



Automated Translation: Generating a Code Generator

Citation

Feigenbaum, Lee D. 2001. Automated Translation: Generating a Code Generator. Harvard Computer Science Technical Report TR-12-01.

Permanent link

<http://nrs.harvard.edu/urn-3:HUL.InstRepos:25104998>

Terms of Use

This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at <http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA>

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. [Submit a story](#).

[Accessibility](#)

Automated translation: generating a code generator

A Thesis presented

by

Lee D. Feigenbaum

to

Computer Science

in partial fulfillment of the honors requirement

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 2, 2001

Abstract

A key problem in retargeting a compiler is to map the compiler’s intermediate representation to the target machine’s instruction set.

One method to write such a mapping is to use grammar-like rules to relate a tree-based intermediate representation with an instruction set. A dynamic-programming algorithm finds the least costly instructions to cover a given tree. Work in this family includes BURG, BEG, and twig. The other method, utilized by gcc and VPO, uses a hand-written “code expander” which expands intermediate representation into naïve code. The naïve code is improved via machine-independent optimizations while maintaining it as a sequence of machine instructions. Because they are inextricably linked to a compiler’s intermediate representation, neither of these mappings can be reused for anything other than retargeting one specific compiler.

λ -RTL is a language for specifying the semantics of an instruction set independent of any particular intermediate representation. We analyze the properties of a machine from its λ -RTL description, then automatically derive the necessary mapping to a target architecture. By separating such analysis from compilers’ intermediate representations, λ -RTL in conjunction with our work allows a single machine description to be used to build multiple compilers, along with other tools such as debuggers or emulators.

Our analysis categorizes a machine’s storage locations as special registers, general-purpose registers, or memory. We construct a data-movement graph by determining the most efficient way to move arbitrary values between locations. We use this information at compile time to determine which temporary locations to use for intermediate results of large computations.

To derive a mapping from an intermediate representation to a target machine, we first assume a compiler-dependent translation from the intermediate representation to register-transfer lists. We discover at compile-time how to translate these register-transfer lists to machine code and also *which* register-transfer lists we can translate. To do this, we observe that values are either constants, fetched from locations, or the results of applying operators to values. Our data-movement graph covers constants and fetched values, while operators require an appropriate instruction to perform the effect of the operator. We search through an instruction set discovering instructions to implement operators via the use of algebraic identities, inverses, and rewrite laws and the introduction of unwanted side effects.

Contents

1	Introduction and Overview	5
2	Background	13
2.1	Machine State	13
2.2	Register-transfer Lists	13
2.3	λ -RTL	16
2.4	Tiny Machine	17
3	Spaces and Location Sets	19
3.1	Motivation	19
3.2	Spaces: Analysis	20
3.3	Spaces: Implementation	24
3.4	Location Sets: Analysis	25
3.5	Location Sets: Implementation	29
4	Choosing Locations for Variables	31
4.1	Motivation	31
4.2	Analysis	32
4.3	Implementation	34
4.4	Choosing Variables' Locations on the Tiny Machine	39
5	From RTL to Machine Code	41
5.1	Motivation	41
5.2	Analysis	42
5.3	Operators: Implementation	47
5.4	BURG Rules: Implementation	57
5.5	Code Generation for the Tiny Machine	65
6	Related Work	69
7	Summary and Conclusions	73
7.1	Summary	73
7.2	Reduction of Full RTLs	73
7.3	Questions Raised	75
7.4	Conclusions	76
	References	81

Chapter 1

Introduction and Overview

A vast array of commercial and academic low-level tools involve the semantics of instruction sets. Emulators, debuggers, profilers, and binary translators are but a few examples of tools involving machine-independent algorithms that must be able to operate on machine-level and often machine-specific code. Perhaps the most basic and widespread of tools in this category is the compiler, charged with generating machine instructions for a particular architecture from a high-level source language.

Particularly in academia, where compilers are used to study many engineering and algorithmic problems and where new hypothetical architectures are continually devised, the ability to quickly and easily retarget an existing compiler to a new architecture is necessary. We seek to automate the creation of code generators from compiler-independent machine descriptions. A look at the stages involved in a typical compiler will help us determine the main problems involved in retargeting a compiler.

The first responsibilities of a compiler are the lexing and parsing of the input language, from which an abstract-syntax tree is created. This process is independent of the target machine, and nothing need change when the compiler is retargeted. Similarly, the compiler may perform static-semantic checking without affecting retargeting. At this point, the code is usually translated into a compiler-specific intermediate representation.

One of the first machine-dependent issues handled by the compiler involves the target architecture's calling conventions. The compiler must arrange for the inputs to and outputs from procedures to be placed in the appropriate locations for the

target machine. The Calling Convention Language, part of the Zephyr Compiler Infrastructure, may be used to describe these conventions in a machine-independent fashion (Bailey and Davidson 1995). We do not deal with calling conventions in the rest of our work, assuming that the inputs and outputs to a procedure have already been moved to the appropriate locations.

At this point, the compiler must choose machine instructions to implement the intermediate representation. If registers are used, the compiler must somehow handle the limited supply of registers that the hardware provides. This is done via the introduction of temporary locations to stand for registers. The use of temporaries allows the compiler to act as if it has an infinite supply of registers to work with, rather than being limited by hardware constraints. Which classes of temporary locations are available depends upon the particular registers available on the target machine, and hence this step is machine-dependent. Following instruction selection, hardware registers are allocated to replace the temporaries. We do not take up register allocation in this paper, but machine-independent register-allocation algorithms have been available for years (Chaitin 1982, Briggs, Cooper, and Torczon 1994, George and Appel 1996, Poletto and Sarkar 1999).

In addition to the above stages, compilers may perform optimizations throughout the entire process. High-level optimizations may be applied to abstract-syntax trees as well as to the intermediate representation. Further optimizations can take place following instruction selection or even after register allocation. Some optimizations may be machine independent, while others may depend on properties of the target architecture.

We see that the major areas that must be addressed in retargeting a compiler are optimizations and the generation of machine code from an intermediate representation. These two areas represent the last barrier towards automating compiler retargeting. Both are usually done by hand, yet there are currently two families of work devoted to aiding the instruction selection and optimization phases of retargeting a compiler.

The Backend Generator (BEG), BURG, iburg, and Twig are all systems based on bottom-up tree rewriting system (BURS) theory. They produce code generators with guarantees of local optimality (Fraser, Hanson, and Proebsting 1992). All are driven

by a set of grammar-like tree-matching rules. These rules specify which computations may develop values into temporary locations, at what cost. Based on these rules, each tool in this family uses a dynamic-programming algorithm to discover sequences of instructions that implement larger computations. Given a tree representing a source language statement, the algorithm finds the tree cover with the lowest cost, based on the rules used.

This approach guarantees local optimality, but says nothing of the overall quality of the code it generates. A different approach is taken by software including the GNU C Compiler (gcc) and Very Portable Optimizer (VPO). This family attempts to gain efficiency via the use of machine-independent optimizations performed on a machine-independent and yet machine-level intermediate representation (Davidson and Fraser 1984, Benitez and Davidson 1988). This strategy encourages handwritten “code expanders” to produce naïve code that can then undergo various code-improving transformations. Typically, software in this category requires three components: a naïve code expander, a set of semantics-preserving optimizations, and a recognizer. The code expander is required to emit a sequence of statements, each representable as a single instruction on the target machine. This property is known as the *machine invariant*. The recognizer tests whether a given statement satisfies the machine invariant. The optimizer repeatedly applies optimizations, using the recognizer after each to ensure that the machine invariant has not been violated. If at any point the invariant does not hold, the offending optimization is undone.

While both of these families are designed to aid in producing an easily retargeted compiler, they both require a compiler author to create a new mapping from an intermediate representation to an instruction set for every new target machine. Such a mapping would also be a part, of course, of creating a new back end for any compiler not explicitly designed for retargetability. Because each such mapping is linked not only to the target architecture but also to a compiler’s particular intermediate representation, the mappings cannot be reused, whether it be in another compiler or in other tools altogether. Under the current state of the art, then, every compiler author must write a mapping for every desired target machine—an $O(m \times n)$ situation.

λ -RTL is a language for specifying the semantics of machines’ instructions in a manner independent of any particular intermediate representation (Ramsey and

Davidson 1998). λ -RTL allows the effect on the state of a machine of every instruction in an instruction set to be specified using *register-transfer lists* (RTLs). Our work demonstrates how the mappings described above can be automatically derived from λ -RTL machine descriptions, instead of written by hand.

If this goal is reached, the $O(m \times n)$ problem from above is reduced to an $O(m+n)$ problem: a compiler author need only write by hand a mapping from the compiler's intermediate representation to our RTLs *once*, and, similarly, a λ -RTL description for any given target machine need only be composed *once*. Moreover, our analyses are sufficiently general to be used to generate other tools, such as emulators, debuggers, or binary translators.

The main challenge in deriving this mapping from a machine description can be stated succinctly as discovering how to use the instruction set to place the results of arbitrary computations into arbitrary locations. We approach this challenge via an analysis of the locations and operators involved in a given target architecture.

Figure 1.1 depicts the entire process, including both what happens at compile-compile time¹ and also the compile-time components of a compiler that we discuss above. The white regions represent the work that we present in this paper. The process begins at compile-compile time with a λ -RTL description of a target machine's instructions' semantics. Our analysis begins once the λ -RTL translator has produced RTLs for the entire instruction set from the machine description. We first perform a location-driven analysis. We classify the target machine's locations via an examination of their use within the instruction set. From this classification we determine the classes of temporaries that will be used throughout the compiler. We also construct a data-movement graph by searching the instruction set for instructions that move values between locations on the machine.

Both the temporary classes and data-movement graph are used within the next phase of our compile-compile-time analysis. This phase is driven by the desire to find instructions to implement all operators that may be involved in arbitrary computations. Particularly on a CISC machine, a single instruction may involve multiple operators, and it is likely that a single operator will occur in multiple instructions

¹Compile-compile time refers to the time when a compiler is generated. Compile time refers to the time when a compiler transforms a source program to assembly or machine code.

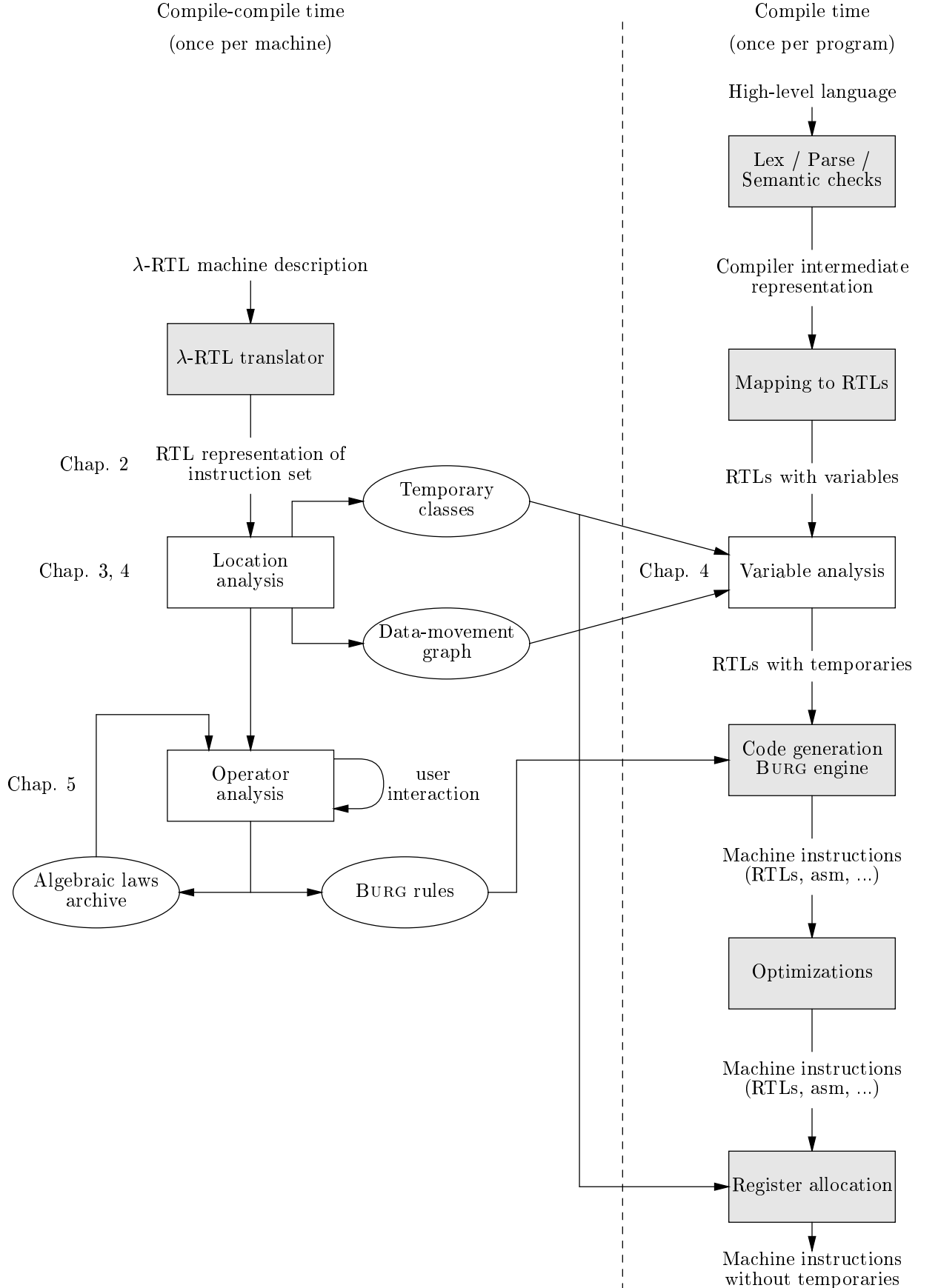


Figure 1.1: The overall process—white regions are presented in this paper

within a target architecture. We present various strategies to find the cheapest way to implement a single computation of each operator. Based on the result of this operator analysis, we discuss a framework to emit BURG rules to drive instruction selection at compile time. We frame our compile-compile-time analyses as an inductive proof of which source RTLs our code generator will be able to translate. If a compiler author composing a mapping from his intermediate representation to RTLs ensures that the only RTLs he produces fall within the set of RTLs that we prove we can translate, he is guaranteed that our back end will successfully generate code for his front end.

At compile time, the compiler front end behaves as normal, lexing and parsing a high-level language and then creating abstract syntax and—perhaps after high-level optimizations and semantic checking—transforming the code into an intermediate representation. The compiler then maps this intermediate representation to RTLs, at which point code generation begins. The RTLs at this point contain variables; we use the data-movement graph and our classes of temporaries to perform a simple analysis that assigns each variable to a temporary. Using the rules generated at compile-compile time, a BURG engine then expands each RTL into a sequence of machine instructions. We leave the representation of the instructions unspecified—it may be RTLs or assembly code or some other format. Following any additional optimizations along the lines of gcc and VPO, the register allocator runs, using the information about our temporary classes in order to assign hardware locations to all temporaries, while spilling temporaries to memory if necessary.

We present several instruction-set analyses in this thesis. In Chapters 3 and 4, we answer the questions, *what storage locations are available on the target architecture* and *where should we store intermediate results of computations*. We first develop formalisms with which we can express the problems and solutions, and then present implementations of the analyses. In Chapter 5 we then study the question: How do we use intermediate results in arbitrary computations? We propose a complete instruction-set analysis to answer this question.

We draw on many examples from actual architectures, particularly the SPARC, as we go along. In order to help illustrate the overall picture, we present a running example involving the translation of a single C procedure into assembly code for a fictitious and very simple machine known affectionately as the Tiny Machine. Our

```

int almostManhattanDistance(int x1, int y1, int x2, int y2) {
    return((x1 - x2) + (y1 - y2));
}

```

Figure 1.2: The C procedure we translate for the Tiny Machine

```

li      20,  %r4      # Set up a 20-byte stack frame
sub     %sp, %sp, %r4

ld      %sp,  0,  %r0      # Load the values of the function's
ld      %sp,  4,  %r1      # parameters from the stack into
ld      %sp,  8,  %r2      # registers
ld      %sp, 12,  %r3

sub     %r0, %r2,  %r5      # Perform the Manhattan distance
sub     %r1, %r3,  %r6      # calculation
add     %r5, %r6,  %r7

st      %sp, 16,  %r7      # Store the return value on the stack

```

Figure 1.3: The final assembly code output from translating `almostManhattanDistance`

example deals with the translation of the procedure in Figure 1.2 that *almost* calculates the Manhattan distance between two points.² As we go through the process outlined above, we will show how this procedure gets translated into the assembly code given in Figure 1.3. The next chapter presents the details of the Tiny Machine's architecture.

