How Generic is a Generic Back End? Using MLRISC as a Back End for the TIL Compiler* (Preliminary Report)

Andrew Bernard**, Robert Harper, and Peter Lee

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract. We describe the integration of MLRISC, a "generic" compiler back end, with TIL, a type-directed compiler for Standard ML. The TIL run-time system uses a form of type information to enable partially tag-free garbage collection. We show how we propagate this information through the final phases of the compiler, even though the back end is unaware of the existence of this information. Additionally, we identify the characteristics of MLRISC that enable us to use it with TIL and suggest ways in which it might better support our compiler. Preliminary performance measurements show that we pay a significant cost for using MLRISC, relative to a custom back end.

1 Introduction

We describe how we integrated *MLRISC*, a "generic" compiler back end, with *TIL*, a type-directed compiler for the Standard ML (SML) programming language. A *type-directed compiler* uses variable type information to guide successive translations between intermediate languages [13]. Type-directed compilers rely on complete variable type information for most or all phases of compilation—thus, types are preserved by intermediate code transformations during each phase. A *generic compiler back end* translates a low-level intermediate language

* This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050, and in part by the National Science Foundation under Grant No. CCR-9502674. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Science Foundation, or the U.S. Government.

** This material is based on work supported under under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

into machine code—the intention is that the back end does not depend on a particular source language or front-end implementation technology. A compiler back end could itself be type directed—both Typed Assembly Language [12] and Proof-Carrying Code [14] [15] encode variable type information at the assembly-language level—although this is not common practice, and, as a matter of fact, MLRISC is not type directed.

TIL translates a source program through a succession of typed intermediate languages until it arrives at conventional assembly code. The typed intermediate languages used in TIL specify types explicitly at all variable binding sites. Thus, TIL can always determine the type of a variable without having to resort to type inference. The universal availability of type information permits TIL to check types after any phase of compilation—this helps to ensure the correctness of compiler optimizations during development. Type information also allows TIL to perform additional optimizations that are not directly available to conventional compilers [13] [10].

A principal benefit of type-directed compilation is that it facilitates sound *tag-free garbage collection* [18]. Sound garbage collection requires that the heap pointers used by an executing program be identified unambiguously. Tag-free garbage collection permits these heap pointers to be identified without perturbing the run-time representations of values.

In TIL's run-time model, word-sized values (*e.g.*, integers and heap pointers) are not tagged, whereas composite values contain tags for their constituent locations. The *location* of a word-sized value is tagged instead of the value itself—this "out-of-band" tagging scheme allows a single location tag, or *trace value*, to identify many different run-time values. A *trace table* is a static encoding of trace values either for a given procedure activation or for a set of static storage locations. A procedure activation requires trace values for the machine registers used by the procedure as well as its stack frame slots. A complete set of trace tables can be synthesized at compile time for a program because all the possible procedure activation shapes can be statically determined.

RTL (Register Transfer Language) is the lowest-level intermediate language used in TIL. A back end for TIL thus translates RTL to assembly language. RTL is an imperative language that resembles the instruction set of a RISC processor, but it also provides complex primitives that are tailored to SML. An RTL *pseudo register* identifies a procedure-local, word-sized storage location: pseudo registers are mapped to machine registers and stack slots by the back end. RTL is not a typed intermediate language, but RTL does annotate pseudo registers with trace values; these are similar to, but distinct from, run-time trace values (the latter are a translation of the former).

There are actually two versions of the TIL compiler: *TIL1* [17] [13] is the first-generation compiler; its successor is called *TILT*. We will refer to the *TIL* compiler when the discussion applies to either compiler interchangeably. Both compilers share a common RTL language: the back end of TIL1 translates RTL directly to assembly language. This back end operates on RTL itself and explicitly propagates trace values through the spilling and register allocation phases.

As only the trace value component of this back end was customized for TIL1, it largely duplicates other work.

MLRISC[1], on the other hand, is a compiler back end implemented in SML that transforms an abstract intermediate language into the assembly language for a particular processor architecture. Taking RTL, MLRISC, and the TIL run-time system as given, our task is to transform RTL code into MLRISC code, and to make the object code generated by MLRISC compatible with the TIL run-time system. We impose an additional constraint on our implementation: we may not customize the interface of MLRISC specifically for TIL, as this would make it necessary to track such customizations across new versions of MLRISC.

As MLRISC does not propagate trace information, we cannot use it as a "drop in" replacement for the TIL1 back end—we must have an additional mechanism that derives run-time trace values from RTL trace values. Run-time trace values are encoded with references to machine register numbers and stack slots. This means that our mechanism must operate in concert with MLRISC, because the global register allocation and spilling phases of MLRISC assign these locations. This is the principal difficulty addressed by our work: how do we translate abstract trace values to concrete trace values "in parallel" with the abstract-to-concrete code translation performed by MLRISC?

Note that significant correctness questions are inevitably raised by the specification of such a translation, because trace values represent invariants—much in the same way that types represent invariants—that may be perturbed by the back end. Trace tables betray an implicit expectation by the run-time system that the object code produced by the back end will (loosely) reflect the original type structure of the program. As the back end is presented with no explicit type structure, how can we expect its code transformations to respect such a type structure? Another contribution of our work is that we describe the implicit constraints that we expect MLRISC to satisfy to ensure the soundness of the trace value translation.

In the remainder of this paper, we focus on how TIL communicates trace information to the run-time system by way of MLRISC. In Section 2 we detail the compiler and run-time system, whereas in Section 3 we present MLRISC. We discuss in Section 4 the techniques we use to marry MLRISC to RTL and the TIL run-time system, and Section 5 is an assessment of our experience. In Section 6 we propose improvements to the current implementation, and in Section 7 we draw together summarizing conclusions.

2 TIL

The TIL compiler is characterized by its aggressive use of type information. TIL compiles programs written in the Standard ML '97 programming language to DEC Alpha assembly language. All transformations (*i.e.*, compiler phases)

[1] We chose to use MLRISC as a back end for our compiler because our research does not directly address back end implementation technology. With MLRISC, we hope to leverage the work of a larger group of researchers.

in TIL are based on explicitly typed intermediate languages. However, RTL, the lowest-level intermediate language used in TIL, is not typed in the same sense that the other intermediate languages are typed. RTL pseudo registers are tagged with trace values that represent a degenerated form of type information that is tailored to the run-time system. Figure 1 is a depiction of the intermediate languages used in TILT; see Morrisett [13] for a description of the intermediate languages used in TIL1.

