A Portable and Optimizing Back End for the SML/NJ Compiler

Lal George[1], Florent Guillame[2], John H. Reppy[1]

[1] AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974, USA
[2] École Normale Supérieure, 45, rue d'Ulm, 75005 Paris, France
Email: {george, jhr}@research.att.com, guillaum@clipper.ens.fr

Abstract. There are two major goals that must be addressed in a portable back end: a good sequence of instructions must be selected making full use of the capabilities of the machine, and it must be possible to orchestrate target-specific optimizations. A key to the first problem is the language MLRISC, intended in part, to represent the simplest and most basic operations implementable in hardware. The importance of MLRISC is that it provides a common representation for expressing the instruction set of any hardware platform. Bottom-up tree pattern matching with dynamic programming, expressed using succinct and clear specifications of the target instruction set, is used to generate target machine code from an MLRISC program. Target-specific optimizations are performed by parameterizing *off-the-shelf* optimization modules with concepts common across architectures. The specification of a variety of architectures, and the ability to mix and match sophisticated optimization algorithms are shown. The resulting back end is independent of the intermediate language used in SML/NJ, and could in principle be used in a compiler for a source language quite different from SML. We argue that porting the compiler to a new architecture requires substantially less effort than the existing abstract machine approach, and report significant gains from preliminary architecture description driven optimizations.

1 Introduction

Portability is crucial to the widespread use and acceptance of any new language. Not only must the compiler be readily portable to a wide variety of architectures, but it must also generate code that is competitive with one where portability is not an issue. The compiler cannot be biased towards one architecture.

The Standard ML of New Jersey system (SML/NJ)[3, 4] is a highly optimizing compiler that uses the Continuation Passing Style (CPS) intermediate form for optimization[2]. Most of the optimizations in the compiler are done at this level. The code generation model has been based on an abstract machine called the `cmachine` for *code machine*. The `cmachine` has a small set of registers, and a fairly high level instruction set. There is a `cmachine` instruction that can expand to several hundred instructions. Registers include: an *allocation pointer* representing the next available location in the heap, a *limit register* representing the highest address in the heap, a set of *miscellaneous registers* for parameter

passing, and others. The compiler is ported to a new architecture by providing a mapping of the **cmachine** registers to physical registers, and *templates* that macro-expand **cmachine** instructions into target machine instructions. Such a port is unsatisfactory in several ways: useful low level optimizations are omitted in the translation to machine code; it may not be possible to use the full capabilities of the target architecture and its instruction set, and it is sometimes difficult to incorporate target-specific optimizations. The back end is responsible for linking, scheduling, span-dependency analysis, and binary instruction output. There is no dependence on the host assembler and linker. Various target specific optimizations, such as scheduling are manually implemented for each architecture.

Two major problems must be addressed in a portable back end, namely: a good sequence of instructions must be selected for the task, making full use of the available registers and instructions, and it must be possible to orchestrate optimizations specific to the architecture, without compromising portability.

2 Overview of Our Approach

Our new approach is not biased towards any architecture. CPS is compiled to a tree language called MLRISC; intended in part, to describe the simplest kinds of operations implementable in hardware. No assumptions are made regarding addressing modes or types of instructions, and because of our register allocation scheme, there are few assumptions made about physical registers. The MLRISC is then converted to a flow graph of target machine instructions, which is optimized using generic optimization modules parameterized over a machine description.

The importance of MLRISC is that it provides a common medium for the specification of any instruction set. There are basic operations implementable in hardware, and instructions are made up of these operations. An instruction ought to be definable using these basic operations.

