Automatically Generating Back Ends for a Portable Assembly Language Using Declarative Machine Descriptions

João Dias
Harvard School of Engineeering and Applied Sciences
dias@eecs.harvard.edu

Norman Ramsey
Harvard School of Engineeering and Applied Sciences
nr@eecs.harvard.edu

Abstract

We show how to generate the back end of an optimizing compiler from a formal description of the syntax and semantics of machine instructions. Our generated back ends for $x86$, ARM, and PowerPC perform as well as their hand-written counterparts. Automatic generation is enabled by two new ideas: a model of machine-level computation that reduces back-end generation to the problem of finding implementations of about a hundred simple, machine-level operations; and an algorithm that finds these implementations by combining machine instructions.

1. Introduction

Writing a back end for an optimizing compiler is difficult. The compiler writer must know not only the syntax and semantics of the target instruction set but also the internal data structures and invariants of the compiler, which can be very complex. For years, the state of the art has been to write back ends using domain-specific languages which combine knowledge of the compiler's data structures with knowledge of the target machine.

We present a new technique which realizes a goal long sought in the compiler community: decoupling compiler knowledge from machine knowledge. Using our system, the compiler writer targets an expressive, low-level language which hides almost all facts about the machine; the back end is generated automatically from *declarative machine descriptions*, which are completely independent of the compiler.

A declarative machine description contains no code; it describes properties of a machine using a well-defined formalism that is independent of any particular tool or programming language. For example, SLED descriptions specify the binary and assembly encodings of machine instructions (Ramsey and Fernández 1997), and λ-RTL descriptions specify the semantics of machine instructions (Ramsey and Davidson 1998). Unlike a "machine description" that stitches together fragments of C code, a declarative machine description is amenable to analysis and can be reused to build multiple tools, including assemblers and disassemblers (Ramsey and Fernández 1995), linkers (Fernández 1995), dynamic code generators (Auslander et al. 1996), debuggers, binary translators (Cifuentes, Van Emmerik, and Ramsey 1999), and compilers (Dias and Ramsey 2006). Declarative machine descriptions are also small and easy to reason about; even the combination of a SLED and a λ-RTL machine description can be one-tenth the size of one of the "machine descriptions" used to retarget `gcc`. And by design, declarative machine descriptions can be written by a machine expert who knows nothing about the internals of any compiler.

The primary contribution of this paper is to show how to generate an instruction selector from a declarative machine description. Building on earlier work (Olinsky, Lindig, and Ramsey 2006; Dias and Ramsey 2006), the work described here enables us to generate back ends automatically. Our contributions are as follows:

- We have developed a new model of machine-level computation. By analyzing this model, we have found a way of decomposing register transfers into *tiles*, where a tile represents a simple operation such as a sign-extending load or a three-register arithmetic operation (Section 2.2). The decomposition is the same for every target machine and is implemented once.
- Given declarative descriptions of a target machine, we have developed an algorithm which enables us to find an implementation of each tile using a sequence of target-machine instructions. Our search algorithm uses three simple ideas: exploiting algebraic laws to find more ways of using instructions (Section 3.2.1); compensating for instructions' unwanted side effects (Section 3.2.2); and identifying distinguishable and indistinguishable register sets and memory locations and finding ways to move data between them (Section 3.2.3).
- We show that given an arbitrary machine, finding implementations of tiles is undecidable (Section 4). But real machines are not arbitrary, and we have generated working back ends for x86, ARM, and PowerPC targets. The generated code is as good as code generated by hand-written back ends (Section 6); the key technique that makes this possible is that optimization runs *after* instruction selection, so it is actually better if the instruction selector emits naïve code (Benitez and Davidson 1994).

The import of these contributions is that a good, reliable back end for an optimizing compiler can be built more quickly and more easily than ever before. Our work also has software-engineering benefits: writing a declarative machine description is significantly easier than writing a back end; the generated back end is much less likely to contain errors than a hand-written back end (Section 6.1.1). Moreover, errors that originate in faulty machine descriptions can often be found by independent testing tools (Fernández and Ramsey 1997; Bailey and Davidson 2003).

2. Instruction selection and register transfers

The heart of a back end is the instruction selector, which maps intermediate code to machine code. We follow the lead of Davidson and Fraser (1980, 1984) in dividing this problem into two parts: first, the compiler writer extends the front end with a hand-written component that translates from intermediate code into low-level *register-transfer lists*, or RTLs. Then, the back end translates these RTLs into machine instructions. In the work of Davidson and Fraser, however, as well as in follow-on work by Benitez and Davidson (1988), *both* parts are machine-dependent and must be written anew for each machine. The reason is that in Davidson's compilers, each

Literal constant	k		
RTL operator	$\oplus$		
Storage space	s		
Expression	e	$::=$	$k \mid l \mid \oplus(\mathtt{e}_1, \cdots, \mathtt{e}_n)$
Location	l	$::=$	$\$s[e] \mid name$
Assignment	f	$::=$	l `:=` e
Guarded Assignment	g	$::=$	e `-->` f
RTL	r	$::=$	$g\ \{\mid g\}$

Figure 1. Grammar for register transfers.

RTL must be representable by a single instruction on the target machine.

Our compilers, by contrast, require only that the front end produce well-typed RTLs in which no expression is wider than a machine word. The front end therefore needs to know almost nothing about the target machine: only the word size and possibly the byte order. It is reasonable to write *one* front end which emits RTLs that will run on *any* 32-bit machine; indeed, one of our colleagues has done so for the pedagogical language Tiger (Appel 1998; Govereau 2004). We have also written a single RTL emitter for the `lcc` C compiler (Fraser and Hanson 1995); we use this emitter for multiple targets by abstracting over byte order and over the `lcc` *metrics*, which give the size of each C type.

Given well-typed RTLs, we translate them into machine instructions using back ends that are generated automatically from SLED and λ-RTL descriptions. These descriptions know nothing about any compiler's intermediate code, and as noted above, they have been reused to generate a variety of other tools.

2.1 Representing code as register transfers

Our surface notation and our internal representation of register transfers are based on those of λ-RTL (Ramsey and Davidson 1998). Register transfers are composed primarily from *expressions*, *locations*, *assignments*, and *RTL operators*. Internally, each expression and location has a width, but normally these widths are inferred using a variation of Damas and Milner's (1982) algorithm. For readability, we therefore omit widths from the examples in this paper and from the simplified grammar in Figure 1.

An expression is either a literal constant, a fetch from a location, or the application of an RTL operator to sub-expressions. Applications of many binary operators can be written using infix notation. An RTL operator always represents a pure computation on bit vectors; our library of about 90 operators includes standard integer, logical, and IEEE floating-point operations. If these don't suffice to describe the semantics of an instruction, a machine description can introduce new operators.

A location is a cell in a *storage space*; every byte in memory and every register is modeled as a cell in a storage space. For convenience, locations can be named. An assignment computes the value of an expression and stores the result in a location. A guarded assignment evaluates a boolean expression called the *guard*, and the result is stored only if the guard is true. For ordinary instructions, the guard is trivially true and is not written; guarded assignments with nontrivial guards typically represent conditional-branch or predicated instructions.

A register-transfer list (RTL) is the parallel composition of multiple guarded assignments; it is like Dijkstra's (1976) multiple assignment except that each assignment has its own guard. RTLs are expressive enough that a single RTL can represent the input/output behavior of any machine instruction. For example, an add instruction might be represented by two trivially guarded assignments: one to store the sum and the other to modify condition codes.

Register-transfer lists have a well-defined semantics and are easily targeted by a front end. To make it easier for authors of front ends to emit RTLs, we provide a language called C--, which provides an ASCII syntax as a thin veneer over RTLs (Peyton Jones, Ramsey, and Reig 1999; Ramsey and Peyton Jones 2000). We have thereby reduced instruction selection to the following problem:

> Given a well-typed RTL expression e and a machine register rd of a suitable width, how can we put the value of e into register rd using only instructions available on the target machine?

Before getting into details, we show example solutions where e is the bitwise complement of a value in a register rs, or in RTL notation, `com($r[`rs`])`.

We might hope to find a single instruction with the semantics `$r[`rd`] := com($r[`rs`])`, but today's architectures lack two-register bitwise-complement instructions. There are, however, plenty of useful alternatives. On the PowerPC, for example, some of the logical instructions compute the complement of the second operand before computing the logical operation. The `orc` instruction computes the logical *or* of a register `$r[`rs_1`]` and the complement of another register `$r[`rs_2`]`, that is, `orc` is represented by the RTL `$r[`rd`] := or($r[`rs_1`], com($r[`rs_2`]))`. If we first store the value 0 in the register `$r[`rs_1`]`, the instruction will compute the of `or(0, com($r[`rs_2`]))`, which is equivalent to the complement of `$r[`rs_2`]`.

Another useful alternative, found on the $x86$, is an instruction that computes the bitwise complement of a value in a register and then places the result in the same register: `$r[`rd`] := com($r[`rd`])`. To compute the complement of some other register `$r[`rs`]`, we must first copy `$r[`rs`]` to `$r[`rd`]`, then execute the complement instruction.

These examples hint at our solution to the problem above: we first show that provided it is not too wide for the machine, any well-typed RTL expression can be placed into a register by a combination of simple RTLs we call *tiles*; the example assignment `$r[`rd`] := com($r[`rs`])` is one such tile. We then present an algorithm that is given a machine description and that gradually enlarges the set of RTLs it knows the machine can compute, stopping when the set contains all the tiles.

2.2 Reducing register transfers to tiles

Our decomposition of RTLs into tiles is a variation on the classic technique of code generation by tree covering. Classically, a tile associates a machine instruction with an intermediate-code expression that it computes; to select instructions for an intermediate-code tree, the compiler covers the tree with tiles, making sure that each source operand of each tile matches the destination where the corresponding sub-tile stores its result. An optimal cover can be found by a bottom-up rewriting system, also called BURS (Pelegrí-Llopart and Graham 1988; Aho, Ganapathi, and Tjiang 1989; Emmelmann, Schröer, and Landwehr 1989; Proebsting 1992; Fraser, Hanson, and Proebsting 1992). In many cases the input to the BURS engine can be generated by a "machine description;" however, such a description describes not the target machine but rather a set of subtrees of the compiler's intermediate code, each of which is associated with a sequence of instructions on the target machine. In such a description, knowledge of the machine and the compiler are inseparably interwoven, and so the description cannot be reused to build other tools.

Our new idea is that instead of choosing a machine-dependent set of tiles based on a particular target's instructions and addressing modes, we have identified a simple, machine-*independent* set of tiles that together can cover any well-typed RTL tree. To identify this set of tiles, we analyzed the grammar in Figure 1, paying particular attention to the RTL operators and their types. For example, almost all the integer and logical operators take two operands of width w and produce a result of width w. Each of these operators must be implemented by a tile of the form rd `:=` $rs_1 \oplus rs_2$. The integer-multiply operator, by contrast, takes operands of width w and produces a result of width $2w$, so its tile has a different form: it takes its operands in registers and puts its double-width result into a *pair* of registers. A unary operator like bitwise complement has yet another form: rd`:=` $\oplus$`(`rs`)`.