²So named as it represents the distance between two points traveling only along horizontal and vertical 'city blocks.' For our function to actually calculate the correct Manhattan distance, we would need to take the absolute values of the two differences.

Chapter 2

Background

2.1 Machine State

λ -RTL posits that a machine's state can be fully represented as a collection of sets of mutable cells (Ramsey and Davidson 1998). We refer to each set of mutable cells as a *space*. We can then uniquely identify a cell by providing the space that it belongs to and its index within that space. For example, on the SPARC, integer registers, floating-point registers, and memory are three different spaces. `$r[3]` refers to the fourth cell within `r`-space, i.e., the fourth integer register. This model of a machine's state extends without difficulty to less traditional locations such as the program counter and condition codes. Thus on the SPARC, the condition codes are nothing more than bits 20-23 of the Process State Register, represented in the λ -RTL machine description as `$i[0]`.

2.2 Register-transfer Lists

A register-transfer list is a list of guarded effects that transform a machine's state. Each guarded effect consists of a guard and an assignment, itself composed of a left-hand side and a right-hand side. The left-hand side of the assignment specifies which locations of the machine's state are modified. The right-hand side specifies the new values for these locations. The guard specifies under what conditions this change of state occurs. Multiple guarded effects within a single RTL occur simultaneously.

RTLs have their roots in the ISP descriptions of Bell and Newell (1971). ISP descriptions allowed for nested sequences of actions predicated on conditions, describing instructions' effects on processors' memories and registers. ISP was not formally specified until it evolved into ISPS (Siewiorek, Bell, and Newell 1982). Davidson adapted register transfers in his original work on machine-independent optimizations (Davidson and Fraser 1984). Richard Stallman borrowed some ideas from these RTLs to create GNU's register-transfer *language*, a mix of semantic effects, pipeline information, and compiler-dependent C-code (Stallman 1999). The current detailed formalism of RTLs that we use was established by Ramsey and Davidson, and eliminates the last vestiges of any machine-dependent aspects (Ramsey and Davidson 1998).

While actual RTLs contain great amounts of detail in a tree form, we use a simplified meta-language borrowed from λ -RTL for convenience in referring to RTLs. We write the general form of an RTL as

$$g_1 \rightarrow l_1 := e_1 \mid g_2 \rightarrow l_2 := e_2 \mid \dots \mid g_n \rightarrow l_n := e_n$$

In this form, each g_i stands for a guard, l_i for a location, and e_i for an expression. We use \rightarrow to indicate that the expression on the left guards the effect on the right. An expression on the right-hand side of an assignment can be a constant, a variable, a value fetched from a location, or an operator applied to a list of one or more expressions. If an assignment is not predicated on a condition (or, equivalently, $g_i = \text{true}$), then we omit the guard for that effect altogether. An empty RTL—a no-op—is represented as `Rtl.Skip`.

For example, the SPARC's `and` instruction might be represented as the RTL `$r[rd] := $r[rs1] \wedge $r[rs2]`. While this example represents the general form of the `and` instruction, we use the same notation to refer to a specific instance of such an instruction, as in `$r[3] := $r[7] \wedge $r[4]`. As another example, consider the SPARC `swap` instruction, `$r[rd] := $m[rs1 + rs2] \mid $m[rs1 + rs2] := $r[rd]`. Because multiple effects within a single RTL occur simultaneously, we can define such an instruction without using temporary values.

Figure 2.1 is a formal specification of the RTLs we use. These RTLs have a rather simple structure. Expressions are built recursively through the application of operators to lists of expressions, with leaf expressions being constants, instruction

<code>width = (int) -- size of a value, in bits</code>	<code>const</code> bit vector
<code>exp = CONST (<i>const</i>)</code>	<code>operator</code> function
<code> FETCH (<i>location</i>, <i>width</i>)</code>	<code>name</code> string
<code> APP (<i>operator</i>, <i>exp</i>*)</code>	<code>space</code> mutable store
<code> OPERAND (<i>name</i>, <i>width</i>)</code>	Meanings of unspecified terminal symbols
<code>location = LOC (<i>space</i>, <i>exp</i>, <i>width</i>)</code>	
<code> VAR (<i>name</i>, <i>width</i>)</code>	
<code>effect = STORE (<i>location</i> <i>dst</i>, <i>exp</i> <i>src</i>, <i>width</i>)</code>	
<code> KILL (<i>location</i>)</code>	
<code>guarded = GUARD (<i>exp</i>, <i>effect</i>)</code>	
<code>rtl = RTL (<i>guarded</i>*)</code>	

Figure 2.1: ASDL specification of the form of RTLs. Variables appear in italics as they can only occur within source RTLs and not within the RTLs that represent a machine’s instruction set.

operands, or fetches from locations. A location may either be a machine location (including temporaries, see Chapter 3), or a variable. Variables may only occur within source RTLs. The RTLs that represent machine instructions do not contain variables.

The formalism used within this paper is a slight simplification of the actual RTLs we use. In particular, the full RTLs distinguish between *cells* and *locations*, allowing the user to dictate aggregations that specify how to interpret multiple cells as a single location. Aggregations are largely orthogonal to our work here, and so we ignore them for expository purposes. We make use of this basic structure of RTLs in our analyses below.

The RTLs used within our system have several properties distinguishing them from other systems. First, every node within a tree has its width (size in bits) fully specified. In particular, the width of every operand of an operator is given in the RTL, along with the width of the result of a computation with that operator. In addition, all fetches are explicit. On the surface, when we write RTLs in our meta-language, we hide the difference between a location and a value fetched from that

location. Within the actual RTLs manipulated by our tools however, this distinction is made explicit. Both of these properties help to fully disambiguate the meaning of a given RTL. Nothing is left implicit and the tools that manipulate these RTLs have all relevant information directly available.

We present some details of the type system used by our RTLs here, in order to develop notation that is used throughout the paper. Consider the RTL `add` operator, which has type `#n bits * #n bits -> #n bits`. This indicates that `add` is a function that takes two inputs, each of type `#n bits` and yields a single result, also of type `#n bits`. Each of these types is made up of two parts: `#n` is a variable of kind number, while the `bits` type constructor transforms values of kind number to types. Operators that contain number variables such as this are polymorphic; for all `n`, the `add` operator has a specific type. That is, an operator is specialized to a particular type by supplying specific values for the variables in the operator's type. Thus `add` might be specialized to `add #32` with type `#32 bits * #32 bits -> #32 bits`. When we use an operator in an instruction, all number variables must be specified to concrete widths.

2.3 λ -RTL

While the RTLs described above are structurally simple enough to be easily manipulated by tools, the high level of detail makes writing them by hand tedious. λ -RTL is a fully typed functional language based largely on Standard ML that provides a convenient environment for specifying RTLs for a target machine's instruction set. The λ -RTL translator is able to infer most widths and aggregations. Additionally, the translator conflates locations and values fetched from location such that writing `$r[4]` can mean either a specific register or the value currently in that register, depending on context.

λ -RTL also allows someone writing a machine description to treat bit slices of cells as full-fledged locations that can be manipulated directly. For example, to set the carry flag of the SPARC's condition codes, one would write the λ -RTL expression `$i[0]@loc[20] := 1`. Without this aid, we would have to read the entire word stored at `$i[0]`, modify bit 20, and then write the entire value back to `$i[0]`.

The availability of a full-fledged functional language greatly simplifies the specification of instructions with semantics that are conditional on their operands by providing a standard `if-then-else` construct. A grouping mechanism provided by λ -RTL facilitates batch definitions of instructions with similar meanings. λ -RTL also defines a set of approximately 70 standard RTL operators that are automatically included in all machine descriptions (Ramsey and Lindig 2001, Ramsey and Davidson 1998). In addition, an author of a λ -RTL machine description may define machine-specific operators.

2.4 Tiny Machine

The Tiny Machine contains a single set of 16 32-bit integer registers, `$r[0] ... $r[15]`, as well as a primary memory. The Tiny Machine uses only base-displacement addressing into the primary memory. The Tiny Machine uses register 15 as the stack pointer, and it passes values to and from procedures via the stack. The instruction set contains the following five instructions:

- `add rs1,rs2,rd`. Sum the values in `$r[rs1]` and `$r[rs2]` and place the result in `$r[rd]`.
- `sub rs1,rs2,rd`. Take the difference of the values in `$r[rs1]` and `$r[rs2]` and place the result in `$r[rd]`.
- `li con22,rd`. Sign-extend the 22-bit constant `con22` to 32 bits and load it into `$r[rd]`.
- `ld rs1,con18,rd`. Load the value at `$m[$r[rs1] + con18]` from memory into `$r[rd]`, where `con18` is an 18-bit constant sign-extended to 32 bits.
- `st rs1,con18,rd`. Store the value in `$r[rd]` into memory at `$m[$r[rs1] + con18]`.

The Tiny Machine's instruction set's semantics are given by the λ -RTL machine description in Figure 2.2. The `aggregate using RTL.AGGL` line in the description indicates that the Tiny Machine is a little-endian machine, though aggregations are

```

module TinyMachine is
  import RTL
  from StdOperators import [sx := + -]

  storage
    'r' is 16 cells of 32 bits called "registers"
    'm' is    cells of  8 bits called "memory" aggregate using RTL.AGGL

  operand [rs1 rs2 rd]  : #4  bits
  operand cons10        : #10 bits
  operand con18         : #18 bits
  operand con22         : #22 bits
  operand address       : #32 bits

  default attribute of
    addr(rs1, con18) : address is $r[rs1] + sx con18

  default attribute of
    li(con22, rd)    is $r[rd] := sx con22
    ld(address, rd)  is $r[rd] := $m[address]
    st(address, rd)  is $m[address] := $r[rd]

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

end

```

Figure 2.2: λ -RTL machine description for the Tiny Machine

immaterial to the rest of this paper. The `default attribute of` sections define both the Tiny Machine's addressing mode (`addr`) as well as the semantics of the Tiny Machine's instructions.

Chapter 3

Spaces and Location Sets

3.1 Motivation

A human being's natural unit of computation is often much larger than a machine's. For example, we would not expect to find a single machine instruction that performs the Manhattan distance calculation $(x1 - x2) + (y1 - y2)$ presented in Figure 1.2. When generating code for a computation too large to be represented by a single instruction on the target architecture, a compiler must break the large computation into smaller computations that can each be implemented by a single machine instruction.

By breaking up large computations in this manner, the compiler has created a new problem: Where on the machine should the intermediate results be stored? We must pick actual hardware locations for these values. More specifically, we would like to store these values within registers. However, not all registers are created equal. Some, such as the program counter or status registers, have defined purposes that prevent them from being used for intermediate results. Others may be intended to store floating-point values, or to store addresses as opposed to data.

Why do we prefer to store intermediate results in registers in the first place? The answer is twofold: First, registers are designed to be accessed more efficiently than memory. Second, the compiler has complete control over choosing—at compile time—which registers to use for which values.¹ Despite these advantages, there is

¹Also, registers are rich and chocolatey!

another difficulty. The number of variables and intermediate results involved in a large computation may be greater than the number of actual hardware registers available.

The standard solution to this problem involves the introduction of temporary locations and the use of a register allocator. To handle the limited supply of hardware locations, we create infinite supplies of temporary locations and allow the code generator to use these temporaries to store the intermediate results of computations. Following code generation, the register allocator maps the temporary locations to actual hardware registers. When there are more live values than available registers, the allocator inserts instructions to spill some values to memory and later reload these values from memory when they are needed again.

We cannot, however, simply deal with a single infinite supply of temporary locations. Because the actual hardware registers themselves are not all the same, we must separate our infinite supply of temporaries into different classes, associating each class of temporaries with one specific set of hardware registers. This relation says, in essence, that the register allocator may replace a location within a given class of temporaries with *any* location from the associated set of registers.

We do not wish to be limited to working with predefined types of register sets. Looking at the SPARC or MIPS architectures, we might conclude that we can classify all register sets as either integer or floating point. This characterization would fail, however, for a machine such as the Motorola 68000 that distinguishes between data and address register sets. As we noted above, the different temporary classes required directly depends on the different register sets available. Thus, before we can think about automatically generating code, we want to automatically discover the appropriate classes of temporary locations from a target machine's λ -RTL semantic description.

3.2 Spaces: Analysis

We have seen that λ -RTL divides locations within a machine into spaces. Before we can discern appropriate classes of temporary locations, we must examine an instruction set and determine which spaces correspond to sets of registers on the target machine. Spaces within a λ -RTL description are not annotated with this information,

and so we make this determination via an examination of *how* locations within spaces are used within the target machine's instruction set.

Achieving this goal is intimately connected with the binding times of expressions found within λ -RTL descriptions. There are three possibilities for this binding time:

- The value of an expression can be known at *specification time*. We can tell the values of these expressions simply by examining an instruction's opcode. The values of these expressions do not depend on the instruction's operands nor on the state of the machine. For example, the SPARC's `call` instruction involves a control transfer to the address `PC + (4 * disp30)`. In this address the value of the constant 4 is known at specification time; no matter when or with what operands `call` is invoked, the value of that expression is always 4.
- The value of an expression can be known at *instruction-creation time*. This is the case for expressions whose values depend on the particular operands with which an instance of an instruction is created. The values of expressions that are known at instruction-creation time cannot depend on the machine's state. In the calculation of the address for the `call` instruction above, `disp30` is an operand to the instruction, and hence the value of `disp30` is bound when a specific instance of the `call` instruction is created; no matter the machine's state when this instance is invoked, the value of this expression is unchanged.
- The value of an expression can be known at *run time*. This is the case for expressions whose values depend on the current state of the machine. The SPARC's `st` instruction stores into memory the value stored within a register, `$r[rs1]`, where `rs1` is an operand to `st`. The value of the expression `$r[rs1]` is bound at run time, as the value fetched from a register depends on the machine's state at the moment `st` is executed.² Expressions with values bound at run time will likely have different values when the *same instance* of an instruction is executed at multiple times.

Aside from atomic expressions such as constants, instruction operands, and values fetched from locations or variables, Figure 2.1 shows us that an expression can involve

²The value of `rs1` is bound at instruction-creation time and hence we know at instruction-creation time to which register `$r[rs1]` refers. But it is not until run time that we know what value is stored *in* that register.

applying an operator to one or more expressions. In this case, the binding time of the value of the resulting expression is directly related to the binding time of the values of the constituent expressions. In fact, we can simply state that the binding time of the über-expression is the *latest* binding time of the values of the operand expressions. Thus, in the `call` instruction the value of the expression `4 * disp30` is bound at instruction-creation time. In one version of the SPARC `add` instruction—`$r[rd] := $r[rs1] + simm13`—a value fetched from a register is added to one of the instruction’s operands, `simm13`. As the operand’s value is bound at instruction-creation time while the value fetched from `$r[rs1]` is bound at run time, the entire expression `$r[rs1] + simm13` has a value that is bound only at run time.

Returning to our view of a machine’s state as collections of sets of mutable cells, we again note that a cell is specified completely by its space and its index (offset) within that space. By examining the binding time of cell’s indexes, we can identify the register sets for a given architecture.

We noted above that one benefit of using registers to hold intermediate results is that the compiler controls which registers are used for which values. As a corollary to this, one defining characteristic of a register set is that specific registers used by an instruction are always known at *instruction-creation time*. That is, no matter how many times an instruction involving a register is executed, and no matter how the machine state varies, that instance of the instruction always refers to the same register.³ On the flip side, it is usually the case that the specific register involved in an instance of an instruction is specified by one of the instruction’s operands.⁴ Given this observation, we can break spaces into three categories, based upon the binding time of the expressions used to index cells within them:

- A *fixed* space is one whose constituent cells are always indexed by expressions with values bound at specification time. Because of this relationship, locations within a fixed space are determined by the instruction’s opcode *only*. The SPARC i-space, containing the integer-unit control and status registers, is an example of a fixed space. Locations within this space include the program

³Though as we have just noted, the value within that register *does* vary with the machine state.