Bottom-up tree pattern matching with dynamic programming (BURG) is central to our approach. The translation from CPS to target machine code proceeds in three major phases, two of which involve BURG specifications (Figure 1). The high level CPS is first translated into a simpler form called **ctrees**, suitable as input to BURG. Several optimizations are performed during this simplification. Using the BURG specification Ψ, the **ctree** language is rewritten to MLRISC trees, optimizing the tagging and untagging of arithmetic operations along the way. BURG is used once again to translate MLRISC to a flowgraph of target machine instructions. The specification Φ is a description of the target architecture *instruction set* and *registers*. Various optimizations such as liveness analysis, scheduling, span-dependency analysis, and graph-coloring register allocation are performed on the target machine instructions. The optimization phase is parameterized over a machine description represented as a set of SML modules. The back end is constructed by a series of functor applications and is a nice demonstration of the flexibility provided by the module system[15]. The

concise description of the instruction set in terms of MLRISC, and the ability to perform architecture description driven optimizations, are the key contributions.

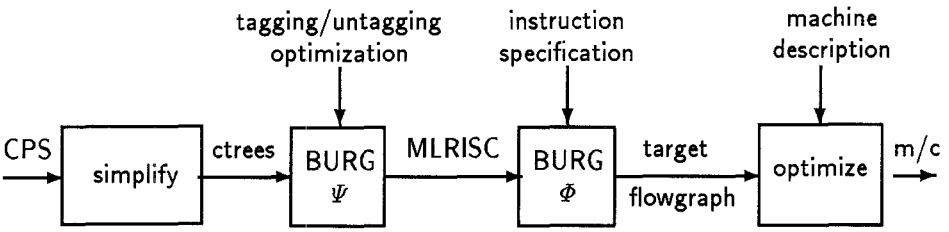

Fig. 1. Flowchart of new code generation model

3 ML-Burg

The new code generation strategy is implemented using a SML version of iBurg[10, 13]. Given a tree rewriting system augmented with costs, ML-Burg generates a program to perform bottom-up tree pattern matching with dynamic programming. A successful reduction of the input tree, corresponds to rewriting the input tree to a special non-terminal symbol called the *start non-terminal*. Upon successful reduction, facilities are provided to walk the tree emitting semantic actions associated with the rules that matched.

Consider the rewrite system specified below:

```
reg :   LIi              (1)  ;
reg :   ADD(reg,LIi)     (1)  ;
reg :   ADD(reg,reg)     (1)  ;
```

`ADD` is a binary node with the usual meaning, and $\texttt{LI}_i$ is a leaf node representing the *integer immediate* i. The integer i is not used in pattern matching and is not part of the rewrite rule, but it is an attribute that may be used in semantic actions . This grammar specifies that: an input tree matching $\texttt{LI}_i$ can be reduced to the *non-terminal* `reg` with a cost of one; an input tree matching $\texttt{ADD(reg,LI}_i\texttt{)}$, where the first child can be reduced to `reg`, can also be reduced to the non-terminal `reg`. The grammar above is clearly ambiguous as there are two ways to reduce the tree $\texttt{ADD(LI}_3\texttt{,LI}_4\texttt{)}$ to the non-terminal `reg`. The two reductions are shown below, where each reduction is annotated with SPARC assembly code. The registers, `%t1`, `%t2`, and `%t3` are pseudo-registers that are assigned to physical registers in a later register allocation pass.

```
ADD(LI3,LI4)                    ADD(LI3,LI4)
     ↓      ;; add %g0,3,%t1         ↓      ;; add %go,3,%t1
ADD(reg,LI4)                    ADD(reg,LI4)
     ↓      ;; add %g0,4,%t2         ↓      ;; add %t1,4,%t2
ADD(reg,reg)                        reg
     ↓      ;; add %t1,%t2,%t3
    reg
```

It is precisely this ambiguity in specification that is the strength of tree rewriting code generation techniques. Dynamic programming finds the cheapest set of instructions to implement the program. The reduction on the left has a cost of three, while the one on the right has a cost of two. This example however, does not demonstrate the ability to describe different register classes available on the target architecture, or non-regular register sets (Section 6).

4 MLRISC

Figure 2 shows the SML signature for **MLRISC**. The instruction set, described by the datatype **mlrisc**, makes no assumptions about addressing modes on the target machine. It is possible to **JMP** to *anything*, and **LOAD/STORE** from *anything*. Each **mlrisc** instruction defines a basic combinator that will be used to describe the target instruction set. A BURG grammar is used to define the instruction set, and the associated semantic actions can be used to effectively utilize the hardware.

