No Assembly Required: Compiling Standard ML to C

David Tarditi, Peter Lee, and Anurag Acharya
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

Abstract

C has been used as a portable target language for implementing languages like Standard ML and Scheme. Previous efforts at compiling these languages to C have produced efficient code, but also compromised on portability and proper tail-recursion. We show how to compile Standard ML to C without making such compromises. The compilation technique is based on converting Standard ML to a continuation-passing style λ-calculus intermediate language, and then compiling the continuation-passing style λ-calculus to C. The generated code achieves an execution speed that is about a factor of two slower than a native code compiler. The generated code is highly portable, yet still supports advanced features like garbage-collection and first-class continuations. We analyze aspects of the compilation method that lead to the observed slowdown, and suggest changes to C compilers that would better support the compilation method.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—compilers; D.3.2[**Programming Languages**]: Language Classifications —Standard ML, Scheme

General Terms: Languages

Additional Keywords and Phrases: continuation-passing style, compilation to C

1 Introduction

With the profusion of new computer architectures and programming languages, compiler writers are increasingly faced with making difficult compromises be-

This research was partially supported by the National Science Foundation under PYI grant #CCR-9057567. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation or the US Government.

tween the efficiency of compiled programs and retargetability of the compiler. One approach that has been used in the past is to compile programs into the C programming language [11], in essence using C as a universal intermediate language. C makes for an efficient intermediate language because it allows relatively low-level access to the machine, yet in a number of important respects is mostly machine independent. Furthermore, it is usually safe to assume that a C compiler will be available on almost any general-purpose machine.

Bartlett has shown that it is possible to compile Scheme programs into efficient C programs [7]. In his approach, constructs in Scheme are mapped directly to similar constructs in C. Although this is a conceptually simple approach, it also leads to certain compromises. First, Bartlett's Scheme implementation, Scheme → C, fails to reflect the pragmatics of the language. Specifically, "proper" tail-recursion [16] is lost, despite a compile-time analysis of procedures for tail-recursive behavior. Second, features such as first-class continuations and garbage collection are implemented with the use of assembly code. The main point of compiling to C, though, is to avoid such machine dependencies.

We are led, then, to consider several questions. How can languages such as Scheme and Standard ML, which support first-class procedures and other powerful control constructs such as exceptions and first-class continuations, be compiled without any use of assembly language? Can languages that depend on proper tail-recursion be faithfully compiled to C? Can this be done without an unacceptable loss of efficiency, and if not, what extensions to C would be necessary to make this possible?

In this paper, we address these questions by describing our experience with compiling Standard ML [14] to C. We present a compiler that compiles the entire Standard ML language into efficient, portable C code. In doing so, we have completely avoided the use of assembly language and make only two assumptions about the architecture: that both pointers and integers are represented in 32 bits and that integer arithmetic is two's-complement.

Our compiler, sml2c, is based entirely on the Standard ML of New Jersey compiler (SML/NJ) [5]. We have replaced the code generator and slightly modified the runtime system.[1] All other modules of the SML/NJ compiler have been used "as is." As a result, our compiler is source-code compatible with the SML/NJ compiler and handles recently proposed extensions to Standard ML for first-class continuations (call/cc) [9], asynchronous signal handling [17], lightweight concurrent processes [8], and an extension for multiprocessing [15]. The implementation has been tested on a number of nontrivial programs—including bootstrapping of the compiler itself—and the generated C code has been tested, without any modification, on Sun-3s, Sparcstations, Decstations, IBM RTs, a Motorola 88000-based machine, and an 80486-based PC.

Our primary considerations in the design of the compiler were, in decreasing

[1] The present runtime system is written in C and requires an operating system compatible with Unix 4.3 BSD.

order of importance: ability to handle the full language (with extensions), faithfulness to the pragmatics of the language, and efficiency of the compiled code. The first two goals were met completely. The price we pay in runtime overhead is typically about 70 to 100% in running time as compared to the native machine code generated by the SML/NJ compiler. In this paper we describe our basic approach, and then discuss some limitations and the optimizations we used to improve significantly the performance of the generated C code. This is followed by a performance analysis, and finally some conclusions. The system is available via anonymous ftp from `dravido.soar.cs.cmu.edu: /usr/nemo/sml2c`.