⁴We say ‘usually’ rather than ‘always’, for many architectures have instructions that refer directly (independent of operands) to specific registers. The `call` instruction on the SPARC is an example of this, as it saves the current value of the program counter into integer register 7: `$r[7]`

counter and the processor-state register, which contains, among other things, the integer condition codes. All instructions that refer to a cell within this space do so directly, rather than via an instruction operand or machine state. For example, every instruction that refers to the processor-state register refers to it as `$i[0]`. In general, all instances of an instruction that refers to locations in fixed spaces refer to the *same* locations within that space, regardless of both the values of the instructions' operands and also the current machine state.

- A *register-like* space is one that is not a fixed space and whose constituent cells are indexed by expressions with values bound either at specification time or at instruction-creation time. Because of this, locations within a register-like space may depend on operands of the instruction, but they are independent of machine state. The SPARC integer register `r`-space is an example of a register-like space, as is the SPARC floating-point register `f`-space. An instruction that refers to a cell within `r`-space either does so directly, as in the term `$r[7]` or (in the usual case) via an instruction operand, such as `$r[rs1]` in the SPARC's `add` instruction. When an instruction that refers to locations within a register-like space occurs more than once in a program, each instance may refer to different locations in that space. However, which cell is being referred to is specified only by the operands of the instruction, and cannot be influenced by machine state.
- A *memory-like* space is one that is neither a fixed nor a register-like space and whose constituent cells are indexed by expressions with values bound either at specification time, instruction-creation time, or run time. As such, locations within a memory-like space will usually depend upon the state of the machine at the time the instruction is executed. Thus, in the SPARC load instruction, `$r[rd] := $m[$r[rs1] + $r[rs2]]`, the value of the address expression, `$r[rs1] + $r[rs2]`, is bound at run time, and hence the specific location in `m`-space being fetched from for a specific instance of this instruction depends upon the machine state (specifically, the values in `$r[rs1]` and `$r[rs2]`). Thus, an instruction that refers to cells in a memory-like space may refer to different cells on different executions of the *same instance* of the instruction, depending on the machine state. Such a situation would arise, for example, when using a

Cells' indexing expressions binding time		Space category
Run time	\Leftrightarrow	Memory-like
\uparrow		\uparrow
Instruction-creation time	\Leftrightarrow	Register-like
\uparrow		\uparrow
Specification time	\Leftrightarrow	Fixed

Figure 3.1: Relationship between index expressions' binding times and space categorization

pointer to iteratively access the elements of an array.

We can place a given space, s , into one of these three categories by examining the individual occurrences of locations in s within an instruction set. If there is at least one mention of s in which the value of the index expression is bound at run time, then s is a memory-like space. If this is not the case and there is at least one mention of x in which the value of the index expression is bound at instruction-creation time, then s is a register-like space. Otherwise, it must be that every mention of s has the value of the index expression bound at specification time, and thus s is a fixed space. More concisely, we consider the *latest* binding time of all expressions that index cells in s . The relation between this binding time and a space categorization is shown in Figure 3.1.

3.3 Spaces: Implementation

The implementation of this analysis is straightforward. We examine the RTLs for the entire instruction set. In particular, we survey all of the locations mentioned within the instruction set. When we find an occurrence of a location in space s , we perform the following steps:

Determine the binding time of the indexing expression. This is done in a manner consistent with the analysis presented above, via induction on the structure of the expression (see Figure 2.1). Constants—with values determined by the instruction itself—are bound at specification time. Instructions' operands are known when a

particular instance of an instruction is created, and hence are instruction-creation time expressions. Values fetched from locations are bound at run time. When an expression results from applying an operator to other expressions, its binding time is the latest binding time of its constituent expressions.

Update our categorization of s . Based on the binding time of the indexing expression, we can make a temporary categorization of s according to the correspondence in Figure 3.1. If we have already placed s into a category, we take the more conservative of the two categories. For example, if we have currently noted that s is a fixed space, and we find an RTL that contains a location in s with an index bound at instruction-creation time, then we update our conclusions to indicate that s is a register-like space. A memory-like space is more conservative than a register-like space, which in turn is more conservative than a fixed space.

We implemented this analysis and ran it on several machines' λ -RTL descriptions. For the SPARC, the analysis correctly showed that both the **r**-space (integer registers) and **f**-space (floating-point registers) are register-like spaces. The **m**-space (main memory) was classified as memory-like, as was the fictitious **w**-space used in the machine description to model the SPARC's register windows. The SPARC spaces representing integer and floating-point control and status registers were determined to be fixed spaces. The MIPS analysis is similar, and also correctly concludes that the **p**-space—the system control coprocessor registers—is register-like. On the Tiny Machine, we determined that **m**-space is memory-like while **r**-space is register like.

3.4 Location Sets: Analysis

Above, we note that not all register sets are equivalent. For example, registers in the SPARC's **r**-space are intended to hold integer data, while those in **f**-space are intended to hold floating-point values. From this, we concluded that we need different classes of temporaries for these different spaces.

This conclusion, however, is insufficient. By itself, it assumes that locations *within* a particular space are interchangeable. This is not always the case. For example, fetches from integer register zero—**\$r[0]**—on the SPARC always return zero. Furthermore, values cannot be stored into register zero. Because of this behavior, **\$r[0]**

is not interchangeable with the other integer registers, $\$r[x]$ where $x \neq 0$. Therefore, it would be a mistake to create an infinite class of temporaries in which any of the temporary locations might stand for either $\$r[0]$ or $\$r[x]$ where $x \neq 0$. Rather than have a class of temporaries associated with all of r -space, we would want a class of temporaries associated with the registers specified by $\$r[x]$ where $x \neq 0$.

We refer to such a class of interchangeable locations as a *location set*. Two locations $l1$ and $l2$ are ‘equivalent in instruction i ’ if and only if there exist two instances of i , i_1 and i_2 , (created by applying i to different operands) such that the RTL semantics of i_2 differs from the semantics of i_1 only in that $l1$ is replaced by $l2$ everywhere. To preclude vacuous relationships, we also require that $l1$ be present in i_1 . This relation is clearly reflexive and symmetric. Given three locations $l1$, $l2$, and $l3$ such that $l1$ and $l2$ are equivalent in i and $l2$ and $l3$ are equivalent in i , we see that substituting $l1$ with $l2$ and then $l2$ with $l3$ does not change an instance’s RTL semantics, and hence $l1$ and $l3$ are equivalent in i . Thus, this is an equivalence relation. Consider the SPARC `add` instruction, $\$r[rd] := \$r[rs1] + \$r[rs2]$. Registers $\$r[3]$ and $\$r[18]$ are equivalent in the `add` instruction. Considering two instances of `add` in which `rs1` = 3 and in which `rs1` = 18 (and in which `rs2` and `rd` are left unchanged), the only difference in the instances’ semantics is the substitution of $\$r[18]$ for $\$r[3]$. On the other hand, $\$r[3]$ and $\$r[0]$ are not equivalent in `add`. Because a value fetched from $\$r[0]$ is always zero, the semantics change when we replace $\$r[3]$ with $\$r[0]$. Also, $\$r[3]$ is not equivalent to $\$f[8]$ in `add`, as there is no instance of `add` that replaces an integer register with a floating-point register.

We now formally define a location set. A location set is a set of locations that are equivalent in one or more instructions. Consider again the SPARC’s `add` instruction. Following the argument above, all locations within the set $\$r[rs1]$ where `rs1` $\neq 0$ are equivalent to one another in `add`, and hence $\$r[rs1]$ where `rs1` $\neq 0$ is a location set. Conversely, $\$r[x]$ where $x = 0$ (or better, $\$r[0]$) forms a singleton location set of its own. On an actual machine, many instructions define the same location sets.

As with the space analysis above, we separate location sets into fixed, register-like, and memory-like categories, as follows:

- A *fixed location set* is a singleton location set. The location within this set is specified directly by one or more instructions, and hence there are no other

locations on the machine that would be interchangeable with it in those instructions. All of the SPARC's instructions that manipulate condition codes refer directly to the integer-unit status/control register `$i[0]`, where the condition codes are stored. No other location can substitute for the role `$i[0]` plays in these instructions, and hence `$i[0]` is a fixed location set. As noted above, `$r[0]` is another fixed location set, due to its unique semantics.

- A *register-like location set* consists of a subset of locations within a register-like space. On the SPARC, `$r[x]` where $x \neq 0$ is an example of a register-like location set. Unlike this example, a register-like location set may be an entire register-like space. The SPARC floating-point register zero does not share the unique semantics of integer register zero, and as such `$f[x]` (all locations in f-space) constitutes a register-like location set.
- A *memory-like location set* consists of some set of locations within a memory-like space of a given width. For example, the 32-bit values within m-space form a memory-like location set on the SPARC. As with register-like location sets, a memory-like location set may correspond to an entire memory-like space. Alternatively, a memory-like location set may correspond to a proper subset of a memory-like space according to, for example, alignment requirements.

In a larger sense, a location on a machine is a place into which values can be put and out of which values can be taken. In this sense, a location is a read-write container for values. As we have seen, a location set is nothing more than a set of locations treated homogeneously by some group of instructions on a target architecture. We would like to extend this abstract idea of a location to value containers that are either read-only or write-only. But to what exactly would such odd notions of a location correspond?

A read-only location is nothing more than a constant.⁵ We can get a value from a constant (the constant itself), but we cannot change the value stored 'in' a constant. How do we group constants into location sets? Machine instructions specify constants of particular widths, such as the 13-bit constant used by the SPARC's `add` instruction,

⁵When we talk of constants here, we speak of values fixed *either* at specification time (constants) or at instruction-creation time (instruction operands).

`$r[rd] := $r[rs1] + simm13`. In this position, we can use any 13-bit constant interchangeably, but we cannot use, say, a 22-bit constant. Hence, we discriminate between read-only location sets based on width: 13-bit constants might comprise one location set while 22-bit constants would make up a different location set. We refer to the location set consisting of constants of width `#U bits` as *conU*.

We also include sign-extended and zero-extended constants as read-only location sets. In fact, `simm13` above is really a 13-bit constant sign-extended to 32 bits. This location set represents all 32-bit constants that can fit within 13 bits. In general, the read-only location set consisting of constants zero- or sign-extended from n to w bits represents those w -bit values that can fit in n -bits.

Figure 2.1 tells us that an expression in a machine instruction that does not contain any operators is either a constant, an instruction operand, or a value fetched from a location. We now observe that any such expression can be classified as either a read-write or a read-only location set.

If a constant may be thought of as a read-only location, what might correspond to a write-only location? By our abstraction above, such a location is a container into which we can place values, but from which we cannot retrieve values. We have seen that memory-like expressions are addressed by arbitrary expressions (depending on the addressing mode). The SPARC can refer to memory locations via register-register addressing: `$m[$r[rs1] + $r[rs2]]` or via base-displacement addressing: `$m[$r[rs1] + simm13]`. If we consider the expressions used to address m-space in this example, we would like to think of the addressing expressions as containers into which we can place arbitrary values (namely an address). We need this ability, for example, to translate code involving arbitrary pointer arithmetic. However, once we have gotten a value into an addressing expression, the value is used to address the memory-like space, and for nothing else. That is, the concept of reading a value out of an addressing expression is meaningless. Addressing expressions, then, fit our abstraction of write-only locations.⁶

We extend these write-only locations to write-only location sets by observing which

⁶An alternative view notes that constants are read-only in the sense that we can move arbitrary values from a constant into read-write (actual hardware) locations, while we cannot move arbitrary values stored in read-write locations ‘into’ constants. Similarly, we want the ability to ‘move’ arbitrary addresses stored in read-write locations into the form of an addressing expression, but once we have an addressing expression we have no need to move that value into a read-write location.

addressing expressions may be used to refer to memory in a given instruction. The SPARC `st` instruction has the following two forms:

$$\begin{aligned} \$m[\$r[rs1] + \$r[rs2]] &= \$r[rd] \\ \$m[\$r[rs1] + \text{sim}13] &= \$r[rd] \end{aligned}$$

As either of these two addressing expressions can be used interchangeably within this instruction, we would create a write-only location set containing the elements $\$r[rs1] + \$r[rs2]$ and $\$r[rs1] + \text{sim}13$.

The concept of these location sets is a refinement and expansion of the space classification discussed above. It is worth noting, however, that fixed location sets may occur within register-like spaces. For example, on the SPARC $\$r[0]$ is a fixed location set within a register-like space, due to its unique semantics already discussed. Furthermore, the SPARC's `call` instruction writes its own address into $\$r[7]$. Because no other location can take this role, $\$r[7]$ is a fixed location set.⁷

3.5 Location Sets: Implementation

Identifying the location sets present within an instruction set is straightforward. Location sets can be determined directly by a completely localized inspection of an RTL. Our interest in these location sets, then, comes from the fact that the same location sets occur time and time again throughout an instruction set. We will use these location sets for much of our analysis below.

Thus, to identify the various classes of temporary locations that we need, we iterate through all of the instructions in a target architecture and identify all of the register-like location sets. We associate each such location set that we find with a new class of temporaries, and we pass this mapping along to the rest of our work. In the next chapter, we will use the ideas developed here to assign temporary locations to variables and to discover how to move values between location sets. In Chapter 5, we will see that each read-write location set corresponds to a non-terminal in the BURG rules we create.

⁷This example also demonstrates that location sets need not be disjoint. For most instructions, $\$r[7]$ is interchangeable with the other integer registers (except for $\$r[0]$) and hence $\$r[7]$ is a member of the location set $\$r[x]$ where $x \neq 0$. For the `call` instruction, $\$r[7]$ is unique, and hence it also comprises (on its own) a fixed location set.

On the Tiny Machine we discover four location sets. First, we have the read-only location set that consists of constants sign-extended from `#22 bits` to `#32 bits`. Next, we have the register-like location set that consists of all locations within `r-space`. Unlike on the SPARC, on the Tiny Machine register zero is no different than the other registers, and hence they all form a single location set. The other read-write location set we discover on the Tiny Machine is all 32-bit locations within `m-space`. Finally, our location set analysis discovers the single write-only location set `$r[rs1] + con18`. From these conclusions, we need only create one class of temporaries to correspond to our only register-like location set. We call this temporary space `t-space`.

This entire analysis can be performed at compile-compile time; all of the necessary information can be gleaned directly from a machine's λ -RTL semantic description.

Chapter 4

Choosing Locations for Variables

4.1 Motivation

Before selecting instructions, a compiler must decide where to store variables that appear in a source procedure. There are two main choices. The compiler can decide either to store variables on the stack or else to store variables in registers. In the former case, the variables are stored on the stack before our code generator ever sees the code, and we need not take further action. In the latter case, it is our responsibility to decide in *which* registers to store each variable. Of course, as per our discussion in the previous chapter, we actually wish to assign each variable to a location in a temporary space associated with a register-like location set. Once the variables have been replaced by temporaries, the code expander can begin instruction selection. If at any time more temporaries in a given class are live than there are registers associated with that class, the register allocator will insert the appropriate spills to and loads from memory.

The goal of this analysis is to store each variable in a register-like location set most appropriate to that variable's usage within a procedure. Thus, if a machine provides both integer and floating-point registers, and a variable y is used primarily for floating-point computations, then it will be most efficient for y to be stored in a floating-point register. If y were stored in an integer register, its value would most likely have to be moved into a floating-point register every time it was used in a computation. The Motorola 68000 provides both data and address registers. We

would rather store a pointer in an address register, while a normal integer variable would be better stored in a data register.

We seek in this section a simple method to determine where to store variables. Our goal here is to present an algorithm that works reasonably well, while avoiding extra complexity. In particular, we do not attempt to split variables' live ranges between multiple locations.

4.2 Analysis

We need to determine ‘how’ each variable is used within a procedure—but what exactly does this mean? For the most part, variables are used either as operands to an operator or to store the result of an operation.¹

It will be useful in the following discussion to have some notation available for referring to results and operands of a given operator. We let op_n refer to the result of operator op if $n = 0$, and to the n th operand of op otherwise. Thus in the RTL $x := y + 3$ we have $+_0 = x$, $+_1 = y$, and $+_2 = 3$. We refer to such a combination of operator and position as, logically enough, an *operator position*.