Non-commutative arithmetic operations specify the order of evaluation of arguments, using the type **order**. The order of evaluation must be recorded to preserve the semantics with respect to arithmetic exceptions. Thus instructions like **SUB** and **DIV**, etc., specify the order of evaluation. The order is assumed to be left-to-right for commutative operators.

There is a commitment to general purpose and floating point registers, **REG** and **FREG** respectively. Nearly all processors today have these sets of registers and provide dedicated instructions to operate on them. This does not preclude the Motorola 68000 that does not have general purpose registers. BURG non-terminals may be used to represent the Motorola 68000, address and data registers (Section 6).

Lastly, MLRISC has no connection to the CPS intermediate representation or dedicated registers, and can be easily divorced from SML/NJ system.

5 Ctree and MLRISC Generation

The **ctree** representation is used to simplify the high level semantics of CPS, and provide a suitable tree representation for input to BURG. Generating a tree representation from the linear CPS input must observe the semantics with respect to arithmetic exceptions and memory. Several low level optimizations may be performed on the **ctree** representation, such as:

```
structure Label : sig
                      datatype label =
                      ...
                  end =
struct  ... end

signature MLRISC = sig
  datatype order = LR | RL                    (* order of evaluation *)
  datatype bcond  = LT | LE | EQ | GEU         (* branch conditions *)
  datatype mlrisc                                   (* instructions *)
     = REG  of int                                     (* register *)
     | FREG of int                               (* floating register *)
     | LI   of int                                (* integer constant *)
     | MV   of mlrisc * mlrisc                                 (* move *)
     | FMV  of mlrisc * mlrisc                 (* floating point move *)
     | ADD  of mlrisc * mlrisc                          (* addition *)
     | SUB  of mlrisc * mlrisc * order               (* subtraction *)
       ...
     | ANDB  of mlrisc * mlrisc                       (* logical AND *)
       ...
     | LOAD  of mlrisc                             (* memory operations *)
     | STORE of mlrisc * mlrisc
       ...
     | CVTI2D of mlrisc                 (* convert integer to double *)
     | FADDD of mlrisc * mlrisc           (* floating point addition *)
       ...
     | BR    of Label.label                   (* branch instructions *)
     | JMP   of mlrisc
     | BCC   of bcond * mlrisc * mlrisc * Label.label * order
     | FBCC  of bcond * mlrisc * mlrisc * Label.label * order
     | SEQ of mlrisc * mlrisc                            (* sequencing *)
end
```

Fig. 2. MLRISC specification

- The detection of situations where a record creation in the heap can be implemented as a tight loop copying consecutive locations from one memory area to the record being created.
- Propagating increments to the allocation pointer, so that it is performed only once at the function exit points.

Dynamic programming is used to optimize the tagging and untagging of arithmetic expressions, in the translation of `ctrees` to `MLRISC`. Integers in SML are tagged with their lowest bit set to one, (i.e., the integer n is represented as $2n + 1$). On the MIPS, the old code generator expands the CPS program (`x:=a-b; z:=x+y`) to:

```
sub     a,b,t1          % t1 := a - b
add     t1,1,x          % x := t1 + 1
sub     x,1,t2          % t2 := x - 1
add     t2,y,z          % z := t2 + y
```

Clearly the intermediate tagging and untagging is unnecessary. Dynamic programming using BURG is a fast and elegant solution to this problem. Peterson's min-cut algorithm is more thorough but expensive ($O(n^3)$)[16]. The BURG specification Ψ contains 124 rules (details appear in an extended version[11]).

The resulting MLRISC program represents the simplest set of operations required to implement the CPS program. The burden of various optimizations till this point, in the abstract machine model, would have been on the person porting the compiler. These optimizations would be repeated for each architecture. Now, it has been transferred once and for all, to a person that is an expert on the internals of the compiler. These basic operations must now be combined to match instructions on the target machine.