2 Related Work

A number of compilers that use C as the target language have been developed. For example, besides Bartlett's Scheme compiler, an existing compiler for the Cedar language was successfully retargeted to C in order to improve portability of Cedar-based systems [6]. Good performance was reported for implementations on SPARC-based and Motorola 68020-based machines.

Our compiler differs significantly from this work in several important respects. Cedar is simpler than Scheme and Standard ML in two crucial ways: there is no pragmatic requirement of proper tail-recursion and functions are not first class. For the implementation of exception handling in Cedar programs, the compiler generates code that traverses the C call stack; this is highly dependent on the particular machine and C compiler which is used. Our implementation does not use any machine-dependent code and is completely portable across a wide class of machines.

3 The Design of the Compiler

Standard ML (SML) is a lexically scoped, call-by-value language with higher-order functions. SML has polymorphic types which are automatically inferred and checked by the compiler. Support for developing large programs is provided by a sophisticated modules system that provides for static type-checking of the interfaces between modules, as in Ada and Modula-II. A type-safe, dynamically scoped exception mechanism allows programs to handle unusual or deviant conditions. Garbage collection is provided to automate the management of heap storage.

Clearly, there is considerable semantic distance between SML and C. Features such as dynamically scoped exceptions, garbage collection, and higher-order functions pose significant problems for an efficient and portable implementation of SML in C. Recently proposed extensions to SML for first-class continuations and concurrent processes further increase this semantic distance.

3.1 Intermediate representation

The key element in the design of sml2cwas the decision to replace the code generator of the Standard ML of New Jersey (SML/NJ) compiler with one that produced C code. The SML/NJ compiler is a publicly available, freely redistributable and modifiable compiler for SML developed at AT&T and Princeton University. Input to the the SML/NJ code generator is represented in a continuation-passing, closure-passing style λ-calculus [4].

There were several reasons for this decision. The first reason was pragmatic. The compiler had to be built by one person working over a summer. This time constraint made it necessary to modify an existing compiler. The second reason was that by focusing on the translation of continuation-passing, closure-passing style programs to C, techniques developed would be applicable to many languages, in particular Scheme. Another reason was the remarkable similarity between programs represented in a continuation-passing, closure-passing style and C programs. It seem likely that these programs could be translated to highly portable C code, although it was unclear whether the C code could be made reasonably efficient.

The continuation-passing, closure-passing style λ-calculus used by the code generator is a refinement of a continuation-passing style λ-calculus (CPS).

A program is transformed into CPS by adding a continuation argument to all user functions. The continuation argument is a function that represents the future of the program. When a function has finished computing its result, instead of returning the function calls its continuation with the result as the argument. Programs that have been transformed into the particular CPS used by the SML/NJ compiler always satisfy the syntactic invariants that function calls are never nested and are always tail-recursive. Since calls are always tail-recursive, the first call to return is essentially the only call to return. Thus, a function call can be considered as a goto with arguments. Conversion of Scheme or ML programs into closure-passing style can be performed by a simple $O(n)$ transformation [3].

There are several advantages to using a CPS intermediate representation [20, 12]. First, all intermediate values are explicitly named. Second, control-flow is explicitly represented by continuations. This makes it easy to implement exceptions and first-class continuations. Third, since all function calls turn into jumps with arguments, tail-recursion elimination is achieved automatically. In addition to the SML/NJ compiler, CPS has been used successfully in the Rabbit [20] and Orbit compilers [13].

A continuation-passing, closure-passing style λ-calculus is a further refinement of CPS in which environment operations are made explicit. All functions are flattened out to one lexical level and explicit record operations are used to represent operations on closures. A description of the conversion of CPS programs into continuation-passing, closure-passing style may be found in [3].

The target machine for a continuation-passing, closure-passing style program

```
datatype cexp =                                        datatype value =
  RECORD of (value * accesspath) list * lvar * cexp      VAR of lvar
| SELECT of int * value * lvar * cexp                  | LABEL of lvar
| OFFSET of int * value * lvar * cexp                  | INT of int
| APP of value * value list                            | REAL of string
| FIX of (lvar * lvar list * cexp) list * cexp         | STRING of string
| SWITCH of value * cexp list
| PRIMOP of primop * value list * lvar list * cexp list
```

Figure 1: Internal representation of CPS used by the Standard ML of New Jersey compiler.

is conceptually simple, consisting of a heap, a set of general-purpose registers, and a set of reserved registers for the heap pointer, heap limit pointer, current exception handler, and arithmetic temporary storage. The instructions required are those typically found on most conventional machines, including loads, stores, logical operations, arithmetic operations with and without overflow checking, floating point operations, register-to-register moves, conditional branches, and jumps.