By examining the target machine's instruction set, we can determine from which location sets every operator takes its operands and places its results. That is, an instruction set might provide an instruction with an add operator that takes its operands from integer registers. If a procedure then contains a variable that is used only as an operand to the add operator, we would want to store that variable in an integer registers.

Of course, the situation is more complex than that; variables will most likely be used with various operators, and those operators may very well be used with different location sets throughout the instruction set. In these cases, we need to determine which location set will be cheapest for a given variable. We would like the cheapest location set to be a register-like location set, as it is for the register-like location sets that we have created temporary classes, and it is registers over which the compiler has the most control. However, there are cases in which it may be cheapest to store

¹A variable can also be the entire right-hand side of an assignment as in $x := y$. But all ‘interesting’ variable uses are as part of a computation.

a given variable in a fixed location set. For example, a 1-bit variable representing the carry bit in an explicit implementation of multiprecision arithmetic might be most efficiently stored within the carry bit of a machine's condition codes.

In order to find the cheapest location set for each variable, we estimate a *cost* for storing a given variable in every fixed or register-like location set for each instruction in which the variable occurs. We estimate the total cost for storing a variable in a given location set by summing the per-occurrence costs of storing the variable in that location set.

The question then becomes: how do we estimate these local costs? Each occurrence of a variable is in a specific operator position. By examining the target machine's instruction set, we can find all location sets that occur in that operator position. For this particular occurrence of this variable, these location sets would be cheapest. We can compute the cost (for this occurrence) of storing the variable in other location sets by finding the cheapest way to move a value from these other location sets into location sets that occur in this operator position within the instruction set. This is, of course, no more than an extremely rough approximation to a true cost of storing a variable in candidate locations.

Such an analysis—made more precise below—views every variable in a procedure independently of the other variables in a procedure. One could imagine a hypothetical machine setup in which choosing the optimal storage for one variable relied on using instructions that made other variables' storage choices extremely costly. Whether or not this is an issue to worry about is an experimental question that has not been taken up in the context of this paper. We hope that the simple analysis we present here is sufficient. If future experimental evidence indicates otherwise, a more complex analysis—such as one that allows for splitting a variable's live range between multiple locations—could easily be substituted.

The variable analysis we have described motivates a more general analysis: how can we move values between location sets, or even between locations within a single location set? By examining the instruction set, we can identify those instructions that do nothing more than move a value between location sets. For example, the `ld` instruction on the SPARC moves a value from memory into the register-like location set `$r[x]` where `x` \neq 0. By identifying data-movement instructions, we can construct

a data-movement graph in which the nodes are location sets and the edges represent instructions that move values between location sets. The edges in the transitive closure of this graph represent *all* pairs of location sets (l_1, l_2) for which there is some sequence of instructions to move a value from l_1 into l_2 . Furthermore, finding the cheapest way to move a value between any two location sets is nothing more than a shortest-path calculation.

4.3 Implementation

We discuss the implementation of the analyses above for two reasons. First, it formalizes the procedures presented. Second, and equally important, it allows us to determine which aspects of the procedures can be conducted at compile-compile time, and which portions can be accomplished only at compile time.

The implementation begins by examining all of the RTLs associated within a given procedure, searching for variables used as the operands or results of operators. When a variable is found in a certain operator position op_n , we search for all read-write location sets l , such that there is an instruction in which $op_n = l$. For example, if we are examining variable x within the source RTL $y := 8 + x$, then we search for all instructions in which $+_2$ is a read-write location set. On the SPARC, one such location set would be $\$r[rs2]$ where $rs2 \neq 0$ from the $\$r[rd] := \$r[rs1] + \$r[rs2]$ version of the `add` instruction. However, we would not include `simm13` from the $\$r[rd] := \$r[rs1] + \text{simm13}$ version of the `add` instruction, since a signed-immediate 13-bit constant is a read-only location set.

Given a variable x in operator position op_n , the search above identifies a set L such that for every location set $l \in L$ there is an instruction that directly uses l in op_n . However, because x may appear in other operator positions that do not share the same L , we actually want to determine, for every source RTL in which x appears, the cost of storing x in *any* (read-write) location set. To accomplish this, we must know the cost of moving values between location sets. Given a function $c(l_1, l_2)$ which returns the cost of moving a value from location set l_1 to location set l_2 , we can write the cost function for storing a variable x at operator position op_n in a particular

location set l as:

$$\text{cost}_l(x, op_n) = \frac{1}{\text{vars}(op)} + \begin{cases} 0 & \text{if } l \in L \\ \min_{l' \in L} c(l, l') & \text{otherwise} \end{cases}$$

where, as above, $L = \{l : op_n = l \text{ for some instruction on the target machine}\}$. Also, we define $\text{vars}(op)$ to be the number of variables that appear in the source RTL that we are currently examining.

The term $\frac{1}{\text{vars}(op)}$ ensures that if every variable in the source RTL is placed in a location in L , then the total cost for this source instruction will be 1. For example, if we examine the source RTL $z := x + y$ while compiling for the SPARC, then placing each of the variables into an integer register would give this instruction a total cost of $\frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 1$. We justify this arrangement more below.

Every time we encounter a variable as we iterate through a procedure's RTLs, we calculate the cost of storing it in every fixed or register-like location set using the cost function above. Before summing up these local costs for a given variable, we must take into consideration the fact that not all source RTLs will be executed with the same frequency. An instruction within a loop, for instance, should be weighted more heavily than an instruction that will only be executed once. Thus, we scale the costs per variable occurrence by the estimated frequency of execution of the source RTL, a value provided by a compile-time analysis that is beyond the scope of this thesis. Taking into account this weighting, the presence of the $\frac{1}{\text{vars}(op)}$ term above means that if every variable in a procedure is placed in an optimal location, the total cost that our analysis yields is equal to the estimate of the total number of source instructions executed in the procedure.

Once we finish examining every RTL within the procedure, we can easily assign variables to location sets. For each variable, the appropriate location set is the one for which our analysis yields the smallest cost. We use a simple allocator to assign each variable to a specific temporary within the class of temporaries associated with this location set (see previous chapter).

We cannot expect that every variable will be stored most cheaply within a register-like location set. In particular, most architectures provide multiply, divide, and/or add with carry instructions that insist on using fixed locations. On the SPARC, for example, the high 32-bits of the product of a multiplication is placed within the

fixed Y register, `$i[3]`. The carry bit from an addition also almost always has a fixed location within the condition codes. Our analysis would likely find that variables used largely for such purposes can be stored most cheaply in the fixed locations themselves.

However, we would rather not deal with such fixed locations in our code generator. Instead, when we find that a variable should be stored within a fixed location, we create a new class of temporaries associated with the singleton set containing only this one location. By doing this, we defer handling the fixed location to the register allocator, which will spill and reload the fixed location as necessary.² We note that spills are very likely in this situation, as every temporary within this new class maps to the same hardware location. It may be worthwhile to add an artificial cost for storing a variable in any fixed location set to represent a spill/reload penalty. Whether such a cost is necessary and what value it should have is an experimental question beyond the scope of this paper. We refer to the process of creating a new temporary class to correspond to a fixed location set as *promoting* the fixed location set.

How, then, do we implement the cost function, $c(l_1, l_2)$? As discussed above, $c(l_1, l_2)$ is the shortest-path from l_1 to l_2 in the graph of data-movement instructions. Recalling the general form of an RTL,

$$g_1 \rightarrow l_1 := e_1 \mid g_2 \rightarrow l_2 := e_2 \mid \dots \mid g_n \rightarrow l_n := e_n$$

a data-movement instruction can be identified as any RTL in which $n = 1$ (there is only a single effect), $g_1 = \text{true}$ (the effect is not predicated on a run-time condition), and e_1 does not contain any operators. Here we take advantage of the fact that any expression without an operator can be classified as either a read-write or read-only location set (see Section 3.4).³

From here it is a straightforward process to examine a full instruction set and create a data-movement graph. We use the Floyd-Warshall all-pairs shortest-paths algorithm (Cormen, Leiserson, and Rivest 1990) to create the transitive closure of the data-movement graph in which the edges are weighted with the lengths of the

²This solution is still not entirely satisfactory. We would like these fixed locations to be spilled to registers rather than memory, if possible.

³Actually, instructions that do not fit this form can also (in some cases) be used as data-movement instructions. On the SPARC, the instruction `$r[rd] := $r[rs1] + simm13` can be used to move values between registers when `simm13` is equal to 0. We discover this and other more complex data-movement possibilities using the techniques presented in the next chapter.

shortest paths from the original graph. We call this second graph the *cost graph*. Both the data-movement graph and the cost graph do not depend on any information beyond the λ -RTL description of the instruction set, and hence they can be created at compile-compile time. Once we have created the cost graph, the cost function c , from above, can be read directly from the graph.

We have implemented the analysis to produce the data-movement and cost graphs from a λ -RTL machine description. We use the publicly available graphing program *dot* (Koutsofios and North 1996) to draw portions of these graphs. Figures 4.1 and 4.2 show part of the data-movement graph and cost graph produced by our analysis of the SPARC's λ -RTL semantics. Specifically, they show the subgraphs that consist solely of 32-bit register-like and memory-like location sets. The full graph, containing location sets of other sizes as well as fixed location sets, constants, and addressing expressions, is much larger. Each edge of the data-movement graph is labeled with the instructions that perform the associated data movement, while the edges of the cost graph are labeled with the associated values of the $c(\cdot, \cdot)$ function.

The lack of a self-loop on the vertex `$r[rs1] where ne(rs1, 0)` seems an obvious omission from the data-movement graph. However, our analysis here finds only those instructions whose sole effect is to move a value between locations. On the SPARC, there is no instruction with the sole effect of moving a value from one integer register to another, and instead a value is moved between two integer registers by using an `or` instruction in which one of the operands is zero. We present techniques to discover such non-obvious move instructions in the next chapter.

We also include the connected portion of the Tiny Machine's data-movement graph (excluding the unconnected write-only location set `$r[rs1] + con18`) as Figure 4.3.

We have not yet implemented the compile-time variable analysis outlined in this chapter. Armed with our temporary classes and the data-movement graph, however (both of which are implemented), the rest of the variable analysis is machine-independent, and hence need only be directly implemented, rather than derived from a λ -RTL description. This direct implementation would, however, refer to the derived data-movement graph for cost information while examining each variable.

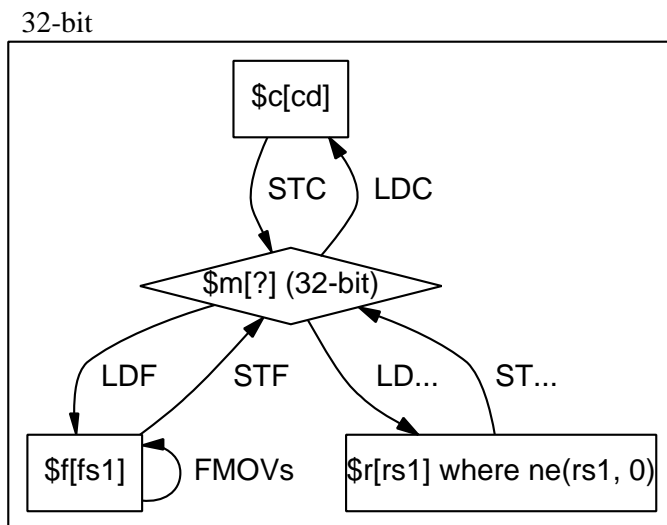


Figure 4.1: Part of the SPARC's data-movement graph. There is no self-loop on the vertex $\$r[rs1]$ where $ne(rs1, 0)$ because the SPARC does not contain an instruction whose sole effect is to move a value between integer registers. We present techniques to remedy this in the next chapter.

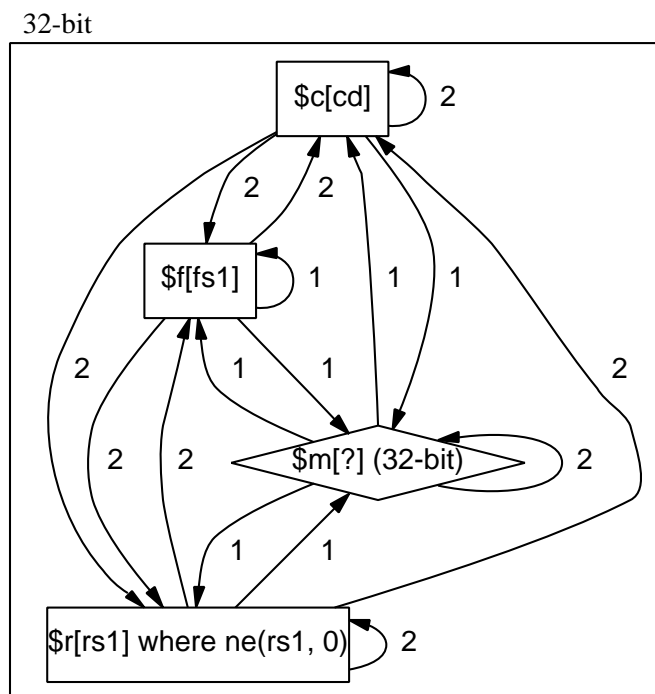


Figure 4.2: Part of the SPARC's cost graph

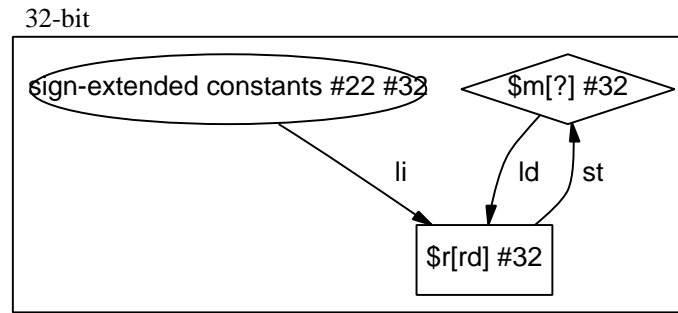


Figure 4.3: The Tiny Machine's data-movement graph.

```
# create a new stack frame by adjusting the stack pointer
$r[15] := $r[15] - 20;

# retrieve parameters into variables
x1 := $m[$r[15] + 0];
y1 := $m[$r[15] + 4];
x2 := $m[$r[15] + 8];
y2 := $m[$r[15] + 12];

# compute and return the appropriate value
$m[$r[15] + 16] := (x1 - x2) + (y1 - y2);
```

Figure 4.4: RTLs with variables produced from the C code in Figure 1.2 on page 11

4.4 Choosing Variables' Locations on the Tiny Machine

We now return to the C procedure from Figure 1.2 that we wish to translate for the Tiny Machine. Figure 4.4 shows the RTLs that a compiler front end might produce for this procedure. They add little to the procedure itself beyond a stack frame and several RTLs to move the procedure's parameters from the stack into local variables. We also assume that the result of the procedure is returned via the stack.

Using the compile-time variable analysis given in this chapter, we would estimate a cost of storing `x1`, `y1`, `x2`, and `y2` in every register-like location set. Because the Tiny Machine only has one such location set—`$r[x]`—all the variables are placed in the corresponding temporary class. Figure 4.5 shows what the RTLs for this procedure would look like once the variables have been replaced with temporaries.

```
# create a new stack frame by adjusting the stack pointer
$r[15] := $r[15] - 20;

# retrieve parameters into variables
$t[0] := $m[$r[15] + 0];
$t[1] := $m[$r[15] + 4];
$t[2] := $m[$r[15] + 8];
$t[3] := $m[$r[15] + 12];

# compute and return the appropriate value
$m[$r[15] + 20] := ($t[0] - $t[2]) + ($t[1] - $t[3]);
```

Figure 4.5: RTLs with temporaries produced from the C code in Figure 1.2 on page 11

Chapter 5

From RTL to Machine Code

5.1 Motivation

The code expander runs after temporary locations have been assigned to variables. It is charged with finding a sequence of machine instructions that implement an arbitrary program statement. More formally, the code expander takes as input an arbitrary source RTL and outputs a sequence of RTLs such that the execution of the sequence has the same observable effect as the execution of the original RTL and each RTL in the list can be represented by an instruction on the target machine. As we have mentioned with `vpo`, we call this latter property the machine invariant (Benitez and Davidson 1988).