Given our large library of RTL operators, plus assignment and control transfer, there are dozens of tiles—too many to enumerate here. But the number of tiles is less important than the number of forms, which we call *shapes*. The shape of a tile is obtained by abstracting over RTL operators and over the widths of arguments and results; shapes are important because if there is already a tile of the appropriate shape, it is trivial to add a new tile or a new RTL operator to our system. We summarize the three different kinds of tiles we have identified, together with the number of shapes of each.

- A *data-movement tile* moves a constant into a location or moves a value between two locations. Shapes include register-register copies, loads from memory, and stores to memory. A data-movement tile can also move data between locations of different widths, so there is also a shape for sign-extending and zero-extending loads, as well as one for instructions that store low bits of a register in a memory location. Data-movement tiles come in a total of 13 shapes.
- A *computational tile* applies an RTL operator to constant or register operands and places a result in one or more registers. Shapes include all the examples above, as well as some others not mentioned, such as the shape used to compute the carry-out bit from an integer add with carry. Computational tiles come in a total of 7 shapes.
- A *control-transfer tile* changes the flow of control. Shapes include conditional and unconditional branches, indirect branches, direct and indirect calls, and returns. Control-transfer tiles come in a total of 9 shapes.

The totals above are for ordinary register-based machines; the floating-point stack of the x86 adds another dozen or so shapes.

The key property that distinguishes our work from previous work on code generation by tree-covering is that in our compiler, every back end uses the same set of tiles, no matter what the target machine. Our compiler covers intermediate-code trees using a single, reusable component called the *generic code expander*. The generic expander uses a simple maximal-munch strategy to reduce any well-typed RTL to a sequence of postexpander tiles, provided only that no intermediate result is too wide to fit in a machine register. For example, given the memory-to-memory move

```
$m[ESP + 8] := $m[ESP + 12]
```

the generic expander may generate a sequence of four tiles, each of which either moves a value or applies a single RTL operator:

```
$t[1]      := ESP + 12
$t[2]      := $m[$t[1]]
$t[3]      := ESP + 8
$m[$t[3]]  := $t[2]
```

This example shows why it is called an "expander": a single RTL is expanded into many simpler RTLs. The expansion factor is large because the tiles are small: each tile contains at most one RTL operator. The expanded code may look horribly inefficient, but in practice it is ideal for peephole optimization and other optimizations that take place after instruction selection (Davidson and Fraser 1980, 1984; Benitez and Davidson 1994).

BURS-based systems use the opposite strategy: optimization is done *before* instruction selection, tiles are as large as possible, and typical BURS engines use dynamic programming to find a minimum-cost tiling. We discuss tradeoffs in Section 6.2.

2.3 Reducing tiles to machine instructions

The set of shapes we use is defined in the machine-independent *postexpander interface*. For any particular target machine, the implementation of each tile is the job of the machine-dependent *postexpander*. In this paper, we show how to generate the postexpander automatically.

The question of whether a particular machine can implement all the tiles turns out to be undecidable. The question is related to the (also undecidable) question of whether two programs are equivalent (Section 4). Undecidability is part of what makes the problem difficult, but real machines are designed to provide the computational capabilities a compiler needs, so it is not shocking that there is an algorithm that works well in practice. Using a pruned search to guarantee termination (Section 4), we have generated postexpanders for the x86, the ARM, and the PowerPC. Generated postexpanders do not degrade the quality of the code produced by the compiler (Section 6.2.4).

3. Generating the postexpander

We generate a postexpander by searching a λ-RTL machine description for combinations of instructions that implement tiles. Before explaining how the search works, we give a quick overview of λ-RTL machine descriptions and the information they contain.

3.1 λ-RTL machine descriptions

Like its sister language SLED (Ramsey and Fernández 1997), λ-RTL models an instruction set using algebraic datatypes that represent the abstract syntax of instructions and operands (Ramsey and Davidson 1998). Each instruction or addressing mode gets a *constructor* that may be applied to operands. For example, the PowerPC's three-register add instruction has the abstract syntax `add(rd, rs1, rs2)`; the `add` constructor is applied to three operands. Operands may be simple bit vectors or may themselves be the result of applying constructors; for example, the x86's bitwise-complement instruction has the abstract syntax `not(Eaddr)`, where `Eaddr` stands for an operand created by applying one of the constructors of effective addresses.

λ-RTL's algebraic datatypes are equivalent to a grammar with named productions. A grammatical view makes it especially easy to explain how λ-RTL specifies semantics: a λ-RTL description determines an *attribute grammar* (Knuth 1968) in which the meaning of an instruction is a synthesized attribute in the form of an RTL. Each production in the grammar is associated with an equation that tells how to synthesize the result's attributes from the attributes of its operands. For example, the meaning of the PowerPC `add` instruction is given as follows:

```
default attribute
  add(rd, rs1, rs2) is $r[rd] := $r[rs1] + $r[rs2]
```

The `rd`, `rs1`, and `rs2` on the left-hand side are binding instances. These identifiers have been declared as 5-bit operands; in the language of attribute grammars, they would be "terminal symbols." When an instruction is generated by the postexpander, the values of these operands are given, so for example the abstract-syntax tree `add(1, 2, 3)` has the semantics `$r[1] := $r[2] + $r[3]`.

The notation $r[$r$] is an example of indexing into a λ-RTL *storage space*. A storage space is simply a part of the hardware from which bit vectors may be read or written; storage spaces typically include memory, general-purpose registers, floating-point registers, special-purpose registers, and so on. As shown in Figure 1, storage-space indexing is how an RTL refers to a location on the target machine.

Storage spaces are part of a λ-RTL specification, and each space has a single-character name. Names are arbitrary, but we conventionally use `r` for registers and `m` for memory; the example above is a three-register add. Moreover, λ-RTL can name locations within a storage space, making descriptions more readable; here's an example:

```
location ESP is $r[4]
```

When a machine has addressing modes, it is convenient to define a nonterminal symbol for each kind of addressing mode. Such a symbol will have its own productions in the grammar, and its attribute may be something other than an RTL. Here, for example, are some frequently used memory-addressing modes on the $x86$; the attribute is an expression, the value of which is a 32-bit address:

```
Indir   (reg)      : Mem is $r[reg]
Disp32  (d32, reg) : Mem is d32 + $r[reg]
Abs32   (a)        : Mem is a
```

`Mem` is the nonterminal symbol created for the in-memory addressing modes; `reg` is a 3-bit register index, `d32` is a 32-bit constant, and `a` is a 32-bit label. The three modes shown are indirect addressing through a register, displacement with an offset from a register, and absolute addressing with a label.

Another frequently used $x86$ addressing mode is `Eaddr`, which defines an effective address as either a fetch from memory or a fetch from a register. The attribute of an effective address is a location:

```
E (Mem) : Eaddr is $m[Mem]
Reg (r) : Eaddr is $r[r]
```

Once defined, these addressing modes can be used in descriptions of instructions; for example, here is the $x86$'s `not` instruction, which uses the RTL operator `com` ("bitwise complement"):

```
not(Eaddr) is Eaddr := com(Eaddr)
```

Our search algorithm synthesizes multiple RTLs for the `not` instruction: one for each possable `Eaddr` mode.

In summary, a λ-RTL description provides the following information:

- A grammar from which all instructions can be generated
- A list of all the locations on the machine
- Equations that show how to compute the observable effect of any instruction, represented as an RTL

3.2 Techniques for finding tiles

One way to try to generate a postexpander is to take each tile and look for a sequence of instructions that implements it; this is more or less what Massalin's (1987) superoptimizer does, for example (see also Granlund and Kenner 1992). Cattell (1980) also uses a goal-directed search strategy. Goal-directed search can be bounded by limiting the number of instructions one is willing to combine, but as explained in Section 4, we believe that the number of instructions is a poor metric. Instead, we use a *forward* search technique that uses two data structures:

- Our algorithm maintains a *pool* of RTLs, each of which is known to be implementable by a sequence of instructions on the target machine.[1] More precisely, we maintain a pool of RTL schemas which are parameterized over bit-vector operands. With each RTL in the pool, we associate a sequence of instructions that implements it. We initialize the pool by expanding every nonterminal symbol in the λ-RTL grammar and taking the resulting set of RTLs. Each RTL in the initial pool is associated with a sequence containing exactly one instruction.
- Our algorithm also maintains a set of unimplemented postexpander tiles. We initialize the set to include all tiles.

[1] Actually, we support implementation by an arbitrary control-flow graph, not just a sequence, but sequences suffice for almost all cases of interest, so in this paper we stick to sequences.

Search proceeds by making one of two steps:

1. Using RTLs in the pool, we construct a sequence of instructions that implements a new RTL, which we then add to the pool.
2. We find a way to implement a postexpander tile using an RTL from the pool; we then remove that tile from the set of unimplemented tiles.

We repeat one step or the other until there are no more unimplemented postexpander tiles. One difficulty is that there is no inherent limit to the size of the pool, so it is not obvious when to stop adding things in step 1. Section 4 describes how we choose RTLs to combine and how we ensure that we don't repeat step 1 indefinitely.

The implementation of step 1 looks for a sequence of RTLs in the pool, such that each RTL in the sequence (except the last) assigns to a location that is read by a later RTL. The sequence of RTLs computes an expression that is more complicated than the expressions used in the individual RTLs; we identify the expression computed by substituting forward for assigned locations.

By design, the postexpander tiles require simple expressions, not complicated ones. A sequence of RTLs that computes a complicated expression may be useful only if its expression is equivalent to a simpler expression that is useful for implementing postexpander tiles (as discussed in Section 4). If so, we add the implementation to the pool.

Although the implementation may sound complicated, it can be understood by considering three simple ideas:

- *Algebraic laws:* An algebraic law asserts that two expressions are equivalent. In particular, it can show that an implementation of a complicated expression may also serve as an implementation of a simpler, equivalent expression. For example, the algebraic law $or(0, e) = e$ is used to find the implementation of bitwise complement on the PowerPC (Section 2.1 above).
- *Compensation for extra assignments:* A sequence of instructions may have more than one assignment; to avoid introducing unwanted assignments in the program we are compiling, we have to compensate for them. For example, when we create a sequence of RTLs, each RTL may assign to distinct locations, and we usually want to implement only one of the assignments.
- *Data movement:* A recurring problem is to understand how to get the value of an expression from where it is computed to where it can be used. This ability is needed to implement the data-movement postexpander tiles and also to save and restore locations. In addition, by intercalating data movement between two RTLs that assign and read different locations, we may be able to add more useful RTLs to the pool.