The internal representation of the CPS used by the SML/NJ compiler is shown in Figure 1. Continuation expressions, or values of type `cexp`, represent CPS programs. Most variants of the `cexp` type bind zero or more variables in the scope of another continuation expression. Variable names are represented by values of type `lvar`. `RECORD`(a, b, c) creates a tuple of values from the values in a, binds the tuple to b, and continues execution in c. `SELECT`(a, b, c, d) selects the ath field of b, binds it to c, and continues execution in d. `OFFSET`(a, b, c, d) takes the tuple bound to b and binds a new tuple starting at the ath field of a to c, and continues execution in d. `APP`(a, b) applies a to the arguments given in b. `FIX`(a, b) binds a list of potentially mutually-recursive functions in a and continues execution into b. Each element of a is a tuple $(f, args, body)$, where f is the name of the function to be bound, and the function is $\lambda args.body$. `SWITCH`(a, b) is a case statement where execution continues with the ath element of b. `PRIMOP`(a, b, c, d) applies a primitive operator a to a tuple of values b, binds some variables c, and continues execution into one of the continuation expressions given in d. Primitive operators include integer arithmetic operators, floating point arithmetic, integer relational operators, floating point relational operators, and operators on arrays.

An example CPS program is given in Figure 2, which shows a simple SML source program and its CPS intermediate representation. By convention, the intermediate representation shown in the figure is presented in a simplified pseudocode style. In particular, infix operators have not been expanded to their true CPS form. Note that the converted code is remarkably similar to C code, except

```
fun foo x =                         fun foo1(x,c) = bar1(x,c)
   let fun bar 0 = "bar"            and bar1(x,c) = if x=0 then c "bar"
         | bar x = baz(x-1)                         else baz1(x-1,c)
       and baz 0 = "baz"            and baz1(x,c) = if x=0 then c "baz"
         | baz x = bar(x-1)                         else bar1(x-1,c)
   in bar x
   end
```

Figure 2: An SML program and its intermediate representation

for the fact that all function calls are tail-recursive.

3.2 The compilation model

One may note the lack of a stack in the target machine. In fact, the issue of whether to allow a stack in the target machine is a fundamental design decision. In principle, the presence of higher-order functions means that activation records for procedures must be allocated on the heap since the environment structure will have the structure of a tree. In practice, heap-allocation of activation records is common in compilers that use a CPS representation, for example the Rabbit and SML/NJ compilers.

With a suitable amount of care, a stack can be used to store most activation records [13]. This leads to the possibility of flattening out all functions in an SML program to a single lexical level, and then mapping SML functions directly to corresponding C functions, thereby making use of the C call stack. Closures would then be represented by records containing a C function pointer and the environment for the function. (Bartlett uses a similar approach in Scheme → C.) Unfortunately, this makes it difficult to implement garbage collection in a portable manner. This is due to the fact that finding the roots for garbage collection requires a nonportable traversal of the C stack. It also makes it difficult to preserve proper tail recursion.[2] Finally, the use of the stack makes it impossible to obtain a portable implementation of the call/cc extension to Standard ML (though efficient stack-based approaches have been proposed [10]). Not even the longjmp and setjmp features provided by some C implementations can provide this functionality.

For these reasons, we adhere to the compilation model used by the SML/NJ compiler which uses heap-allocation of activation records and makes no use of a stack.

[2]Some C compilers implement some tail-recursion optimization for C functions which are immediately tail recursive (that is, they call themselves). They generally fail on mutually recursive tail-calls and tail-calls to functions that are not known until runtime, which are both quite frequent in a programming style using higher-order functions.