SML ➤ Abstract Syntax ➤ HIL ➤ MIL ➤ RTL ➤ MLRISC ➤ Assembly Language

Fig. 1. Intermediate Languages in TILT

2.1 HIL and MIL

The TILT elaborator translates programs from abstract syntax to *HIL* (High-level Intermediate Language). HIL is an explicitly-typed refinement of the SML programming language, including the module system; a detailed discussion of HIL is beyond the scope of this paper (see Harper and Stone [11] for further details). HIL is translated to *MIL* (Mid-level Intermediate Language) by the *phase splitter*, which is responsible for eliminating modules and breaking abstraction barriers. MIL is a lower-level, explicitly-typed, polymorphic intermediate language that does not provide modules.

2.2 RTL

TILT translates MIL to *RTL* (Register Transfer Language) after performing closure conversion, determining data representations, and making heap allocation explicit, among other things. RTL resembles an abstract assembly language in which there are an unbounded supply of local *pseudo registers* for each procedure. Pseudo registers are identified by positive integers and are automatically mapped to machine registers and stack slots by the back end. Each pseudo register is annotated with a trace value that classifies the kinds of values that the register can contain—one can think of trace values as degenerated type information that is present only for the benefit of the run-time system. RTL trace values are derived directly from MIL variable types.

Figure 3 contains one possible RTL translation[2] of the SML function in Figure 2[3]. Pseudo registers are given names (*e.g.*, `x`) in this example to clarify

[2] This is not the actual RTL code currently produced by TILT for this function: it has been simplified by hand to clarify its correspondence to the original SML code. This correspondence is obscured by the poor RTL code that TILT currently generates.

[3] This function was not written to compute anything interesting. Rather, the particular intermediate code it translates to helps to illustrate points later in this paper.

the presentation; in an actual RTL program, pseudo registers are identified by positive integers. Variables introduced by the compiler are prefixed by an underscore. Pseudo register trace values are written following the pseudo-register name in parentheses—trace values encode the "traceability" of a pseudo register, perhaps by projecting the contents of another pseudo register (Table 1). Trace values identify pointers into the heap to the run-time system—we explain the role of run-time trace values in Section 2.3. Section 2.3 also contains a discussion of type environments, of which `_tenv1(trace)` is one example.

```
fun f(x, n: int, l: int list, l2: int list) =
      g(x, n, if length l>0 then hd l else 1)
```

Fig. 2. An SML Function to Take the Head of a List

`trace`	A pointer into the heap
`notrace_int`	An integer
`notrace_code`	A pointer to machine code
`notrace_real`	A floating-point number
`label`	A pointer to data, but not into the heap
`compute` *path*	May be a pointer into the heap: *path* is an expression that evaluates (at run time) to the actual trace value
`unset`	Uninitialized
`locative`	A pointer into the middle of an item in the heap (cannot be traced)

Table 1. Trace Values for an RTL Pseudo Register

On entry to the body of the procedure, the argument pseudo registers listed in the procedure header contain the values of the actual arguments passed to the procedure. Similarly, on exit from the body, the result pseudo register listed in the procedure header will be copied into the actual machine-level result register, if necessary. The arguments and results of each procedure call are simply listed in order, as RTL uses an implicit calling convention. When a given pseudo register needs to be moved to/from a specific argument/result machine register according to a particular calling convention, this code is generated as part of the call/entry/exit sequence.

2.3 The Run-time System

Run-time Type Information Certain benefits arise from the use of typed intermediate languages: for example, types can be checked after compiler passes to help ensure correctness. Additionally, type-based information can be used at

```
procedure f:
  arguments = [_tenv1(trace),               ; arguments to f
               x(compute _tenv1(trace).0),
               n(notrace_int),
               l(trace),
               l2(trace)]
  results   = [_t3(notrace_int)]            ; result of f
{
  call    "length"                          ; _t1 <- length(l)
            arguments = [l(trace)]          ; (call #1)
            results   = [_t1(notrace_int)]
  bcndi2  le, _t1(notrace_int), 0, _L1      ; if _t1<=0 goto _L1
  call    "hd"                              ; _t2 <- hd(l)
            arguments = [l(trace)]          ; (call #2)
            results   = [_t2(notrace_int)]
  br      _L2                               ; goto _L2
_L1:
  mv      1, _t2(notrace_int)               ; _t2 <- 1
_L2:
  call    "g"                               ; _t3 <- g(x, n, _t2)
            arguments =                     ; (call #3)
              [_tenv1(trace),
               x(compute _tenv1(trace).0),
               n(notrace_int),
               _t2(notrace_int)]
            results   = [_t3(notrace_int)]
}
```

Fig. 3. A Translation of the code in Figure 2 to RTL

run time for dynamic type dispatch and tag-free garbage collection[13]. In TIL, the main ramification for the back end is that the run-time system uses a simple form of type information to reclaim storage.

The TIL run-time system uses a tracing, copying garbage collector to reclaim unused values in the heap. When the garbage collector is invoked, it must determine the locations of all the heap pointers that are in use so that it does not reclaim accessible memory. The garbage collector is said to *trace* these pointers to determine the layout of the objects they address, and to copy these objects to new locations. Traceable pointers may reside in registers, on the stack, in the data segment, or in the heap. However, these locations may also contain a variety of non-pointer values (*e.g.*, word-sized integers) that *cannot* be safely traced. The "traceability" of a given location is determined by its type: trace values, which are derived directly from types, specify to the garbage collector which locations should be traced. Machine registers and stack frame slots are tagged with trace values according to a static table that is indexed by the return address of an active procedure. Static (*i.e.*, global) storage locations are tagged by a corresponding set of static tables. The header of a heap value contains trace values for its slots. Tagging locations is potentially more efficient than tagging values because a single location tag can be shared for many values.

Note that the usual run-time model for garbage collection is *not* tag free. A typical implementation of tagged garbage collection makes the representations of heap pointers and non-heap pointers disjoint by encoding "tags" in the low-order bits of each word. This approach introduces extra overhead, constrains the range of representable values, and complicates interoperability with other languages. Part of the purpose of TIL is to determine whether these pitfalls can be avoided in a garbage-collected programming language implementation.