We desire also to classify at compile-compile time the set of RTLs that our code expander is able to translate. Such a categorization is crucial to compiler authors writing a mapping from their intermediate representation to RTLs. If such an author ensures that the RTLs generated by his front end all fall within our set of translatable RTLs, then we guarantee to the author that those RTLs will be successfully compiled to machine instructions.

To generate code, we must know something about the semantics of the operators involved. To know that we must use the SPARC's `orn` instruction `$r[rd] := $r[rs1] \vee \neg $r[rs2]` with `$r[rs1] = 0` in order to perform bitwise-complement, we either must have this particular knowledge about this instruction, or else we must know that $0 \vee x = x$. The first requires *a priori* machine-dependent knowledge. The

second is a universal truth. Thus we approach instruction selection by investigating the properties of operators' semantics, independent of any particular machine. By applying this knowledge to a particular machine, we glean the machine-specific knowledge we need to generate code.

5.2 Analysis

Following our analysis in the last chapter, we now must deal with the challenge of translating source RTLs *with no variables* into sequences of machine instructions. This challenge boils down to being able to move the result of an arbitrary computation into any arbitrary location on the machine. Our framework here does not deal with either guards or simultaneous effects. We have not taken up either of these issues within this paper, although we do address them briefly in Chapter 7.

As we develop techniques for translating code, we wish to have a framework for proving which computations we are able to translate. We approach this goal via structural induction over an RTL, with the aim of proving:

We can get the result of any computation into any location of the appropriate size.

Instrumental in establishing this result is the data-movement graph discussed in Section 4.2. If we determine that we can get a certain value into a location within location set l , then the data-movement graph immediately tells us that we can get that value into any location within any location set reachable from l in the data-movement graph.

Ideally, we would like the set of read-write location sets within the data-movement graph to be strongly connected.¹ This would indicate that we have a way to move a value from any machine location to any other machine location. Additionally, we would want every read-only location set to contain an edge to at least one read-write location set, indicating that we can move constant values into any location in the machine. Finally, we would want every write-only location set to have an incoming edge from at least one read-write location set, indicating that we can use values stored

¹An alternative view observes that we would like the cost graph—the transitive closure of the data-movement graph—to contain a clique among all the read-write location sets.

in arbitrary locations as addresses into memory. For now, we assume these properties to hold. Below, we discuss how the same techniques that we use in dealing with operators can also be exploited to expand the data-movement graph to fulfill these desired properties.

Following this assumption, the proposition that we are attempting to prove is reduced to:

We can get the result of any computation into some location of the appropriate size.

With this in mind, we observe that the denotation of an expression is a value, and that there are three possible kinds of expressions in our RTL formalism. A value can be fetched from a location, a value can be a constant, or a value can be the result of applying an operator to one or more other values. We also make the distinction here between fetching from fixed or register-like locations and fetching from arbitrary memory locations. Constants and fetches from either fixed or register-like locations comprise the base case of our inductive framework. Applications of operators to other expressions are our inductive step and also establish our ability to fetch values from arbitrary memory locations.

Both base cases need only our data-movement graph. The edges of this graph are exactly those pairs of location sets $(l1, l2)$ for which we can move a value from any location in $l1$ to any location in $l2$. As we include constants (discriminated by width) as read-only location sets within this graph, the graph establishes our base case. We can immediately add to our set of translatable values all X -bit constants such that $(conX, y)$ is an edge in the data-movement graph for some y . Similarly, we can add all read-write locations from any location set l such that (l, y) is an edge in the data-movement graph for some y .

The inductive case requires us to show that we can translate the application of a given operator to one or more operands. By our inductive hypothesis, we assume that we are able to move the values of all operands into any location on a machine, and hence we need only find some way to implement each possible operator, given that the operands are within some read-write location sets.

Before continuing, it is reasonable to ask what is meant by “each possible operator.” Recall from Section 2.2 that operator definitions may be polymorphic in the

widths of their operands, such as in the type of `add`: $\forall n \text{ \#}n \text{ bits} * \text{ \#}n \text{ bits} \rightarrow \text{ \#}n \text{ bits}$. An instruction set could theoretically contain an instance of the operator `add \#n` for any value of `n`. As such, there are an infinite number of possible operators; we use a combination of guesswork and user interaction to limit the number of operators for which we attempt to find implementations.

On an 8-bit machine, we would not expect to find an instruction to implement 32-bit addition, but we would expect such an instruction on a 32-bit machine. For operators with types that involve a single width variable, such as `add`, we seek that operator specified to the width of the target machine's word size. For example, on a 32-bit machine, we seek an implementation of `add \#32`, while on a 64-bit machine we seek `add \#64`.

Some operators contain more than one width variable. For example, the `mul` multiplication operator has type $\forall n, m \text{ \#}n \text{ bits} * \text{ \#}n \text{ bits} \rightarrow \text{ \#}m \text{ bits}$. For this operator we would be interested in the specialized forms

```
#32 bits * #32 bits -> ? bits
? bits *    ? bits -> #32 bits
```

In practice, of course, $m = 2n$ and hence the first form above is 32-bit multiplication while the second form is 16-bit multiplication. However, this constraint is outside of the RTL type system, and so we will rely on the user to inform us which values we should be seeking for `?`. Via this combination of heuristics and user interaction, we can ensure that we only seek implementations of a finite number of operators.

Thus, it seems that to be able to expand arbitrary source computations, we must find within an instruction set a way to implement every operator. We employ three techniques in this search:

1. *Direct implementations.* The SPARC `add` instruction, `$r[rd] := $r[rs1] + $r[rs2]`, is a direct implementation of `add \#32`. The most straightforward way of implementing an operator is if the architecture provides an instruction that does nothing other than compute that operation. If we find such an instruction for a particular operator, no additional work need be done at compile-compile time for that operator.
2. *Algebraic laws.* For instructions that involve operators other than the operator in which we are interested, we can attempt to exploit algebraic laws to transform

Name	Law	What We Need
Identity	$\oplus(v_1, \dots, v_{k-1}, x, v_{k+1}, \dots, v_n) = x$	$k, (v_1, \dots, v_{k-1}, v_{k+1}, \dots, v_n), n \geq 2$
Inverse	$f(\oplus(x)) = x$	f , where \oplus does not appear in f
Rewrite	$\oplus(x_1, \dots, x_n) = E$	E , where \oplus does not appear in E

Figure 5.1: Using algebraic laws to eliminate the effect of operators. The equations are universally quantified over x and the x_i 's, but not over f , E , \oplus , or the v_i 's.

the effect of the instruction to that of our desired operator. We can divide the use of algebraic laws into three subcategories:

- Identity laws. These laws can be used to eliminate the effect of certain operators within an instruction. For example, the SPARC's `orn` instruction, `$r[rd] := $r[rs1] \vee \neg $r[rs2]`, can be used to implement the bitwise complement operator (\neg) if we know the bitwise-or identity: $x = 0 \vee x$. By this identity, $\neg x = 0 \vee \neg x$.
- Inverse laws. These laws can be used to eliminate the effect of unary operators within an instruction. For example, \neg is its own inverse: $\neg(\neg x) = x$. Using this inverse, we could use `orn` to implement bitwise-or if we have an implementation of bitwise-complement: $x \vee y = x \vee \neg(\neg y)$.
- Rewrite laws. In this case, we are seeking laws that allow us to implement one operator in terms of others. On an architecture that contains integer multiplication and division instructions, we could use this instruction to implement `mod`, the modulus operator, if we know that $x \bmod y = x - x * (x/y)$.

These three categories are summarized in Figure 5.1. In this figure, the x 's represent variables, the v 's represent values, and n is the arity of \oplus . E represents an arbitrary expression and f stands for an arbitrary function of one variable. Each equation should be considered universally quantified over the x 's—that is, the laws must hold for any values of the x 's. This table makes clear what is meant by “eliminating the effect of an operator.” For all three types of laws, the operator of interest— \oplus —does not appear on the right-hand side of the law. \oplus 's semantics have been eliminated.

For inverses and rewrite laws, knowing the law is not sufficient for being able

to use the law. To use an inverse, we must know how to implement every operator involved in the function f . To use a rewrite law, we must know how to implement every operator within E .

3. *Unwanted side effects.* Some instructions may implement a desired operator and yet also have other effects, such as performing another operation or setting the condition codes. We can use such instructions to implement an operator if we are willing to introduce these unwanted side effects—which must then be dealt with. On the Pentium, almost all arithmetic and logical instruction (for example `add` and `adc`) also sets the status flags (condition codes). We could use these instructions as regular implementations of the logical operators, provided we then mark the status flags invalid in some manner. (We see how this is done below.)

Of course, the use of algebraic laws and the introduction of unwanted side effects are not mutually exclusive. We may find that the only viable implementation of a given operator requires us to both exploit one or more algebraic identities and also to deal with unwanted side effects. By using these techniques, we hope to find *some* implementation for every operator.

Returning to the question raised above about the properties of the data-movement graph, we now see that these same techniques are applicable for attempting to strongly connect the read-write location sets and find edges out of read-only location sets and into write-only location sets. The data-movement graph as presented in the previous chapter discovers only those instructions that do nothing more than move a value between locations. However, algebraic identities may allow us to utilize other instructions for data movement. Thus, we can use the SPARC's or-immediate instruction `$r[rd] := $r[rs1] ∨ simm13` in conjunction with the right identity on \vee — $x \vee 0 = x$ —in order to use this instruction to move values between integer registers. Thus, algebraic identities allow us to discover the self-loop on `$r[rs1]` where `ne(rs1, 0)` that we observed was missing in the original data-movement graph (Figure 4.1 on page 38).

Similarly, by introducing unwanted side effects we can use an instruction for data movement even if it has other effects as well. For example, the VAX move instructions

(`movab`, `movaw`, `moval`, etc.), all move data between locations with one effect, while also setting the condition codes.

Once we have discovered methods for translating operator applications and have strongly connected our data-movement graph, we can also translate fetches from memory locations indexed by arbitrary expressions. We know at this point that we can get the value of an arbitrary addressing expression into any read-write location. But we also know that every write-only location set (which consists of the addressing expressions recognized by the target machines' instructions) has at least one edge into it from some read-write location set. By combining these two results, we conclude that we can fetch values from arbitrary memory locations.

Thus, if our compile-compile time analysis connects the data-movement graph, and finds implementations of every operator, it can prove its ability to place the results of arbitrary computations in arbitrary locations on the machine. Even if the graph is not completely connected or not every operator is implemented, the framework presented here will allow the code-generator generator to produce a precise characterization of those RTLs that the code expander is able to translate.

Our implementation of this analysis is split into two parts. In the first section, we seek instructions to implement operators. In the second section, we use this information and the entire instruction set to emit BURG rules for a BURG engine. This entire process occurs at compile-compile time.

5.3 Operators: Implementation

On the most general level, our code-generator generator must emit BURG rules such that a BURG engine using these rules will be able to cover as many source RTLs as is possible. Before we can emit these BURG rules, however, we must perform the operator-based analysis discussed above. The particular implementation specified here includes interactions with a user at compile-compile time. We archive the results of these interactions, however, in order that the amount of interaction needed will decrease with subsequent runs of our system.

The general strategy behind the implementation of this analysis is to find ways of implementing every operator by looking individually at the RTLs that make up

the target instruction set. The overall operator-analysis process is presented in Figures 5.3-5.7 in pseudocode. The process consists of three phases. In the first phase, we identify direct implementations within the instruction set, and also identify candidate operator implementations. In the second phase, we attempt to satisfy the identities and inverses that we identified as necessary for candidate implementations in phase one, first via our archive and second—if necessary—via interaction with the user. In the final phase, we ask the user for rewrite laws to implement those operators for which we have not yet found implementations.

We associate each operator with a list of implementations. Each implementation includes information on the location sets of the operands and result of the operator, the instruction that provides the implementation, as well as additional information that may relate to that particular implementation (more on this later). As we proceed, we move operators between four sets:

- *K, the set of known operators.* This set contains operators associated with satisfactory implementations. We know how to compute operators in *K*, but there may still be better implementations of these operators.
- *P, the set of preferred operators.* This set contains operators associated with implementations that we prefer over other implementations. We need not search for additional implementations of operators in *P*. The distinction between the sets *K* and *P* is, to some extent, the distinction between what operators we can translate and what operators we can translate efficiently.
- *C, the set of candidate operators.* This set contains operators that either have no associated implementations, or have associated implementations predicated on conditions that have not yet been fulfilled.
- *U, the set of unwanted operators.* This set contains operators that the user has indicated we need not implement.

All operators begin in *C*, associated with no implementations. To fully establish the inductive proof discussed above, every operator must end up in either *K* or *P*. The relation between these four sets is depicted as a state diagram in Figure 5.2.

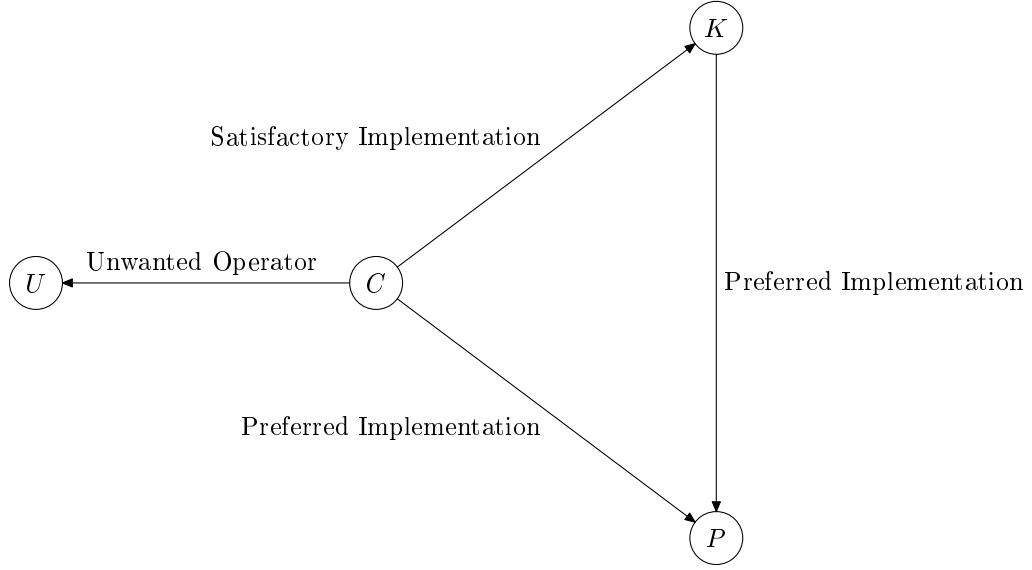


Figure 5.2: The four operator-implementation sets

5.3.1 Phase one

In phase one, we check each instruction within the target machine's instruction set. For each operator within each instruction, we check whether and under what conditions the instruction implements the operator for each of our three strategies: directly, using algebraic laws, and with unwanted effects. Figure 5.3 contains pseudocode for this phase.

As we did with constructing the data-movement graph, recall that we can write the general form of an RTL representing a machine instruction as a list of simultaneous effects:

$$g_1 \rightarrow l_1 := e_1 \mid g_2 \rightarrow l_2 := e_2 \mid \dots \mid g_n \rightarrow l_n := e_n$$

We speak of a single *effect*, i , as a direct implementation of an operator \oplus if g_i is always true and if e_i has the form $\oplus(e_i^1, \dots, e_i^k)$ and each of the e_i^j is a value fetched from a location. To be able to perform arbitrary computations of \oplus , we require that each operand be a value from a location, rather than *any* leaf node (which would include constants). This restriction, in conjunction with the data-movement graph (as discussed above), ensures that when we find a direct implementation of \oplus we can compute \oplus with arbitrary operands.