4 Code Generation

It is straightforward to implement the target machine resources in C. The registers and heap are implemented by integer arrays. (Use of actual machine registers is discussed in the next section.) Most of the target machine instructions are also straightforward. Only the jump and arithmetic operations (with overflow checking) complicate matters. Recall that function calls are tail-recursive after CPS conversion and are thus compiled into jumps with arguments passed in registers. The implementation of jump is problematic because labels are not first-class values in C. In particular, we cannot store a label in memory or jump to it from an arbitrary point in a C program. The only way to obtain the address of a block of C code is to encapsulate it in a C function. But if C function calls are used naively, the stack would quickly overflow.

Instead, we borrow a mechanism from the Rabbit compiler, called the *UUO handler*. We refer to it as the *apply-like* procedure. The apply-like procedure emulates the apply operation. Code execution begins with the apply-like procedure calling the first function to be executed. When this function wants to call another function, it returns to the apply-like procedure with the address of the next function to call. In general, when a function f needs to call another function g, it returns the address of g to the apply-like procedure, which then calls g. Arguments to functions are passed in memory in the simulated machine registers. In this way the depth of the stack of activation records never grows to more than two.

The implementation of integer arithmetic is complicated by the fact that overflow checking is not normally provided in C. The definition of Standard ML, however, mandates overflow checking. To implement overflow checking, bit-level checks on the operands and the result are used.

Register allocation is based on the spilling transformation used in the SML/NJ compiler. This guarantees that at every point in the program, a sufficient number of registers are available for all variables. Our code generator then performs register assignment on the fly via register-tracking. A variety of heuristics are used in order to minimize shuffling of registers at function calls.

Figure 3 contains a simplified version of the C code that would be generated for the sample SML code shown in Figure 2. The figure also shows the apply-like procedure stripped of its initialization code.

5 Runtime Interaction

The implementation of the CPS target machine as a C program is supported by a modified version of the SML/NJ runtime system [2], which provides operating system services, garbage collection, and asynchronous signal handling. The generated code and the SML/NJ runtime system work together as coroutines. For the purposes of multiprocessing, per-processor state is threaded throughout

```
int foo1(),bar1(),baz1();

int apply(start,r)
int (*start)();
int *r;
{ while (1)
    start = (int (*)()) (*start)(r);
}

int bar1 (r)
int *r;
{ if (r[1]==0)
   { r[1] = "bar"; return r[2]; }
  else
   { r[1] = r[1] - 1; return baz1; }
}

int foo1 (r)
int *r;
{ return((int) bar1); }

int baz1 (r)
int *r;
{ if (r[1]==0)
   { r[1] = "baz"; return r[2]); }
  else
   { r[1] = r[1] - 1; return bar1; }
}
```

Figure 3: C code and apply-like procedure (simplified)

the runtime system and the generated code. As far as the generated code is concerned, the per-processor state is maintained in a dedicated element of the register array.

Execution begins in the runtime system, which then transfers control to the generated code. The generated code must transfer control back to the runtime system for successful program completion, requests for operating system services, machine faults such as floating point overflow, garbage collection, and asynchronous signal handling.

The code transfers control to the runtime system by doing a *longjmp* back into the apply-like procedure, and returning. The target of the *longjmp* is set by doing a *setjmp* on the first pass through the apply-like procedure. Of course, the jump buffer is part of the per-processor state. The beauty of this approach is that it allows the heart of the apply-like procedure to be extremely simple. In fact, the main loop of the procedure in actual use is merely an unrolled version of the code shown in Figure 3.

Garbage collection is initiated by having the generated code call the garbage collector as a subroutine when the heap is exhausted. Like the SML/NJ compiler, our compiler avoids heap checks on every allocation by inserting a single heap check at the beginning of every function. The heap check is accomplished by adding the heap pointer and an offset that is the maximum amount of allocation that could be done along any path through the function and checking that this value is less than the heap limit pointer. If this is not true, the garbage collector is called. Code generation ensures that at such points all roots are in the register array.

6 Optimizations

There are a number of problems with the implementation described in Section 4. The generated C code fails to make effective use of actual machine registers, functions calls are expensive, and arithmetic involves a high overhead for portable overflow checking. The following subsections describe optimizations that address these problems.

