Automated Discovery of Machine-Specific Code Improvements

By

Peter Bernard Kessler

B.S. (Yale University) 1973
M.S. (University of California) 1980

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: Susan J. Graham Nov. 12, 1984
Chairman Date

Richard J. Fateman 11/7/84

Marc Davis Nov 7, 1984

.....

Automated Discovery of Machine-Specific Code Improvements

by

Peter Bernard Kessler

If I had the arranging of things in this world they would be different, I can tell you. I don't mean to say I would go round improving things right and left. I think every improvement ought to be looked in the mouth first: to make sure it isn't an improvement for the worse.

G. B. Edwards,
The Book of Ebenezer Le Page,
Alfred A. Knopf, Inc., N.Y., N.Y., 1981.

Abstract

Retargetable compilers generate code using machine-independent algorithms guided by the results of an analysis of the target machine. The analysis may be performed either by the compiler writer or by a compiler phase construction program. An implementation must be found for each operation of the source language.

Additional analysis may reveal special features of the target architecture that may be exploited to generate efficient code. Such analysis is optional; it affects only the quality of the code produced. This dissertation examines one method of automating the examination of a machine description to discover machine-specific *idioms* for inefficient instruction sequences.

Previous techniques discovered idioms by composing instructions into sequences and searching for more efficient implementations of those sequences. Analysis by composition takes time exponential in the length of the sequences examined. Practical considerations limit such analysis to sequences composed of pairs of instructions. The thesis of this dissertation is that an interesting class of idioms can be discovered by decomposing instructions into inefficient sequences to be avoided during compilation. Decomposition is shown to take no more time than composition in the worst case, and is shown to take less time on the average. Analysis by decomposition naturally discovers idioms for sequences longer than pairs of instructions.

Idioms on a target machine and predicates for their applicability are discovered during compiler construction. Consequently, alternative instruction sequences must be identified without knowledge of the particulars of individual programs, such as instances of instruction sequences, operands, or data flow contexts. These program-dependent properties can be exploited by recording the conditions under which one instruction sequence is equivalent to another. Program-independent predicates on instruction equivalence are evaluated during compiler construction, leaving only program-dependent predicates to be evaluated during compilation.

The design of an idiom discoverer is discussed and a prototype implementation is examined. One application of idioms is demonstrated with a prototype retargetable transformer of assembler source code. This transformer can be used to replace hand-written case analysis routines in a retargetable code generator. Other applications of idioms are suggested.

Table of Contents

Table of Contents i

List of Figures iv

Acknowledgments v

Chapter 1 Introduction 1

Analysis of a Target Machine 2

Chapter 2 Work by Others 5

2.1 R. Steven Glanville 5

2.2 Robert R. Henry 6

2.3 Mahadevan Ganapathi 8

2.4 Roderic G. G. Cattell 9

2.5 Jack Davidson and Christopher Fraser 10

2.6 Robert R. Kessler 12

2.7 Robert Giegerich 14

Chapter 3 Motivation and Scope of Idiom Discovery 17

3.1 Limitations of Retargetable Compilers 17

3.2 Binding Idioms 19

3.3 Set Idioms 21

3.4 Composite Idioms 22

3.5 Addressing Mode Idioms 23

3.6 Idioms Not Searched For (and Not Found) 24

3.6.1 Binary Image Idioms 24

3.6.2 Program Flow Graph Idioms 25

3.6.3 Available Expression Idioms 25

3.6.4 Tree Substitution Idioms 26

3.6.4.1 Duals to Binding and Set Idioms 28

3.7 Idiom Discovery 29

3.7.1 Instruction Comparison 30

3.7.1.1 Composition versus Decomposition 30

3.7.1.2 Forward versus Backward Comparison 33

3.7.2 Data Flow Context Information 35

Chapter 4 Machine Descriptions 37
4.1 Trees 37
4.2 Properties of Tree Operators 38
4.3 Type Descriptions 40
4.4 Processor State Variables 40
4.4.1 Limitations on Flow Analysis 41
4.5 Cost Information 42
4.6 Addressing Modes 42
4.7 Address Mode Classes 43
4.8 Instruction Descriptions 44
4.9 Experience with Machine Descriptions 47

Chapter 5 An Implementation of Idiom Discovery 49
5.1 Data Structures for Decomposition 50
5.1.1 Operand Attributes 50
5.1.2 Data Flow Information Attributes 51
5.2 An Overview of Decomposition 52
5.3 Data Flow Analysis 53
5.4 Instruction Classification 55
5.5 Decomposition of a Subject Instruction 56
5.6 Tree Matching 61
5.6.1 Forest Matching 61
5.6.2 Operand Matching 62
5.6.2.1 Binding Constraint Discovery 63
5.6.2.2 Set Constraint Discovery 63
5.6.2.3 Binding Subject Operands 64
5.6.3 Pruning 65
5.6.4 Assignment Matching 66
5.6.5 Type Unification 67
5.6.6 Axioms of Tree Operators 68
5.7 Addressing Mode Side-Effect Idiom Discovery 70
5.8 Artifacts of My Implementation 71
5.8.1 Trees versus Instruction Sequences 71
5.8.2 Allowing Side-Effects on Addressing Modes 71
5.9 Experience with Idiom Discovery 73

Chapter 6 An Implementation of Idiom Application 75
6.1 Information Required for Idiom Application 76
6.2 An Implementation of an Idiom Applier 77
6.2.1 Pattern Matching and Replacement 78

6.2.2 Identifying Patterns and Forming Replacements 79
6.2.3 Addressing Mode Side-Effect Idioms 80
6.3 Interactions of Idiom Application Phases 81
6.4 Experience with Idiom Application 81

Chapter 7 Summary and Conclusions 83
Critique of the System 84
Future Research 85

Bibliography 87

List of Figures

Figure 4.1 Built-in Operators and Leaves 39
Figure 4.2 Post-Increment Addressing Mode 44
Figure 4.3 Description of **sub[bwl]3** 46
Figure 5.1 Tree Describing **incl** 57
Figure 5.2 Tree Describing **addl3** 58
Figure 5.3 Tree Describing **tstl** 59

Acknowledgments

This dissertation is dedicated to Monica, for keeping my life interesting the whole time, and to my parents, for encouraging me to lead an interesting life.