Therefore, given an instruction and an operator, we can check if the instruction is

Entries in the four sets C , K , P , and U are of the form $(op, inst, laws, locs)$ where op is the operator, $inst$ is the relevant instruction $laws$ is a possibly empty set of algebraic laws (either actual or desired laws) that this implementation is predicated upon, and $locs$ is a possibly empty set of locations affected by effects of $inst$ other than the effect that implements op . Every law in $laws$ is marked as either ‘found’ or ‘desired’. Found laws are of the form ‘ $0 \vee x = x$ ’ while desired laws can be identities, inverses, or rewrite laws.

```

for each operator  $\oplus$  do
   $C \leftarrow (\oplus, \text{none}, \emptyset, \emptyset)$ 
for each instruction  $i$  do
  for each operator  $\oplus$  in  $i$  do
    if  $i$  is a direct implementation of  $\oplus$  then
       $P \leftarrow (\oplus, i, \emptyset, \emptyset)$ 
      Remove  $(\oplus, \text{none}, \emptyset, \emptyset)$  from  $C$ 
    else if  $\oplus$  is not in  $P$  then
       $Laws \leftarrow$  Laws needed to eliminate effects of operators other than  $\oplus$ , if any
       $Locs \leftarrow$  Locations affected by unwanted effects, if any
       $C \leftarrow (\oplus, i, Laws, Locs)$ 

```

Figure 5.3: Phase one of the operator-analysis procedure. Glean operator information from the instruction set to identify preferred and candidate implementations.

a direct implementation by checking that $n = 1$ and that the single effect fulfills the above criteria. For such a direct implementation, our code-generator generator need only identify the relevant instruction and operator. If we find a direct implementation of an operator \oplus , then we add \oplus to P . For such a direct implementation, it is straightforward to record the location sets from which \oplus ’s operands are taken and into which its result goes.

We approach the use of algebraic laws differently. Rather than assume that we know every algebraic law that might ever be of use, we attempt to discern whether a given instruction might potentially implement a given operator if the proper algebraic laws were available. Given the **orn** instruction discussed above, we identify that it can implement \neg if we can find a left identity on \vee . Also, **orn** could be used to implement \vee if we can find an inverse of \neg . That is, the ability to solve $z = \neg y$ for y given z would allow us to use **orn** as an implementation of \vee . Until these conditions are satisfied, these implementations are only candidates, and remain in set C . If we were to find a left identity on \vee we would add \neg to K associated with the **orn** instruction

and the \vee identity.

In this case we specifically need a *left* identity on \vee . Consider a fictitious multiply-add instruction of the form $\$r[rd] := \$r[rs1] + (\$r[rs2] * \$r[rs3])$. As above, a left identity on $+$ would allow us to use this instruction to implement multiplication. If we wish to use this instruction to implement addition, however, we can use *any* identity on $*$ —left or right.

Based on this intuition, we wish to develop general rules for determining the specific algebraic-law conditions on which a particular operator implementation depends. Given an RTL r and an operator \oplus that appears in r , we must find appropriate laws to allow us to eliminate the effect of all other operators that appear within r . For a unary operator \odot within r , this simply means finding an inverse of \odot . For an operator \odot with more than one operand, we must find an identity on \odot ; but do we need a specific identity, or will any identity do? To answer this question, we must consider the relation between \oplus and \odot within the RTL tree. There are two possibilities:

- \oplus is a descendant of \odot in the tree. In this case, \oplus (the operator we are attempting to implement) occurs within one of \odot 's operands. Because of this, we are *not* free to find any identity on \odot . Instead, we must search for an identity that preserves the operand of \odot that contains \oplus . This corresponds to the case above of using the multiply-add instruction to implement multiplication. The first tree in Figure 5.4 illustrates this relationship between the desired operator \oplus and another operator \odot .²
- \oplus is an ancestor of \odot or the two have no ancestral relation in the tree. In this case, we need only eliminate the effect of \odot in some manner, and hence any identity will suffice.³ The second two trees in Figure 5.4 illustrate the relationships between \oplus and \odot that allow us to use any identity on \odot .

²This case holds whenever \oplus is a descendant of \odot , not only when \oplus is a child of \odot . Consider a situation in which \oplus is buried deep within a subtree of one of \odot 's operands. To use this tree to compute \oplus we need the computation of \oplus to percolate up the tree to the top. Eventually, it will have to percolate through the operand of \odot in which it is embedded. To do this, we need an identity on \odot that preserves this particular operand.

³Actually, we restrict ourselves to finding an identity on \odot that preserves an operand that is either a read-write location set or else is itself the result of another operator application. This helps ensure that the eventual implementation of \oplus takes all operands from read-write location sets.

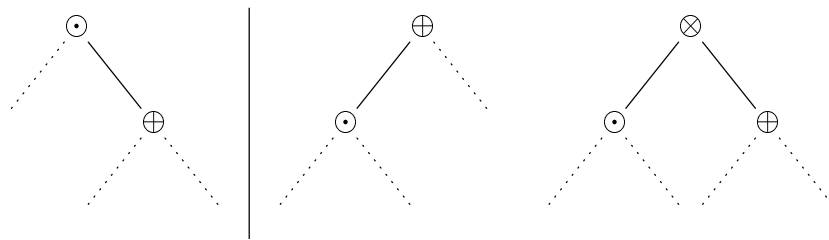


Figure 5.4: Possible operator relationships within an RTL

Until we determine the actual identities and inverses that would allow us to use this implementation, we cannot predict the location sets that this implementation would use for the operands and result of \oplus . We leave \oplus , associated with the appropriate conditions, within C . In the case of the SPARC `orn` instruction discussed above, we would keep \neg in C , associating it with the prerequisite of a left identity on \vee . We would also keep \vee in C , associating it with the conditions of an inverse of \neg .

In this phase, we are concerned only with algebraic identities and inverses (the first two entries in Figure 5.1), and not with those laws that rewrite one operator in terms of others. We mention this latter case below, in our phase-three interactions with the user.

Recognizing an implementation of a given operator with unwanted side effects is much simpler. Given an RTL with multiple effects, we simply determine if one of the effects is a direct implementation of the operator. If so, we examine the instruction's other effects, noting the location affected by each. If we use this instruction to implement this operator, we will need to save and restore the values in any locations affected by the side effects.

An operator that may be implemented with the introduction of unwanted side effects is put into K , along with the relevant instruction and a list of other affected locations. We now know how to implement this operator, but we may still encounter a better implementation.

Once we can recognize these three implementation methods, we can also easily recognize an instruction that may implement an operator via both the use of algebraic laws and the introduction of unwanted side effects. Such an implementation remains in C —along with the information that we record for both of the individual strategies—until the necessary algebraic laws are found, at which time it is moved to K . The procedure for moving implementations from C to K is given in pseudocode

in Figure 5.7 and is explained below.

By examining every instruction in this manner, we build up a large set of possible implementations for each operator. However, we do not want to include two possible implementations in our BURG rules when it is clear that one implementation is superior to another. For example, given the SPARC’s `add` instruction that does a direct implementation of addition, we do not also want to include `addcc`—addition with the unwanted effect of setting condition codes—as an implementation of addition. We cannot, however, simply compare any two implementations and decide which we prefer; we do not know which algebraic laws we may successfully find, nor which of two potential implementations may be cheaper.

We are able, however, to define a simple partial order that avoids obvious redundancies. Namely, we prefer a direct implementation over all other strategies. We make no preference between the other strategies. This partial order, then, is the purpose of differentiating between the known operators set K and the preferred operators set P . If we find that an operator has an implementation within P , we no longer seek implementations of it within K . That is, if we have found a direct implementation of an operator, we no longer search for algebraic-law or side-effects-based implementations of that operator. Thus, when we discover that the SPARC’s `or` instruction provides a direct implementation of \vee , we remove the entry for \vee in C that we created above from the `orn` instruction.

We also note that we can distinguish between implementations within a particular strategy. We classify every potential implementation of an operator not only based on the strategy involved but also based on the location set of the result of the operation. In general, we prefer operator implementations that place results in large register-like location sets. The smaller the cardinality of a location set involved in an operator implementation, the more likely it is that the register allocator will need to spill and reload locations in the set. We can cope with this by assigning a higher BURG cost to implementations that place operations’ results in small location sets.⁴ The necessity and value of such a cost are experimental questions that we have not yet answered.

⁴Of course, the lowest of cardinalities is a register-like location set of size one—namely, a fixed location set. If we include an operator implementation that uses a fixed location set, we must promote that fixed location set in order that we can generate code with temporaries for it. See Section 4.3 for more information on promoting a fixed location set.

```

SatisfyCandidateImplementationsViaArchive()
for each law still unsatisfied in C do
  Ask user for law
  if user provides the law, law' then
    Mark law found as law' in Archive
  else if user guarantees law is not satisfiable then
    Mark law not satisfiable in Archive
    Remove all entries from C that contain law
  else
    Remove all entries from C that contain law
SatisfyCandidateImplementationsViaArchive()

```

Figure 5.5: Phase two of the operator-analysis procedure. Search the archive to satisfy laws needed for candidate operations. If there are still laws that we do not know, we ask the user for them and remember the response in the archive.

5.3.2 Phase two

In phase one, we identify the laws that we would need to use certain instructions to implement certain operators. At this point, we enter a phase in which we attempt to satisfy as many of these conditions on implementations in *C* as possible. First, we scan our archive to see which of the desired laws we already know. Alternatively, we may already know that a desired law does not exist at all.

If we are unable to find all laws in our archive, then we conduct an interactive session that allows us to gather laws from the user. As a user is likely to use this system to retarget a compiler for multiple machines, we do not only gather these laws to produce a single code generator. Instead, we archive the knowledge that we gain from the user, and use this knowledge in code-generator generation for future target machines. In this way, we reduce the length of this interactive phase on subsequent runs of our system.

As we discover new algebraic laws, we look through *C* and our archive to determine which conditions have been satisfied. For a condition to be satisfied, we must have found the desired law, and we must already know implementations for any operators that appear on the right-hand side of the law that we have found (see Figure 5.1). Whenever the last condition on an operator implementation is satisfied, we move that implementation from *C* into *K*. Figure 5.7 gives the pseudocode for the procedure

SatisfyCandidateImplementationsViaArchive() to perform this check.

In this phase, we seek those identities and inverses identified in our analysis. We begin by searching those laws that we have archived from past runs. Assuming that this search still leaves some identities or inverses unfulfilled, we then prompt the user and ask for the desired laws. The user may give one of three answers:

- The user may provide the desired law. In this case, we add it to our archive, and remove it as a condition in C wherever it appears,⁵ moving an implementation to K if no other conditions remain. Consider the case of our fictitious multiply-add instruction $\$r[rd] := \$r[rs1] + (\$r[rs2] * \$r[rs2])$. To implement multiplication using this instruction, we ask the user for a left identity on $+$. If the user informs us that $0 + x = x$, then we may move this instruction from C to K , associating it with this law. (In terms of Figure 5.1, we would note that $k = 2$, and $v_1 = 0$.)
- The user may not know of an appropriate law. In this case, we search through C for any implementations that is predicated on this law. If we find any, we remove them from C . We do not make changes to our archive. If a user is unable to provide a left identity on \vee (and we do not have one in our archive), then we would be unable to use $\$r[rd] := \$r[rs1] \vee \neg \$r[rs2]$ to implement bitwise-complement, and so we remove this possibility from C .
- The user may guarantee that no such law exists. We proceed as in the previous case, removing implementations that require this condition from C . We also archive the fact that this identity or inverse does not exist; on future runs, we will not search for such a law in the first place. If a potential operator implementation was predicated on a left identity of integer division (that is, a v such that $v/x = x$ for all x), the user might inform us that no such identity exists. We remove this candidate implementation, and also record this fact for all posterity.

Returning to our example of the `orn` instruction, we concluded before that with

⁵The pseudocode in Figures 5.3-5.6 waits until the end of each phase to check which conditions have been satisfied. An actual implementation would be more efficient checking as new laws are discovered, as is presented in the text.

```

for each  $(\oplus, \text{none}, \emptyset, \emptyset)$  in  $C$  do
  Ask user for a rewrite law for  $\oplus$ 
  if user provides a rewrite law,  $law$  then
    Add  $law$  to Archive
  SatisfyCandidateImplementationsViaArchive()

```

Figure 5.6: Phase three of the operator-analysis procedure. Ask the user for rewrite laws for operators for which we have not yet found implementations. Remember these rewrite laws in the archive.

```

procedure SatisfyCandidateImplementationsViaArchive()
  for each  $(\oplus, i, Laws, Locs)$  in  $C$  do
    if  $Laws$  are found in the Archive and
      all needed operators for  $Laws$  are in  $P$  or  $K$  then
         $K \leftarrow (\oplus, i, Laws, Locs)$ 
        Remove  $(\oplus, \text{none}, \emptyset, \emptyset)$  from  $C$ 
    if  $Laws$  are found to be not satisfiable in the Archive then
      Remove  $(\oplus, i, Laws, Locs)$  from  $C$ 
  Repeat the procedure until no new implementations are satisfied
end procedure

```

Figure 5.7: This procedure determines which candidate implementations in C are satisfied by algebraic laws in our archive and moves them to K , the known implementations set. It also determines (from the archive) which implementations are not satisfiable and removes them from C .

a left identity of \vee , we could implement \neg . If our archive or the user is able to tell us the left identity, $x = 0 \vee x$, then this condition is satisfied, and we can move this implementation of \neg into the known operators set, K .

5.3.3 Phase three

Once we have found all of the identities that we can, we may still have a set of operators that we cannot yet implement. For each of these, we ask the user whether there is an algebraic law that would allow us to rewrite this operator in terms of another. For example, if we have not found an implementation of \neg , the user may inform us that we can rewrite $\neg x$ as $-1 - x$. If we already have an implementation of subtraction in K or P , then we may now add \neg to K , associating it with this rewrite law. If we do not yet have an implementation of subtraction, then we would

put this implementation of \neg into C , predicated on a subtraction implementation. Another example would rewrite the modulus operator in terms of integer multiplication and division by noting that $x \bmod y = x - x * (x/y)$. In this case, our potential implementation of `mod` would have two conditions: an implementation of `*` and an implementation of `/`. We would move this implementation from C to K only when both of these conditions were satisfied. By thinking of such conditions as edges within a directed graph, we can direct our search for further operator implementations by querying first for an operator with a minimal number of predecessors in such a graph.

Throughout this process, a user may inform us that he is uninterested in a particular operator. We then place that operator into U , and remove all candidate implementations in C that depend on it. Eventually, we will have no more laws to exploit, and we will have to discard any operators that have not found their way into K or P . We now use the information we have gleaned to emit the BURG rules that drive the code expander.

5.4 Burg Rules: Implementation

We have already discussed BURG as a member of the family of systems that use dynamic programming over a tree-based intermediate representation to produce locally optimal code. Our work makes use of a BURG engine to perform the actual code expansion, and as such the general structure of BURG deserves more mention (Fraser and Hanson 1995). A BURG engine is driven by a set of rules similar to those within a context-free grammar. Each rule consists of a non-terminal symbol, a tree fragment, a cost, and associated semantics, such as:

```
reg: reg + scon13  (1) = <<semantics>>
```

Such a rule might be produced for an architecture's add-immediate instruction. It instructs the engine that it can match the tree pattern `reg + scon13` and replace it with the non-terminal `reg` at a cost of 1. As a tree is covered, BURG composes the associated semantic actions. Eventually, the semantic functions that correspond to the cover with the least cost are executed. We discuss these parts in more detail below, in the context of our work.

While much of the operator analysis above has been implemented (the interactive phases have not yet been implemented), the production of BURG rules described in this section has not yet been implemented. We explain here the algorithm we plan to use to create BURG rules from the operator analysis already presented.

All BURG rules consist of the same four parts, a non-terminal symbol, a tree fragment to be matched, a cost, and associated semantic code. Before discussing the details of the rules we emit, it is helpful to discuss the meaning of the non-terminal symbols. There are three types of non-terminal symbols:

- The *top-level* non-terminal, `stmt`. The `stmt` non-terminal is equivalent to the start symbol in a grammar. Top-level rules represent a tree fragment matched for no other purpose than its effect on the machine state. The tree fragments associated with `stmt` non-terminals consist of an assignment to a location, reflecting this change of state. When the BURG engine picks rules to cover a source RTL, the top-most rule picked will be a `stmt` non-terminal. As we have no interest in these rules other than their side effects, the semantic action associated with them is nothing more than a sequence of machine instructions needed to effect the state change.
- The *production* non-terminals. Each production non-terminal corresponds to a location set that we may want to use to hold the intermediate results of a computation. We will have one production non-terminal per location set that is used in some instruction as the destination of an assignment. The associated tree fragment tells us an expression that may produce a value in a location within this location set. For example, the tree fragment `reg + reg` being associated with the production non-terminal `reg` might tell us on the SPARC that we can develop the sum of two values stored within the location set `$r[x]` where `x` \neq 0 into a location within that same location set. For these rules, we are interested both in the sequence of instructions that develops a value into the location set associated with the non-terminal and also in the (hardware or temporary) location in which the value is produced. The semantic code associated with a production non-terminal must return both pieces of information.