6.1 Register Caching

Recall that the target machine registers are implemented by an integer array. Global variables could be used, but most C compilers will not move global variables into real machine registers[18]. Furthermore, this approach is not viable for a multiprocessor implementation. To make effective use of real registers, we cache target machine registers in local C variables for the duration of function calls. We then depend on the C compiler to place the local variables in registers when possible.

The register caching optimization uses a simple static count of the number of uses of a register along each possible path through a function body to decide whether to cache a target machine register for the duration of the function call. A consequence of the continuation-passing, closure-passing style is that function bodies are extended basic blocks. In other words, they have only forward branches, no loops. If the number of uses along any path is greater than two we cache the register. We must be careful when caching certain registers that must be valid whenever a machine fault might occur. These registers are the heap pointer register and the exception continuation register. We never cache the exception continuation register, and the heap pointer register is spilled back to the corresponding global variable before executing any instruction that may cause a machine fault. Fortunately, the only instructions which may cause machine faults in our implementation are division by zero and floating point operations, so the need to spill is fairly rare.

6.2 Function integration

Functions calls are expensive since every function call involves a return to the apply-like procedure and then a call from the apply-like procedure to the target function. This involves at least two jumps. The expense is usually more than this, since many C compilers implement the return by a jump to a piece of "postlude" code at the end of a function which cleans things up and then does the actual jump back to the apply-like procedure. In addition, values stored in actual machine registers might be saved to memory upon entry to a function and then restored when upon exit. In fact, this occurs in practice on callee-save systems even though our apply-like procedure has just one variable live across function calls. Thus, the true cost of a function call is usually three jumps and some indeterminate number of loads and stores.

In order to eliminate some of this overhead, we use an analysis of known functions. A *known* function is one whose possible call sites are all known. If all call paths to a known function f pass through a single function g, we can integrate f into the body of g. This means that f gets placed in the body of g and that all calls to f turn into goto's. We also perform direct tail-recursion elimination, turning all calls to g within its body to gotos. Function integration is useful for avoiding passes through the apply-like function. In particular, it is useful for optimizing tight tail-recursive loops.

Computing whether all call paths to a known function pass through another function can be cast as the classical problem of computing dominators in a call graph with multiple start nodes, where each unknown function is a start node. An algorithm for computing dominator information can be found in [1]. Define a maximal dominator to be a function which is only dominated by itself. This captures the intuitive concept of a highest-level dominator. We first compute the set of dominators for each known function. For each known function f, it is an easily proved fact that there is a unique maximal dominator for f. If the maximal dominator of f is not itself, then we integrate f into its maximal dominator.

Consider the SML code in Figure 2. The functions `bar` and `baz` are known and `foo` is the maximal dominator for them. Thus, we can integrate `bar` and `baz` into `foo`. Figure 2 shows the pseudo-CPS code that represents these functions as they are presented to the code generator. Figure 4 shows a simplified version of C code generated for these functions with function integration and register caching. Note that the mutually tail-recursive functions have been compiled into a tight loop with the loop counter placed in a register. (The redundant gotos appearing in the code are optimized away by most C compilers.)

6.3 Overflow checking

Finally, integer arithmetic is expensive because most C compilers provide no portable way to use the integer overflow signal. Implementing addition and subtraction with portable overflow checking requires complicated explicit bitwise tests, and a function call is needed for multiplication. For division, some comparisons and branches suffice. We can simplify the overflow checks by constant-folding them in the cases where one operand is known at compile time. For example, the overflow check for addition can be reduced from a complicated boolean expression involving five operators to a single comparison and branch and the procedure call for multiplication can be replaced with two compares and a branch.

```
int foo1 (r)
int *r;
{ register int r1;
  r1=r[1];
 goto bar1;
bar1:
  if (r1==0) { r1 = "bar"; r[1]=r1; return ((int) r[2]); }
  else { r1 = r1 - 1; goto baz1; }
baz1:
  if (r1==0) { r1 = "baz"; r[1]=r1; return ((int) r[2]); }
  else { r1 = r1 - 1; goto bar1; }
}
```

Figure 4: Optimized C code

7 Benchmarks

To measure the performance of sml2c, we tested sml2cwith and without the optimizations turned on, and version 0.75 of the SML/NJ compiler on two different architectures. The benchmark suite is a subset of the suite used in [3]. It consists of the following programs: *Life*, a program for the Game of Life; *Lex*, a lexical analyzer generator; *Yacc*, an LALR(1) parser generator; *Knuth-B*, an implementation of the Knuth-Bendix completion algorithm; and *VLIW*, an instruction scheduler that performs software pipelining on register-transfer machines. These programs range in size from 117 to 5800 lines of SML code. *Lex* was run on a lexical description of Standard ML, while *Yacc* was run on a grammar for Standard ML.