We describe each of these techniques, then describe the implementation in the λ-RTL toolkit.

3.2.1 Algebraic laws and equivalence of expressions

Equivalent expressions are often syntactically identical, but in general we use algebraic laws to show equivalence. For example, the algebraic law `or(0, `e`)` = e helps show that the PowerPC's `orc` instruction can implement bitwise complement. Many laws are left or right identities of binary operators. We also use operator-inverse laws, such as `neg(neg(`e`))` = e. Both inverses and identities are examples of a larger class of laws in which an expression is shown to be equivalent to a more complicated expression involving itself. Sometimes the expression can appear more than once, as in the law `or((`e `>>` n`) <<` n`, zx(lobits`$_n$`(`e`)))` = e. This law is useful on RISC machines to show that we can load a 32-bit constant into a register by a sequence of load-upper-immediate and or-immediate instructions.

Another very useful kind of law shows how to implement a single operator in terms of other operators. These laws embody some of the kind of domain-specific knowledge presented by Warren (2003). For example, we can implement bitwise complement using integer negation and integer subtraction, or we can implement integer negation using bitwise complement and integer addition:

`com(`e`) = sub(neg(e), 1)`
`neg(`e`) = add(com(e), 1).`

Although dividing the laws into different kinds helps us understand what is going on, our algorithm treats every law in the same way: any expression that matches an algebraic law can be replaced with the equivalent expression specified by the law. In this way, we use algebraic laws to conclude that an implementation of one RTL in the RTL pool can also be used to implement an equivalent RTL. Because algebraic laws are used to add the most new RTLs to the pool, the details are closely connected to ensuring termination (Section 4).

3.2.2 Compensating for extra assignments

Many of the RTLs in the pool will contain unwanted assignments. Assignments to condition codes are almost always unwanted, because the postexpander tiling does not assume a machine has condition codes; computational tiles assign only to their result registers, and conditional-branch tiles test conditions directly. Another source of unwanted assignments is the combination of RTLs in sequence. Earlier assignments in a sequence are typically not mentioned in a tile. For example, the sequence

```
$r[rs1] := 0
$r[rd]  := $r[rs1] + $r[rs2]
```

is equivalent to the RTL

```
$r[rs1] := 0 | $r[rd] := $r[rs2]
```

This RTL would be a data-movement tile if not for the unwanted assignment to `$r[rs1]`.

If we find an RTL in the pool that is equivalent to an unimplemented postexpander tile plus an extra assignment to location l, we must either ensure that the assignment to l cannot be observed by the rest of the program, or we must compensate for the unwanted assignment by saving and restoring l. Saving and restoring means introducing a prologue of the form l' `:=` l and epilogue of the form l `:=` l', where l' is not mentioned in the main RTL.[2] But the assignment to l' is also unwanted, so it is essential that eventually we assign to some location that cannot be observed.

An assignment to a location l cannot be observed if we can guarantee that it never reaches an RTL that reads l. We provide the guarantee by making l a *fresh temporary* or a *scratch register*.

Our compiler provides an infinite supply of fresh temporaries, each of which is guaranteed to be distinct from every other location used in the program. If we use a fresh temporary on the left-hand side of an unwanted assignment, the assignment cannot be observed. In the example above, we can use temporary `$t[0]` instead of `$r[rs1]`:[3]

```
$t[0]   := 0
$r[rd]  := $t[0] + $r[rs2]
```

In a similar fashion, we can use a fresh stack slot to compensate for an extra assignment to a memory location.

There's nothing magical about fresh temporaries; an infinite supply of distinct locations is a standard abstraction—to map such locations to finite hardware, we use a register allocator. But not every hardware resource should be managed by a register allocator; for a unique resource like a condition-code register, we can save and restore, or we can deploy our other strategy: scratch registers.

A back end can take any set of hardware registers and make them unnameable in source code and unavailable to the register allocator. These registers then become available as *scratch registers*. Scratch registers may be used freely within the implementation of a single postexpander tile, provided that no scratch register is live in or live out at a tile. Together, these properties guarantee that an assignment to a scratch register will not affect the observable behavior of a program. Some care is still required: within the implementation of a single tile, the postexpander must store at most one value at a time in each scratch register.

An example of an ideal scratch register is the condition-code register. It is not exposed to source code, it cannot usefully be managed by the register allocator, and in the presence of conditional-branch instructions it would be awkward to save and restore: restore instructions would have to be inserted not only after the branch instruction but also at the branch target. Making the condition-code register a scratch register allows the postexpander generator to mutate it at will.

3.2.3 Data-movement graph

To implement the data-movement tiles, the postexpander must be able to move a value from any register to any other register as well as between registers and memory. Because data-movement tiles are also used to save and restore locations and to connect multiple RTLs in sequence, they play a key role in generating the postexpander. In general, as we discover implementations of new data-movement tiles in step 2, we create new opportunities to add useful RTLs to the pool in step 1. For example, to implement the postexpander tile for `carry`, on the $x86$ we first perform an addition instruction, then use a data-movement tile to move the carry-out bit from the condition-code register into a general-purpose register. We cannot find the implementation of `carry` until we have found a way to move bits from the condition-code register to a general-purpose register. To help find data-movement tiles, we have a special analysis and data structure.

To find data-movement tiles, we build a directed *data-movement graph*. Each node in the graph represents a set of locations on the machine, determined using the *location-set analysis* of Dias and Ramsey (2006). This analysis identifies locations that are interchangeable for use in some subset of instructions.[4] For example, memory locations are typically interchangeable because most instructions that refer to one memory location can refer to any memory location. As another example, most machines have a set of general-purpose registers that are interchangeable for most purposes, although some registers may play special roles in call instructions or multiply instructions. What is new in this paper is that we not only identify what locations are interchangeable, but we automatically find ways to move values among and between interchangeable locations. That is, we find instructions that move values from one register to another of the same kind, instructions that move values between registers of different kinds, and instructions that move values between registers and memory. On some platforms, values have to be moved indirectly, e.g., from integer registers to floating-point registers via memory. This is why we have a data-movement *graph*; to move a value from one location to another, it is sufficient that there is a path from one corresponding graph node to the other.

[2] In a program in which l is not live out, dead-assignment elimination will remove the save and restore instructions, even if they occur in a loop (Lerner, Grove, and Chambers 2002).

[3] Because register `$r[0]` is hardwired to 0 in many instructions, we could implement this assignment with just one instruction, but this example illustrates a common pattern. And after postexpansion, our peephole optimizer will combine these two instructions anyway.

[4] The results of the interchangeability analysis also play a key role in register allocation (Smith, Ramsey, and Holloway 2004). For example, if a machine has floating-point registers, they are almost certainly *not* interchangeable with integer registers, and the register allocator must distinguish floating-point temporaries from integer temporaries.

UseKnownAssignment

$$\frac{\langle l' := e, G\rangle \in pool \qquad \langle l := l', G'\rangle \in pool \qquad l \leftrightarrow G_{save_l}, G_{restore_l} \qquad l' \leftrightarrow G_{save_{l'}}, G_{restore_{l'}}}{(G_{save_l} \cdot G_{save_{l'}} \cdot G \cdot G'), l, G_{restore_{l'}} \cdot G_{restore_l} \Downarrow e}$$

ApplyAlgLaw

$$\frac{G_e, e, G'_e \Downarrow e_1 \qquad (e_1 = e_2) \in AlgLaws}{G_e, e, G'_e \Downarrow e_2}$$

Figure 2. Simplified versions of two inference rules used to find implementations of postexpander tiles.

Ideally the data-movement graph would be complete: able to move a value from any location on the machine to any other location of the same size. In practice, the data-movement graph usually lacks some edges involving fixed registers. For example, on the $x86$, only 8 bits of the 16-bit condition-code register can be moved to other locations. Luckily, these missing edges have little practical consequence because with just two exceptions, postexpander tiles mention only general-purpose registers.

3.2.4 Specifying and implementing the algorithm

We have described the techniques our algorithm uses to find implementations of new RTLs. Now we describe how the algorithm is specified and implemented.

We specify the algorithm using syntax-directed, nondeterministic inference rules. The main judgment form is $G, r, G' \Downarrow r'$ which states that the sequence of the graph G followed by the RTL r and graph G', executes assignments equivalent to those performed by the RTL r'. In other words, the sequence G, r, G' implements the RTL r'. Using this judgment, we can define step 1 and step 2 precisely. Step 1 must find a satisfying instance of the judgment $G, r, G' \Downarrow r'$, where G, r, and G' contain only RTLs from the pool, and r' is not in the pool. If we find a satisfying instance, we add the RTL r' to the pool. Similarly, step 2 must find a satisfying instance of the judgement $G, r, G' \Downarrow r'$, where G, r, and G' contain only RTLs from the pool, and r' is an RTL representing a postexpander tile. If we find a satisfying implementation, it is used to implement the postexpander tile.

Our definition of the judgment on RTLs relies on two related judgments. The first judgment is designed to compensate for extra assignments: $l \leftrightarrow G_{save_l}, G_{restore_l}$. The judgment states that the graph G_{save_l} saves the value in location l, and the graph $G_{restore_l}$ restores the value back to the location l. The second judgment is similar to the main judgment on RTLs: $G, e, G' \Downarrow e'$, which states that if we first execute the graph G, then evaluate the expression e, it will have a value equivalent to e'; furthermore, executing G' will undo any assignments made in G. In other words, the expression e is equivalent to e' if we first perform a setup graph G, then a clean-up graph G'. The clean-up graph usually restores the value of any location overwritten by an extra assignment in G.

Each judgment is defined by a set of syntax-directed, nondeterministic inference rules. Space limitations prevent us from including them all here, but we show simplified versions of two inference rules in Figure 2. The first example rule, UseKnownAssignment, formalizes the operation of forward substituting an assignment in the pool: If we execute a graph G assigning $l' := e$ and we can move the value to location l using a graph G', then an expression that fetches from the location l is equivalent to the expression e. This inference rule relies on two of our basic techniques: using data movement and compensating for extra assignments. By using the movement graph G', we allow the value to be used in any location l' reachable from l. And to avoid introducing extra assignments to locations l and l', the inference rule uses the $\leftrightarrow$ relation to find fix-up graphs that compensate for the extra assignments.

Another inference rule, ApplyAlgLaw, specifies our technique of using algebraic laws to conclude that one expression is equivalent to another. The rule's first premise is that the expression e can be used to implement e_1. (It may be that e_1 is identical to e, in which case the graphs G_e and $G_{e'}$ would be empty.) If there exists an algebraic law $e_1 = e_2$, we can then conclude that the expression e can also be used to implement the expression e_2.

Specifying the algorithm in terms of judgments defined using inference rules has several advantages:

- When expressed as a set of rules, a new idea or heuristic is easy to add and evaluate.
- Nondeterministic rules may combine in powerful ways.
- We can reason about the correctness of each rule in isolation.