Thanks to my advisor, Susan L. Graham, for her support and encouragement of this research, and to my readers, Richard J. Fateman and Marc Davis, for their careful readings and helpful criticisms on my dissertation. Thanks also to my colleagues, especially Robert Henry and Eduardo Pelegrí-Llopart for their enthusiastic discussions of my work and compiler construction philosophy. Bill Joy showed me how to see what could be, and Kirk McKusick helped make some of those visions work.

Financial support for this research was provided by the Department of Defense, under contracts N00039-82-C-0235 and N00039-84-C-0098.

CHAPTER 1

Introduction

A *compiler* is a translator between a source language written by a programmer and a target language acted on by a machine. Compilation involves several steps: for example, reading the input program, recognizing constructs of the programming language, choosing an implementation of those constructs from the operation codes of the target machine, preparing an executable image of the operations, and so forth. Compilers are often organized into *phases*, each of which performs one step in the translation. The early phases of compilation – lexical analysis, syntactic analysis, and semantic analysis – are *language-specific* but often *machine-independent*. That is, these phases incorporate knowledge of the source language, but do not refer to features of the target machine. These early phases are sometimes referred to as the *front-end* of a compiler. The later phases of compilation – resource allocation, instruction selection, and executable image assembling – are *machine-specific* but *language-independent*. That is, these phases incorporate knowledge about the target machine, but do not refer to features of the source language. These later phases are sometimes referred to as the *back-end* of a compiler. The partitioning of language-specific and machine-specific phases encourages a more modular compiler design.

The design of language-specific compiler phases has benefited from advances in formal language theory. Models and algorithms from formal language theory have resulted in efficient and provably correct implementations of some front-end phases. These implementations are in the form of language-independent algorithms guided by tables for a particular source language. The construction of these tables has been automated by tools that examine a description of the source language and produce tables needed by the language-independent algorithms. A tool to automate the production of a compiler phase is called a phase constructor.

Research on machine-specific compiler phases has not yet resulted in a uniform model for those phases. Algorithms for the efficient and correct implementation of some machine-specific phases have been developed, but others are still performed by *ad hoc* routines. Much work has been done in this field due to the diversity of machine architectures and the cost of developing quality compilers. One goal of research in this field is the development of *retargetable* compiler phases, using machine-independent

algorithms guided by machine-specific tables. These tables should be produced by phase constructors that analyze a description of the target machine.

A retargetable code generator more easily exploits unrestricted instructions, *e.g.*, an addition instruction, than instructions to perform special cases of operations, *e.g.*, an "increment by one" instruction. The algorithms for code generation are machine-independent, though they are qualified by the results of the analysis of the target machine. It may be difficult to specify to those algorithms the restrictions needed to use a particular instruction. Alternatively, some instructions may be impossible to exploit because of limitations of the code generation algorithms, *e.g.*, instructions that cover, or span, several statements of the source language.

Limitations of a code generator with respect to a particular machine have traditionally been handled by a compiler phase running after the code generation phase. This phase examines generated code and transforms instruction sequences to employ special case instructions of the target machine. Since the number of instructions being examined at any one time is typically small, this phase is sometimes called "peephole optimization" [McKeeman 65]. I prefer to call these transformations "code improvements".

This dissertation presents the design of a retargetable compiler phase to improve generated code, and a novel method of phase construction to discover machine-specific improvements from a description of the target architecture.

Analysis of a Target Machine

A code generator selects instructions of the target machine to implement the program being compiled. The quality of the generated code – the space instructions occupy, the speed with which they execute, and their efficient use of architectural resources – depends in part on information about the target machine incorporated into the code generator. This information is the result of an analysis of the target machine. In early compilers, the analysis of the target machine was performed by the compiler writer and the results included directly in the code of the compiler [Backus et al. 57]. Such a compiler is adapted to a new machine by analyzing the new target machine and then rewriting all the machine-specific compiler phases. Such rewriting is often as difficult as writing the original code, and so hardly qualifies the compiler as retargetable. The first retargetable compilers still depended on an analysis of the target architecture by the compiler writer. However, the results of that analysis were encoded in tables used by code generation algorithms applicable to a large class of machine architectures [Johnson 77]. Thus the code selection algorithms need only be written once. Recent proposals have suggested automating the analysis of the target architecture to produce retargetable

compilers from a description of the target machine. Several of these proposals are reviewed in Chapter 2.

Two kinds of target machine analysis are useful for code generation. First, an implementation for each operator of the programming language must be found on the target machine. This analysis is a necessary and sufficient condition to permit code generation for every program. Second, the target architecture may include special features – special cases of instructions or complex addressing modes – that can be exploited to implement programs more efficiently. This second analysis is optional, as it affects the quality of an implementation but not its correctness.

