple PC-relative addresses. This powerful technique cannot be used when compiling to `C` because there is no portable way to control the relative placement of the code and data areas.

   This particular difficulty arises, ultimately, only in dynamically typed languages (such as SCHEME, COMMONLISP[17], and DYLAN), where the compiler cannot in general decide when an object being manipulated is a procedure. In a statically typed language (e.g. `ML`) the compiler can always represent a procedure as two words embedded into any data structure where the procedure can be included, since any procedure that may manipulate such data structure can be specialized to handle the flattened representation. In practice, however, polymorphic types introduce the same difficulty in order to avoid potentially explosive replication of code.

5. While an assembler takes time and space roughly proportional to the size of the input code, modern `C` compilers, even without optimization, do not. Directly compiling an entire SCHEME source file into a single `C` procedure would permit simple cases of tail-recursion to be implemented using `goto` statements; but the resulting procedure is often too large for a `C` compiler to handle effectively. For example, the `C` compilation of a single (admittedly large) file in the MIT SCHEME runtime system using this technique required over 120 megabytes of swap space and took about five hours using a (roughly) 40 `SPECint92` HPPA processor.

3 Related Work

There has been a good deal of work in compiling higher-order languages to `C`, starting with Bartlett's work[1] on a compiler from SCHEME to `C` in 1989 and including the more recent work on compiling HASKELL[13], `SML`[20], and the Bigloo[14] and Hobbit[19] implementations of SCHEME. This work can generally be divided into two groups, based on the goals set for the work: Bartlett's compiler, Bigloo, and Hobbit are motivated primarily by a desire to interface from SCHEME to code written directly in `C`; our work, the Glasgow HASKELL compiler, and `SML` work are motivated by a desire to easily port the system to new computer systems. This difference in goals leads to a major difference in philosophy: the work in the first group has taken the requirement for tail-recursive behavior as a desirable feature, while the latter efforts regard it as a mandatory part of the implementation. (In fact, the Hobbit compiler regards the requirement for higher-order procedures as a desideratum as well.)

Not surprisingly, then, the techniques used by the two groups are somewhat different. Bartlett has concentrated primarily on the difficult issue of garbage collection in a mixed memory management system. His system provides tail-recursive behavior within a single module (in simple cases), but all top-level procedures are generated to adhere to the `C` calling convention. This makes it quite easy for SCHEME programs both to call external procedures written in `C` and to be called back from external `C` procedures; but calling one of these procedures always requires space on the `C` runtime stack.

The work on HASKELL and `SML` are both closer to our own work. The mechanisms developed for the `SML`-to-`C` translator described in [20] are similar to our own, although done in the context of a statically typed language. It is hardly a surprise, then, that their `C` back-end is also roughly a factor of two slower than their native code generator.

Perhaps the most interesting comparison is to the work on HASKELL reported in [13]. Here the goal was to gain portability and "get code that appears to be significantly better than we could generate using any hand-built code generator." The performance analysis given here leads us to conclude rather strongly to the contrary: in only 4 benchmarks out of 60 did we see *any* performance improvement when using `C`, and even then the best improvement was less than 20%. Furthermore, in examining the technique used in the Glasgow HASKELL distribution (version 0.16) we found that the compiler generates distinctly non-portable code: it runs only with the `gcc` compiler, uses in-line assembly language, and embodies an assumption which is invalid on several architectures.

4 Mechanisms

Both GAMBIT and MIT SCHEME use a single stack to store return addresses and variable bindings. They divide the stack into frames, and satisfy the full tail-recursion requirement by maintaining a simple invariant: a frame persists on the stack only as long as it is in the lexical chain[1] of an *active* frame on the stack. The current (topmost) frame is active, as is any frame awaiting a value from one of its subproblems.

While most `C` implementations also use a stack for storing variables and return addresses, there is no requirement that they maintain the tail-recursion invariant. Both SCHEME systems chose to continue with their stack-based implementation of environments, and simply emulated all of the stack operations using a `C` array and pointer operations. The translation is straightforward and most `C` compilers optimize this kind of array and pointer arithmetic. Once this decision has been made, it is simple to implement `call-with-current-continuation` portably using any one of several well-known techniques[2, 8].