Our implementations of step 1 and step 2 are similar to implementations of type inference and type checking, respectively. Our implementation of step 1 begins by choosing an RTL r as input from the pool; the goal is to infer an output RTL r' along with output graphs G and G' that satisfy the judgment $G, r, G' \Downarrow r'$. The implementation proceeds by structural induction on the input RTL r, attempting to construct a proof tree using syntax-directed inference rules that match the input RTL r. But the inference rules are non-deterministic, so there more be more than one possible proof tree; for example, we may be able to apply many different algebraic laws to conclude that an expression e is equivalent to many syntactically distinct expressions. If multiple rules may be applied, our algorithm applies each rule in turn and collects the results from all of them. For each satisfying sequence G, r, G', we add the equivalent RTL r' to the pool.

Our implementation of step 2 finds an implementation of a postexpander tile in a similar fashion, but this time, it is given an RTL r' representing the postexpander tile. Step 2 proceeds by choosing an RTL r from the pool, and trying to match it to r' to satisfy the judgment $G, r, G' \Downarrow r'$. Again, we proceed by finding inference rules matching both input RTLs, and if a satisfying implementation is found, it is used to implement the postexpander tile. If multiple implementations are found, we choose the cheapest one, according to a heuristic.

The most important optimization in our implementation is subsumption checking: if we have two RTLs r and r' in the pool, and we prove that we can use r to implement r', then we can remove r' from the pool. More formally, if we can prove the judgment $nop, r, nop \Downarrow r'$, then r is equivalent to r', and consequently, r' may be discarded. By discarding extra implementations, we reduce the number of RTLs we might consider using in rules such as UseKnownAssignment. Pleasantly, the subsumption checking is a direct application of the same code used to implement step 2.

Even with subsumption checking, we can always find implementations of new RTLs; in the next section, we discuss how we ensure that our algorithm terminates.

4. Ensuring termination

How are we to avoid adding RTLs to the pool indefinitely? We could limit the number or complexity of RTLs added to the pool, but then we might miss the implementations of some tiles. Unfortunately this tension cannot be resolved, because the underlying problem is undecidable: there is no terminating algorithm that is guaranteed to find an implementation of a postexpander tile if one exists. Our proof proceeds by reduction from the halting problem; we adapted well-known proofs of undecidability for strong normalization in term-rewriting systems (Huet and Lankford 1978; Klop 1992; Bezem, Klop, and Roel de Vrijer 2003). Because of space limitations, we do not discuss the proof, but the problem of finding a combination of instructions that is equivalent to a postexpander tile is closely related to the more general, undecidable problem of determining if two programs are equivalent.

Given the undecidability, our idea is to limit the number of RTLs added to the pool in such a way that on real machines, we have a good chance of finding all the postexpander tiles. We have developed a heuristic which meets this criterion as well as two additional criteria: the total number of RTLs added to the pool is small enough that the postexpander generator runs in minutes, not hours; and the instruction sequences corresponding to those RTLs are sequences that expose opportunities to the optimizer. Our criteria are not satisfied by standard heuristics that limit the size of the RTL, either by limiting the number of machine instructions combined to produce an RTL or by limiting the height of the abstract-syntax tree that represents an RTL.

Limiting the number of machine instructions is explicitly at odds with our strategy of producing simple code in the expander, then relying on the optimizer to improve the code. If we limited the number of machine instructions that could be combined, our algorithm would have to find a short, efficient sequence of instructions, taking advantage of complex instructions and complex addressing modes. Not only is it more difficult to generate a postexpander that uses the most efficient implementations, but it does not even result in better code. Such implementations, while ideal for a BURS tiling, are the worst possible input for a Davidson-style optimizer, which does best when fed long sequences of very simple instructions (Benitez and Davidson 1994). And without the help of the optimizer, we cannot expect our approach to generate better code than previous work on generating BURG-style instruction selectors, in which generated instruction selectors have produced significantly slower code than hand-written instruction selectors (Section 7).

Another standard heuristic limits the height of abstract-syntax trees, but in the presence of algebraic laws, the height of a tree is not a very good measure of whether it will help implement a postexpander tile. On some machines, we *need* high abstract syntax trees; for example, in order to load a 32-bit constant into a register on a typical RISC machine, we need to apply the algebraic law `or((`e` >> `n`) << `n`, zx(lobits`$_n$`(`e`))) = `e, which is applicable only to an abstract-syntax tree of height at least 5.

Another heuristic, which is commonly used in term rewriting, is to use algebraic laws to conclude only that an expression is equivalent to a *smaller* expression (Bezem, Klop, and Roel de Vrijer 2003). We use this techniqe to reduce the number of implementations we find, but in our domain, the technique by itself is not sufficient to guarantee termination. Even without using algebraic laws, we can produce larger expressions through forward substitution. For example, if we have an instruction that adds two registers `$r[`rd`] := $r[`rs_1`] + $r[`rs_2`]`, we can use two such instructions in sequence to compute a larger expression: the sum of three registers.

Unlike the standard heuristics, our novel heuristic is designed to indicate how likely it is that an RTL will be involved in an implementation of a postexpander tile: given an RTL, our heuristic predicts *the number of algebraic laws that may have to be applied* before the RTL may yield a postexpander tile. Our heuristic is founded in the observation that real machines are designed with compilers in mind: to compute a value on a real machine, you might have to put together a long sequence of instructions, but you probably won't have to do a whole lot of reasoning beyond forward substitution. Many interesting low-level computations can be implemented by sequences of such very simple instructions as shifting, logical operations, and arithmetic (Warren 2003). So instead of limiting the length of such sequences or the complexity of the expressions they compute, we limit the amount of *reasoning* that we expect to be necessary to show that a sequence of instructions is useful. We postulate that the more algebraic laws have to be applied, the less likely an implementation is to be used to implement a postexpander tile.

Because we cannot know ahead of time the actual number of algebraic laws we will apply, our heuristic makes an educated guess. Before adding an implementation of an RTL to the pool, we consider each expression e computed by each assignment in the RTL. Our heuristic works by covering the expression e with postexpander tiles and fragments of algebraic laws. A fragment of an algebraic law $e_1 = e_2$ is a subexpression of either e_1 or e_2. By counting the number of algebraic laws that partially match e, we estimate the number of algebraic laws that must be applied before we might find an implementation of a postexpander tile. We add an implementation to the pool only if we estimate that we might find the implementation of a postexpander tile by applying at most 3 algebraic laws. If more than 3 laws seem to be required, we discard the implementation. The number 3 is chosen empirically; on the ARM, PowerPC, and $x86$, applying 3 laws is enough to get from combinations of instructions to postexpander tiles.

5. Discussion

5.1 Generating postexpanders on many different machines

Because some instructions are easier to implement than others, the success of our postexpander generator is fundamentally affected by the choice of postexpander tiles. But different instructions are implemented more easily on different machines. To give some intuition about how our algorithm can generate instruction selectors for different machines, we answer two questions about the postexpander tiles:

- Is a single postexpander tile implementable on all machines?
- What if a postexpander tile is not implementable on a given machine?

First, we do not expect a single postexpander tile to be implementable on any machine. Instead, we classify machines into a very small number of groups based on how they move data and compute values. For example, some machines such as the x86 and the PowerPC support computation on registers: they move data between registers and apply operators to arguments in registers. Other machines support computation using a stack: the x86 floating-point unit maintains a stack discipline with floating-point values. We do not expect a postexpander tile describing floating-point addition on a register machine to be implementable on a machine that performs floating-point operations on a stack. Therefore, we identify a set of tiles once for each class of machines; currently, we have designed tiles for two ubiquitous classes of machines: register machines and stack machines. We may need a new set of tiles when we attempt to support a new class of machines, such as accumulator machines.

But even if we choose a postexpander tile for a particular class of machines, why do we expect the tile to be implementable on every machine in that class? The postexpander tiles are chosen carefully to implement only the most basic actions: movement between two locations on the machine, application of a single operator, and control flow. These actions are necessary to implement even simple programs written in a general-purpose language; therefore, we expect that any general-purpose machine to support a simple implementation of these instructions. Of course, we may need to do some reasoning to find an implementation, as we demonstrated in Section 3.

5.2 Intuition behind our pruning strategy

Because searching for implementations of postexpander tiles is an undecidable problem, we have to prune the search by discarding an implementation from the pool. The hard question is: how do we decide when to discard an implementation? One common metric measures the depth of the search that results in the implementation; in our application, we could count the number of algebraic laws we applied to arrive at the implementation. But this metric may discard an implementation that is only one algebraic law away from producing a postexpander tile.

A much better metric would be to count the number of algebraic laws we need to apply to arrive at an implementation of a postexpander tile. Of course, computing this number is also undecidable, so we settle for predicting the number of algebraic laws we may need to apply.

6. Evaluation

Our goal is to produce reliable compilers quickly. To evaluate how well we meet this goal, we evaluate our compiler in two steps: first an internal evaluation comparing the generated back end to the hand-written back end, then a comparative evaluation between our compiler and the portable compilers `lcc`, vpo, and gcc.

To evaluate the compilers, we rely on descriptions, examples, experiments, and line counts to assess how each compiler satisfies the following criteria:

- *How much effort is required to build one back end?* In particular, how complicated are the algorithms implemented in the back end? If machine descriptions are used, how complicated is a machine description, and how complicated are the tools that manipulate the machine descriptions?
- *How much effort is required to build multiple back ends?* After a single back end has been built, what is the marginal effort required to add a new back end?
- *How do we test a new back end?* After a back end is added to the compiler, how do we verify that it generates correct code?
- *How well does the compiler perform?* We evaluate the compile times of the compiler, as well as the run times of compiled programs. If the compiler requires pre-processing of machine descriptions, we also assess whether the compile-compile time is reasonable.
- *How much effort is required to add a new front end?* Specifically, what is the intermediate representation, and how well is the front end isolated from the code generator?

6.1 Internal Evaluation

Our internal evaluation compares the hand-written back ends in our compiler with our generated back ends. With the generated back ends, we write straightforward machine descriptions and automatically generate the complicated parts of the back end. Unlike previous work, our generated back ends perform as well as our hand-written back ends.

We begin by describing how we write back ends by hand, then we describe the generated back ends. After a performance analysis, we describe how our compiler supports the addition of new front ends, which is common to both hand-written back ends and generated back ends.