Much of the analysis of the target machine may be done when the compiler is constructed, at *phase generation time*; rather than when the compiler runs, at *phase application time*. A phase is generated once for each target machine, whereas the generated phase is applied each time a program is translated for that target machine. Thus analyses that might be prohibitively expensive at phase application time are tolerable at phase generation time. There are details of individual programs that must be analyzed during compilation, but identifying machine-specific characteristics before compilation reduces the time spent on analysis during compilation. For example, a code generation phase could be written that would be given the program to translate and descriptions of the target machine instructions. This phase would discover, and rediscover, that additions in the program could be implemented by an addition instruction on the target machine [Cattell 78]. The alternative is to provide tables that incorporate the knowledge that the addition instruction is the instruction of choice for additions in the program. The search for an implementation of additions can be done once at code generator construction time, to save repeated searches at code generation time. Due to the tedious and exacting nature of target machine analysis, it is best performed by programs, rather than by human compiler writers.

Special purpose instructions are identified by comparisons with other instructions or instruction sequences. The result of a comparison is a set of conditions under which a special purpose instruction may replace other instructions. The conditions are of two sorts: program-dependent and program-independent. Program-dependent conditions refer to attributes of instructions, *e.g.*, particular values of operands, that can not be evaluated except with respect to a particular program. Program-independent conditions reference only attributes of instructions fixed by the architecture, and can be evaluated at compiler construction time. The goal of compiler construction time analysis is to evaluate all the program-independent conditions for replacement, reducing the work required to complete a replacement to only program-dependent conditions. Several classes of special case instructions are distinguished, and techniques to identify members of those classes from a

machine description are discussed in Chapter 3. These techniques are the primary contribution of this dissertation.

The machine descriptions used in this research are presented in Chapter 4. The major feature of these descriptions is their completeness in describing the instructions and addressing modes of the target machine, especially the description of multiple effects of instructions. Chapter 5 examines a prototype phase constructor for a retargetable code improver. Comparison of instructions is via a technique called "backward decomposition" that identifies all instruction sequences that can be replaced by special purpose instructions of the target machine. The results of the machine analysis are used, for example, by the code improver described in Chapter 6. The prototype code improver transforms assembler source code, though other opportunities for applying transformations are proposed. Chapter 7 appraises the ideas presented in the dissertation, and suggests future research in machine-specific phase construction.

CHAPTER 2

Work by Others

Recent research in retargetable compiler construction has experimented with several different techniques for analyzing target machines. Below is a survey of recent work in this area. The focus of these summaries is on how the analysis of the target machine is carried out and how the results of that analysis are incorporated into the generated code generators. The interested reader is referred to the original works for details about the code generation techniques, *per se*.

This compendium omits systems in which the compiler writer analyzes the target machine. Such systems range from the ones described in [Tanenbaum, van Staveren and Stevenson 82] and [Lamb 81], where the compiler writer analyzes the target machine by hand, to the system described in [Morgan and Rowe 82], where an interactive verifier aids the compiler writer in assuring that two instruction sequences are equivalent.

The first five reviews describe code generation schemes that incorporate progressively more machine-specific analysis into the code generator. The last two reviews describe systems that apply machine-specific improvements to code from naive code generators. This dissertation argues that a combination of these two approaches is better than either one by itself.

2.1. R. Steven Glanville

Steven Glanville describes a code generator based on tree pattern matching using tables produced from a description of the target machine [Glanville 77] [Glanville and Graham 78]. The key ideas are the reduction of the tree matching problem to the string matching problem through the use of LR parsing and the automated production of LR parsers by parser constructors.

The code generator accepts a preorder linearization of program expression trees and uses an SLR(1) parser to match patterns in the tree corresponding to instructions on the target machine. The parser is produced automatically from a grammar describing how the primitive operations of the program tree are implemented on the target machine. The terminals of the machine description grammar are the primitive operators and leaves of the program trees. The right hand side of each production is a preorder linearization of a

tree describing the computation of a target machine instruction. The left hand side of each production represents the result (if any) of that instruction. For example, an add instruction might be used to implement a primitive addition operator, as described by:

```
dest ::= '+' src1 src2
         "add src2,src1,dest"
```

The LR parser constructed from the machine description grammar determines a derivation of an input program tree and emits the corresponding instructions.

Much of the analysis of a target architecture is performed by the grammar writer. The grammar writer chooses implementations for each of the primitive operators from the program tree. More than one way to implement each primitive operator may be specified in the machine description grammar. If several implementations for an operator exist, the grammar is ambiguous. The only automatic analysis of the target machine is the production of parser tables from the grammar at compiler construction time. The generated parser tables incorporate static heuristics to choose between alternative derivations (*i.e.*, code sequences). For example, longer rules, and therefore more complex instructions, are preferred over shorter rules and simpler instructions. The parsers used for code generation are SLR(1) parsers. The limited lookahead available makes it possible for these code generators to "block" (that is, be unable to parse valid input) if the machine description includes instructions whose use is restricted and the alternatives can not be distinguished with the context available via the lookahead.

Glanville's code generators operate on one source statement at a time, and so can miss opportunities to use complex instructions on the target machine. Glanville incorporates a restricted set of semantic tests into the parser to allow the code generator to identify semantically restricted instructions. These semantic tests could be replaced by a transformation system such as the one described by this dissertation.

Glanville's code generators produce translations from program trees to instructions that are provably correct with respect to the description grammar. The use of LR parsing guarantees that the generated code generator will run in time proportional to the size of the input program.