Both systems distribute the program's executable `C` code among a set of `C` procedures called *host procedures*. Each host procedure contains a set of *control points*: places in its code that can be the target of a control transfer (i.e. procedure entry points and return points). A procedure object is basically an encoding of a control point from which it is easy to find the corresponding host procedure and the correct control point within that host procedure.

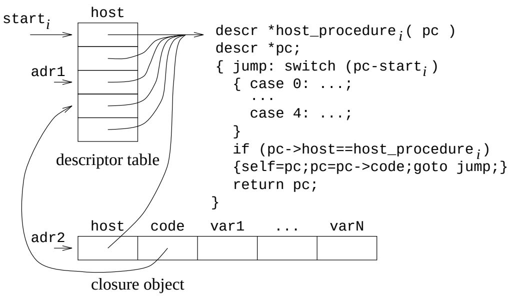
The two systems use slightly different encoding methods to support separate compilation (Figure 1). They both generate, with each separately compiled module, a table of descriptors having one entry per control point in the module. A non-closure procedure is represented as a tagged pointer to the descriptor for its entry point. Return addresses are represented in the same way (as if the program had been CPS-converted). In MIT SCHEME, a globally unique index is assigned to each control point in the program at program initialization time and the corresponding descriptor is set to this index. The indices for all the control points within a given host procedure are consecutive. A table of all the control points in the program is also created (the *global host table*). Each entry contains a pointer to its host procedure and the starting index for that host procedure. In GAMBIT, a descriptor is simply a pointer to the corresponding host procedure. To enable the selection of the correct control point within a host procedure, a pointer to the descriptor of the first control point in host procedure "i" is kept in $\texttt{start}_i$.

As in Steele's Rabbit system[16], there is a driver loop (the *dispatcher*) which is responsible for transferring control between host procedures. The dispatcher maintains in `pc` (a `C` variable local to the host procedure) the control point to be jumped to. At each iteration of the loop, the corresponding host procedure is obtained from `pc` and called. Execution begins by setting up `pc` to a predetermined control point (the SCHEME equivalent to a `C main` procedure) which starts execution of the program.

The dispatcher passes `pc` as an argument to host procedures to enable them to determine which of their control points is the target of the control transfer. The details of this computation differ between MIT SCHEME and GAMBIT (see Figure 1). Any SCHEME arguments or return values needed at the control point are passed by a completely independent SCHEME calling convention (a combination of values on the SCHEME stack and `C` global variables serving as virtual registers). In the case of MIT SCHEME, some free variables are accessed using compile-time known offsets from `pc`. The execute caches and other structures described in [11] are also accessed in this way.

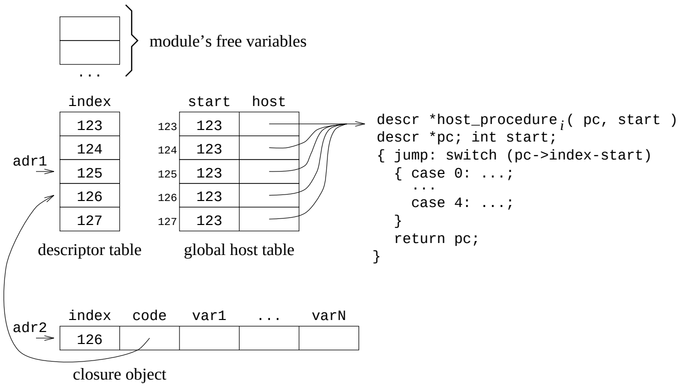
Host procedures never call other host procedures directly. Instead, they return a value to the

[1] GAMBIT uses a variation on lambda-lifting[10] to flatten the environment structure so that lexical chains are often of depth one. MIT SCHEME uses a combination of static analysis and runtime conditionals[6] to preserve the invariant.




```
void dispatcher()
{ descr *pc = initial_pc;
  while (1)
    pc = pc->host( pc );
}
```

GAMBIT




```
void dispatcher()
{ descr *pc = initial_pc;
  while (1)
  { host *h = &global_host_table[ pc->index ];
    pc = h->host( pc, h->start );
  }
}
```

MIT SCHEME

Figure 1: C procedures generated by GAMBIT and MIT SCHEME

dispatcher: the new control point to jump to. It is the dispatcher's role to transfer control to the appropriate host procedure. Thus, a host procedure can be thought of as a way to encapsulate a number of SCHEME control points; it receives as input the current control point and returns as output the next control point to be invoked. The code within the procedure manipulates explicit representations (written in C, of course) for the registers and the stack that would have been present in a native back-end.