Representing Type Information At run time, vestigial type information is represented as *type environments* and *trace tables*. Type environments supply type information for variables whose types cannot be resolved at compile time (*e.g.*, polymorphic variables). For example, in Figure 2, the type of `x` is polymorphic and thus cannot be statically determined by the compiler—`x` could take the value 3, `"three"`, `[1, 1, 1]`, or any of a number of values that have distinct representations at run time. A type environment for `f` has the caller pass an explicit representation of `x`'s type so that both the run-time system and `f` can operate on it [18] [10]. Type environments are needed only for functions such as `f` where complete type information is not available at compile time. A type environment is a record of values that encode properties of types that are important to the run-time system. Note that type environments are unlike the explicit value descriptors used in many other language implementations, in that type environments are constructed only in contexts where they are specifically needed, as opposed to being an integral part of every value.

Trace tables map machine registers, stack slots, and static locations to trace values. A trace table gives a value of `yes` to those locations that are known to contain pointers into the heap. Other trace values allow the status of a location to depend on a type environment or on the dynamic caller's trace table: the trace values that can be attached to a storage location by a trace table are documented in Table 2. This table resembles Table 1 because run-time trace values are derived from the corresponding RTL trace values. By contrast, objects in the heap have special headers that specify trace values for locations in the object.

`yes`	Contains a pointer into the heap
`no`	Does not contain a pointer into the heap
`callee` *id*	Contains the *saved value* of a callee-save register: *id* identifies a machine register in the dynamic caller's activation whose trace value should be used for this location.
`stack` *offset*, *index*	A polymorphic location: *offset* is the offset in the current stack frame of a pointer to a type environment that contains the trace value of this location at index *index*.
`global` *label*, *index*	A polymorphic location: *label* is the label of a type environment in the data segment that contains the trace value of this location at index *index*.
`unset`	Uninitialized
`impossible`	Contains a heap pointer, but cannot be traced

Table 2. Trace Values for a Trace Table Storage Location

Locating Pointers Trace tables are consulted by the run-time system only when the garbage collector is invoked. This invocation takes the form of a library call from an active procedure that is unable to allocate storage. At this time, the collector must locate and trace all pointers into the heap: this process is nontrivial because pointers can potentially reside in any machine register, stack location, or static variable, and because the pointers themselves contain no identifying information (*e.g.*, tags). For static variables, we create a table in a known location that points to the trace tables for all static regions. Tracing the stack is more difficult because the stack depends on the dynamic behavior of the program. However, there are only a statically computable number of distinct activations, each of which can be determined by the compiler. Thus, we index a static table for each activation record according to the return address of a call site in the corresponding procedure. Because the collector is invoked via a procedure call, we simply use the return address in its activation record to locate the trace table of the most recent stack frame. Each trace table includes the size of its stack frame, so we can use these offsets to "walk up" the stack. This means that we need to generate a trace table for each direct call to the collector and for each call to another procedure that might indirectly call the collector[4].

At collection time, a callee-save register initialized by an active procedure may have had its value saved on the stack by another procedure, or the original

[4] In practice, we simply assume that all procedures might indirectly call the collector.

value may be left intact. The trace table of the most recent procedure activation holds the correct trace values for the machine register file at the time the collector is invoked. If any callee-save registers are not allocated by the most recently called procedure, then their trace values are determined according to the trace table of the next most recently called procedure: this is the function of the `callee` trace value. A `callee` trace value is also possible when a callee-save register is saved to the stack (and the register is presumably overwritten)—in this case, the proper stack location is given the `callee` trace value and the trace table of the next most recent activation record is consulted to determine the status of the stack location. This process can continue as long as the trace value of a location is specified as `callee`.

Example In Figure 4, we show a possible DEC Alpha Assembly Language translation of the example function of Figure 2, whereas in Figure 5 we document the registers used in Figure 4. `$_tenv`, `$x`, `$n`, `$l`, and `$l2` are assigned to callee-save temporaries in this procedure (*e.g.*, `$11`, `$12`, etc. according to the standard calling convention). `$_tenv1` contains the type environment for the function. The procedure begins by allocating a stack frame of size 32 and saving the return address and the callee-save registers on the stack. The arguments to the procedure are then moved from registers defined by the calling convention into the corresponding callee-save temporaries. Next, a call is made to `length` with the value of `l` as an argument (the standard calling convention requires all calls to jump through `$pv`). The result of the length call is then compared against zero and a branch is taken to `_L1` if it is not strictly positive. Assuming the branch is not taken, a call is made to `hd` and the result is saved in `$t2`; otherwise `$t2` gets the value `1`. The two control paths next converge at a call to `g` with arguments `x`, `n`, and `$t2`—the result of this call becomes the result of `f` after it restores the caller's register file from the stack frame. The `ldgp` instructions are a peculiarity of the Alpha standard calling convention.

The important things to notice in Figure 4 are the three call sites (commented `(call #`n`)`) for which we must construct trace tables. These trace tables must correctly identify the trace status of values in the register file and the local stack frame at the time of the corresponding call. In Figure 6, we show a trace table for call #2—notice that the trace values of stack slots saving callee-save registers depend on the dynamic caller's trace table (these slots are given trace status `callee` n).

3 MLRISC

MLRISC [8] is a generic compiler back end developed by Lal George at Bell Laboratories. MLRISC is "generic" in the sense that it can be used to compile many different programming languages. The interface language to MLRISC, also called "MLRISC", is essentially an architecture-independent assembly language: MLRISC is thus suited to compiling programming languages for which a translation to assembly language is feasible; to date, MLRISC has been used to compile SML

```
f:   ldgp    $gp, 0($pv)      ; set global pointer
     subl    $sp, 32, $sp     ; alloc frame
     stl     $ra, 0($sp)      ; save return address
     stl     $_tenv, 8($sp)   ; save callee save
     stl     $x, 12($sp)
     stl     $n, 16($sp)
     stl     $l, 20($sp)
     stl     $l2, 24($sp)
     mov     $arg0, $_tenv    ; get arguments
     mov     $arg1, $x
     mov     $arg2, $n
     mov     $arg3, $l
     mov     $arg4, $l2
     stl     $_tenv, 28($sp)  ; save type environment
     mov     $l, $arg0        ; $t1 <- length(l)
     lda     $pv, length
     jsr     $ra, ($pv)       ; (call #1)
     ldgp    $gp, 0($ra)      ; set global pointer
     cmple   $res, 0, $t0     ; if $t1<=0 goto _L1
     bne     $t0, _L1
     mov     $l, $arg0        ; $t2 <- hd(l)
     lda     $pv, hd
     jsr     $ra, ($pv)       ; (call #2)
     ldgp    $gp, 0($ra)      ; set global pointer
     mov     $res, $t2
     br      $zero, _L2       ; goto _L2
_L1: lda     $t2, 1           ; $t2 <- 1
_L2: mov     $_tenv, $arg0    ; $t3 <- g(x, n, $t2)
     mov     $x, $arg1
     mov     $n, $arg2
     mov     $t2, $arg3
     lda     $pv, g
     jsr     $ra, ($pv)       ; (call #3)
     ldgp    $gp, 0($ra)      ; set global pointer
     ldl     $ra, 0($sp)      ; restore return address
     ldl     $_tenv, 8($sp)   ; restore callee save
     ldl     $x, 12($sp)
     ldl     $n, 16($sp)
     ldl     $l, 20($sp)
     ldl     $l2, 24($sp)
     addl    $sp, 32, $sp     ; dealloc frame
     jmp     $zero, ($ra)     ; return
```