2.2. Robert R. Henry

Robert Henry addresses many of the problems with Graham-Glanville code generators [Henry 84]. Machine description grammars are factored into separate productions representing addressing modes and productions representing instructions. This factoring allows real, moderately complicated, target machines to be described by reasonably sized grammars. In addition, Henry adds algorithms to the parser generator to alleviate blocking in the produced code generator. The grammar writer may therefore

describe some special purpose instructions, in addition to general purpose instructions, without fear of introducing blocks into the constructed code generator. Henry adheres strictly to the syntax-driven parsing model for code generation. Within this framework, he advocates "syntax instead of semantics" to exploit special properties of the target machine. For example, an instruction whose description requires a specific constant is modeled by adding that constant as a new terminal symbol in the grammar. The restricted instructions can then be described syntactically using this symbol. The cost of adding this symbol is that the lexical analyzer of the input program primitives must distinguish the constant, and that the parser tables for the produced code generator are larger than without the new symbol. Similarly, an instruction with multiple results, *e.g.*, a computation and the setting of condition codes, would be modeled by adding a new non-terminal to represent both effects. Again, the cost is in larger parser tables to be examined during code generation. Both of these techniques must be used judiciously to keep the size of the parser tables reasonable. The generated code generators use only SLR(1) parsing to recognize instruction tree patterns, and so are provably correct and run in time proportional to the size of the program trees.

Certain restrictions on instructions can not be modeled with purely syntactic productions. Chief among these restrictions is the inability to specify the required relationships between operands of an instruction. For example, a 2-operand addition instruction computes the sum of its operands and stores the result in one of those operands. The correspondence between one of the addends and the sum can not be modeled with pure syntactic productions without enumerating all possible operand correspondences, an unreasonably large enumeration.

To alleviate the restrictions of the purely syntactic pattern matcher, the production Graham-Glanville-Henry code generators include hand-written routines to exploit certain semantically restricted instructions of the target architecture. For example, these routines identify opportunities to use 2-operand instructions in place of 3-operand instructions selected by the code generator. The replacement of these hand-written routines was a starting point of my research. A conclusion of my research is that all of the improvements performed by these routines can be discovered automatically by examining a suitable description of the target machine (see Chapter 3). In addition, if a retargetable code transformation phase is added to the compiler, the descriptions of many special purpose instructions can be removed from the machine description grammar. The removal of these special cases implies less analysis of the target machine is required of the grammar writer, the reduced grammar requires less parser construction time, and smaller parser tables are used during code generation.

The code generator used in my research is a Graham-Glanville-Henry code generator. I will note where that choice of code generator affects discussions in this dissertation.

2.3. Mahadevan Ganapathi

Mahadevan Ganapathi, [Ganapathi 80], tries to formalize the semantic restrictions on tree pattern matching via parsing by using attribute grammars [Knuth 68]. In Ganapathi's machine descriptions, semantic information may be *attributed* to syntactic symbols. The right hand side of a production may include predicates on the attributes to control reductions, and functions of the attributes to synthesize new attribute values. For example, the choice between a 2-operand or a 3-operand addition instruction discussed above would be represented by the pair of productions:

reg.*dest* ::= ':=' reg.*dest* '+' reg.*src2* reg.*src1*
IfEqual(*dest*, *src2*)
Emit("**add2**", *src2*, *dest*)
reg.*dest* ::= ':=' reg.*dest* '+' reg.*src2* reg.*src1*
Emit("**add3**", *src1*, *src2*, *dest*)

where attributes are shown in italics, "IfEqual" is an attribute predicate that tests if two sets of attributes are equal, and "Emit" is an attribute function that prints instructions. The ability to control reductions based on attributes containing program-dependent data is an advantage in exploiting instructions with semantic restrictions. Ganapathi's system can only synthesize, rather than inherit, attributes of the left hand side of productions, so semantic information only passes up the derivation tree. That is, no context information is inherited by the production to influence code selection.

Ganapathi's attributed grammars describe how to implement the primitive operations of the input program tree, including predicates that must be tested to use semantically restricted instructions. The machine describer analyzes the target machine to determine the attributes, and predicates on those attributes, that must be tested to use certain instructions. The description writer then augments the grammar to synthesize the attributes, and writes the predicates to test them. Each rule restricted by attribute predicates must be accompanied by an unrestricted rule for the same tree. Without these unrestricted rules the parser will block on input trees that fail to satisfy the semantic predicates. A machine description is developed by first writing unrestricted (*i.e.*, syntactic) rules to use general instructions, and subsequently adding restricted rules to use special purpose instructions. Essentially, the description writer supplies both the code to be produced for each reduction and a set of optimizations to be performed as code is produced. Ganapathi's published examples demonstrate the omission of well-known assembler source code optimizations, as well as the unevenness of application of optimizations.

The added descriptive power of Ganapathi's approach is paid for by added time needed to evaluate attribute predicates at code generation time. Since there are no restrictions on attribute predicates or functions, Ganapathi can not claim that his attribute directed parsing code generators run in time proportional to the size of the input program. In practice, this is not a problem, since the predicates and functions only formalize semantic checks and computations that would otherwise be performed in semantic routines. Correctness of translation is also an issue, since the hand-written attribute predicates influence the pattern matching of the code generator.

As in the previously described code generators, Ganapathi's code generators operate in a single pass over the input tree. To allow some improvements across instruction boundaries, Ganapathi describes a buffering mechanism for generated code. This buffering allows multiple passes over the code after code generation to perform additional transformations, but can not be considered part of the attributed parsing code generation technique.

My research shows that most of the improvements in generated code made possible by attributed parsing can be discovered automatically in a machine description, and applied uniformly by table-driven code transformation. I chose to implement my prototype transformer as a separate compiler pass. An area of future research would be the incorporation of automatically discovered code improvements into an attributed parsing code generator.

2.4. Roderic G. G. Cattell

Rick Cattell describes the first code generator constructor that performs case analysis of the target machine at code generator construction time [Cattell 78]. The code generator constructor is in the form of a description driven code generator. This code generator takes as input a description of the target machine and a program tree, and produces a code sequence to implement the program tree on the target machine. The program tree and code sequence are recorded in a table. This table of mappings from trees to instructions is used to retarget a table-driven code generator.