6 Instruction Set Specification

As a concrete example, Figure 3 introduces a fragment of the SPARC specification. An effective address on the SPARC can either be a *register+displacement* or a *register+register*. This is specified using the non-terminal **ea**. The semantic actions associated with rules that reduce to **ea**, return a value of type **eaValue**. The operand to a **LOAD** must be reduced to the non-terminal **ea**, and the code to emit is a simple case statement over the various **eaValue** constructors. Fortunately, no restrictions were imposed on the operand to **LOAD** in the MLRISC design. This example extends to handle the full set of addressing modes and instructions found on CISC machines such as the Intel i486 or Motorola 68000. A description of the i486 addressing modes involves just 10 lines of BURG specification.

An example from the Motorola 68000, illustrates how simple specifications can later on yield high quality code, and the use of non-terminals to denote various kinds of register classes. On the 68000, certain kinds of registers are not permitted as operands to instructions. For example, the operand to **LOAD** must be reducible to an address register. The result of the load may be either an address or data register. This is fairly easy to specify by devoting a non-terminal to address registers. A possible fragment of the 68000 specification is shown in Figure 4.

For correctness, a **movl** is required in the implementation of **ADD**. Since we assume an infinite number of registers, which are later assigned to physical registers, these moves normally turn out to be harmless. Coalescing non-interfering live ranges in a graph-coloring register allocation algorithm[9], collapses **rd** and $\mathbf{dreg}_1$ to the same physical register where possible, eliminating the redundant move. This technique is used quite effectively to handle the non-regular register set on the Intel i486. These specifications and semantic actions are very simple, yet they describe quite varied and complex systems.

```
datatype eaValue = DISPea of register * int
                 | INDXea of register * register
...
ea:   ADD(reg,LIi)       (0) DISPea(reg,i) ;;
ea:   ADD(reg1,reg2)     (0) INDXea(reg1,reg2) ;;
ea:   SUB(reg,LIi)       (0) DISPea(reg,~i) ;;
ea:   reg                (0) DISPea(reg,0) ;;
reg:  LOAD(ea)     (1) let val rd : register = newReg()
                       in
                          case ea
                            of DISPea(rt,n) => emit(ld(rt,IMMED n,rd))
                             | INDXea(rs,rt)=> emit(ld(rs,REG rt,rd))
                          (* esac *);
                          rd
                        end ;;
```

Fig. 3. SPARC instruction set specification

```
areg:   LOAD(areg)        (1) ...
dreg:   LOAD(areg)        (1) ...
dreg:   ADD(dreg1,dreg2)  (1) let val rd = newDreg()
                              in
                                 emit(movl(rd,dreg1));
                                 emit(addl(rd,dreg2))
                              end
```

Fig. 4. Motorola 68000 instruction set specification

The combination of ML-Burg and MLRISC is an elegant way to solve the instruction selection problem. BURG is expressive enough to allow the concise specification of most instruction set. A similar observation was reported by Appel[5], who wrote TWIG[1] specifications for the VAX and Motorola 68000; detailed information was encoded in the cost function to aid in the selection of the best rule. Porting the compiler does not require knowledge of any compiler internals, such as tagging schemes, runtime representations, and semantics of high level abstract instructions (often specific to SML). The instruction set must be specifiable in MLRISC, which is then used to pick the cheapest instructions to emit with respect to the cost function. Since the generated MLRISC data structure is larger than the source CPS, the instruction selection is done in small units.

7 Target Machine Architectural Description

Once instruction selection has been performed, facilities exist to generate a generic control flowgraph, where the nodes contain target machine instructions. It is not possible to directly output the binary representation of instructions as they contain pseudo-registers and symbolic labels. Instruction scheduling, span-dependency resolution, and further optimizations, may be necessary before final binary code emission. Writing target-specific optimizations for each architecture would be a portability nightmare. Instead, we use a scheme where *off-the-shelf* optimization modules are parameterized over a description of the target machine.