Host procedures start with a `switch` statement that branches to the correct control point (a C labeled statement) based on the `pc` argument. In general, the code branched to corresponds to a single basic block which terminates by loading the descriptor for a desired destination back into `pc` and jumping back to the `switch` statement. The `switch` has one entry for each control point within the host, and the default action (which occurs when `pc`'s host is not the current procedure) is to return `pc` to the dispatcher (but see below for the handling of closures in GAMBIT).

Notice that this technique guarantees that the C stack never grows in size by more than a constant amount, since SCHEME code invokes code in other C procedures only after first returning from the current C procedure. This guarantees correct tail-recursive behavior, assuming that the SCHEME code was compiled into code that correctly handles the explicit SCHEME stack. At the same time, the cost of a C return, dispatch, and call is not trivial, as our measurements (below) indicate. Therefore one important optimization performed by both compilers is the detection of jumps to known control points in the same host procedure. When these are discovered they are converted into the corresponding C `goto` statements[2]. In all other cases, the compilers emit C code to compute the destination procedure representation (as described above), store it into `pc`, and then jump back to the `switch` using a `goto`.

[2]This reflects our current choice to have every SCHEME compilation unit – a file or a procedure depending on compilation switches – generate exactly one host procedure; thus known destinations do not cross host procedure boundaries.

Closures are handled slightly differently in MIT SCHEME and GAMBIT. In MIT SCHEME, a closure is not distinguished from a non-closure procedure by either the driver loop or the internal dispatch mechanism. A closure contains an entry index, distinct from its parent procedure, but included in the mapping mechanism used by the driver loop, which causes the internal dispatch mechanism to jump to the closure's code. In GAMBIT, a closure looks like a normal descriptor so the dispatcher will transfer control to the correct host procedure. However, since `pc` does not point within the descriptor table, the host's internal `switch` will fall through to the default case. The code placed there is a "closure handler" that will detect that `pc` corresponds to a closure and will proceed to transfer control to the closure's code pointer after having saved a pointer to the closure in `self` so that the closure's code can access the variables. Note that only host procedures which contain closure entry points need to include this closure handler. There is an interesting tradeoff here: the MIT SCHEME mechanism makes the cost of calling a closure and a non-closure procedure the same, but each requires a reference into the descriptor; the GAMBIT approach makes calling a closure more expensive, but calling a non-closure procedure does not require a reference into the descriptor. We have not measured the effect of this design decision in either case.

Finally, both systems generate code to cache frequently accessed resources (such as `pc`, the stack pointer and the virtual registers) for the duration of a host procedure. These resources are located in global variables and it would be inefficient if every access required a memory reference. Instead, the resources accessed within the host are copied into local variables on entry to the host procedure and (if modified within the host) are copied back to memory on exit. Examination of the assembly code generated by the `C` compilers shows that these local variables are essentially always allocated to machine registers (a "`register`" declaration is included to help out the `C` compiler).

5 Performance Measurements

The goal of our performance measurements has been to determine two things: the relative performance of a native code generator and a `C` code generator; and the source of the performance differences. The first is always a difficult question for any large system, since it involves a very large number of components, each of which affects the overall performance in a largely unknown manner. Some of the components include the quality of the `C` compiler being used (including various optimization flags), the cache performance of programs compiled one way versus another, and the architecture of the machine being measured. We report here only the simplest measurements, and we have based them on the Gabriel benchmarks[5] initially written in COMMONLISP and directly translated into SCHEME. To this set of benchmarks we've added two large programs originally written in SCHEME, `conform` and `peval`. `Conform` computes an uninteresting function, but it is the only benchmark that makes heavy use of higher-order procedures; `peval` is a simple partial evaluator. Figure 2 summarizes the results of these experiments.