- The *address* non-terminals. An address non-terminal matches expressions that may be used to address memory-like location sets. The associated semantic code must return an expression that is a valid memory address expression, as well as a sequence of instructions necessary before the expression can be used. On an architecture that allows only register-register addressing, an address non-terminal may match the tree fragment `reg`. The semantic code would need to indicate that a single register can be used as an address if the constant zero can be loaded into another register. The expression returned would then be the sum of the two registers, and the prerequisite instruction might be a load immediate of zero into the second register.

Examples of all three types of non-terminals are below. While most BURG rules that we emit fall into one of these three categories, there are other rules necessary to fit things together. We present such rules as they are necessary in the examples that follow.

For the SPARC `add` instruction `$r[rd] := $r[rs1] + $r[rs2]`, we would want to include the general rule,

```
reg: Apply(("add", [32]), [reg_0, reg_1]) (1) = <<semantics>>
```

telling us one possible way to develop a value into the `reg` temporary class. The `reg` non-terminal corresponds to the location set `$r[x]` where `x ≠ 0`. We annotate non-terminals on the right-hand side of a rule with numbers in order that we can uniquely refer to them within the semantic code. The BURG engine ignores these annotations in identifying the non-terminals. For the rest of our examples, we simplify the full patterns for readability. The above example becomes:

```
reg: reg_0 + reg_1 (1) = <<semantics>>
```

The semantics associated with this rule would allocate a temporary for the result of this computation and specify the RTL that implements it. We would produce code for these semantics such as:

```
let val (t0, i0) = reg_0
    val (t1, i1) = reg_1
    val t2 = getreg #"t"
```

```

in
  (t2, (Sparc.add (to, t1, t2)) :: (i1 @ i0))
end

```

The first two lines of this code decompose the right-hand side non-terminals into the locations they represent and the list of instructions required to develop a value into that location. The third line allocates a new temporary for the result. The code snippet returns the location that stores the result of the addition (`t2`). We also add the `add` instruction to the lists of instructions that we already have for `reg_0` and `reg_1` and also return this list (in reverse order). We use an abstract `Sparc.add` function as we do not care whether we actually produce RTLs, assembly code, or some other representation at this point.

We also need to indicate that addition can be a top-level effect. We include the top-level rule:

```
stmt: reg_lhs_0 := reg_1 + reg_2 (1) = <<semantics>>
```

telling us that an instruction can be the storing of the addition of two values stored in the `reg` temporary class into a *specific* location within that temporary class. Whereas the `reg` non-terminal represents any expression that can develop a value into a register, the `reg_lhs` non-terminal represents a register *l-value*—either an actual hardware register within the location set $\$r[x]$ where $x \neq 0$ or else a temporary location associated with this location set.

When faced with a source RTL in which `+` is the upper-most operator, BURG's algorithm is able to utilize this rule if `+`'s operands can be developed into locations within the `reg` class, and if the destination of the computation is in the location set $\$r[x]$ where $x \neq 0$, or in the temporary space associated with this location set. The semantics for a top-level rule such as this need only prepend the `add` instruction onto the instruction lists associated with `reg_lhs_0`, `reg_1`, and `reg_2`.

We have already seen that each temporary class corresponds to a set of actual hardware registers, and in particular that in our ongoing examples the `t` temporary class corresponds to the $\$r[x]$ where $x \neq 0$ location set on the SPARC. Not only must our `reg` non-terminal be able to refer to the temporary locations, they also must be able to refer directly to the actual registers in `r-space`. Thus, we must include the following rules:

```

reg_lhs: $t[x]                (0)          = $t[x]
reg_lhs: $r[x]                (<<cost>>)    = $r[x]
      reg: Fetch(reg_lhs_0, 32) (0)          = (reg_lhs_0, [])

```

The semantics for the first two rules simply return the appropriate locations, while the semantics for the third rule returns the location along with an empty list of instructions. The cost for the second rule depends on the value of x . If x is within the range of the desired location set—in this case $1 \leq x \leq 31$ —then the cost is zero; otherwise the cost is infinity.

More generally, for every non-terminal that corresponds to a class of temporaries we also create a left-hand-side (lhs) non-terminal that represents either the temporaries or the hardware registers. We also add the third rule above to specify that the main non-terminal (`reg`) is a superset of the left-hand side version (`reg_lhs`).

We emit a top-level rule for *every* instruction within an instruction set. We also emit production rules for all instructions within an instruction set that contain a single effect.

We also must emit rules for all of the operator implementations found in the previous analysis. There is no need to take special action for direct implementations, as these correspond directly to unmodified instructions, and hence we already have rules for them. We must emit special rules for those operator implementations derived from algebraic laws or the introduction of unwanted side effects. We present several examples of the various forms of these rules.

Consider the use of the SPARC `orn` instruction to implement bitwise negation, as discussed previously. Our analysis above tells us that we can implement \neg from `$r[rd] := $r[rs1] \vee \neg $r[rs2]` using the identity $x = 0 \vee x$. We emit the following production rule. (We also emit a very similar top-level rule that differs from this rule in exactly the same manner as the top-level and production rules we have already seen.)

```

reg:  $\neg$ reg_0 (2) = <<semantics>>

```

with semantics:

```

let val (t0, i0) = reg_0
    val (t1, i1) = moveConst (0, "cons22", "reg")

```

```

    val t2 = getreg #"t"
  in
    (t2, (Sparc.orn(t1, t0, t2)) :: (il @ i0))
end

```

We create the `moveConst` function from the transitive closure of the data-movement graph at compile-compile time. It yields the instructions necessary to move a constant value into a location within a location set, and also the resulting location. We also create a similar function, `moveValue` to move a value stored in a particular location into a location within a given location set. Both functions are derived in a straightforward manner from the data-movement graph once the Chapter 4 version has been completed using the techniques given in this chapter. The cost for this rule is 2, as we must take into account the instruction needed to move zero into a register.⁶

We next consider the structure of a BURG rule for an operator implementation that involves the introduction of extra side effects. Ignoring the presence of the SPARC's standard `add` instruction, consider the `addcc` instruction, which performs an add and also sets the condition codes, which are stored within the processor state register, `$i[0]`. To handle the unwanted effect, we must promote the fixed location set `$i[0]` (see Section 4.3) in order to associate it with a class of temporaries. This allows the register allocator to spill and reload a value in `$i[0]` that would otherwise be overwritten by our use of `addcc`. Assuming that we associate the temporary j-space with `$i[0]`, we would emit the following production rule (and, again, a very similar top-level rule):

```

reg: reg_0 + reg_1  (1) = <<semantics>>

with semantics

let val (t0, i0) = reg_0
    val (t1, i1) = reg_1
    val t2 = getreg #"t"
    val t3 = getreg #"j"
  in
    (t2, (Sparc.addcc'(t0, t1, t2, t3)) :: (il @ i0))
end

```

⁶ We would, of course, actually like to use `$r[0]` for the value zero, rather than loading zero into some other register. Luckily, a peephole optimizer combined with a recognizer will recognize this and optimize away the load immediate. The BURG engine needn't do all the work!

As with our other examples, `Sparc.addcc'` is an abstract procedure that generates some representation of the `addcc` instruction. It also accepts an extra parameter representing a temporary to substitute for `$i[0]`.

We conclude this section by examining three more rules. The first two demonstrate the structure of rules that have an address non-terminal, while the third shows the use of this address non-terminal in a rule for a load-from-memory instruction.

Consider the SPARC's register-register addressing mode. We create the following BURG rule for it:

```
addr: reg_0 + reg_1 (0) =

let val (t0, i0) = reg_0
    val (t1, i1) = reg_1
in
    (t0 + t1, i0 @ i1)
end
```

There are two differences between this rule and previous examples we have seen. First, this rule does not require any instructions beyond those needed to get values into `reg_0` and `reg_1`, and hence it has a cost of zero. Second, rather than returning a location, as is the case for our production rules, we instead return an address expression.

Recall that one of the manners in which we use the techniques presented in this chapter is to ensure that every write-only (address) location set in the data-movement graph contains at least one incoming edge from some read-write location set. Via the algebraic identity on addition, $x = x + 0$, we could create an edge from the `$r[x]` where $x \neq 0$ location set to the address location set containing (in part) the base-displacement addressing mode. From this edge, we create the following rule:

```
addr: reg_0 (0) =

let val (t0, i0) = reg_0
in
    (t0 + 0, i0)
end
```

Again, we do not use any instructions beyond those that develop a value into `reg_0`, and so the cost of this rule is zero. We create an addressing expression from an address stored in a register as the addition of that register with the constant zero.

Given these (and other) rules for memory addresses, we can easily write a rule for the SPARC's `st` store-to-memory instruction.

```
stmt: $m[addr_0] := reg_1  (1) =
let val (e0, i0) = addr_0
    val (t1, i1) = reg_1
in
    (Sparc.st (e0, t1)) :: (i0 @ i1)
end
```

Whereas we decompose the `reg_1` production non-terminal into a location and a list of instructions, we decompose the address non-terminal into an addressing expression and a list of instructions. We pass this expression along to our abstract instruction function.

We have not shown our complete procedure for emitting all the BURG rules we need. We enumerate the rest of the necessary rules here, but as they are very similar to those presented above, we do not give examples of them. First, we must emit rules for any edges added to complete the data-movement graph using the techniques described in this chapter. Those rules, however, look very similar to those already presented. Furthermore, we must emit production rules for developing values into memory locations, rather than temporary and register locations. These production rules are similar to those we have already seen, except that we require a method to allocate stack slots to store temporary values. Also, we want to artificially increase the cost of these production rules to model the expense of storing intermediate results of computations in memory. The exact nature of the stack slot allocator (e.g., how is the address of an appropriate stack slot determined?) and the amount by which we increase the cost of such production rules have not yet been determined.

Returning to our inductive proof, we can now look back at how we have achieved our goal. Via algebraic laws and unwanted effects, we can complete the data-movement graph to the point where the read-write location sets are strongly connected and every read-only and write-only location set is connected to some read-write location set. These properties ensure us that we can move arbitrary values between arbitrary locations of the appropriate size. The operator analysis allows us to emit BURG rules to implement every operator on its own, completing the inductive step in the proof.

Even if we are unable to fully complete the graph or find implementations of every operator, we can still emit a precise characterization of the source RTLs that our code generator can translate. This characterization would take the form “we can translate source RTLs that contain the operators `add #32`, `sub #32`, or `mul #32 #64`, take operands from 13-bit sign-extended constants, `$r[x]` where $x \neq 0$, 32-bit cells in `m-space`, or `$f[x]`, and place their result anywhere in `$r[x]` where $x \neq 0$, 32-bit cells in `m-space`, or `$f[x]`”. Such a characterization is important for a compiler author writing a mapping from his intermediate representation to RTLs.

5.5 Code Generation for the Tiny Machine

We now have developed the machinery that we need to put together a full translation from the `almostManhattanDistance` procedure in Figure 1.2 to the assembly code in Figure 1.3. We have already seen the transformation from source code to RTLs in which variables have been assigned to temporary locations in Figure 4.5 on page 40. To move from this state to assembly code we must have BURG rules for the Tiny Machine’s instruction set. Below is a complete, annotated set of BURG rules for this machine.⁷

The left-hand side register non-terminal, `reglhs` can either be an actual register (in which case the index must be less than 16) or else a location in the `t` temporary space associated with the registers. A `reg` non-terminal may be a value fetched from either a register or a temporary.⁸

```
reglhs: $t[x]    (0)                                = $t[x]
reglhs: $r[x]    (if x < 16 then 0 else infinity) = $r[x]

reg: Fetch(reglhs)  (0) =  reglhs
```

We have one top-level rule for every instruction provided by the Tiny Machine. The cost of each rule is 1, and each rule returns a list of abstract instructions. For

⁷We do not include results of a hypothetical operator analysis here. Such an analysis would allow us to conclude, for example, that we can use the Tiny Machine’s `add` instruction to move values between registers. We have omitted such details to keep the size of the example manageable.

⁸In this example, we only include the `_x` at the end of non-terminals when it is necessary to disambiguate between multiple occurrences of the same non-terminal.

example, the first top-level rule is for the Tiny Machine's `add` instruction. The instructions returned by the rule are the `add` instruction itself in addition to whatever rules are necessary to move the appropriate values into the addition's operands, `reg_0` and `reg_1`.

```
stmt: reglhs := reg_0 + reg_1  (1) =
  let val (t0, i0) = reg_0
    val (t1, i1) = reg_1
  in
    (Tiny.add(t0, t1, reglhs)) :: (i0 @ i1)
  end
```

```
stmt: reglhs_2 := reg_0 - reg_1  (1) =
  let val (t0, i0) = reg_0
    val (t1, i1) = reg_1
  in
    (Tiny.sub(t0, t1, reglhs)) :: (i0 @ i1)
  end
```

```
stmt: reglhs := con22  (1) =
  Tiny.li(con22, reglhs)
```

```
stmt: reglhs := $m[addr]  (1) =
  let val (e0, i0) = addr
  in
    (Tiny.ld(e0, reglhs)) :: i0
  end
```

```
stmt: $m[addr] := reg  (1) =
  let val (e0, i0) = addr
    val (t1, i1) = reg
  in
    (Tiny.st(e0, t1)) :: (i0 @ i1)
  end
```

We include one production rule for each rule that produces a value in a register. These rules use `getreg` to allocate a temporary to hold this intermediate result. They return both the temporary that holds the intermediate results as well as the list of instructions to develop the appropriate value into that temporary.

```
reg: reg_0 + reg_1  (1) =
  let val (t0, i0) = reg_0
    val (t1, i1) = reg_1
```

```

        val t2 = getreg #"t"
    in
        (t2, (Tiny.add(t0, t1, t2)) :: (i0 @ i1))
    end

reg: reg_0 - reg_1 (1) =
    let val (t0, i0) = reg_0
        val (t1, i1) = reg_1
        val t2 = getreg #"t"
    in
        (t2, (Tiny.sub(t0, t1, t2)) :: (i0 @ i1))
    end

reg: con22 (1) =
    let val t1 = getreg #"t"
    in
        (t1, Tiny.li(con22, t1))
    end

reg: $m[addr] (1) =
    let val (e0, i0) = addr
        val t1 = getreg #"t"
    in
        (t1, Tiny.ld(e0, t1))
    end
end

```

Because the Tiny Machine only has a single addressing mode, we need only one address rule. This rule simply returns the addressing expression and instructions necessary to use a base-displacement address.

```

addr: reg + con18 (1) =
    let val (t0, i0) = reg_0
    in
        (t0 + con18, i0)
    end
end

```

Finally, we define the non-terminals for 18-bit and 22-bit constants. These rules do nothing other than check that the value they match fits in the appropriate number of bits. If a value does not fit in the appropriate number of bits, then the cost of the rule is infinity, and so the rule never matches.

```

con18: x (if fitsInBits(18, x) then 0 else infinity) = x
con22: x (if fitsInBits(22, x) then 0 else infinity) = x

```

Feeding these rules to a BURG engine would enable it to generate assembly code for the RTLs in Figure 4.5. For this small procedure even the Tiny Machine has enough registers to hold all of the intermediate results, and so register allocation would not insert any spills to or loads from memory. Following code generation and register allocation, we end up with the following assembly code (equivalent RTLs are given in comments), as promised in Chapter 1:

```
li      20,  %r4      # $r[4] := 20
sub    %sp, %sp, %r4   # $r[15] := $r[15] - $r[4]

ld     %sp,  0,  %r0   # $r[0] := $m[$r[15] + 0]
ld     %sp,  4,  %r1   # $r[1] := $m[$r[15] + 4]
ld     %sp,  8,  %r2   # $r[2] := $m[$r[15] + 8]
ld     %sp, 12,  %r3   # $r[3] := $m[$r[15] + 12]

sub    %r0, %r2, %r5   # $r[5] := $r[0] - $r[2]
sub    %r1, %r3, %r6   # $r[6] := $r[1] - $r[3]
add    %r5, %r6, %r7   # $r[7] := $r[5] + $r[6]

st     %sp, 16,  %r7   # $m[$r[15] + 16] := $r[7]
```

Chapter 6

Related Work

Most work in creating easily retargetable compilers involves in some manner writing a code generator by hand. In particular, we have already seen that the two main families of work in this area both require a mapping from intermediate representation to machine instructions to be written by hand. Systems built on BURS theory, such as BURG, BEG, or twig, require this mapping in the form of grammar-like rules (Fraser, Hanson, and Proebsting 1992, Fraser and Hanson 1995). Systems such as gcc and VPO that produce naïve code and then apply machine-independent optimizations, require the naïve code generator to be written by hand (Davidson and Fraser 1984, Stallman 1999).