The benchmarks were run on a Decstation 5000/200 with 48 Mbytes of memory, and a Sun 4/330 with 24 Mbytes. We used built-in timing functions available in the SML/NJ system which are based on the Unix *getrusage* utility. We factored out the garbage collection costs, so our numbers represent only code speed. Garbage collection costs are irrelevant for the sake of comparison since all versions of code generated by sml2cand the SML/NJ compiler generate the same amount of garbage and use the same garbage collector. Ignoring garbage collection costs is in fact slightly unfair to sml2csince programs generated by sml2cactually have less live data than SML/NJ programs, as the SML/NJ system places the compiled code in the heap. The C code produced by sml2c, on the other hand, is outside the heap. For both sml2cand the native code compiler, we used the same high optimization settings provided by the SML/NJ compiler. The C code was compiled on both machines using version 1.40 of *gcc* [19] with the *-O* and the *-fomit-frame-pointer* flags.

Tables 1 and 2 show the execution times for the various benchmark programs. The code generated by our system for the benchmark programs generally runs

Program	SML/NJ (α) (sec)	Opt (β) (sec)	β/α	Unopt (δ) (sec)	δ/α
life	25.3	50.2	1.98	90.0	3.56
lex	17.3	30.4	1.76	57.3	3.31
yacc	5.4	10.8	2.00	19.2	3.56
Knu-B	14.4	32.1	2.23	57.5	3.99
vliw	30.0	77.7	2.60	142.4	4.75

Table 1: Benchmarks for Decstation 5000/200

Program	SML/NJ (α) (sec)	Opt (β) (sec)	β/α	Unopt (δ) (sec)	δ/α
life	56.3	105.9	1.88	161.0	2.85
lex	39.3	64.0	1.62	109.3	2.78
yacc	14.8	24.2	1.63	35.2	2.37
Knu-B	43.4	78.2	1.80	103.0	2.37
vliw	80.4	133.2	1.65	207.6	2.58

Table 2: Benchmarks for Sun 4/330

Program	SML/NJ (α) (sec)	Opt (β) (sec)	(β/α)
life	11.2	28.2	2.5
lex	74.2	134.6	1.8
yacc	395.8	602.5	1.5
Knu-B	26.3	62.1	2.4
vliw	367.0	646.8	1.8

Table 3: Compilation times on Decstation 5000/200

at about half the speed of the native code generated by SML/NJ.

We also measured compilation times for our benchmarks to show that the times were still reasonable. The compilation times include the time spent by sml2cand the time spent by *gcc* compiling the generated C code. The compilations were done on a Decstation 5000/200 using our system with optimization and version 0.75 of the SML/NJ compiler, both run with the settings described previously. The times are shown in Table 3. The compilation times for the long programs, *Lex*, *Yacc*, and *VLIW* are less than a factor of two slower than the SML/NJ times.

Several aspects of our compilation method lead to the observed slowdown in the benchmark programs. One possible source of overhead comes from the loads and stores to the register array. Another possible source are operations that are more expensive to implement in C than in assembly language, namely the jumps

Program	s ('000s)	w ('000s)	r ('000s)	c ('000s)	ifc ('000s)
life	256,421	71,502	109,710	37,763	10,348
lex	172,358	31,255	54,376	15,070	5,976
yacc	55,948	17,685	10,875	4,248	832
Knu-B	158,533	43,410	59,928	12,630	284
vliw	170,849	52,781	72,624	18,267	2,993

Table 4: Results from instrumented programs

Program	s/c	ifc/c
life	6.8	27%
lex	11.4	39%
yacc	13.2	20%
Knu-B	12.6	2.2%
vliw	9.4	16.4%

Table 5: Frequency of function calls and integrated function calls

Program	$(r+w)/s$
life	0.70
lex	0.50
yacc	0.51
Knu-B	0.65
vliw	0.73

Table 6: Overhead of register array usage

(which are implemented by transfers through the apply-like procedure) and the heap check (which in the native code typically requires a single instruction, but in C requires a conditional test and branch). There is also added overhead for integer arithmetic, but previous benchmark studies showed this to have a negligible effect.