While all machines are different in detail, they are all very similar in concept. The idea behind our machine description is to describe those concepts that are common across architectures, and use them in generic optimization modules. The structure of the machine description is shown in Figure 5. At the lowest level of the module dependency is a description of the storage units on the machine, specified by the signature `CELLS`. Several dataflow problems require efficient operations over sets of cells, so we require the type `cellset` and the usual set operations over them. These are easily constructed using modules defined in the SML/NJ Library[6]. The signature `INSTRUCTION` is a specification of the available instructions on the machine in terms of its cells. This hierarchy corresponds to the fundamental design of von Neumann machines. Lastly, the signature `INSN_PROPERTIES` contains the bulk of the machine description. Useful properties of the instruction set are collected here, and used in generic optimizations modules. For example: the type `kind`, returned by the function `instrKind`, is used to classify instructions as being either a NOP (`IK_NOP`), a jump instruction (`IK_JUMP`), or any other (`IK_INSTR`); the type `target` returned by `branchTargets` is used to describe the target of branch instructions. `instrKind` and `branchTargets` are used to implement a generic module that produces a flowgraph specialized over instructions of the target machine.

Figure 6 shows the machine description for the SPARC. The type `cell` includes: an unlimited supply of general and floating point registers (`Reg` and `Freg`, respectively), the condition code register (`CC`), and the floating point condition code register (`FCC`). The stack (`STACK`) and memory (`MEM`), which are not normally considered to be in the same category as registers, are also included. Instructions that access the memory or stack, will be marked as accessing the `MEM` or `STACK` resource. This information is used during instruction scheduling. `SparcInstr` is the module matching `INSTRUCTION` in the machine description. The type operand has been simplified for expository purposes. `SparcProps` shows a fragment of the module matching `INSN_PROPERTIES`. The module is a total of 460 lines, most of which is boiler-plate.

As additional optimization modules are developed, one may expect the signature for `INSN_PROPERTIES` to grow, in order to meet the demands for more information about the target architecture. After a certain point in this evolution, generating high quality code for a new architecture will involve mixing and matching off-the-shelf optimization modules to suit the architecture.

```
signature CELLS = sig
  type cell
  type cellset
  val cardinality : cellset -> int
  val union       : cellset * cellset -> cellset
  val add         : cell * cellset -> cellset
  ...
end

signature INSTRUCTION = sig
  structure C : CELLS
  type instruction
end

signature INSN_PROPERTIES = sig
  structure I : INSTRUCTION
  structure C : CELLS
  sharing I.C = C

  datatype kind = IK_NOP | IK_JUMP | IK_INSTR
  datatype target = LABELED of Label.label | FALLTHROUGH | ESCAPES
  val instrKind     : I.instruction -> kind
  val defUse        : I.instruction -> C.cellset * C.cellset
  val branchTargets : I.instruction -> target list
  ...
end
```

Fig. 5. Machine description

8 Target Machine Optimization

A generic basic block scheduler that is parameterized by a machine description, described above, has been developed. Figure 7 shows the machine properties required for this purpose. We describe each component individually in more detail to illustrate their complexity (or more appropriately, lack of):

`branchDelayedArch` is a boolean flag that indicates if the architecture requires a branch delay slot. Special considerations are used for picking this instruction if needed.

`latency(`*instr*`)` is a function that returns the number of cycles needed to execute the instruction *instr*.