We now turn our attention to the reasons for these performance differences. We begin with the hypothesis that there are three dominant components to the performance difference (for a given computer, operating system, `C` compiler, and SCHEME system): the cost of a SCHEME procedure call or return; the cost of arithmetic overflow checking; and differences in the optimization strategy for the SCHEME and `C` compilers.

Since all of the benchmark programs use arithmetic on small integers, the performance numbers in Figure 2 were generated using SCHEME compiler flags that suppressed the type dispatch and overflow checking that would normally be present. We performed a direct comparison, using MIT SCHEME on the HPPA, and found the cost of the type dispatch and overflow checking to be 7% on both the native and `C` code generators. This can be compared to the figures of an average of 2% and a maximum of 8% in [18]. It is therefore not surprising that the overall measured performance

	GAMBIT	MIT SCHEME		
	68K	68K	Alpha	HPPA
Boyer	2.30	2.62	2.25	2.10
Browse	2.91	1.90	1.34	1.82
Conform	2.68	3.20	2.74	2.63
Cpstak	1.13	0.95	1.03	0.57
Ctak	2.21	1.17	1.09	0.86
Dderiv	4.15	4.10	3.02	3.99
Deriv	3.85	3.61	2.89	3.79
Destructive	1.11	1.43	1.00	0.79
Div	1.00	1.91	1.71	1.33
Peval	2.58	4.27	4.02	5.16
Puzzle	1.14	1.95	1.69	0.97
Tak	2.08	2.42	1.72	1.45
Takl	1.31	2.35	2.38	1.67
Traverse	1.76	2.31	1.63	1.51
Triangle	1.42	2.13	1.57	1.14
Arithmetic Mean	2.11	2.42	2.01	1.98
Geometric Mean	1.91	2.23	1.85	1.64

Figure 2: Ratio of C to Native compiled speed

difference between the `C` and the native back-ends is not affected by this particular optimization (less than 2% in our measurements).

To isolate the remaining two performance components, we developed a simple performance model that divided all procedure calls and returns into three distinct categories: those in which the destination was known at compile time, those in which the destination is not known but turns out (at run time) to be within the same host procedure, and those that cross procedure boundaries. We used the test programs shown in Figure 3 to measure the cost of each of these. These were compiled to suppress the use of generic arithmetic, and so that each top-level SCHEME procedure generated a single host procedure. The loops were then run 10 million times, compiled either direct to native code or to `C` code. The results are shown in Figure 4.

These measurements provide a good deal of insight into the performance of the `C` back-end. In the native back-ends, the cost of a procedure call (including the execution of a trivial body) to an unknown address never costs more than $\frac{.49}{.163} \approx 3.01$ times (GAMBIT on the 68K) the cost of a call to a known address. Such a difference is noticeable, to be sure, but will be less important for more realistic programs with larger procedure bodies. By contrast, in the `C` back-end on Alpha this factor grows to $\frac{.144}{.023} \approx 6.3$ for calls *within* the same host procedure, but goes up to an astounding factor of $\frac{.833}{.023} \approx 36.2$ for calls to other host procedures. This large factor motivates an attempt to keep as many procedure calls as possible within the same host procedure.

MIT SCHEME and GAMBIT provide two different compilation modes to control the number of cross procedure calls: *per-procedure compilation* in which every top-level SCHEME procedure becomes a separate host procedure and *block compilation* in which the entire SCHEME source file becomes a single host procedure. The numbers shown in Figure 2 are those from block compilation, and constitute a best-case performance of the benchmarks given the standard source code. Figure 5 shows the mean performance figures from Figure 2 along with two other compilation techniques. *Full per-procedure* compilation, basically a worst case analysis, has the entire runtime system as well as the benchmark code compiled in per-procedure mode. Thus, all procedure calls to unknown addresses cross host procedure boundaries. *Complete benchmark* compilation modifies the benchmark

```
(let ()

  (define (proc1 n)
    (if (> n 0)
        (proc2 (- n 1))
        n))

  (define (proc2 n)
    (if (> n 0)
        (proc1 (- n 1))
        n))

  ; All calls to known addresses
  (show-time (lambda () (proc1 #e1e7))))
```

```
(define (proc1 n)
  (if (> n 0)
      (loop1 (- n 1))
      n))

(define (proc2 n)
  (if (> n 0)
      (loop2 (- n 1))
      n))

; All calls within a single procedure
(set! loop1 proc1)
(show-time (lambda () (proc1 #e1e7)))

; All calls cross procedure boundary
(set! loop1 proc2) (set! loop2 proc1)
(show-time (lambda () (proc1 #e1e7)))
```

Note: These are slightly simplified. The actual programs are more elaborate in order to defeat certain SCHEME compiler optimizations.