In order to better understand the nature of these sources of overhead, we instrumented the generated C code. The instrumentation counts the following things: s, the number of C statements executed, excluding assignment statements which move values between the register array and local variables, w, the number of assignments to the register array, r, the number of reads from the register array, c, the number of function calls, and ifc, the number of calls to integrated functions. The results of instrumenting the benchmark programs are given in Table 4.

It is clear from the data that function calls occur with enormous frequency, and furthermore that most function calls transfer through the apply-like procedure. Table 5 gives the average number of statements executed in a function (including the last jump), and the percentage of function calls that are calls to integrated functions, *i.e.*, turned into gotos. It also seems clear that there is a tremendous amount of reading and writing of values in the register array, relative to the total number of statements executed. Table 6 gives the number of reads and writes as a fraction of the statement count. The reading and writing is to be expected with small function bodies and the passing of parameters in memory via the register array.

This overhead could be reduced greatly by extending C to support global register variables and by modifying C compilers to support proper tail-recursion. The introduction of global register variables would allow us to eliminate the register array. Thus, the generated C code would no longer make reads and writes to the register array. Global register variables would be compatible with multi-processing or multi-threading, since naturally each process would have its own global register variables that are distinct from other processes, just as processes have their own distinct local register variables. Extending C compilers to support proper tail-recursion or some form of interprocedural goto would allow us to eliminate the use of the apply-like procedure in the generated code. Instead the generated code could use C function calls as its function call mechanism.

8 Conclusions

Our experience shows that it is possible to compile languages such as Scheme and SML without using any assembly language. It is possible to develop portable implementations of such languages without sacrificing either proper tail-recursion or an inordinate amount of efficiency.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley Publishing Company, 1986.

[2] A. W. Appel. A Runtime System. *Lisp and Symbolic Computation*, 3(4):343–380, Nov. 1990.

[3] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[4] A. W. Appel and T. Y. Jim. Continuation-Passing, Closure-Passing Style. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 293–302. ACM, Jan. 1989.

[5] A. W. Appel and D. B. MacQueen. A Standard ML Compiler. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274, pages 301–324. Springer-Verlag, 1987.

[6] R. Atkinson, A. Demers, C. Hauser, C. Jacobi, P. Kessler, and M. Weiser. Experience Creating a Portable Cedar. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 322–329. ACM, 1989.

[7] J. F. Bartlett. SCHEME→C: A Portable Scheme-to-C Compiler. Technical report, DEC Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, CA 94301 USA, Jan. 1989.

[8] E. C. Cooper and J. G. Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, Dec. 1990.

[9] B. F. Duba, R. Harper, and D. MacQueen. Typing First-Class Continuations in ML. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*. ACM, Jan. 1991.

[10] R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing Control in the Presence of First-Class Continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77. ACM, 1990.

[11] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[12] D. Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Yale University, Feb. 1988.

[13] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the SIGPLAN '86 Conference on Programming Language Design and Implementation*, pages 219–233. ACM, 1986.

[14] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[15] J. G. Morrisett. A multiprocessor interface for Standard ML. In *The Third International Workshop on Standard ML*. School of Computer Science, Carnegie Mellon University, September 26-27, 1991.

[16] J. Rees and W. C. (Eds.). Revised report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, 1986.

[17] J. H. Reppy. Asynchronous Signals in Standard ML. Technical Report 90-1144, Department of Computer Science, Cornell University, Aug. 1990.

[18] V. Santhanam and D. Odnert. Register Allocation Across Procedure and Module Boundaries. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 28–39. ACM, June 1990.

[19] R. M. Stallman. Using and Porting GNU CC. GNU CC is a widely available C compiler developed by the Free Software Foundation, Sept. 1989.

[20] G. L. Steele Jr. Rabbit: A Compiler for Scheme (A Study in Compiler Optimization). Master's thesis, AI Laboratory Technical Report AI-TR-474, Massachusetts Institute of Technology, May 1978.