`needsNop(`*instr*`,`*instrs*`)` during scheduling there may not be enough instructions available to keep the pipeline busy while executing high latency instructions. Further, some architectures require an explicit NOP (No OPeration) instruction between two instructions under such circumstances. For example, on the MIPS, a `MFHI` instruction must occur at least two instructions after a

```
structure SparcCells = struct
  structure S = SortedList
  datatype cell = Reg of int
                | Freg of int
                | CC | FCC | STACK | MEM
  type cellset = int list * int list * int list
  fun cardinality(r,f,e) = length r + length f + length e
  fun union((r1,f1,e1),(r2,f2,e2)) =
      (S.merge(r1,r2),S.merge(f1,f2),S.merge(e1,e2))
  ...
end

structure SparcInstr = struct
  structure C = SparcCells
  datatype operand = REGrand of int
                   | IMrand of int
                   | LABrand of Label.label
  datatype cond_code = CC_A | CC_E  | CC_NE | CC_G | CC_GE
                     | CC_L | CC_LE | CC_GEU | CC_LEU
  datatype instruction
    = NOP
    | LD    of int * operand * int
    | ADD   of int * operand * int
    | ADDCC of int * operand * int
    | JMPL  of int * Label.label list
    | BCC   of cond_code * Label.label
    | FBCC  of cond_code * Label.label
    ...
end

structure SparcProps = struct
  structure I = SparcInstr
  structure C = SparcCells
  datatype kind = IK_NOP | IK_JUMP | IK_INSTR
  datatype target = LABELED of Label.label | FALLTHROUGH | ESCAPES
  fun instrKind(I.NOP)       = IK_NOP
    | instrKind(I.BCC _)     = IK_JUMP
    | instrKind(I.JMPL _)    = IK_JUMP
    | instrKind(I.FBCC _)    = IK_JUMP
    | instrKind _            = IK_INSTR
  fun branchTargets(I.BCC(I.CC_A,lab))  = [LABELLED lab]
    | branchTargets(I.BCC(_,lab))       = [LABELLED lab,FALLTHROUGH]
    ...
  ...
end
```

Fig. 6. SPARC machine description

```
val branchDelayedArch : bool
val latency  : I.instruction -> int
val needsNop : I.instruction * I.instruction list -> int
val defUse   : I.instruction -> int list * int list
val isSdi    : I.instruction -> bool
val minSize  : I.instruction -> int
val maxSize  : I.instruction -> int
val sdiSize  : I.instruction * (int -> int) * int -> int
val expand   : I.instruction * int * (int -> int) ->
                                        I.instruction list
```

Fig. 7. Machine properties for basic block scheduling

`MULT` instruction. `needsNop` returns the number of NOPs required between *instr* (the instruction being emitted), and *instrs* (the previous instructions emitted).

`defUse(`*instr*`)` returns list of resources defined and used by the instruction. This is used to construct the data dependency graph.

`isSdi(`*instr*`)` returns true if *instr* is a span-dependent instruction whose size is determined by the final value of labels.

`minSize/maxSize(`*instr*`)` returns the minimum/maximum size of the instruction *instr*. These two functions are used to schedule blocks with span-dependent instructions. The value of labels is calculated assuming all instructions expand to their minimum size. Another calculation is performed assuming all labels expand to their maximum size. If the size of a span-dependent instruction does not vary under these extremities, then it may be expanded, and scheduled along with the other instructions in that block. Such a block is said to be *stable*. Scheduling a basic block refines the value of labels under these two extremities and may stablize an otherwise unstable block. If unstable blocks still persist, then there is no option but to expand the span-dependent instructions to their maximum size.

`sdiSize(`*instr*`,`*labMap*`,`*loc*`)` returns the size of the span-dependent instruction *instr*, under the assignment of labels given by *labMap*, where the current location counter is *loc*.

`expand(`*instr*`,`*size*`,`*labMap*`)` returns the sequence of instructions when the span-dependent instruction *instr* is expanded to *size* number of instructions, assuming the assignment of labels given by *labMap*.

The generic basic block scheduler is 397 lines of SML code. The module to perform span-dependency analysis is 384 lines. In a similar fashion to basic block scheduling, we have developed a generic graph-coloring register allocator used to allocate general purpose and floating point registers on most target machines. In addition, on the IBM RS/6000 it is used to allocate pseudo condition code registers among the eight condition code registers available. More optimizations are planned in the near future.