Figure 3: Exploring the costs of procedure call

MIT SCHEME

Destination	68K C	68K Native	68K δ	Alpha C	Alpha Native	Alpha δ	HPPA C	HPPA Native	HPPA δ
Known	1.23	0.774		0.161	0.145		0.323	0.284	
Within procedure	1.55	0.856	0.327	0.211	0.140*	0.050	0.509	0.366	0.186
Across procedures	5.10	0.854	3.87	0.881	0.140*	0.721	2.221	0.364	1.90

GAMBIT

Destination	68K C	68K Native	68K δ	Alpha C	Alpha Native	Alpha δ
Known	0.163	0.163		0.023		
Within procedure	1.10	0.490	0.936	0.144		
Across procedures	4.29	0.490	4.13	0.833		

Numbers are in microseconds per iteration, based on 10 million iterations of the loop.

*: We believe these small, unexpected performance improvements result from a shift in the instruction stream allowing for dual issuing in the unknown destination case but not in the known address case.

Figure 4: Measured cost of procedure call

programs to include the source code for any procedures they require that are normally supplied by the runtime system. These self-contained benchmark programs are then block compiled. In this situation, all procedure calls are guaranteed to be within the same host procedure.

	Gambit	MIT Scheme		
Compilation Technique	68K	68K	Alpha	HPPA
Full per-procedure	2.97 (2.50)	2.73 (2.42)	1.98 (1.80)	2.35 (1.87)
Block compilation	2.11 (1.91)	2.42 (2.23)	2.01 (1.85)	1.98 (1.64)
Complete benchmark	1.73 (1.63)*	2.00 (1.90)*	1.98 (1.88)*	1.34 (1.27)

*: Because of their size, the benchmarks `conform` and `peval` could not be compiled with optimization by the `C` compiler. They are not included in this calculation.

Arithmetic (geometric) mean of $\frac{\text{C code}}{\text{native code}}$ over the entire set of benchmarks shown in Figure 2. See the text for a description of the compilation techniques.

Figure 5: Performance variation by compilation technique

The information in Figure 5 tells us operationally the performance impact of each compilation technique. There remains the interesting question, however, of whether the overall performance shown there can be predicted solely on the basis of the cost of procedure calls, or whether the third performance component, optimization by the `C` compiler, has any noticeable effect on these numbers. This last question is extremely hard to answer, but we can begin to address it by using the information in Figure 4 along with some careful analysis of the code for the benchmarks used for that figure. We are interested in estimating how much of the performance difference shown in Figure 2 can be attributed solely to the difference in procedure call efficiency. This is done by using the measurements in Figure 4 to estimate the cost of each kind of procedure call, counting the number of each kind of call that occurs in a given benchmark, and comparing estimated performance to measured performance: $Est(\texttt{C}) = Time(\text{native}) + N_{int} * \delta_{int} + N_{ext} * \delta_{ext} + N_{known} * \delta_{known}$, here the δ function is the estimated additional cost of a call to an unknown address within (int) or across (ext) host procedures or to a $known$ address in the `C` code compared to the native code, and the N function is the count of the number of occurrences of each kind of jump.

Careful examination of the assembly code generated by both the native and `C` back-end for the simple loop with all known addresses used in Figure 4 revealed that δ_{known} was precisely 0, with all of the difference coming from factors unrelated to the actual jump instruction. Given this, we are able to directly calculate δ_{int} and δ_{ext}: $\delta_{int} = time_{int} - time_{known}$, where $time_\alpha$ is the time in microseconds derived from the appropriate entry in Figure 4 (similarly for δ_{ext}). These values are shown in the column labeled δ in Figure 4.