The separation of code generator construction from code generation has the advantage that the code generator constructor can be arbitrarily thorough in searching for code sequences to implement tree patterns. Cattell describes a heuristic tree matching strategy for identifying code sequences for trees using decomposition and axioms. A collection of sample program trees are given to the tree matcher. The best code sequence for each sample program tree is recorded as the code to generate for that tree. The sample program trees used to construct the tables include trees for all the primitive tree operations, including transfers between all members of the storage hierarchy, and

conversions between all data types. Thus Cattell's code generators will be able to find instruction sequences for all trees (*i.e.*, they will not block on any trees). In addition, Cattell records the trees describing each individual instruction on the target machine for use during code generation. Thus Cattell's code generators should be able to use any special purpose instructions available on the target machine.

Unfortunately, Cattell's table-driven code generation algorithm does not run in time proportional to the size of the tree being translated. The inclusion of trees for all the instructions delays much of the analysis of alternative matches until code generation. In addition, Cattell's code generators are not used in the one-pass style of previously described code generators. The multi-pass PQCC compilers, for which Cattell's work was intended, perform extensive flow analysis and register allocation that affects code generation and the quality of generated code [Leverett et al. 79]. These factors make it difficult to evaluate the extent to which PQCC compilers would benefit from the techniques discussed in this dissertation.

2.5. Jack Davidson and Christopher Fraser

Christopher Fraser [Fraser 79] and Jack Davidson [Davidson 81] have designed a retargetable code generator that analyzes the target machine at code generation time. Their system, PO, performs machine-specific transformations on the intermediate form of the program before instruction selection [Davidson and Fraser 80].

The front-end of a Davidson-Fraser compiler translates a program into instructions for an abstract machine. These abstract instructions are then translated to register transfers [Bell and Newell 71]. The register transfer form of the program is manipulated, and finally target machine instructions are generated.

The translation of abstract instructions to register transfers is accomplished by macro expansion, with a new set of macros for each target machine. The register transfers incorporate implementation strategies (*e.g.*, computations in registers or on a stack) appropriate for the target machine. The instructions of the target machine are described by register transfers. Register transfer expressions for addressing modes are factored out of the instruction descriptions. The macro expansion of abstract instructions to register transfers and the subsequent transformation of that form maintains the assertion that the register transfers can be mapped, many to one, to target machine instructions by syntactic matching. The abstract instructions implement an intersection of operations available on a wide class of target machines, so the assertion is not difficult to satisfy for the output of macro expansion. Since the abstract instructions are so primitive, mapping to target instructions without machine-specific transformations produces inefficient code. Therefore the register transfers representing a program are

combined in an attempt to match more complex target machine instructions.

The first step in the transformation of a program gathers data flow information from the register transfers. Links are established from uses of values to the expressions defining those values. This data flow information is used to perform several machine-independent transformations to eliminate redundant transfers and reuse available expressions.

A second pass performs machine-specific transformations. For each register transfer in the program representation, PO follows the use–to–definition links and substitutes defining expressions in place of the uses of values. If the composite register transfer describes an instruction on the target machine, that register transfer is substituted for the original register transfers in the representation of the program. If the composite register transfer is not a legal instruction, and there remain additional uses of variables for which defining expressions may be substituted, PO attempts a second substitution. If this additional substitution results in a register transfer that matches an instruction description, the three register transfers are replaced by the composite register transfer. PO never considers combining the effects of more than three register transfers. Davidson argues that substitution translates the load–operate–store model of the abstract stack machine to instructions operating on multiple operands. When no more substitutions can be made into a register transfer, the next sequential register transfer in the program representation is examined. When no more substitutions can be made in the program, the instructions represented by the register transfers are emitted.

The register transfers that are the input to PO represent individual operations of larger expression trees. The structure of these trees is implicit in the definition and use of temporary variables. PO discovers the structure of the expression tree by establishing use–to–definition links. PO then assembles the individual operations into explicit trees matching instructions by collapsing links between definitions and uses. Each substitution may eliminate the need for the variable defined by the substituted tree. (Since common subexpressions have been identified, substitution may only decrease a use count.) PO works from the first register transfer forward, assembling instruction trees bottom-up. When PO is finished, the structure of the expression tree is explicit between operations combined into instructions, and implicit between instructions in the definition and use of remaining temporary variables. In contrast, the parsing code generators are given the operations and structure of the expression tree explicitly. The parser recognizes subtrees representing instructions with a top-down pass over the expression tree. Instruction trees are then linked by the introduction of compiler temporary variables.

Since the substitution and matching in PO are done lexically, special cases (*e.g.*, the constant 1) are easily specified. A disadvantage of lexical substitution and matching is that the substituted portions of a register transfer are matched each time they are

substituted. Since substitution is performed only for addressing modes, which are typically small, this repeated matching should not be a problem in practice. The matching in PO is not strictly lexical, since conventions exist for specifying correspondences between one addressing mode and another. (It appears that it is not possible to specify a necessary correspondence between components of an addressing mode: *e.g.*, that two base registers are equivalent, though the offsets from the base registers differ.) The Davidson-Fraser machine descriptions also include escapes for letting the description writer provide predicates and functions to direct matching. As with the predicates and functions used in Ganapathi's attribute directed parsers, these escapes detract from the claim of automatically discovered transformations.

The system described above is a retargetable code generator, not a code generator constructor. No analysis of the target machine instructions takes place at compiler construction time. The transformations are all discovered at code generation time. Further, those transformations are rediscovered each time they are applicable in the program being transformed. Davidson and Fraser suggest saving the transformations and building a table-driven transformer [Davidson and Fraser 84]. The code generator would be run on a set of programs and the transformations discovered in those programs would become the tables for a separate compile-time transformer.