6.1.1 Hand-written back ends

The bulk of a hand-written back end is the code generator, which consists of two components: the postexpander and the recognizer. Because the postexpander implements only a small number of simple tiles, most of the postexpander code is short and simple. For example, on the x86 we implement the tile for subtraction using the following instructions, which also sets the condition codes:[5]

```
let sub dst op x y = rtl (R.par
  [R.store (R.reg dst)
     (R.app (R.opr ("sub", [32]))
        [R.fetch (R.reg x); R.fetch (R.reg y)]) 32;
   R.store (R.reg eflags)
     (R.app (R.opr ("x86_subflags", [32]))
        [R.fetch (R.reg x); R.fetch (R.reg y)]) 32])
```

The main pitfall in writing a postexpander is that it is easy to write code that usually works but may cause a bug in some corner cases. For example, our hand-written implementation of the x86's shift instruction failed to specify an assignment to the condition codes, which could allow the optimizer to generate incorrect code. Our hand-written back ends contained many of these types of errors.

The implementation of a hand-written recognizer is relatively simple because we rely on well-understood BURG technology (Fraser, Henry, and Proebsting 1992). Our implementation of the BURG engine is simple: it does not precompute state tables. The actual BURG specification has two parts: we write patterns to match the machine instructions, and we write code to convert RTLs to the input trees consumed by the BURG algorithm. The BURG pattern used to match the subtract instruction uses a constructor `Withaflags` to represent a binary operator that modifies the condition codes:

```
inst : Withaflags(dst:eaddrl, "sub", src:reg, w,
                                "x86_subflags")
  {: s "sub%s %s,%s" (suffix w) src dst :}
```

When the recognizer is invoked, the following pattern matches the subtract RTL and constructs the BURG input tree, using the `conWithaflags` constructor:

```
| RP.Rtl [(RP.Const(RP.Bool true),
            RP.Store(l, RP.App((o, _),
              [RP.Fetch(l',_); r]), w));
          (RP.Const(RP.Bool true),
            RP.Store(RP.Reg(('c', _, _),
                             flag_index, _),
              RP.App((fo, _), [RP.Fetch(l'',_);
                                 r']), _))
  ] when l =/ l' && l =/ l'' && r =// r'
                && flag_index = SS.indices.SS.cc ->
      conWithaflags (loc l) o (exp r) w fo
```

This code is not only verbose and tedious, but it is another possible source of tricky bugs: it is easy to write a pattern that does not check all the preconditions of an instruction. For example, if the pattern did not check that the first operand of the operation is also the destination, then the compiler could generate code using an invalid machine instruction. Fortunately, this class of bugs is usually detected by the assembler.

The other major part of our hand-written back end is the handling of calling conventions and stack layout. We write a concise, formal specification of each in a domain-specific language that is interpreted by our compiler (Lindig and Ramsey 2004; Olinsky, Lindig, and Ramsey 2006; Dias and Ramsey 2006). The specifications are interpreted by fairly straight-forward implementations of state machines. Writing a specification is no more complicated than understanding the conventions used by the target machine.

Once a single back end has been implemented, we can reuse the recognizer's BURG engine and the domain-specific languages used for calling conventions and stack layout. But parts of each additional back end must be written from scratch:

- The postexpander tiles

[5] The given code is an inlined version of the actual code in the compiler. Because most of the binary-operator instructions on the x86 are nearly identical, the actual compiler code is factored over the operator, allowing a shorter implementation of the operators.

- The recognizer's BURG patterns and the code that produces input trees for the BURG engine
- The calling-convention and stack-layout specifications

To test a back end, our hand-written back ends rely on end-to-end testing. Our test suite includes tests from our C front end, tests that have caused bugs in the past, and tests that exercise each RTL operator individually. We also have the mechanisms to generate tests for our calling conventions, which can be used to verify that our compiler generates code that is interoperable with code generated by an existing compiler on the target machine.

6.1.2 Generated back ends

Our goal in generating a back end is to produce reliable compilers quickly, not to push automatic generation to its extreme. Some code should be automatically generated; come code could be generated, but might not be worth the effort; and some code cannot even be generated in principle.

We generate the postexpander and recognizer because they are not only the most complicated parts of the back end but also the largest, comprising 67% of the back end, as measured in lines of code (see Table 1). The code in these components is also the most error-prone code in the back end: in the course of performing experiments with the generated back ends, we uncovered about half a dozen bugs in the hand-written versions.

The machine descriptions used to generate a back end are as straightforward as the target machine. We describe the x86 subtraction instruction from our earlier example with the following λ-RTL code:[6]

```
SUB^mr^od (Eaddr, reg) is
    Eaddr      := sub (Eaddr, reg)
  | Reg.EFLAGS := x86_subflags(Eaddr, reg)
```

Compared to the hand-written postexpander or recognizer, the machine description is short and simple: we encode the semantics of the machine instruction without concern for the data structures or algorithms used by the compiler. As a point of comparison, the SLED and λ-RTL machine descriptions use a total of 1,948 lines to describe 639 distinct instructions, whereas the hand-written postexpander and recognizer require 1,286 lines of code and can generate only 233 distinct instructions. By generating a recognizer that accepts more instructions, our generated back end allows the optimizer to produce better code: the optimizer only executes a transformation if the resulting RTLs are accepted by the recognizer.

Of course, the cost of implementing the λ-RTL toolkit must be considered: the algorithms described in this paper and in previous work (Dias and Ramsey 2006) are somewhat complicated. But they are implemented only once, and a proper implementation should avoid the bugs introduced in hand-written expanders and recognizers. In total, the algorithms related to generating the postexpander and the recognizer comprise thousands of lines of Standard ML in the λ-RTL toolkit. It is not clear whether we might be able to use our implementation of these algorithms to generate code for more than one compiler (Ramsey 2003).

The rest of a generated back end is unchanged from a hand-written back end. We could generate the list of available registers and their aliasing relationships. But because the code is short and hard to get wrong, we continue to write it by hand. Another 20% of the back end is glue code: this large amount of code reflects the number of people who have had their fingers in that part of the compiler, and it is far more than necessary. It might be possible to generate this code, but although the size of this code is embarrassing, writing it is neither difficult nor time-consuming.

[6] λ-RTL has several features that help the machine expert factor common patterns in machine instructions, such as binary operations that set the flags. Over multiple instructions, this factoring allows us to define several machine instructions in very few lines of code, resulting in a compact description of the target machine. But for the purpose of comparing various compilers, we have inlined the functions used in the λ-RTL description to highlight the description of our example instruction.

Back-end component	# lines
Postexpander	522
Recognizer	764
Stack layout	128
Calling conventions	73
Register specs	46
Calling convention glue	199
Back-end glue	179
Total	1,911

Table 1. Line counts for each component in the hand-written $x86$ back end.

The remaining 13% of the code consists of the specifications of calling conventions and stack layout. We cannot generate the code for the stack layout and calling conventions for the target's standard C calling convention because both are a matter of human convention; they are not properties of the machine. But considering that we support five different calling conventions on the $x86$, the 201 lines of formal specification are reasonably short.

Although the standard calling conventions cannot be generated, we could probably generate native (non-standard) calling conventions automatically. We could then automate calling-convention experiments that others have performed by hand (Davidson and Whalley 1991) and generate a calling convention that performs well on a benchmark of choice.

Once a single back end has been implemented, we can reuse the λ-RTL toolkit to generate postexpanders and recognizers for new target machines. Adding a new back end requires writing the following components:

- Machine descriptions for the target machine
- The calling-convention and stack-layout specifications

In addition to all the end-to-end tests used in our hand-written compiler, our generated compiler opens new possibilities for testing the code generator. We can test the SLED machine description to verify that it correctly describes the encodings of machine instructions (Fernández and Ramsey 1997); in future work, we hope to develop similar mechanisms to test λ-RTL descriptions. Furthermore, we are interested in applying techniques from certified compilation (Leroy 2006) to prove that the generated postexpanders produce correct code. We believe that the inference rules for finding implementations of postexpander tiles can form the basis of a proof strategy.

Our experiments demonstrate that the performance of our generated back ends is equivalent to performance of our hand-written back ends. To compile benchmarks written in C, we have added a new back end to the `lcc` compiler (Fraser and Hanson 1995); the back end emits C-- (Peyton Jones, Ramsey, and Reig 1999; Ramsey and Peyton Jones 2000). Because `lcc` compiles only programs written in ANSI C, we can use benchmarks only if they are written in ANSI C. We have chosen suitable benchmarks from the SPEC CPU95 and SPEC CPU2006 benchmark suites. For each benchmark, we measure the wall-clock running time, averaged over five runs. The CPU95 benchmarks show results on the reference set of inputs; for the CPU2006 benchmarks, we ran only the training set of inputs due to time constraints.

The experimental results in Table 2 verify that programs compiled by our automatically generated back end for the x86 run at least as fast as the programs compiled by the hand-written back end, The one exception is the vortex-95 benchmark, and the unnormalized results show that the difference in vortex-95 is less than three-tenths of one percent.

6.1.3 Front ends

Our compiler supports a variety of source languages by providing a flexible interface to front ends. The front end generates code in C--, which represents instructions using the full expressive power of the RTL language. With 90-odd operators, RTLs are capable of expressing a wide variety of machine-level computation, and new operators can be added easily.

Because RTLs describe instructions at the machine level, our intermediate language is not biased toward any particular source language. The flexibility of this approach is demonstrated by the variety of languages for which front ends have been written to target our compiler: C code with the `lcc` compiler, Standard ML with the MLton compiler (MLton 2005), Java with the Whirlwind compiler, and the pedagogical language Tiger with a front end written by a colleague (Govereau 2004).

6.2 Comparative Evaluation

We also compare our approach to generating back ends with three compilers that are designed to be portable: lcc, vpo, and gcc. lcc is a simple, non-optimizing compiler that uses a traditional strategy for code generation: the compiler chooses instructions, then immediately allocates registers and emits assembly code. vpo, on the other hand, is Davidson's highly optimizing compiler, which uses the code-generation strategy that we adopted in our compiler. gcc is a widely used, highly optimizing compiler that is also based on Davidson's code-generation strategy.[7]

Because the instruction selection is the bulk of the machine-dependent code in the back end, we focus our discussion on the implementation of instruction selection in each compiler. In each case, we find that, like our hand-written back end, each back end uses some form of pattern matching along with an additional set of predicates used to constrain the instructions matched. To make the issues in each compiler clear, we study how a subtraction instruction is emitted by the back end. The subtract instruction is useful for showing how a back end solves most of the problems in instruction selection without bringing in too much complexity. For example, the subtract instruction affects the condition codes on most machines, but it has few other complications. For some back ends, we need a more complicated instruction to highlight an interesting part of the code generator. In these cases, we look at how the back end handles a multiplication instruction.

6.2.1 lcc

The lcc compiler is a well-designed, portable C compiler that quickly generates reasonable, but mostly unoptimized, assembly code. The only optimization in the compiler is common subexpression elimination, and the compiler uses a local register allocator.