9 Mix and Match

Figure 8 shows the construction of the SPARC code generator, which is formed by linking several optimization phases. The **FlowGraph** functor produces a flowgraph data structure specialized over the SPARC instructions. The **Liveness** functor exports a function called **liveness**, which will annotate the flowgraph with liveness information at block boundaries. Optimizations are mixed and matched using functor applications. The functors **RegAllocator** and **FlowGraphGen** implement a certain optimization, and requires a function **codegen** that will be invoked to perform the rest of the optimizations. The functor **BBSched** that performs basic block scheduling, is the last in the chain, and exports a function called **finish** that does the final machine code output. The SPARC code generator, in Figure 8, strings together: flowgraph generation that includes liveness analysis (**FlowGen**); integer register allocation (**IntRAlloc**); floating point register allocation (**FloatRAlloc**), and finally basic block scheduling (**BBsched**). The various parameters to these functors are unimportant except to note that they are specified by signatures that describe generic properties of architectures. The example illustrates that the use of functor application makes it easy to mix and match generic optimization modules to suit the SPARC architecture.

```
structure SparcFlow = FlowGraph(structure Instr = SparcInstr)

structure SparcLive = Liveness(structure Flowgraph = SparcFlow
                               structure InsnProps = SparcProps)

structure BBsched = BBSched(structure Flowgraph = SparcFlow
                            structure InsnProps = SparcProps
                            structure Emitter   = SparcMCEmitter)

structure FloatRAlloc = RegAllocator(structure Ra = FloatRA_Arg
                                     val codegen = BBsched.bbsched)

structure IntRAlloc =   RegAllocator(structure Ra = IntRA_Arg
                                     val codegen  = FloatRAlloc.ra)

structure FlowGen = FlowGraphGen(
                      structure Flowgraph = SparcFlow
                      structure InsnProps = SparcProps
                      val codegen = SparcLive.liveness)
```

Fig. 8. Gluing the SPARC code generator together

10 Results

At the time of writing, we have working code generators for the MIPS, IBM RS/6000 and SPARC, an an untested specification for the Intel i486. The preliminary results reported here are only for the SPARC.

A fairly standard set of SML benchmarks are used[2]. We first measure the improvements from using a more sophisticated register allocation scheme. SML/NJ supports a register passing style for parameters, and it is essential that operands be computed in the *right* register. Register constraints may require that the operand be first computed into a temporary and later moved into the correct register before a function call. The first column of Figure 9 shows the number of register-register moves required at function call boundaries in the existing compiler (version 0.93). The second column shows the performance of our new graph-coloring register allocator. At least 40% of the original register-register moves are removed.

Figure 10 shows the improvements from dynamic programming and the allocation pointer optimization described (Section 5). The first column shows the static code size (in number of instructions) without any optimization, and the second with these optimizations. There is a static code size improvement of 2-5%. In terms of dynamic instruction counts, this corresponds to roughly 1-3% improvement. This is encouraging as these improvements have come largely for free in our attempt to improve portability. Machines such as the Dec Alpha or the IBM RS/6000, should do even better, because multiple overflow checks may be collapsed into one.

While compile time speeds are acceptable, we do not report them since the new system is not tuned or optimized for this. The back end in the current SML/NJ compiler takes about 25% of the total compilation time. This percentage does not include CPS optimization. The new back end is currently about 3-4 times slower.

11 Future Work

Machine descriptions are required for all the architectures that the SML/NJ compiler currently supports, which include, the Motorola 68000, and the HPPA. Work is in progress on a DEC Alpha port. The main areas for future work relate to the speed of compilation, and further optimizations relevant to RISC processors. Our compilation scheme is highly symbolic — developing fast table driven optimizations[17] derived from a more concise machine description, and the use of partial evaluation[7] ought to produce a faster backend. Composing BURG specifications similar to that done for attribute grammars may also prove worthwhile[8]. Lastly, a pre-pass global scheduler is extremely important for superpiplined and superscalar machines[12, 14].