Fig. 4. A Translation of the SML function in Figure 2 to DEC Alpha Assembly Language

`$sp` Stack pointer	`$arg`n Argument n
`$pv` Call address	`$res` Result
`$ra` Return address	`$t`n Caller-save temporary n
	`$zero` Always zero

Fig. 5. Registers Used in Figure 4

`$_tenv`	`yes`	Always trace
`$x`	`stack 28, 0`	Trace according to `$_tenv`
`$n`	`no`	Never trace
`8($sp)`	`callee $_tenv`	Use dynamic caller's trace value for `$_tenv`
`12($sp)`	`callee $x`	Use dynamic caller's trace value for `$x`
`16($sp)`	`callee $n`	Use dynamic caller's trace value for `$n`
`20($sp)`	`callee $l`	Use dynamic caller's trace value for `$l`
`24($sp)`	`callee $l2`	Use dynamic caller's trace value for `$l2`
`28($sp)`	`yes`	Always trace

Fig. 6. A Trace Table for Call #2 of Figure 4

and Tiger [3]. Our compiler differs from other compilers using MLRISC [4] [2] [3], however, in that TIL does not use dynamic tag bits to distinguish heap pointers from other word-sized values.

In MLRISC, as in RTL, local storage locations are identified by numbered *pseudo registers*. Pseudo registers are transparently mapped to machine registers or spilled to the stack by MLRISC. Pseudo registers in MLRISC, however, carry no trace values or other type information; there are distinct classes of integer, floating-point, and condition-code pseudo registers, but an integer pseudo register that happens to be used as a heap pointer is not distinguished in any way. The principal challenge in integrating MLRISC with TIL, then, is to propagate pseudo-register trace values to the run-time system in the form of run-time trace values. Because pseudo registers are transformed into machine registers and stack slots, trace values for these locations will be based on the code transformations performed by MLRISC (*e.g.*, register allocation, spilling).

In Figure 7, we show the SML function in Figure 2 as it might be translated to MLRISC. Pseudo registers are given names (*e.g.*, `x`) in this example to clarify the presentation; in an actual MLRISC program, pseudo registers are identified by positive integers (*e.g.*, `500`). Machine registers are referred to by a small positive integer (*e.g.*, `16`) and can be used interchangeably with pseudo registers in MLRISC code. Names generated by the compiler are prefixed by an underscore; `_tenv1` contains the type environment (see Section 2.3) for the function and `cs1` through `cs5` are used to hold the saved values of the callee-save registers. Following the Alpha standard calling convention, this code uses machine registers `16` through `20` to hold arguments and register `0` to hold the result; in MLRISC, unlike RTL, calling conventions are explicitly specified in terms of primitive operations. An MLRISC procedure is a sequence of imperative *statements*, each of which may refer to applicative *expressions*; the terms "statement" and "expression" have the normal connotations of programming language terminology. In Table 3 and Table 4, we document the MLRISC constructs used in this example. Expressions can be nested to an arbitrary depth, so, in general, a single statement can generate many assembly language instructions.

`bcc` *cexp*, *label*	Branch to label *label* if the result of evaluating conditional expression *cexp* is true.
`call` *addr*	Call the procedure at the address formed by evaluating expression *addr*.
`copy` *dst*, *src*	Copy the registers listed in *src* into the corresponding registers listed in *dst*; this is a "parallel" operation: no register can appear more than once in the union of *src* and *dst*. `copy` statements are coalesced [9] by MLRISC whenever possible.
`mv` *dst*, *exp*	Move the result of evaluating expression *exp* into register *dst*.
`jmp` *addr*	Jump to the code at the address formed by evaluating expression *addr*.
`ret`	Return from the current procedure.
`store32` *addr*, *exp*	Store the result of evaluating expression *exp* as a 32-bit value at the address formed by evaluating expression *addr*.

Table 3. Selected MLRISC Statements

4 Techniques

This section discusses the translation techniques we use to integrate MLRISC with TIL. In Section 4.1 we touch on the technology that translates RTL code to MLRISC code, whereas in Section 4.2 we outline how we construct trace tables for MLRISC from RTL trace values. Finally, in Section 4.3 we justify the correctness of trace values for translated code.

4.1 From RTL to MLRISC

Translating RTL "instructions" to MLRISC "statements" is relatively straightforward—the principal difficulty lies in generating efficient code for conditional branches. RTL provides two forms of conditional branch instruction: one that compares a pseudo register against zero, and one that compares two pseudo registers. The current translation from MIL to RTL favors the former kind of branch, even for comparisons between two pseudo registers. It does this by storing the boolean result of each comparison in a third pseudo register and then testing the third pseudo register against zero. Although this idiom matches the use of conditionals in certain RISC architectures (*e.g.*, the Alpha), it cannot

```
f:
  mv      gp, reg pv                    ; set global pointer
  mv      sp, sub(reg sp, const frame) ; alloc frame
  store32 add(reg sp, li 0), reg ra    ; save return address
  copy    [cs1, cs2, cs3, cs4, cs5],
          [l1, l2, l3, l4, l5]         ; save callee save
  copy    [_tenv1, x, n, l, l2],
          [l6, l7, l8, l9, l20]        ; get arguments
  store32 add(reg sp,
              const _tenv1_offset),
          reg _tenv1                   ; save type environment
  copy    [l6], [l]                    ; _t1 <- length(l)
  mv      pv, label "length"
  call    reg pv                       ; (call #1)
  mv      gp, reg pv                   ; set global pointer
  copy    [_t1], [0]
  bcc     cmp(le, reg _t1, li 0), _L1  ; if _t1<=0 goto _L1
  copy    [l6], [l]                    ; _t2 <- hd(l)
  mv      pv, label "hd"
  call    reg pv                       ; (call #2)
  mv      gp, reg pv                   ; set global pointer
  copy    [_t2], [0]
  jmp     label _L2                    ; goto _L2
_L1:
  mv      _t2, li 1                    ; _t2 <- 1
_L2:
  copy    [l6, l7, l8, l9],
          [_tenv1, x, n, _t2]          ; _t3 <- g(x, n, _t2)
  mv      pv, label "g"
  call    reg pv                       ; (call #3)
  mv      gp, reg pv                   ; set global pointer
  copy    [_t3], [0]
  copy    [0], [_t3]
  mv      ra, add(reg sp, li 0),       ; restore return address
  copy    [l1, l2, l3, l4, l5],
          [cs1, cs2, cs3, cs4, cs5]    ; restore callee save
  mv      sp, add(reg sp, const frame) ; dealloc frame
  ret                                  ; return
```