A back end in lcc provides two major pieces: type metrics that help the front end lay out data and a code generator to emits assembly code for the target machine. At the heart of the code generator is a BURG description mapping the compiler's intermediate representation to machine instructions. The advantages of using BURG are that it is easy to write a simple BURG engine (Fraser, Hanson, and Proebsting 1992), and it is easy to write BURG patterns that match the tree-based intermediate representation. For example, lcc uses the following BURG pattern to implement the subtraction node in the compiler's intermediate representation:

```
reg: SUBI4(reg,mrc) "?movl %0,%c\nsubl %1,%c\n" 1
```

Because the x86 subtract instruction `subl` places its result in the same register as its first source operand, `lcc` emits a `movl` instruction to copy the source operand `%0` to the destination `%c`.[8] In many cases, the mapping from the intermediate representation to machine instructions is straightforward.

But the BURG description does not provide all the information about the selection instruction: in some cases, the BURG code may generate an instruction that requires its operands to be place in specific registers. For example, the x86's multiple instruction requires one of the operands to be placed in the `EAX` register. To ensure that the register allocator places the operands in the proper locations, the back end also provides a function `target` to ensures that the operands of an operator are placed in suitable locations:

```
case MUL+U:
  setreg(p, quo);
  rtarget(p, 0, intreg[EAX]);
```

The output of the multiply instruction is the `EAX` register, which is defined elsewhere as `quo`, and the first input operand must be the `EAX` register.

As compilers go, an lcc back end is relatively simple, requiring only 939 lines of non-blank, non-comment lines for the x86 back end. But the simplicity comes with a cost: the compiler does not support optimization on the selected machine instructions. The BURG engine; promises only to produce locally optimal code, where optimality is defined by the patterns in the BURG description.

More significantly, the effort required to add a new back end to lcc includes the effort of understanding lcc's complicated interface between the front end and the back end. Writing the type metrics is easy, and most instructions in the BURG description are relatively straightforward. But to understand complicated cases, such as the x86 multiply instruction, we cannot reason locally about the BURG code. Instead, we have to understand that there is a separate `target` function that specifies where to allocate an operand in the intermediate code. Similarly, we have to know that the back end requires a `clobber` function to specify unintended side effects of the assembly code.

The lcc approach contrasts strongly with our approach of generating the code generator from declarative machine descriptions. Our generated compilers support optimization on machine instructions by following Davidson's code-generation strategy. And we achieve a separation of concerns by requiring only declarative descriptions of the target machine: we can describe the instructions on the target machine without understanding how the compiler will use the generated components.

Once a single back end has been implemented, the BURG engine can be reused. But each additional back end must be written from scratch, requiring these machine-dependent components:

- Type metrics
- BURG code mapping intermediate representation to machine instructions
- Functions constraining register use in instructions

To test a new back end, the lcc compiler relies on end-to-end testing. The compiler has been tested on the Plum-Hall Validation Suite for ANSI C compilers, in addition to the compiler's custom test suite, including previously reported bugs.

[7] gcc uses multiple intermediate representations on which some optimizations are performed, but the code generator continues to follow Davidson's strategy, complete with optimizations.

[8] The "?" prefix ensures that the move instruction is only emitted if the source and destination are different locations.

Our experiments demonstrate that most programs compiled by lcc are slower than when compiled by our generated back end (see Table 2); the one exception is the benchmark `vortex-95`. Although neither compiler performs much optimization, we hypothesize that the peephole optimizer in our generated compiler accounts for the superior speed of programs compiled with our compiler.

Although it is easy to add a new back end to lcc, it is not easy to add a new front end. The front end and the back end are tightly coupled: the interface consists of 19 functions with callbacks made in both directions across the interface. Data structures are also shared across the interface, most notably symbols and types. The intermediate representation uses directed acyclic graphs with a 36-operator language to represent instructions. The types and operators are precisely those needed for a C compiler, which may make it difficult to shoehorn another source language into the given intermediate representation. This design contrasts strongly with C--, which provides a broad range of operators and lets the front end make the decisions about data representation. Fraser and Hanson (1995) acknowledge these limitations while explaining that adding front ends easily was not a design goal for lcc, "changes to front end may affect back ends... This complication is less important for standardized languages like ANSI C because there will be few changes to the language."

6.2.2 vpo

The vpo compiler is a highly optimizing compiler designed to be ported easily to new target machines. With a collection of scalar and loop optimizations, vpo demonstrates that a Davidson-style compiler can produce extremely fast code (Benitez and Davidson 1988).

The main components of a back end in vpo are the expander and the recognizer. The expander is like a combination of our tiler and postexpander: it is a single machine-dependent component that takes an intermediate representation of the source program as input and returns semantically equivalent RTLs, each of which is representable by a single instruction on the target machine. For example, the following case expands a subtraction instruction in a manner very similar to our hand-written postexpander:[9]

```
case O_SUB:
  src2_size = get_location_size(src2);
  src1_size = get_location_size(src1);
  VPOi_rtl(sub(src1, Rtl_fetch(src2,src2_size*8),
               src1_size),
           VPOi_locSetBuild(EFLAG,src2,0));
  push_loc(src1);
```

The `VPOi_rtl` function outputs an RTL and a set of registers killed by the instruction. In this case, the condition-code register `EFLAG` is killed, as is the source operand. The RTL representing the subtract instruction is constructed by the `sub` function:

```
Rtl_ty_rtl
sub(Rtl_ty_loc reg_or_mem,
    reg_or_immInstance reg_or_imm, int nbytes) {
  Rtl_ty_expr expr;
  expr = Rtl_binary(Rtl_op_sub,
           Rtl_fetch(reg_or_mem,nbytes*8), reg_or_imm);
  return Rtl_assign(reg_or_mem, nbytes*8, expr);
}
```

As in our hand-written postexpander, the vpo expander constructs the familiar subtract instruction on the x86, which stores the result in the first source operand. To ensure that vpo's expander will treat `src1` as the location where the result is stored, the destination location was noted by the command `push_loc(src1)` in the earlier snippet:

As in our compiler, vpo's recognizer implements a predicate that decides whether an RTL is representable by a single instruction on the target machine; the recognizer can also return assembly code for the instruction. Because vpo represents an RTL as a string, vpo's recognizer uses a parsing engine, such as YACC, to match instructions. The implementation of the parsing engine requires some effort, but techniques for parser generation are well known, and existing implementations are ubiquitous. Once the parsing engine is written, the compiler writer effort writes a grammar that matches valid RTLs for the target machine. For example, the following YACC patterns match the x86 subtract instruction:

```
inst  : subi               { binst($1, SN);}
subi  : dst '=' rhssub ';' { $$ = binopi($1, $3);}
rhssub: src1 '-' src2
                   { $$ = brecord('-', $1, $3, AN);}
```

But these patterns also match a subtraction instruction in which the destination is not the same as the first source operand, which is not a valid instruction on the x86. The recognizer uses a library of functions to check the validity of the instructions that are parsed by the grammar.For example, the action of the `subi` pattern invokes the function `binopi` to check that, among other conditions, the destination matches the first source operand. Because a YACC pattern may match several instructions, the conditions in a recognizer function may be very complicated: the function `binopi` is 58 lines of code.

Overall, vpo's recognizer is very similar to our hand-written recognizer: both recognizers use matching technology to facilitate the task of writing instruction patterns, and both recognizers require a raft of complicated condition checking to ensure that the recognizer accepts only valid RTLs.

Like our hand-written back end, the vpo back end is significantly more complicated than a declarative machine description. In total, vpo's expander for the x86 consists of 922 non-blank, non-comment lines of code, and the recognizer consists of 1,840 lines. But more importantly, the code is complicated: the expander must select machine instructions for arbitrary source code, and the recognizer requires not only a YACC description to parse the RTLs but also a raft of helper functions to verify that the RTLs are actually valid on the target machine.

Aside from the parsing engine, there are no reusable parts in the code generator. For each additional back end, the expander and recognizer must be written by hand.

To test a new back end, vpo uses end-to-end testing with an extensive test suite. Additionally, the idea of specifying and generating tests for calling conventions was pioneered in vpo (Bailey and Davidson 2003).

This technical report lacks performance results for vpo.

In vpo, adding a new source language requires work proportional to the number of target machines we want to support. The front end in vpo is expected to convert the source program to vpo's intermediate representation: RTLs with the invariant that each RTL is representable by a single instruction on the target machine. The conversion is performed the expander, which therefore must be both specific to the source language and the target machine. The consequence is that support for a new language requires a separate expander for each target machine.

6.2.3 gcc

The gcc compiler is a widely used, highly optimizing compiler. An army of contributors have implemented a variety of scalar and loop optimizations, as well as man different back ends. The result is a compiler that generates good code on many different machines.

[9] An extra case for floating-point has been ellided to simplify the exposition.

The structure of gcc is a mixture of the canonical 3-part compiler and Davidson's code generation strategy. Each front end generates code in a (mostly) language-independent, machine-independent tree-based intermediate representation. The tree-based representation is converted to SSA form, and a number of optimizations are performed. Then, the code generator takes over.

The code generator is based on Davidson's structure of a compiler: the code is converted to a form of RTLs using an expander, and the machine invariant is maintained by a recognizer. But unlike vpo, in gcc the expander and recognizer are not written by hand. Instead, the compiler writer composes a "machine description," from which the expander and the recognizer are generated.

But gcc's machine descriptions are not like λ-RTL machine descriptions. A gcc machine description is a collection of definitions that are used to implement the expander, the recognizer, and sometimes parts of the optimizer. For example, the expansion of a subtraction instructions is defined as follows on the x86:

```
(define_expand "subsi3"
 [(parallel
   [(set (match_operand:SI 0 "nonimmediate_operand" "")
         (minus:SI
           (match_operand:SI 1 "nonimmediate_operand" "")
           (match_operand:SI 2 "general_operand" "")))
    (clobber (reg:CC FLAGS_REG))])]
  ""
  "ix86_expand_binary_operator (MINUS, SImode,
                                 operands);
   DONE;")
```

The first argument `"subsi3"` states that the code expands a subtraction of 4-byte integers. The second argument is an RTL with two effects: one performing the subtraction and the other clobbering the condition conde register. The careful reader will note that the RTL does not constrain the destination location to be the same as the first source operand. But strangely, it appears that this RTL is never used. Instead, the final argument of the `define_expand` definition evaluates commands before generating the expanded RTL. The `ix86_expand_binary_operator` function emits the correct RTL, with the destination register properly constrainted. Then, the `DONE` command ensures that the expander before generating the incorrect RTL given in the `define_expand` definition.