2.6. Robert R. Kessler

Robert Kessler[1] has implemented the first table constructor for retargetable code improvers [Kessler 81][Kessler 84]. A description of the target machine is examined at compiler construction time, and tables are produced for a compile-time code improver.

The front-end of Robert Kessler's compiler produces instructions for an abstract machine. Macro expansion translates these abstract instructions to target machine instructions. Since the abstract instructions are primitive, the code generated by macro expansion is often an inefficient implementation of the program on the target machine. Machine-specific improvements are implemented by a simple pattern matching and replacement technique. Robert Kessler's code improver transforms assembler source; that is, code improvements are made after code generation and resource allocation. All the analysis of when a pattern can be replaced is performed at compiler construction time.

Machine descriptions represent instructions as forests of computation trees. Addressing modes are factored and described separately. Restrictions on addressing modes of instruction operands are represented by addressing mode sets for each operand.

[1] Robert Kessler is not a relative of the author of this dissertation.

The code improver applies several machine-independent transformations exploiting program-dependent data flow information. The data flow information is gathered by machine-independent algorithms using tables produced from the machine description. The patterns and replacements are discovered by composing all possible pairs of instructions and searching for a single instruction that performs the composed computations.

If the first instruction of a pair of instructions can define a value used by the second instruction of the pair, the defining expression is substituted in place of the use of the value in the description. The target machine description is then searched for an instruction whose description matches this composed computation. Description matching is aided by algebraic manipulations of the description trees. Composition along definition–to–use chains is similar to the substitution done by Davidson and Fraser. The key difference is that Davidson and Fraser do the substitution and search at code generation time, where Robert Kessler's system does composition and searching at compiler construction time.

If the first instruction is independent of the second instruction, the descriptions of the instructions are concatenated, and the target machine description is searched for an instruction that performs the multiple effect represented. In contrast, Davidson and Fraser do not have to check for concatenations of effects, since their instructions are selected after transformation. Note that neither composition nor concatenation causes a search for one–to–one instruction replacements. Such replacements are unnecessary in Robert Kessler's system because code is generated by hand-written macro expansions of abstract machine instructions. Thus some analysis of the target machine is left to the compiler writer. The analysis by the compiler writer presumably includes all single instruction transformations, such as using an increment instruction for additions of one, since there is no automated analysis of single instructions.

Analyzing the target architecture at compiler construction time permits more time to be devoted to the analysis than would be acceptable if the analysis were performed at code generation time. Thus Robert Kessler can afford to describe complete machines. Each instruction is composed or concatenated with each other instruction. Thus n^2 patterns are matched against the instruction descriptions. It is obvious that composition and concatenation can be applied to triples of instructions. It is not obvious that the time required to analyze triples would be acceptable even at compiler construction time.

Intuitively, Robert Kessler's system (and Davidson and Fraser's system) construct inefficient code sequences as patterns and then try to find more efficient replacements on the target architecture. In contrast, the system described in this dissertation starts with efficient sequences and determines what inefficient code sequences the efficient code can replace.

2.7. Robert Giegerich

The place of honor in these reviews must go to Robert Giegerich. A formal framework is presented, in [Giegerich 83], in which to analyze complete descriptions of a target machine. The analysis distinguishes three sets of properties: properties of the target machine, properties of particular programs, and properties of particular program execution states. This framework is used to describe how one could automate the production of a machine-specific improver of assembler source code. Much of the analysis proposed has a wider application to the automated construction of compiler components than just transforming the assembler source code representation of programs. Other representations of programs can be analyzed for machine-specific transformations using Giegerich's approach.

Much of Robert Giegerich's concern is with the automatic generation of data flow analyses [Allen and Cocke 76]. Giegerich's machine descriptions are based on a characterization of the storage hierarchy of the target machine. Addressing mode descriptions depict how elements of the storage hierarchy can be referenced. Instruction semantics are given in terms of their effect on the state of the storage.

The description of the storage hierarchy allows Giegerich to derive predicates to test if two operands overlap, *i.e.*, reference the same storage location. The predicates are derived at compiler construction time, to be evaluated at code generation time. In the case of statically addressed storage (*e.g.*, registers, global variables), the overlap predicates are accurate at code generation time. In the case of dynamically addressed storage (*e.g.*, indirection through pointers, displacements from base registers), the overlap predicates must be conservative. The predicates make clear the distinction between program dependencies and execution state dependencies.

Giegerich's analysis of the target machine derives functions to gather data flow context information at code generation time. Data flow context is divided into a "left context" and a "right context". The left context consists of sets of addressing modes that reference the same value. These equivalences can be used to transform an operand to a cheaper reference to a value. The right context is a set of storage locations whose values are not used. This information can be used to remove extraneous computations from a program. Both contexts can be maintained by machine-independent data flow analysis algorithms, given the predicates on storage and instruction semantics Giegerich derives from machine descriptions. Giegerich also derives predicates to indicate when transformations of a program affect data flow information gathered for that program. Such predicates reduce the amount of data flow analysis required when transforming a program.

All of Robert Giegerich's work on data flow dependence and data flow analysis can be exploited to improve a program in any representation. In particular, much of his research seems applicable to code motion and register allocation.