Fig. 7. A Translation of the SML function in Figure 2 to MLRISC

`add` (*exp1*, *exp2*)	Evaluates to the result of adding the results of evaluating *exp1* and *exp2*.
`cmp` (*cmp*, *exp1*, *exp2*)	Evaluates to true if the result of evaluating *exp1* and *exp2* are ordered according to comparison *cmp*. This expression evaluates to a condition code, as opposed to an integer.
`const` *fn*	Evaluates to the result of calling the function *fn* during the final code generation phase. `const` allows constants in the final assembly language program to depend on the results of earlier phases (*e.g.*, spilling).
`label` *string*	Evaluates to the address of label *string*.
`li` *n*	Evaluates to the integer *n*.
`load32` *addr*	Evaluates to the result of loading a 32-bit value from the address formed by evaluating expression *addr*.
`reg` *id*	Evaluates to the contents of pseudo register *id*.
`sub` (*exp1*, *exp2*)	Evaluates to the result of subtracting the result of evaluating *exp2* from the result of evaluating *exp1*.

Table 4. Selected MLRISC Expressions

be expressed efficiently in MLRISC, because there is no statement to move the result of a comparison directly into an integer pseudo register. We address this problem by "preprocessing" the RTL code into two-operand conditional branch form whenever the result of a compare instruction is used by an immediately following branch instruction, *and* the boolean result is not used anywhere else in the procedure.

Although RTL and MLRISC treat pseudo registers in much the same way (*i.e.*, as local storage locations for a procedure), one cannot interchange the two notions. For example, the MLRISC translation of an RTL instruction that refers to pseudo register `500` cannot simply refer to pseudo register `500`, because pseudo registers in MLRISC code must be allocated explicitly through MLRISC. To overcome this difficulty, we maintain a mapping from RTL pseudo registers to MLRISC pseudo registers and allocate a new pseudo register from MLRISC whenever we see an RTL pseudo register that does not have an existing mapping. As MLRISC pseudo registers, unlike RTL pseudo registers, carry no explicit trace values, we construct a separate mapping from MLRISC pseudo registers to run-time trace values. Storing the trace values "off to the side" allows us to forget about RTL pseudo registers entirely for the later phases of the translation.

Our translation "forces" certain pseudo registers to spill by manually replacing them with memory accesses; this transformation is accomplished as a separate pass over the MLRISC code just before we pass it to MLRISC. Pseudo registers that are forced to spill include those holding type environments referred to by trace values, those saving callee-save registers in the presence of exception handlers, as well as any global registers that do not fit in the machine register file. Because the trace value of a given pseudo register can refer to a type environment on the stack to resolve its status (*e.g.*, `stack` in Table 2), we must ensure that these type environments are in fact on the stack and not being held in a machine register. The callee-save registers must be restored to their former values at the end of an exception handler, so we force the pseudo registers that are used to save these registers to be spilled to the stack so that we can later restore them. Finally, for performance, TIL reserves a small number of machine registers to hold global values that are used by most procedures (*e.g.*, the current heap and limit pointers). Unfortunately, certain machine architectures (most notably, the Intel x86) do not have enough registers for this scheme to be feasible, so we rewrite code using these registers with references to global memory locations.

4.2 Constructing Trace Tables

The most interesting part of the translation from RTL to MLRISC is constructing trace tables for call sites. As MLRISC does not explicitly propagate type information, we construct trace tables by passing trace values "around" MLRISC's code generator. Trace tables are represented as *data* pseudo operations that are compiled into the data segment of the program.

Because trace tables are encoded in terms of machine register numbers and stack offsets, and because trace values are attached indirectly to pseudo registers, we must account for the results of spilling and register allocation during trace table generation. For example, if pseudo register `500` has the run-time trace value `yes` and is mapped to machine register `12`, then a trace table should contain a `yes` entry for machine register `12`. This implies that we must generate code and trace table data in separate phases—first we translate the code to obtain a pseudo-register mapping, then we generate trace table data based on this mapping. We must also generate a single trace table for all the static locations in a module: this is accomplished by mapping the RTL label for each static location to a corresponding MLRISC label. In Figure 8 we illustrate how an RTL module containing procedures and static variables is transformed into MLRISC code statements and data directives. Note that for reasons of expediency, trace tables are generated first in terms of RTL data directives which are then translated to MLRISC data directives. This allows us to reuse the trace table module from the TIL1 back end.

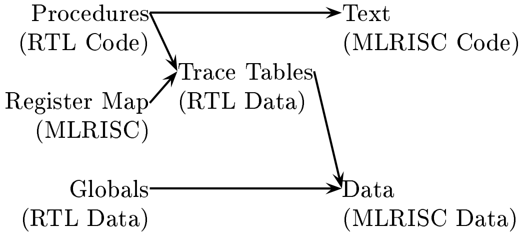

Fig. 8. Generating MLRISC Code and Trace Tables from RTL

The results of register allocation and spilling are not difficult to obtain from MLRISC. The mapping from pseudo registers to machine registers is exported as a data structure by the MLRISC interface. We can construct the mapping from spilled pseudo registers to stack offsets because MLRISC spills pseudo registers via a "call back" to our code. It is important to understand that the mappings used by these phases must be accessible for our technique to work—for a back end that does not export this information, we cannot determine how the pseudo registers are represented at run time, and therefore cannot construct trace tables from pseudo-register trace values. This problem is explored further in Section 5.2

4.3 Register Allocation

Suppose that in Figure 7, pseudo-registers `l2` (trace value `yes`) and `_t2` (trace value `no`) have both been mapped to machine register `4` by MLRISC's register allocator. Which trace value should we give machine register `4` when constructing a trace table? Obviously, we cannot resolve this conflict with just the pseudo-register mapping—we should look at the code to determine which pseudo register was *defined* most recently on the control path to the call site in question. However, because the code may contain arbitrary branches and loops, a linear scan will not resolve this ambiguity in general. Notice that pseudo registers can be mapped to the same machine register only if they have non-overlapping live ranges: otherwise, definitions of the pseudo registers would interfere with each other. Thus, for a given call site we can resolve a conflicting register assignment by choosing the trace value of the pseudo register that is live across the call site [7], as there can be only one.