The recognizer uses a similar definition to describe machine instructions. The following definition defines a pattern (the first argument to `define_insn`) that matches the subtraction instruction:

```
(define_insn "*subsi_3"
 [(set (reg FLAGS_REG)
    (compare (match_operand:SI 1
                "nonimmediate_operand" "0,0")
      (match_operand:SI 2 "general_operand" "ri,rm")))
  (set (match_operand:SI 0 "nonimmediate_operand"
                         "=rm,r")
       (minus:SI (match_dup 1) (match_dup 2)))]
 "ix86_match_ccmode (insn, CCmode)
  && ix86_binary_operator_ok (MINUS, SImode, operands)"
 "sub{l}\t{%2, %0|%0, %2}"
 [(set_attr "type" "alu")
  (set_attr "mode" "SI")])
```

Much like our hand-written expander and vpo, gcc defines predicates to verify that the matched RTL is really representable by a valid machine instruction; for example, the `ix86_binary_operator_ok` predicate verifies that the destination of the RTL is the same as the first source operand.

In total, the machine description for the x86 defines 754 instructions, 320 expansions, and 222 optimization opportunities. But the machine description is 18,462 non-blank, non-comment lines of code, which is an order of magnitued longer than the λ-RTL machine description.

The code that generates the expander and recognizer is complicated, but it is written only once, then reused with all the compiler's back ends. The code to generate an expander is 727 non-comment, non-blank lines of C code; the code to generate a recognizer is 2,289 lines of C code.

Once a single back end has been implemented, the additional effort required to add a new code generator is writing a new gcc machine description:

- Definitions of the machine instructions
- Definitions for the expander

To test a new back end, gcc relies on end-to-end testing, using an extensive test suite.

Our experiments demonstrate that our generated compilers produce code comparable to code produced by gcc with little optimization. When gcc compiles the benchmarks with optimization turned off (`gcc -O0`), our generated back ends produce faster code for most of the benchmarks, with the exception of the `vortex-95` benchmark (see Table 2). But when gcc uses a variety of scalar optimizations (`gcc -O1`), it produces code that is 1.7% to 29.5% faster than code produced by our back end. Until we have implemented a set of optimizations in our compiler, we can not expect it to compete with an optimizing compiler such as gcc.

In addition to supporting a large number of back ends, gcc can support a new front end through an interface that is only partially decoupled from the rest of the compiler. The front end is resposible for converting a source program into code in a language-independent, machine-independent tree-based intermediate representation. gcc then translates the tree-based intermediate representation into another internal representation called `GIMPLE`. Unlike C--, the tree-based intermediate representation was designed for C programs, making it well-suited to procedural languages but not necessarily other languages. If a front end can not compile a language feature into the tree-based representation, the compiler writer has one recourse: he can describe the feature using a language-dependent extension to the tree-based representation and provide callback functions to convert the extension into `GIMPLE`. In short, if the tree-based representation is not expressive enough to capture the semantics of the source language, the writer of the front end can use a mechanism to extend the early phases of the compiler. Despite the effort that may be involved in adding a front end for a non-procedural language, gcc supports front ends not only for C, C++, and Java, but also for the object-oriented language Java and the hybrid logic-functional language Mercury.

6.2.4 Experimental results

To demonstrate that an automatically generated back end produces code that is as good as a hand-written back end, we compare our hand-written back end for the $x86$ with a generated back end.[10] To compile C programs, we have added a new back end to the `lcc` compiler (Fraser and Hanson 1995); the back end emits C-- (Peyton Jones, Ramsey, and Reig 1999; Ramsey and Peyton Jones 2000). Because `lcc` compiles only programs written in ANSI C, we can use benchmarks only if they are written in ANSI C. We have chosen suitable benchmarks from the SPEC CPU95 and SPEC CPU2006 benchmark suites. For each benchmark, we measure the wall-clock running time, averaged over five runs. The CPU95 benchmarks show results on the reference set of inputs; for the CPU2006 benchmarks, we ran only the training set of inputs due to time constraints.

[10] We ran out of time to do the same experiments on the PowerPC and ARM.

To show that our hand-written and generated back ends both produce reasonable code, we also compare them with `lcc`'s native back end and with `gcc`. `lcc`'s only optimization is common-subexpression elimination, but it does use BURG to perform optimal local instruction selection. With `gcc`, we use two different compilations, setting the optimization flags to `-O0` and `-O1`: the first option compiles the program without optimization, whereas the second option uses a variety of scalar optimizations. Because of the immaturity of our optimizer, we expect our code to be competitive only with `gcc -O0`.

The experimental results in Table 2 validate our hypotheses. Except for the vortex-95 benchmark, programs compiled by the automatically generated back end run at least as fast as the programs compiled by the hand-written back end, and the unnormalized results show that the difference in vortex-95 is less than three-tenths of one percent. In the other benchmark programs, both versions of our compiler outperform `lcc` and `gcc -O0`, presumably because of our peephole optimizer. As expected, the greater number of optimizations performed by `gcc -O1` produce code that is 1.7% to 29.5% faster than our generated back end.

7. Related Work

People have been working on retargetable compilers since 1958 (Strong et al. 1958). Instead of surveying all that work, we discuss three kinds of related work: support for front ends in compilers sharing the same code-generation strategy, machine-description languages, and compiler back ends generated from machine descriptions. We focus on the compilers.

7.1 Support for front ends in Davidson-style compilers

Davidson's code-generation strategy of choosing machine instructions in the expander, then using a machine-independent optimizer to improve the machine instructions is used both by Davidson's highly-optimizing compiler vpo and the widely used, optimizing compiler `gcc`, as well as by our compiler. But each of the three compilers has a different interface between the front end of the compiler and the back end, which affects how easy it is to add support for a new source language.

Davidson's vpo uses RTLs as the intermediate code input to the back end, with the restriction that each RTL must be representable by a single instruction on the target machine. The front end is responsible for choosing instructions on the target machine by implementing an expander. The drawback of this approach is that a new front end is required for each combination of source language and target machine.

This drawback is avoided by `gcc`, in which each front end in generates code in a single machine-independent, tree-based intermediate language, which is later translated to machine-dependent RTLs. This approach reduces the cost of adding a new front end, but the savings comes with a cost: it may be difficult to map the semantics of a new language onto the tree-based representation.

In our compiler, each front end generates RTLs with almost no restrictions: the only constraints are that the front end may not use operations that are too wide for the target machine,[11] so things like 73-bit addition are ruled out, and the generated RTLs must use the same address size and byte order as the target machine. In practice, most machines use either 32-bit or 64-bit addresses and either little-endian or big-endian byte order, so it is not difficult to parameterize a single front end over the target word size and byte order; we have done so with `lcc`'s front end, for example.

The most significant benefit our approach is that the author of the front end has access to the full expressive power of the RTL language. With its 90-odd operators, this language is capable of expressing a wide variety of machine-level computations—and new operators can be added easily. The flexibility of this approach is demonstrated by the variety of languages for which front ends have been written to target our compiler: C code with the `lcc` compiler, Standard ML with the MLton compiler (MLton 2005), Java with the Whirlwind compiler, and the pedagogical language Tiger with a front end written by a colleague (Govereau 2004).

[11] Redwine and Ramsey (2004) present an efficient algorithm for dealing with operations that are too narrow.

7.2 Machine descriptions

There has been a lot of work on machine descriptions, but not all machine descriptions are the same. A declarative machine-description language such as λ-RTL describes a property of the machine independent of any tool. Machine descriptions such as `nML` and `LISA` have similar goals to λ-RTL, but the language designs are very different (Ramsey and Davidson 1998). Each of these machine descriptions has been particularly successful in the embedded communities, where a description of an architecture is used in the design of processors.

In some cases, the term machine description is used to describe a mapping from the compiler's data structures to machine instructions. For example, a BURG specification and a `gcc` machine description specify how to convert the compiler's intermediate code to machine instructions. In some cases, the compiler-specific information is placed inside a larger architecture description language, which may include other properties of the machine, such as functional units (Farfeleder et al. 2006). These descriptions can be very useful for producing an efficient code generator, but they are not reusable.

As a point of comparison, we contrast our declarative machine descriptions of the $x86$ with the machine description used by gcc. The λ-RTL description is 1,160 non-blank, non-comment lines, and the SLED description is 788 non-blank, non-comment lines. The `gcc` machine description for the $x86$ is an order of magnitude larger: 18,462 non-blank, non-comment lines of code.

But the benefits of a declarative machine description are not limited to reducing the amount of code we write by hand: it also reduces the complexity of the code we write. The author of a declarative machine description can focus solely on the task of describing the semantics of the target instruction set. Furthermore, declarative machine descriptions can be tested independent of the compiler. Previous work showed how to test the correctness of a SLED description (Fernández and Ramsey 1997); in future work, we hope to develop similar mechanisms for λ-RTL descriptions.

On the other hand, the writer of a hand-written back end must not only understand the semantics of the target instruction set, but also the semantics of the compiler's intermediate code. He must then define a mapping from one semantics (the intermediate code) onto another (the machine instructions). And of course, the mapping must be written in terms of the compiler's data structures, and must satisfy the compiler's invariants. In the case of a `gcc` machine description, the compiler writer defines a machine description using a collection of compiler-specific macros with the occasional snippet of compiler-specific C code. The macros are used to define machine-dependent components of the compiler, including the expander and the recognizer. Because the description is defined in terms of the compiler, it cannot be reused for other applications.

The observant reader may have noticed that our machine descriptions are longer than the hand-written postexpander and recognizer. But it must be noted that the machine descriptions describe 639 instructions on the $x86$; the hand-written components, on the other hand, can use only 233 instructions. By generating a recognizer that accepts more instructions, we allow the optimizer to produce better code: the optimizer can only execute a transforma-

Benchmark	lcc	Hand / lcc	Generated / lcc	`gcc` -O0 / lcc	`gcc` -O1 / lcc
compress-95	43.32 s	0.84	0.80	1.01	0.69
go-95	25.84 s	0.98	0.88	1.01	0.61
vortex-95	55.84 s	1.09	1.10	1.05	0.77
mcf-2006	230.24 s	0.94	0.92	0.97	0.90
bzip2-2006	316.06 s	0.91	0.90	1.06	0.68

Table 2. Running times for compiled benchmarks: We use `lcc` as the baseline and give run times in seconds; other compilers are normalized to `lcc`. The Hand and Generated columns represent our compiler using hand-written and automatically generated x86 back ends. Benchmarks were run on an AMD Athlon MP 2800+ with 2 GB of memory.

tion if the resulting machine code is accepted by the recognizer.[12] Furthermore, because a declarative machine-description language is independent of any particular tool, it can be reused to generate components for many different applications, such as binary translators (Cifuentes, Lewis, and Ung 2002), linkers (Fernández 1995), debuggers, and compilers. The effort of writing a declarative machine description can be amortized over all of these applications.