Giegerich applies his analysis of target machines to automating the construction of machine-specific code improvers. He proposes deriving predicates, at compiler construction time, to indicate when one instruction may be replaced by another, cheaper, instruction. These predicates start from machine-independent axioms and are qualified by the improver generator with specifics about the target machine, *e.g.*, instructions, addressing modes, left contexts, and right contexts required to perform safe transformations. The predicates on the instructions and addressing modes can be evaluated at compiler construction time. The restrictions of left and right context must be delayed for evaluation at code generation time. Thus Giegerich's transformer constructor separates machine dependencies from program dependencies, leaving only the program dependencies for evaluation during code generation. Robert Giegerich's proposal is an improvement over Robert Kessler's system in that it incorporates data flow context predicates into the transformations discovered at compiler construction time.

My research is based on Robert Giegerich's framework. Giegerich does not describe how instructions and addressing modes are compared at compiler construction time. Much of this dissertation is concerned with the details of instruction comparison. Further, Giegerich discusses only the comparison of one instruction to another. Therefore, his constructed transformer will only replace single instructions. My research extends this comparison and transformation to sequences of instructions. Giegerich's framework derives transformations predicated on both left and right context. My implementation uses only right context predicates. I argue that the transformations based on left context, *e.g.*, common subexpression elimination and redundant code elimination, are better done before code selection than by transformations of assembler code (see Section 3.6.3).

CHAPTER 3

Motivation and Scope of Idiom Discovery

This dissertation proposes the automation of a portion of retargetable compiler construction. First this research must be motivated within the framework of a retargetable compiler, and second it must be shown how this proposal automates the construction of a portion of such a compiler. This chapter outlines the scope of the problem of automated discovery of machine-specific code improvements and a solution to that problem.

3.1. Limitations of Retargetable Compilers

In a retargetable compiler the lexical, syntactic, and semantic analyses phases are performed by language-dependent but machine-independent routines, while the code generation phases are performed by language-independent but machine-dependent routines. The language-dependent and machine-dependent phases communicate via a set of largely language-independent and machine-independent primitive operations. The compiler is "retargeted" by changing the code generation routines to those for a new target machine.

Recent retargetable code generation techniques have concentrated on machine-independent code generation algorithms driven by tables. Retargeting is automated by providing table constructors driven by a description of the target machine. In this scheme, the compiler writer describes how to implement the primitive operations on the target machine and a "code generator generator" produces tables from that description. The more comprehensive the description, the better the code generator produced, possibly at the expense of larger tables and slower code generation.

Limitations of code generation techniques, both hand-written and automatically generated, often make it difficult or impossible to exploit all the facilities of a target architecture. For example, a code generator may operate on a single source statement at a time, making it impossible to use instructions with several effects: *e.g.*, an addition instruction that also tests its result against zero. It may be difficult or impossible to represent certain special cases under which specific instructions may be used: *e.g.*, that a restricted addition instruction can only add constants in the range [−8,...,+7]. It has been

traditional to alleviate these limitations with a "peephole optimization" phase that transforms generated instruction sequences to equivalent, but cheaper, instructions. It has also been traditional to neglect certain case analyses during code generation, knowing that they will be handled by such transformations. The separation of machine-specific transformations from code generation leads to a more modular retargetable compiler design. Many of the uses originally proposed for such peephole optimizations have been shown to be better handled by other phases of the compiler: *e.g.*, constant folding, common subexpression elimination, register allocation, *etc.* There continue to be opportunities for machine-specific improvements of the output of most code generators. Many of these improvements can be found by examining a description of the target machine. I propose automating this examination to discover machine-specific improvements. These improvements can be recorded for use by a table driven code transformer.

Each target machine is analyzed once, producing tables that are used to transform code generated for any program. The analysis of the target machine is performed independent of the code produced for any particular program. The output of the analysis is pairs of equivalent instruction sequences and constraints needed to assure equivalence. These pairs are used to retarget a machine-independent program transformer. Constraints on instruction pairs identify program dependencies that must be verified during compilation. Restrictions may be placed on operands of instructions, and also on the data flow context in which the instruction sequences appear. Each particular program is examined for sequences and satisfaction of constraints from the pairs. The elements of a pair need not have equivalent costs. Replacing the more expensive sequence with the cheaper sequence will reduce the cost of the transformed code. The cheaper of the sequences is an *idiom* for the more expensive sequence, by analogy with the idioms used by native speakers of natural languages. The analysis of the target machine I call *idiom discovery*. The transformation of generated code I call *idiom application*.

The main advantages of automated idiom discovery are its completeness and its correctness. No instances of the improvements available on the target machine will be missed by the automated analysis. Furthermore the transformations recorded, especially the preconditions for their application, will be correct with respect to the machine description (if the idiom discoverer is correct). A secondary advantage of automated idiom discovery is that the same categories of improvements will be discovered on each target machine. This consistency allows comparable quality retargetable compilers to be constructed.

The main advantage of table-driven idiom application over more *ad hoc* techniques is that the improvements are applied uniformly. No opportunity to apply a transformation

will be missed. The idiom applier examines assembler source code. As such the idiom applier is not limited to improving the output of a single code generator, but can transform code from any source, compiler or human. A disadvantage of this freedom is that the idiom applier cannot use annotations (for example, data flow information) that might be available from some compilers but not others. Assembler source code transformation was chosen because it separates idiom application from any particular code generator for the target machine.

The improvements for which the idiom discoverer searches exploit special cases on the target machine. These special cases include, for example, the restricted addition instruction above, that only adds constants in the range [--8,...,+7]; or an addressing mode that references an operand, but also has a "side-effect" of incrementing a register. In general, architectures that include special case instructions also include unrestricted instructions. The unrestricted cases for the above examples might be a general addition instruction, or an addressing mode that references the operand without the side-effect. A person retargeting the code generator is concerned with the general cases: the code generator must generate correct code. If time is available, and the code generator specification permits, some case analysis might be done by the description writer and special cases added to the description by hand. The case analysis involved is tedious and prone to error. For example, the description writer may specify that addition of one can be implemented by the restricted addition instruction, but may forget that subtraction of one can be implemented by the same instruction. It also may be that the code generator description language is not powerful enough to specify predicates needed to select an instruction. For instance, the '1' in the previous examples has to be distinguished as a constant and distinguished from other constants. Another common predicate is that two (or more) operands of a general instruction must refer to the same location before that instruction can be replaced with a special purpose instruction with fewer operands. Finally, even with the most complete machine description, limitations on the code generation strategy may introduce opportunities for improvement: *e.g.*, generating code for one source statement at a time. The idioms searched for in my prototype idiom discoverer are described in detail below.