Figure 7 additionally illustrates a deeper problem, in that the correct trace value for machine register `4` at `call #3` depends on the run-time contents of pseudo-register `_t1`: if `_t1` is greater than zero, then machine register `4` will be overwritten by the result of `call #2`. This example suggests that there are cases where the code generation transformations induced by the back end will make it impossible to give a fixed trace value to a particular machine register. Fortunately, such unpredictable trace values can only arise when none of the pseudo registers in question are live across the call site—otherwise, the generated code would be incorrect, because the definition of one pseudo register could interfere with the later use of another.

Returning to our example, we know that neither `l2` nor `_t2` can be used after `call #3`, because such a use might read the value of the wrong pseudo register. This observation suggests that we must take into account the next *use* of a pseudo register *after* the call site as well as its *definition before* the call site—it is not sufficient to simply note the trace value at the most recent definition. We can give a machine register the trace value `no` if the contents of that register will not be used after the corresponding call site: because the register will not be used, its contents need not be retained by the garbage collector. This happy coincidence allows us to give the trace value `no` to machine register `4` for our example.

Thus, to construct a trace table for a given call site, we map the pseudo registers live across the site through the pseudo-register mapping and pair the resulting machine registers with the trace values for the corresponding pseudo registers. All other machine registers are given the trace value `no`. Liveness analysis has the added benefit of minimizing storage retained during garbage collection. This, in turn, enhances performance by reducing the load on the collector and also enables certain programs to terminate that would not otherwise [16]. Our liveness analysis is based on well-understood data-flow techniques [1]. Note that the call-site liveness analysis must be at least as precise as the register-allocation liveness analysis for this technique to work.

We give trace values to stack slots holding spilled pseudo registers with an analogous technique—in Figure 9 we show the construction of a trace table for `call #2` of Figure 7 from the live pseudo-register set, a run-time trace value mapping, and a sample MLRISC register mapping.

Live Set

`_tenv1`
`x`
`n`

MLRISC Register Map

`_tenv1`	$\mapsto$	`11`
`x`	$\mapsto$	`12`
`n`	$\mapsto$	`13`
	...	

RTL Trace Map

`_tenv1`	$\mapsto$	`yes`
`x`	$\mapsto$	`compute _tenv1.0`
`n`	$\mapsto$	`no`
	...	

$\downarrow$

Trace Table

`11`	$\mapsto$	`yes`
`12`	$\mapsto$	`stack 12, 0`
`13`	$\mapsto$	`no`

Fig. 9. Constructing a Trace Table

5 Assessment

5.1 TIL

RTL is not well-suited as a source language for MLRISC because the languages are similar enough that the translation between them is essentially wasted work. The principal difference between RTL and MLRISC is that RTL pseudo registers are annotated with trace information; as we describe in Section 4.1, it is not difficult to simulate this capability for MLRISC pseudo registers, so there is no compelling reason to use RTL as a separate translation step. We thus see the use of RTL as an intermediate language as vestigial: we plan to translate directly from the MIL intermediate language in the future. We originally decided to translate from RTL to expedite development of the compiler, as MIL was in a fluid state of development at the time.

5.2 MLRISC

This section seeks to identify the specific features of MLRISC that made it possible for us to integrate it with TIL. We also suggest additional features not found in MLRISC that would have made our job easier, or would have resulted in more efficient code generation. We speculate that our experience may be of use to designers of other generic back-ends [5] [6].

We divide the relevant features into two classes: those that are essential to the propagation of type information, and those that can enhance performance or simplify translation when using type-directed techniques. An underlying theme of our characterization is that the back end needs to do more than simply emit assembly code on behalf of the client—it should also return information about *how* the translation was accomplished.

We first present a brief summary of our conclusions. These are the key features of MLRISC that enable the type-directed translation techniques of our compiler:

- A visible pseudo-register-to-machine-register mapping
- A machine-register mapping that is unique for a given pseudo register
- A visible pseudo-register-to-stack-slot mapping
- A spilled pseudo register will not also be mapped to a machine register

These are features not found in MLRISC that might enhance the performance of our translation:

- Visible liveness information
- An extensible spill mechanism

Essential Features As the TIL run-time system uses trace tables that contain machine register numbers, a client-accessible pseudo-register mapping is needed to propagate trace values. Input trace values are attached to pseudo registers, so we must be able to uncover which pseudo registers are mapped to which machine registers if we are to encode trace value mappings for the latter. Although it might be possible to deduce the pseudo-register mapping by comparing the output object code with the input pseudo code, this is likely to be difficult for a back end that performs aggressive optimizations (*e.g.*, global instruction scheduling).

The shape of the mapping to machine registers can also present problems to the implementor. MLRISC maps each pseudo register to at most one machine register [9]; thus, when an unspilled pseudo register is live across a call site, we can always precisely identify which machine register it is mapped to. If a single pseudo register might be mapped to one of several machine registers at different points in the code, then MLRISC would need to tell us the mapping ranges for each register. Again, it might be possible to deduce this information from the object code—or even from the pseudo code, if we know the back end's register allocation algorithm—but such a deduction algorithm is likely to be complex and correspondingly inefficient.

Given the mapping to machine registers, we must have a similar mapping to stack slots for pseudo registers that have been spilled transparently by the back end. For MLRISC, there is no special interface to this information, but because we implement the spill mechanism ourselves[5], we can easily reconstruct it. If a back end does not provide a customizable spill mechanism, it must allow the client to query the spill status and location of a given pseudo register so that trace value mappings can be constructed for the stack.

Note that to simplify trace table generation, we ensure that a spilled pseudo register will never be mapped to a machine register (*i.e.*, a spilled pseudo register is always on the stack for its entire lifetime). This is accomplished by allocating a new temporary and rewriting the instruction referring to the spilled pseudo register with a reference to the temporary instead; a store instruction (or load in the case of a reload) is then appended (or prepended) to the rewritten instruction. Because the lifetime of the temporary is only between the rewritten instruction and the store (or load), we can assume that it will never be live across a call site [7], and thus need not be traced. Although such a temporary will never be traced, the stack slot containing the original value still might be traced if it is live across the call site in question. Making the "spilled to the stack" and "mapped to a machine register" states exclusive for a given pseudo register simplifies the process of constructing trace tables—if we could not assume this, then we would have to track *where* the pseudo register is moved to or from the stack and which location(s) (stack slot or register) its next use(s) expect it to be in. As this information is not exported by MLRISC, it is not clear how we would recover it.