7.3 Generating code generators from machine descriptions

The most closely related work on generating a back end was conducted over 20 years ago by Cattell (1980). Cattell presents ideas that are similar to our work, including the techniques of using algebraic laws and compensating for extra assignments. Unlike our search algorithm, Cattell's search for implementations starts from the goal and uses a means-ends analysis to search for machine instructions. At each step of the search, Cattell's algorithm looks at the instruction it is trying to implement and rewrites the instruction using an algebraic law. A heuristic chooses which algebraic law to apply by estimating which law will produce an instruction that is closer, in some sense, to machine instructions. He prunes his search using a depth limit that caps the number of algebraic laws that may be applied. Like other work on generating a back end, Cattell works with a compiler where instruction selection is one of the final phases, so instead of being able to generate naïve code and improve it later, Cattell's automatically generated instruction selector had to generate good code. Unfortunately, experimental results were not published.

Recent work with `LISA` discussed how to generate a BURG-style instruction selector using a LISA machine description (Ceng et al. 2005). The code produced by the generated back end ran about 5% slower than the code produced by a hand-written back end. This slow-down is indicative of the difficulty of generating a back end when the instruction selector is one of the last phases of the compiler: inefficient code produced by the instruction selector results in an inefficient program. Our novel approach, which builds on Davidson and Fraser's compilation strategy, allows the optimizer to clean up the inefficient code produced by our expander.

8. Future work

We have shown how to generate the back end of an optimizing compiler using declarative machine descriptions written using λ-RTL. But λ-RTL is not the last word in machine descriptions: we have no automated techniques to check the correctness of λ-RTL descriptions, and our work with λ-RTL has exposed flaws in its design.

In future work, we plan to check the correctness of λ-RTL machine descriptions, which would further improve the reliability of a generated back end. Furthermore, we are interested in applying techniques from certified compilation (Leroy 2006) to prove that the generated postexpanders produce correct code, assuming that the λ-RTL description is correct.

Generating tools from machine descriptions is a common strategy, but an important problem that is still unsolved is how to write a single, declarative machine description that can be used to generate a complete toolchain. A significant reason that this problem remains unsolved is that different tools view the machine at different levels of abstraction, and a machine-description framework will not be adopted unless it permits each user to omit descriptions at levels of abstraction he doesn't care about. A flaw we have identified in λ-RTL is that it works best at the level of abstraction used by compiler writers.

A compiler does not care which bits of the condition code register are modified by a subtraction instruction; it cares only that the mutation of the condition codes can be used to implement a conditional branch. Therefore, when we write a machine description with the compiler in mind, we follow a standard compiler technique of defining an operator to abstract over the mutation of the individual bits in the condition codes. In fact, to our compiler, *all* RTL operators are abstract; the only thing the compiler knows about the semantics of the operators is that the algebraic laws hold.

The designer of a bit-level simulator, on the other hand, needs to know precisely which bits of the condition-code register are modified, which means the bit-level semantics of each RTL operator must be specified. Unfortunately, λ-RTL cannot even express the idea that an RTL operator has a particular semantics; to get a lower level of abstraction, one must replace the operator with a function giving its semantics. This limitation militates toward creating multiple versions of each λ-RTL description, each to be used to generate a different tool—and outcome we wish most strongly to avoid. Because our long-term goal is to use *exactly* the same machine description to generate different tools, the problem of supporting both low-level semantics and higher-level abstractions in a single description is important for future work.

A closely related problem is that a family of machine architectures may include several versions, each of which is very similar to the rest. We would like to find ways to use a single description to describe not just one version, but the entire family, as do Hoover and Zadeck (1996).

[12] To focus our experiments solely on the quality of the generated postexpander, our experiments using the hand-written postexpander were run with the machine-generated recognizer.

References

Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. 1989 (October). Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11 (4):491–516.

Andrew W. Appel. 1998. *Modern Compiler Implementation*. Cambridge University Press, Cambridge, UK. Available in three editions: C, Java, and ML.

Joel Auslander, Matthai Philipose, Craig Chambers, Susan Eggers, and Brian Bershad. 1996 (May). Fast, effective dynamic compilation. *Proceedings of the ACM SIGPLAN '96 Conference on Programming*

Language Design and Implementation, in *SIGPLAN Notices*, 31(5):149–159.

Mark W. Bailey and Jack W. Davidson. 2003 (November). Automatic detection and diagnosis of faults in generated code for procedure calls. *IEEE Transactions on Software Engineering*, 29(11):1031–1042.

Manuel E. Benitez and Jack W. Davidson. 1988 (July). A portable global optimizer and linker. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 23(7):329–338.

Manuel E. Benitez and Jack W. Davidson. 1994 (March). The advantages of machine-dependent global optimization. In *Programming Languages and System Architectures*, *LNCS* volume 782, pages 105–124. Springer Verlag.

Mark Bezem, Jan Willem Klop, and eds. Roel de Vrijer. 2003. *Term Rewriting Systems*. Cambridge University Press, Cambridge, UK.

Roderic G. G. Cattell. 1980 (April). Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems*, 2(2):173–190.

Jianjiang Ceng, Manuel Hohenauer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Gunnar Braun. 2005. C compiler retargeting based on instruction semantics models. In *DATE*, pages 1150–1155. IEEE Computer Society. ISBN 0-7695-2288-2. URL `http://doi.ieeecomputersociety.org/10.1109/DATE.2005.88`.

Cristina Cifuentes, Brian Lewis, and David Ung. 2002. Walkabout — a retargetable dynamic binary translation framework. Technical report, Sun Microsystems.

Cristina Cifuentes, Mike Van Emmerik, and Norman Ramsey. 1999 (October). The design of a resourceable and retargetable binary translator. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE'99)*, pages 280–291. IEEE CS Press. URL `http://www.eecs.harvard.edu/~nr/pubs/wcre99.ps`.

Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–12, New York. ACM Press.

Jack W. Davidson and Christopher W. Fraser. 1980 (April). The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202.

Jack W. Davidson and Christopher W. Fraser. 1984 (October). Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526.

Jack W. Davidson and David B. Whalley. 1991. Methods for saving and restoring register values across function calls. *Software—Practice & Experience*, 21(2):149–165. URL `citeseer.nj.nec.com/davidson91methods.html`.

João Dias and Norman Ramsey. 2006 (March). Converting intermediate code to assembly code using declarative machine descriptions. In *15th International Conference on Compiler Construction (CC 2006)*, *LNCS* volume 3923, pages 217–231.

Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ.

Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. 1989 (July). BEG—a generator for efficient back ends. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 24(7):227–237.

Stefan Farfeleder, Andreas Krall, Edwin Steiner, and Florian Brandner. 2006. Effective compiler generation by architecture description. In Mary Jane Irwin and Koen De Bosschere, editors, *LCTES*, pages 145–152. ACM. ISBN 1-59593-362-X. URL `http://doi.acm.org/10.1145/1134650.1134671`.

Mary F. Fernández and Norman Ramsey. 1997 (May). Automatic checking of instruction specifications. In *Proceedings of the International Conference on Software Engineering*, pages 326–336.

Mary F. Fernández. 1995 (June). Simple and effective link-time optimization of Modula-3 programs. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 30(6):103–115.

Christopher W. Fraser and David R. Hanson. 1995. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, Redwood City, CA.

Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. 1992 (September). Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226.

Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. 1992 (April). BURG—fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76.

Paul Govereau. 2004. A Tiger compiler front end for `C--`. Presented at a tutorial at PLDI 2004 and available for download at `http://www.eecs.harvard.edu/~govereau/tigerc/`.

Torbjoern Granlund and Richard Kenner. 1992. Eliminating branches using a superoptimizer and the GNU C compiler. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 27(7):341–352.

Roger Hoover and Kenneth Zadeck. 1996. Generating machine specific optimizing compilers. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 219–229, St. Petersburg Beach, FL.

Gérard Huet and Dallas S. Lankford. 1978. On the uniform halting problem for term rewriting systems. Technical report, INRIA.

Jan Willem Klop. 1992. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press.

Donald E. Knuth. 1968 (Jun). Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145. Errata: volume 5, number 1 (1971), 95-96.

Sorin Lerner, David Grove, and Craig Chambers. 2002 (January). Composing dataflow analyses and transformations. *Conference Record of the 29th Annual ACM Symposium on Principles of Programming Languages,* in *SIGPLAN Notices*, 31(1):270–282.

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Conference Record of the 33rd Annual ACM Symposium on Principles of Programming Languages*, pages 42–54. ACM. ISBN 1-59593-027-2. URL `http://doi.acm.org/10.1145/1111037.1111042`.

Christian Lindig and Norman Ramsey. 2004 (April). Declarative composition of stack frames. In *13th International Conference on Compiler Construction (CC 2004)*, *LNCS* volume 2985, pages 298–312.

Henry Massalin. 1987 (October). Superoptimizer: A look at the smallest program. *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS II),* in *SIGPLAN Notices*, 22(10):122–127.

MLton. 2005. The MLton SML compiler. See `http://mlton.org`.

Reuben Olinsky, Christian Lindig, and Norman Ramsey. 2006 (January). Staged allocation: A compositional technique for specifying and implementing procedure calling conventions. In *Proceedings of the 33rd ACM Symposium on the Principles of Programming Languages*, pages 409–421.

Eduardo Pelegrí-Llopart and Susan L. Graham. 1988 (January). Optimal code generation for expression trees: An application of BURS theory. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 294–308, San Diego, CA.

Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. 1999 (September). `C--`: A portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, *LNCS* volume 1702, pages 1–28. Springer Verlag.

Todd A. Proebsting. 1992 (June). Simple and efficient BURS table generation. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 27 (7):331–340.

Norman Ramsey. 2003 (May). Pragmatic aspects of reusable program generators. *Journal of Functional Programming*, 13(3):601–646. A preliminary version of this paper appeared in *Semantics, Application,* *and Implementation of Program Generation*, LNCS 1924, pages 149–171.

Norman Ramsey and Jack W. Davidson. 1998 (June). Machine descriptions to build tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, *LNCS* volume 1474, pages 172–188. Springer Verlag.

Norman Ramsey and Mary F. Fernández. 1995 (January). The New Jersey Machine-Code Toolkit. In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, New Orleans, LA.

Norman Ramsey and Mary F. Fernández. 1997 (May). Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524.

Norman Ramsey and Simon L. Peyton Jones. 2000 (May). A single intermediate language that supports multiple implementations of exceptions. *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 35 (5):285–298.

Kevin Redwine and Norman Ramsey. 2004 (April). Widening integer arithmetic. In *13th International Conference on Compiler Construction (CC 2004)*, *LNCS* volume 2985, pages 232–249.

Michael D. Smith, Norman Ramsey, and Glenn Holloway. 2004 (June). A generalized algorithm for graph-coloring register allocation. *ACM SIGPLAN '04 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 39(6):277–288.

J. Strong, J. H. Wegstein, A. Tritter, J. Olsztyn, Owen R. Mock, and T. Steel. 1958. The problem of programming communication with changing machines A proposed solution (Part 2). *Communications of the ACM*, 1(9):9–16. ISSN 0001-0782.

Henry S. Warren. 2003. *Hacker's Delight*. Addison-Wesley.