To calculate the values of N_{int} and N_{ext} (the value of N_{known} being irrelevant since δ_{known} is 0), we changed the compiler to force *all* unknown procedure calls to appear to be external to the current host procedure, and we modified the driver loop to count all external calls. These two changes allow us to count the number of calls to unknown addresses and subdivide these between those internal and external to the host procedure. Notice that the values of N_α are determined by the compiler we use and are unaffected by the architecture, while the δ_α values are determined by both the architecture and the compiler.

Given this information, it was a direct (if time-consuming) task to estimate the performance of the `C` back-end for any given benchmark under any compiler on any architecture. The predictions, however, were consistently lower than the measured values; typically about a factor of two lower than measurement. After some consideration we concluded that it was possible that the loops shown in Figure 4 were *too* simple: the `C` compiler was able to optimize them in ways that were not possible with the larger benchmarks (for example, they contain few control points and the `C` compiler on several machines converts `switch` statements with a small number of cases into more efficient "jump chains"; they also use fewer registers so the call and return spills fewer than is typical of the benchmark procedures).

MIT Scheme

			68K $\delta_{int} = 1.31$ $\delta_{ext} = 4.42$		Alpha $\delta_{int} = 0.211$ $\delta_{ext} = 0.755$		HPPA $\delta_{int} = 0.316$ $\delta_{ext} = 1.77$	
	N_{int}	N_{ext}	Meas.	$\frac{Meas.}{Est.}$	Meas.	$\frac{Meas.}{Est.}$	Meas.	$\frac{Meas.}{Est.}$
Boyer	2533209	303052	11000	1.25	1850	1.32	2840	1.11
Browse	95591	45604	1080	1.33	338	1.12	385	1.29
Conform	4008909	474115	15000	1.63	3560	1.81	4670	1.45
Cpstak	47710	2	502	0.829	113	0.947	247	0.534
Ctak	63612	190829	19400	1.12	5210	1.05	9490	0.829
Dderiv	210003	200002	1940	1.64	466	1.75	746	1.74
Deriv	150003	160002	1560	1.53	363	1.67	611	1.73
Destructive	10975	2	358	1.37	72	0.972	171	0.772
Div	244805	2	892	1.22	189	1.23	291	0.976
Peval	295658	481355	4050	1.62	929	2.17	1520	1.93
Puzzle	40696	10224	3110	1.9	723	1.66	1140	0.947
Tak	1113162	2	2820	1.18	499	0.908	896	0.882
Takl	647985	2	1590	1.09	270	1.17	487	0.965
Traverse	13153074	202	34900	1.17	7220	1.00	11100	0.937
Triangle	11608244	3102	49800	1.48	9620	1.16	17000	0.888
Arithmetic Mean			9870	1.36	2090	1.33	3440	1.13
Geometric Mean			3530	1.33	768	1.28	1280	1.07

All numbers are rounded to three significant figures. Measured times are in milliseconds. The values of δ, in microseconds per call, were computed from a lambda-lifted version of `destructive` as described in the text.

Figure 6: Estimated vs Measured Performance

To improve our performance prediction, we chose one of the smaller benchmark programs (`destructive`) and rewrote it in a fully lambda-lifted version. We then compiled this procedure and ran it to derive a new set of values for the δs. These values of δ are shown in Figure 6, along with the values of N_{int} and N_{ext} for each benchmark, our performance prediction for the benchmark, and the actual timing.

Overall, our predictions are in very close accord with measurement, although there are individual anomalies. This leads us to conclude that almost all of the performance difference between our native compilers and our `C` back-end can be accounted for by the overhead required by the use of `C`'s procedure call mechanism to implement tail-recursive behavior. Where our predictions differ from measurement, we generally *underestimate* the performance of the programs, indicating that the translation to `C` is causing even more performance problems than those introduced by procedure calls alone. Since all of our benchmark results (except as indicated in Figure 5) were performed with the maximum level of optimization permitted by our `C` compilers, it follows that this optimization is simply unable to overcome this single intrinsic cost: `C` is simply not a good language for implementing tail-recursion in any but the smallest example programs.