Our assumption about the lifetimes of spill temporaries may not hold in the presence of global instruction scheduling, but as MLRISC does not currently perform global instruction scheduling, our implementation is sound for the moment. To implement our translation in the presence of global instruction scheduling, we would need access to the liveness analysis of the back end, or we would need to be able to constrain the scheduling of instructions referring to potentially traceable values. Note that the former solution has the added benefit of reducing the overhead incurred by our translation.

Desirable Features It is unfortunate that we must perform a data-flow analysis to determine liveness across call sites, because this work will be largely duplicated by MLRISC's register allocator. It would be more efficient if we could derive call site liveness from the register allocator's own liveness information. This might be accomplished in a hypothetical version of MLRISC by returning the pseudo registers that are live into and out of each basic block as part of the translation to assembly language. Additionally, each call site must be isolated in its own basic block: this could be done by TIL if MLRISC were to provide a way to explicitly delimit basic blocks. Note that because spilling is performed in a pass prior to register allocation, liveness information may be lost for spilled pseudo

[5] When MLRISC decides to spill a pseudo register, it calls a client-supplied function to return an architecture-specific code sequence for the spill.

registers when they are replaced by memory accesses—unless MLRISC retains information showing that they cannot be aliased, it will have to assume that they are always live after the first definition. It might be possible to deduce the liveness of spilled pseudo registers by analyzing the spill and reload patterns via another data-flow analysis, but this seems counterproductive. It is not correct to simply assume that spilled pseudo registers are always live, because the contents of a given stack slot may not be initialized until after the first call site.

Our translation to MLRISC includes a "forced spill" pass (see Section 4.1) that replaces certain pseudo registers with memory accesses. As MLRISC implements a similar spilling pass, it would save implementation time if MLRISC were to allow the client to provide an additional spill set as a part of code generation. This would avoid the extra spill pass that currently handles our special spill cases.

5.3 Performance

Benchmarks Because the optimizer in the TILT compiler is still under construction, we cannot yet take meaningful performance measurements of the object code produced by MLRISC in conjunction with TILT. However, because TILT and TIL1 use the same RTL intermediate language, we can use MLRISC as a back end for the TIL1 compiler: in Table 5, we present the relative execution times of some of the benchmark programs from Tarditi *et al.* [17]. These measurements show that by using MLRISC as a back end for TIL1, we introduce a significant amount of overhead into the generated code. We believe that this overhead is due to complications in the translation of RTL code to MLRISC code, and is not due to MLRISC itself.

Program	TIL1	TIL1-MLRISC	TIL1-MLRISC/TIL1
FFT	2.02	2.49	1.23
Knuth-Bendix	2.28	2.70	1.18
Lexgen	2.66	3.09	1.16
Life	2.07	2.51	1.21
Matmult	2.66	2.61	0.98
Simple	11.91	14.03	1.18

Table 5. TIL1-MLRISC Execution Time Relative to TIL1

The execution times in Table 5 are the time in seconds required to execute the programs on a DEC Alpha 3000/600 workstation with 96mb of RAM. This workstation has a 175MHz Alpha 21064 processor with 8k primary instruction and data caches and a 2mb unified secondary cache. Each figure is the arithmetic mean of ten consecutive runs of the corresponding program. See Tarditi *et al.* [17] for descriptions of the benchmark programs.

We made one change to MLRISC for the purpose of benchmarking: MLRISC ordinarily generates floating-point arithmetic instructions with the `sud` flags set in the instruction word. Because these instructions are emulated in software on our workstation, we replaced them with the equivalent "garden-variety" instructions (e.g. `addt` instead of `addt/sud`). The `sud` flags control the precise semantics of floating-point operations—see the *Alpha Architecture Handbook* for more information. As use of the `sud` flags makes the FFT benchmark about 300 times slower on our workstation, it is not meaningful to take performance measurements with them set.

We used a calling convention without integer callee-save registers for these benchmarks because, when used with TIL1, MLRISC often allocates pseudo registers to callee-save registers in such a way that violates the constraints of our trace table encoding. In particular, our encoding requires that the contents of a callee-save register either be saved on the stack or be left in the original machine register during the activation of a procedure. When used with TIL1, however, MLRISC often allocates the pseudo registers used to save the callee-save registers to other (different) callee-save registers—this is not expressible in our trace table encoding. We have encountered this problem with much less frequency when using MLRISC as a back end for TILT, but it remains unresolved.

Target Code The principal techniques outlined in this paper for interfacing MLRISC to TIL operate only on type information, and therefore should not have a direct effect on object code quality. However, there are sources of inefficiency in the code transformations performed by our translation. Additionally, the limitations of these techniques may introduce performance-limiting constraints when used with other back ends.

To elaborate on the former point, the details of the translation from RTL to MLRISC has a significant effect on the ultimate quality of the object code, as is indicated by the discussion of conditional branch translation in Section 4.1. It is clear, however, that this particular difficulty arises as an artifact of an unfortunate mismatch between the semantics of conditional values in RTL and MLRISC, and does not represent a general problem with the interaction between TIL and MLRISC.

Another valid question might arise about whether the "forced spill" phase outlined in Section 5.2 will introduce so many new spills as to significantly degrade performance. Although it seems unlikely that the indiscriminate spilling of type environments will have a measurable effect on performance, one cannot so easily dismiss the spilling of the callee-save registers and the pervasive global registers. Note, however, that in each of these cases, spilling is introduced as a consequence of constraints imposed by the run-time system, and not as a consequence of a poor interaction between the compiler and the back end. Thus, the forced spill phase is really a function of the run-time architecture used with TIL and will be required in some form whether or not the object code is generated by MLRISC. A general discussion of the performance of type-directed run-time architectures is beyond the scope of this paper, but see Tarditi *et al.* [17] and Morrisett [13].