	SML/NJ 0.93	New back end	ratio
mandelbrot	46	5	0.11
life	299	212	0.71
kbendix	968	594	0.61
simple	1089	657	0.60
lexgen	1965	1121	0.57
yacc	4090	2222	0.54

Fig. 9. register-register moves with 6 callee-save registers

	static code size			dynamic instruction count
	before	after	% improvement	% improvement
mandelbrot	625	561	11.4	16.2
life	6333	6032	5.0	0.8
kbendix	13209	12808	3.1	3.2
simple	30492	29263	4.2	1.9
lexgen	24671	24217	1.9	2.2
yacc	96925	94666	2.4	2.3

Fig. 10. Static code size and dynamic instruction count improvements

12 Conclusions

A highly portable and optimizing back end has been described. It addresses the problems of target machine instruction selection and machine specific optimization. Porting the compiler to a new architectural platform is expected to be trivial for someone ignorant of the internal of the compiler, but familiar with the architecture. Off-the-shelf optimization modules can be easily constructed to suit a particular machine. The set of optimizations currently implemented show encouraging results.

References

1. AHO, A., GANAPATHI, M., AND TJIANG, S. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems 11*, 4 (Oct. 1989), 491–516.
2. APPEL, A. *Compiling with Continuations.* Cambridge Univ. Press, 1992.
3. APPEL, A., AND MACQUEEN, D. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, G. Kahn, Ed. Springer-Verlag, 1987, pp. 301–324. LCNS No.274.
4. APPEL, A., AND MACQUEEN, D. Standard ML of New Jersey. In *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming* (New York, August 1991), M. Wirsing, Ed., Springer-Verlag. (in press).

5. APPEL, A. W. Concise specifiations of locally optimal code generators. Tech. Rep. CS-TR-080-87, Princeton University, Feb. 1987. Dept. of Computer Science.
6. AT&T. *The Standard ML of New Jersey Library, Reference Manual*, 0.2 ed. AT&T Bell Laboratories, 600 Mountain Ave, Murray Hill, NJ 07974, 1993.
7. BIRKEDAL, L., AND WELINDER, M. Partial evaluation of Standard ML. Master's thesis, University of Copenhagen, October 22 1993. Dept. of Computer Science.
8. BOYLAND, J., AND GRAHAM, S. Composing tree attributions. In *POPL '94: 21st ACM SIGPLAN-SIGACT symposium on principles of programming languages* (January 1994), ACM, pp. 375–388. Portland,Oregon.
9. CHAITIN, G. Register allocation and spilling via graph coloring. *SIGPLAN Notices 17(6)* (June 1982), 98–105. Proceeding of the ACM SIGPLAN '82 Symposium on Compiler Construction.
10. FRASER, C., HANSON, D., AND PROEBSTING, T. Engineering a simple, efficient code generator generator. In *Letters on Programming Languages and Systems* (1992), ACM.
11. GEORGE, L., GUILLAME, F., AND REPPY, J. A portable and optimizing back end for the SML/NJ compiler. Tech. Rep. BL112610-931103-38TM, AT&T Bell Laboratories, November 1993.
12. GOODMAN, J., AND HSU, W.-C. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 1988 International Conference on Supercomputing* (July 1988), ACM, pp. 442–452.
13. GUILLAUME, F., AND GEORGE, L. *ML-Burg — Documentation*, 1.0 ed. AT&T Bell Laboratories, 600 Mountain Ave, Murray Hill, NJ 07974, 1993.
14. GWENNAP, L. Cyrix describes pentium competitor. *Microprocessor Report 7*, 14 (October 1993), 5–10.
15. MACQUEEN, D. Modules for Standard ML. In *Proc. 1984 ACM Conf. on LISP and Functional Programming* (New York, 1984), ACM Press, pp. 198–207.
16. PETERSON, J. Untagged data in tagged environments: Choosing optimal representations at compile time. In *Functional programming languages and computer architecture* (September 1989), ACM, pp. 89–99.
17. PROEBSTING, T., AND FRASER, C. Detecting pipeline structural hazards quickly. In *Principles of Programming Languages* (January 1994), ACM.