6 Conclusions

The system structure described by Steele for his implementation of Scheme almost twenty years ago continues to be a viable technique for compiling a higher-order language to a first-order language and preserving fully tail-recursive semantics: there is an outer driver loop that dispatches to a set of procedures, each representing some set of control points, and these in turn return another control point to be dispatched. This basic mechanism is used by our systems, by the `SML` to `C` compiler from CMU, and by the Glasgow Haskell compiler. In all cases where the performance of this system can be compared to the performance of a native compiler, the `C`-based system performs at less than half the speed of the native system.

The basic difficulty with the structure is that the cost of a `C` procedure call, return, and dispatch can be as much as a factor of 36 times slower than a call implemented with a `goto` instruction. Since `C` compilers can accomodate only moderately sized input procedures even the best compiler produces code that must cross the `C` procedure boundary at runtime; and the more often this occurs, the worse the performance. This problem is exacerbated by the need to compile the large runtime library for languages like Scheme and `SML` separately from user code. Thus every call by the user to a runtime utility that cannot be open coded at compile time will involve the high overhead of a cross-procedure jump.

While we have not been able to make detailed measurements of all the parameters involved in the performance differences, it is clear that the differences are real and are not likely to be overshadowed by any additional improvements in the technology of `C` compilers. The problems are fundamental to the language design of `C` when the goal is a tail-recursive implementation. The trade-off is clear: lose a factor of two (or more), or invest the labor to make a native code implementation. Given the current technology, it is relatively easy to make a portable code generator (for example, by using a rule-base RTL similar to that in `gcc`) and amortize the implementation cost across a number of back-ends.

References

[1] Joel Bartlett. `Scheme->C` a portable Scheme-to-C compiler. Technical report, Digital Equipment Corp. Western Research Lab., 1989.

[2] William Clinger, Anne Hartheimer, and Eric Ost. Implementation strategies for continuations. In *Proc. ACM Symp. LISP and Functional Progr.*, pages 124–131, 1988.

[3] Apple Computer. *Dylan^{tm} An Object-Oriented Dynamic Language.* Apple Computer, Inc., Cupertino, CA, 1992.

[4] Marc Feeley and James S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Proc. ACM Symp. LISP and Functional Progr.*, June 1990.

[5] Richard P. Gabriel. *Performance and Evaluation of LISP Systems.* MIT Press, 1985.

[6] Chris Hanson. Efficient stack allocation for tail-recursive languages. In *Conference on Lisp and Functional Programming*, Nice, France, June 1990. ACM.

[7] Chris Hanson et al. MIT Scheme reference manual. Technical Report 1281, Mass. Inst. of Technology, Artificial Intelligence Laboratory, Cambridge, MA, January 1991.

[8] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proc. ACM Prog. Lang. Design and Impl.*, pages 66–77, 1990.

[9] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language.* Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[10] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* International Series in Computer Science. Prentice-Hall, New York, 1987.

[11] James Miller and Guillermo Rozas. Free variables and first-class environments. *Journal of Lisp and Symbolic Computation*, 1991.

[12] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[13] Simon L Peyton-Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology*, 1992.

[14] Manuel Serrano. Bigloo user's manual version 1.4. INRIA–Rocquencourt, August 1993.

[15] Guy Lewis Steele Jr. Debunking the 'expensive procedure call' myth. AI Memo 443, Mass. Inst. of Technology, Artificial Intelligence Laboratory, Cambridge, MA, October 1977.

[16] Guy Lewis Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Mass. Inst. of Technology, 1978.

[17] Guy Lewis Steele Jr. *Common LISP The Language.* Digital Press, second edition, 1990.

[18] P. Steenkiste and J. Hennessy. Tags and type checking in lisp: Hardware and software approaches. In *2nd Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 50–59, October 1987.

[19] Tanel Tammet. Documentation for Hobbit version 2. Department of Computer Sciences, Chalmers University of Technology, 1993.

[20] David Tarditi, Anunrag Acharya, and Peter Lee. No assembly required: Compiling standard ML to C. Technical report, School of Computer Science, Carnegie Mellon University, November 1990. CMU-CS-90-187.