There are a few significant works that focus on analyzing a machine description to produce a code generator. Cattell (1980) makes use of heuristic search methods inspired by the field of artificial intelligence to derive—at compile-compile time—tables to be used for tree-matching code generation at compile time. Cattell’s notion of “operand classes” corresponds loosely with our location sets. Cattell’s analysis is driven by the search to find ways to implement operators on the target machine, as is our analysis in the previous chapter. Cattell approaches this by attempting to transform a goal (an operator implementation perhaps) into a sequence of machine instructions, whereas we begin with machine instructions and employ strategies to find operator implementations within them. Cattell’s search also encompasses part of our data-movement graph, though he does not include constants, memory, or addressing expressions within this search. These searches are driven by a fixed set of logical

axioms. Although he mentions the possible need for machine-specific axioms, Cattell’s system does not include an archival strategy for allowing the set of axioms to be easily extended. While Cattell does acknowledge instructions with multiple effects, he does not take advantage of unwanted side effects when searching for operator implementations and data-movement instruction sequences.

Cattell’s machine descriptions are somewhat more complex than λ -RTL descriptions. Cattell does not derive the properties of a machine’s locations as we do. Instead, his machine descriptions classify “storage bases” as general purpose (locations that may hold values), temporary (condition codes), or reserved (locations that may not hold values, such as the program counter). Cattell’s axioms deal directly with hardware registers rather than temporary locations. As such, it is unclear how his code generator would fare when generating code for which there are more intermediate results than available hardware registers.

Cattell requires addressing modes to be explicitly specified in the machine description, whereas we infer these modes from the λ -RTL semantics.¹ Additionally, Cattell’s machine descriptions include a cost for every instruction, which is used to select the best implementations for inclusion in the compile-time tables.

At compile time, Cattell’s code generator is table driven, performing top-down pattern matching on source trees. This method is unable to guarantee the local optimality that BURG’s bottom-up approach provides for us.

Zadeck and Hoover’s work (1996) on the Tailored Optimization And Semantic Translation (TOAST) compiler begins with a machine description containing much more information than both λ -RTL descriptions and those used by Cattell. Specifically, TOAST’s machine descriptions must include a timing model for the target architecture as well as a model of the machine’s available resources such as multiple processors. TOAST analyzes a machine description to create data structures and program fragments that are plugged into an abstract compiler. The compiler produced from a machine description operates over a *machine-specific* intermediate representation, and hence TOAST must also produce the machine-dependent glue necessary for *any* desired optimizations. Conversely, the code generators we produce work over

¹ λ -RTL allows a machine description author to define addressing modes for ease of writing the instruction set semantics, but all addressing modes are expanded out by the λ -RTL translator.

a machine-independent representation, and hence optimizations (and other components) can be added to compilers built with our work that do not necessarily have to be derived from our analyses. Because TOAST’s machine descriptions contain much more information than a λ -RTL description, they can also be used to derive machine dependent parts of schedulers and register allocators (and in fact, TOAST must be used for this, because of the machine-specific intermediate language).

Zadeck and Hoover require “register patterns”—their equivalent of our location sets—to be specified explicitly within each of the instruction records that comprise part of a machine description. Much of Zadeck and Hoover’s work is centered around a “semantic comparator” that exhaustively applies transformations to a “desired operation” attempting to find a sequence of instructions that are semantically equivalent. As with Cattell’s work, these transformations are applied in the opposite direction from our work. Whereas we look first at instructions and attempt to transform them to obtain operator implementations, the semantic comparator attempts to transform a desired operation to match the semantics of machine instructions.

The semantic comparator’s search progresses by first attempting to find “toeprints”—single effects of instructions that are semantically equivalent to subgraphs of a desired operation. Toeprints are expanded to “footprints” by considering entire instructions rather than single effects. Zadeck and Hoover do not specify how they handle any unwanted effects that this introduces. Finally, the search finds “translations” by attempting to tile the full graph of the desired operation with footprints. The search tries every possible tiling to find as many translations as it can. The in-depth timing model—such as cycles per instruction—included in the machine descriptions allows TOAST to pick the best translations.

Zadeck and Hoover’s exhaustive search occurs at compile-compile time in order to discover machine instructions to implement every possible desired operation that a compiler front end can produce. We approach this from the other direction as well, characterizing the front end RTLs that we can translate based on the results of our location and operator analyses. As the semantic comparator uses an exhaustive search rather than the directed search that we use, it finds many more possible translations than we do. Zadeck and Hoover state that producing code-improver tables for the RS6000 took 36 cpu hours on an RS6000. Although our work is not fully implemented,

running our location and operator analyses for a SPARC λ -RTL description on a Pentium takes about 5 seconds.

The GNU Superoptimizer (Massalin 1987) is a unique approach to discovering implementations of desired operations. An interpreter compares the desired operation with each possible sequence of machine instructions less than a certain small length for several carefully chosen input values. Sequences of instructions that match output values with the desired operations for all attempted inputs are used to implement the desired operations. While this undirected search method will discover even more implementations than Zadeck and Hoover's work, the nature of the exhaustive search makes it prohibitively expensive (Hoover and Zadeck 1996).

Chapter 7

Summary and Conclusions

7.1 Summary

We have implemented and proposed several instruction-set analyses in this thesis. The space analysis and location-set analysis presented in Chapter 3 are implemented and can produce a mapping between register-like location sets and classes of temporaries. We also can create data-movement graphs, as well as the cost function that derives from the weighted transitive closure of the data-movement graph.

In Chapter 5 we propose analyses to find viable implementations of operators within the instruction set and to use this information to emit BURG rules to drive the code generator. We have partially implemented phase one of the operator analysis. We can determine which instructions are direct implementations of operators, and also which identities, inverses, or unwanted effects would be involved in using instructions to implement operators they contain. We have not yet implemented the remainder of the analyses presented in Chapter 5.

7.2 Reduction of Full RTLs

Recall once more the general form of a full RTL:

$$g_1 \rightarrow l_1 := e_1 \mid g_2 \rightarrow l_2 := e_2 \mid \dots \mid g_n \rightarrow l_n := e_n$$

A full RTL consists of multiple guarded effects. The solution that we have presented in this thesis deals with translating a single effect— $l := e$ —into a sequence of machine

instructions. A full solution must also encompass multiple and guarded effects.

Multiple effects within an RTL occur simultaneously. We expect that rewriting multiple simultaneous effects as a sequence of single effects is a straightforward process via the introduction of temporaries. Thus a swap statement such as

$$x := y \mid y := x$$

would get rewritten as

$$\begin{aligned} t &:= x \\ x &:= y \\ y &:= t \end{aligned}$$

where t is a newly introduced temporary.

We have then reduced translating multiple guarded effects to translating single guarded effects, $g \rightarrow l := e$. This can be rewritten as the three statements:

$$\begin{aligned} &\neg g \rightarrow \text{goto } L \\ &l := e \\ &L: \end{aligned}$$

The guard g must be a Boolean, and there are only a limited number of operators which produce a value of type `bool`. These include the binary logical operators (`eq`, `ne`, `lt`, `le`, `gt`, `ge`) in both signed and unsigned forms, and also negation, disjunction, and conjunction. Negation, conjunction, and disjunction can be rewritten using control flow, similarly to what we did above. Thus we have reduced general guards to comparisons of the form $r_1 \text{ cmp } r_2$ for some binary logical operator cmp . We can translate an effect guarded by a comparison into the two statements:

$$\begin{aligned} &res := \text{compare}(r_1, r_2) \\ &\text{test}_{\text{cmp}}(res) \rightarrow l := e \end{aligned}$$

such that the equivalence $\text{test}_{\text{cmp}}(\text{compare}(r_1, r_2)) \Leftrightarrow r_1 = r_2$ holds. That is, we first compare r_1 and r_2 and then test the result of that comparison to see if the comparison operator cmp holds. The difficulty is that every machine has a different representation for the results of a comparison (res). For some machines, such as the SPARC, res is the condition codes. For others, such as MIPS, res is stored in a general-purpose register. To translate a guarded effect, we must be able to determine what the abstractions compare and test_{cmp} might be. We hope that this problem will

yield to techniques similar to those used in Chapter 5 in this thesis, but for now this problem remains unsolved.

7.3 Questions Raised

Our work has raised several theoretical and experiment questions for future work. The current design is split between a location-driven analysis (resulting in the data-movement graph) and an operator-driven analysis. In Chapter 5 we have seen that many of the same techniques may be used to complete the data-movement graph. In fact, the data-movement graph as first presented in Chapter 4 seems to correspond to our notion of direct implementations of operators in Chapter 5. It would be nice to factor out the similarities in these two analyses, perhaps into an abstraction of the three main strategies we have identified (direct effects, effects requiring algebraic laws, and effects with unwanted side-effects).

Our data-movement techniques do not yet find methods to load 32-bit constants into 32-bit registers. To do so requires on many architectures two load-immediates, a shift-left, and an addition or bitwise-or. We believe that by finding implementations of these operators, we will be able to compose a method for completing this link in the data-movement graph.

Three important experimental questions raised by our work are:

1. How effective is our simple compile-time variable analysis? If the cost-based analysis we present in Chapter 4 should prove too coarse, more complex analyses must be tried. In addition, it may in some cases be cheapest to store a variable in memory. This remains to be seen.
2. What cost is appropriate for promoting a fixed location set and associating it with a class of temporaries? Such a cost might occur either in the compile-time variable analysis or in BURG rules for operator implementations that involve fixed locations (or even register-like sets with small cardinalities). The cost would represent the extra spill/reload penalty that a register allocator would incur when mapping all of the locations in one temporary class to the same hardware location.

3. How does the local optimality provided by BURG affect subsequent machine-independent optimizations in the spirit of gcc and VPO? That is, do the code-improving machine-independent transformations that work well with the results of naïve code expanders also work well with the locally optimal code produced by a BURG engine?

7.4 Conclusions

The instruction-set analyses of Chapters 3 and 4 are already useful to classify a machine’s storage locations—a problem which other researches (Cattell 1980, Hoover and Zadeck 1996) have thought must be solved by hand annotations within machine descriptions. Combined with our analyses from Chapter 5, we have a framework in which we can automatically generate code generators, independent of any particular compiler intermediate representation. By taking advantage of BURG’s local optimizations and the proven ability of gcc and VPO to apply machine-independent code-improving transformations to naïve code, we have been able to sidestep efficiency issues and concentrate on translation itself.

Our binding-time analysis demonstrates that we can determine the various classes of temporaries needed to generate code for a target machine without needing predefined types of register sets. We have also seen the effectiveness of partitioning hardware locations into location sets, as these location sets correspond with the production non-terminals when we emit BURG rules.

The structure of the inductive proof presented in Chapter 5 falls almost directly out of the formal structure of RTLs (Figure 2.1). By treating constants as read-only locations, the data-movement graph that our location analysis yields establishes the base case of this proof, leaving the operator analysis to prove the inductive step. While searching for implementations of individual operators to complete the inductive step, we learn how to exploit algebraic identities, inverses, and rewrite laws, as well as instructions with multiple effects. In turn, we see that these strategies are also applicable to fill in any missing paths within the data-movement graph.

The inductive nature of the proof also allows us to emit a precise characterization of the RTLs that our code expander can translate. This ability is crucial in order

that compiler writers may use our back end with a guarantee that the RTLs that they generate from their intermediate representation will be successfully translated.

A full solution crafted in the manner presented in this paper would change the traditional $O(m \times n)$ compiler retargeting problem to a more appealing $O(m + n)$. For each new machine, a single λ -RTL machine description need be written. For each compiler, a single mapping from the compiler's intermediate representation to our RTLs need be written. Because our work is independent of a particular intermediate representation, it can also be applied towards creating emulators, binary translators, and other low-level tools. We have achieved this independence of a particular intermediate representation by deriving all machine-dependent information from a single λ -RTL description of a machine's instructions' semantics and we believe that the techniques and framework presented in this thesis bring us close to a full solution.

Acknowledgments

I am extraordinarily indebted to my advisor, Professor Norman Ramsey, for his inspiring, educational, and tireless guidance and friendship throughout the development of this thesis over the past seven months. His seemingly unbounded enthusiasm and always positive attitude are a tribute to the enormous effort and care with which he advises. Thanks, Professor Ramsey.

I also owe much gratitude to my good friends who volunteered to read drafts of my thesis and offer comments and suggestions to better it. Russ Cox, Jon Eddy, Stuart Schechter, and AJ Shankar, thanks for all of your incredibly helpful advice.

Last—but certainly not least—thank you to my family, to Dodzie Sogah, and to Lynn Zuckerman, the jewel of my heart, for all of your support and encouragement both while writing this thesis and during the past 22, 4, and 6 years, respectively.

References

- Bailey, Mark W. and Jack W. Davidson. 1995 (January).
A formal model and specification language for procedure calling conventions.
In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, pages 298–310, New York, NY.
- Bell, C. Gordon and Allen Newell. 1971.
Computer Structures: Readings and Examples.
New York: McGraw-Hill.
- Benitez, Manuel E. and Jack W. Davidson. 1988 (June).
A portable global optimizer and linker.
In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 329–338, Atlanta, Georgia, USA.
- Briggs, Preston, Keith D. Cooper, and Linda Torczon. 1994 (May).
Improvements to graph coloring register allocation.
ACM Transactions on Programming Languages and Systems, 16(3):428–455.
- Cattell, Roderic G. G. 1980 (April).
Automatic derivation of code generators from machine descriptions.
ACM Transactions on Programming Languages and Systems, 2(2):173–190.
- Chaitin, Gregory J. 1982.
Register allocation spilling via graph coloring.
In *SIGPLAN Symposium on Compiler Construction*, pages 98–105.
- Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. 1990.
Introduction to Algorithms.
Cambridge, MA: MIT Press.
- Davidson, Jack W. and Christopher W. Fraser. 1984 (October).
Code selection through object code optimization.
ACM Transactions on Programming Languages and Systems, 6(4):505–526.
- Fraser, Christopher W. and David R. Hanson. 1995.
A Retargetable C Compiler: Design and Implementation.
Redwood City, CA: Benjamin/Cummings.

- Fraser, Christopher W., 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.
- George, Lal and Andrew W. Appel. 1996 (May).
Iterated register coalescing.
ACM Transactions on Programming Languages and Systems, 18(3):300–324.
- Hoover, Roger and Kenneth Zadeck. 1996.
Generating machine specific optimizing compilers.
pages 219–229.
- Kane, Gerry. 1988.
MIPS Risc Architecture.
Englewood Cliffs, NJ: Prentice Hall.
- Koutsoufios, Eleftherios and Stephen C. North. 1996.
Drawing graphs with dot.
AT & T Bell Laboratories, Murray Hill, NJ.
- Massalin, Henry. 1987 (October).
Superoptimizer – A look at the smallest program.
In *Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Vol. 22, pages 122–127, Palo Alto, California.
- Poletto, Massimiliano and Vivek Sarkar. 1999.
Linear scan register allocation.
ACM Transactions on Programming Languages and Systems, 21(5):895–913.
- Ramsey, Norman and Jack W. Davidson. 1998 (June).
Machine descriptions to build tools for embedded systems.
pages 172–188. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '98).
- Ramsey, Norman and Christian Lindig. 2001.
The C- Reference Manual pre-Version 2.0.
Available at <http://www.cminusminus.org>.
- Siewiorek, Daniel P., C. Gordon Bell, and Allen Newell. 1982.
Computer Structures= Principles and Examples. Computer Science.
McGraw-Hill.
- SPARC International Inc. 1992.
The SPARC Architecture Manual: Version 8.
Prentice Hall, Englewood Cliffs, New Jersey 07632.
- Stallman, Richard. 1999.
Using and Porting GNU CC (Version 2.95.3).
Free Software Foundation.
Available at <http://gcc.gnu.org>.