A potential performance problem that *is* directly related to the use of MLRISC as a back end for TIL concerns the constraints that our techniques impose on a back end to simplify trace table generation. These restrictions are discussed in Section 5.2, and although none of them appear to be especially restrictive, it will be difficult to demonstrate this without measurements. Unfortunately, this is particularly awkward to do for our technology, as only one of these constraints (spilled pseudo registers) can be alleviated in MLRISC. Even if we were to remove this limitation, however, we would not be able to execute the resulting code because of the absence of trace tables. It might be more productive to examine individual measurements of these code generation features on other compiler platforms and then use the results as a guide to forming conclusions about the potential drawbacks of our techniques.

Compilation Speed A final performance consideration relates to how the use of our techniques affects the speed of compilation. Because we perform an extra liveness analysis before code generation (see Section 4.3), there is a potential for inefficiency here. Preliminary measurements show that our use of MLRISC has a significant performance cost: the combined RTL-to-MLRISC translation and the subsequent MLRISC code generation phases together perform at less than half the speed of the TIL1 back end when used with TILT. These same measurements also indicate that the bulk of the time is being spent in translation code external to MLRISC; MLRISC on its own is usually faster than the TIL1 back end. Unfortunately, we have not yet isolated the source of this inefficiency: the extra liveness analysis by itself only accounts for a fraction of the translation overhead—it typically consumes less than 5 percent of the total compilation time. It is certainly possible that most of the translation overhead is caused by unoptimized code on our end. In our opinion, it is too early to draw meaningful conclusions about the performance of the translation itself, as there may still be room for substantial optimization.

6 Future Work

In this section, we discuss features of MLRISC that are currently underutilized.

MLRISC is able to perform inter-procedural register allocation on procedures in the same call graph. We do not currently take advantage of this feature, but hope to utilize it once we fully understand the complications with regard to trace table construction. Because ML programs typically use function calls for looping, the performance benefits of this optimization may be significant when the compiler has not entirely optimized away procedure calls.

MLRISC does not currently perform any global instruction scheduling, but we expect that it will eventually do so. We anticipate that this optimization will introduce complications into our call-site liveness analysis because the live ranges of pseudo registers will be perturbed across basic blocks. For example, if the first definition of a traceable pseudo register is moved forward past a call site, then the garbage collector will trace an uninitialized value at that call site if no corrective action is taken. Because basic blocks in ML programs tend to be so small that local instruction scheduling has little benefit, we think it will be important to find a solution to this problem that does not unduly constrain the back end. This topic is also discussed in Section 5.2

MLRISC provides condition-code pseudo registers in addition to integer and floating-point pseudo registers. We currently do not use these pseudo registers because RTL does not distinguish condition codes from integers. A direct translation from MIL might make it easier to take advantage of these registers and also to correct some additional inefficiencies in the translation of conditional branches.

Finally, we hope to isolate the source of the current translation inefficiency so that using MLRISC with TILT is not significantly slower than using the TIL1 back end. We also hope to improve the performance of the object code generated by MLRISC once implementation of the TILT optimizer is complete.

7 Conclusion

We have presented our approach to integrating MLRISC, a generic back end, with TIL, a type-directed compiler. Our work is a solution to a specific instance of a more general problem: how can abstract trace information be mapped to concrete trace information, given that the correct mapping is a function of a parallel code translation performed by the back end? Register allocation and spilling are the critical code translations that must be reproduced to translate trace information. MLRISC exports its register and spill mappings: it is this property of MLRISC that makes it possible to use it with our compiler.

As important parts of TILT are still being developed, we cannot draw definitive conclusions yet about the merits of our approach. It is currently unclear if the use of MLRISC will give us a significant improvement in object code quality. It is also unclear whether the "scaffolding" we have constructed around MLRISC can be made efficient enough not to seriously degrade compilation time.

It is reasonable to object to our use of RTL as an intermediate language between MIL and MLRISC, because RTL serves essentially the same purpose as MLRISC. We chose to retain RTL from TIL1 only to better compartmentalize our development effort. One could argue that some of the problems we have encountered are due more to the use of RTL than to the use of MLRISC. In particular, we expect that in a hypothetical translation from MIL to MLRISC, a redundant liveness analysis on MIL code would be less onerous due to its more structured control flow. This would appear to undermine our contention that the back end should export the results of its liveness analysis for use by the rest of the compiler. However, we do not believe that simply performing the call-site liveness analysis on MIL code is an adequate long-term solution, because it is not clear that liveness of variables in MIL code necessarily corresponds to liveness of machine registers and stack slots in machine code—we even know that this correspondence will not hold in the presence of global instruction scheduling. For this reason, we think that the availability of liveness information from MLRISC will be crucial to the long-term success of this effort.

Our work attests that MLRISC is "generic enough" to be reused as the back end of our compiler, even though TIL is substantially different from Standard ML of New Jersey [4], the compiler for which MLRISC was originally developed. Reuse has attendant costs, however, and the most significant of these appear to be related to the speed of compilation. We suggest that generic compiler technology is a valuable asset, but that more developers will benefit from it if interfaces are made flexible enough to encompass dissimilar compilation strategies.

Acknowledgements

Special thanks to Lal George for his helpful insights during the integration of MLRISC and TIL.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, tools.* Addison-Wesley, 1986.
2. Andrew W. Appel. A runtime system. *Lisp and Symbolic Computation 3*, pages 343–380, 1990.
3. Andrew W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.
4. Andrew W. Appel and David B. MacQueen. Standard ML of new jersey. In *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13. Springer-Verlag, August 1991.
5. Andrew W. Appel et al. The national compiler infrastructure project.
6. Robert P. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. Technical report, Computer Systems Laboratory, Stanford University.
7. Lal George. Personal Communication.
8. Lal George. MLRISC: Customizable and reusable code generators. Technical report, Bell Labs, December 1996. submitted to PLDI.
9. Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.
10. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141. ACM, January 1995.
11. Robert Harper and Chris Stone. A type-theroretic interpretation of standard ML. Technical report, Carnegie Mellon University, 1997. submitted for publication.
12. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97. ACM, January 1998.

13. J. Gregory Morrisett. *Compiling with Types.* PhD thesis, Carnegie Mellon University, December 1995. Published as CMU Technical Report CMU-CS-95-226.
14. George C. Necula. Proof-carrying code. In *Conference Record of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, January 1997.
15. George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemantation*, New York, 1998. ACM Press.
16. Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Conference on Lisp and Functional programming*, June 94.
17. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL : A type-directed optimizing compiler for ML. In *Proceedingsof the ACM SIGPLAN Conference on Programming Language Design and Implemantation*, pages 181–192, New York, May21–24 1996. ACM Press.
18. Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings 1994 ACM Conference on Lisp and Functional Programming*, June 1994.