3.2. Binding Idioms

A *binding idiom* is a special purpose instruction that replaces a more general instruction when several of the operands of the general instruction are the same. For example, on the VAX-11, the 2-operand long integer addition instruction

```
addl2      src,dest          ; dest := dest + src
```

is usually a binding idiom for the 3-operand long integer addition instruction

```
addl3      src1,src2,dest    ; dest := src2 + src1
```

when the second source operand is the same as the destination[1]. A careful reader will have noticed the "usually" in the previous sentence. A counterexample is when the reference to the second source operand and the destination are the same syntactically, but have side-effects that change the operand they reference. The presence or absence of addressing modes with side-effects on the target machine is (obviously) machine dependent. Whether a particular instance of an **addl3** operand has side-effects depends on the program being translated. But the disallowing of side-effects when binding operands is machine and program independent, and is one of the rules incorporated in the idiom discoverer.

The definition of binding idioms above states that the operands being bound must be the same. The same in what sense? Comparison of operands examines several program dependent attributes of those operands. One can define the program dependent attributes of an operand as consisting of an offset, a base register, and an index register, and cover a large class of interesting machines. Certainly, also, the addressing mode (*e.g.*, register-direct, indexed-displacement-deferred, *etc.*) is an attribute of the operand that must be compared. A case can similarly be made for considering the type of an operand as one of its attributes [Bell and Newell 71]. Most machines specify the type of the operands in the instruction, and so the types of operands are inherited from the instruction descriptions. Alternatively, an architecture might specify the type of an operand as part of the operand descriptor, independent of the instruction in which that operand appears. Still more dynamic would be a type tagged architecture, where the type of the operand is part of the data for the operand. This is another example of the difference between target machine dependence (*i.e.*, a function of the instruction set), program dependence (*i.e.*, a function of

---

[1] Throughout this document, I will use a consistent notation for assembler instructions. Operation codes will be shown in **boldface**, with operands in roman, and comments in *italic*. In general, source and destination operands read from left to right. If an operation has a destination, that operand will be the rightmost operand. Destination operands will be usually named 'dest' or occasionally just 'dst'. Any source operands will appear on the left. Operations may have multiple source operands. Source operands will usually be named 'src', or where there is more than one, 'src1', 'src2', *etc.* Occasionally, the destination operand is also a source for the operation. Instruction sequences will appear as

```
add        src1,src2,dest    ; dest := src2 + src1
compare    dest,0            ; if (dest - 0 ...
jless      label             ; ... < 0) then goto label
```

or, where confusion would not result from abbreviation, as "**add**; **compare**; **jless**". Transformations of one instruction sequence to another will be shown in two columns, as for the example above,

```
addl2      src,dest          ↔   addl3      src1,src2,dest
```

the assembler source being translated), and data dependence (*i.e.*, a function of the state of the program during execution). Since operand types can be a function of the instruction set, I include them as operand attributes. Operands are considered "the same" when these attributes (addressing mode, type, offset, base register and index register) are equivalent.

Binding idioms are difficult to identify during code generation because their correct application depends not on the operations of the instructions or operands (the main concern of the code generator) but on the correspondence between attributes of the operands. For example, in a Graham-Glanville-Henry (*e.g.*, syntax directed) code generator, correspondence between operands cannot be checked because the operand attributes are not available in the syntax of the primitive operations (see Section 2.2). If Ganapathi's attributed parsing were used to get around this limitation (see Section 2.3), the case analysis would be done by the description writer and the attribute predicates would be written by hand.

3.3. Set Idioms

A *set idiom* is a special purpose instruction that replaces a more general instruction when one of the operands of the more general instruction has attributes whose values are members of a particular set. For example, the MC68000 **addq** instruction ("quick" constant addition) is[2] a set idiom for the **addi** instruction (unrestricted constant addition) when the addend is in the set {1,...,8}. The usual cases of set idioms are immediate constant addressing modes in which the operand represents a value from a set of constants (*e.g.*, 1, or {1,...,8}). However, immediate constants are only one source of set idioms. Another example is the VAX-11 **pushl** instruction (push a long integer onto the stack), which is a set idiom for the **movl** instruction (move a long integer) when the destination of the **movl** specifies predecrement addressing mode using the stack pointer as the base register. That is:

pushl src ⇔ **movl** src,−(sp)

is a set idiom on the set of addressing mode attributes that reference the stack pointer with the appropriate pre-decrement. The addressing mode of the destination operand of the **movl** is constrained to a single addressing mode, with the base register constrained to a single value. As an example of a set of values that are not a contiguous range of values, the VAX-11 has an addressing mode that can represent any one of a set of predefined floating point constant values.

[2] almost always

Distinguishing set idioms in a retargetable code generator is expensive in terms of the description writer's time, the code generator generator's time, and the size of the tables used at code generation time. First, the description writer must identify attributes (*e.g.*, values of constants) to be distinguished as new primitives. Second, the code generation description must be written to use the new primitives to recognize idiomatic instructions. The larger description takes longer to process and increases the size of the code generation tables. The new distinguished attributes increase the number of primitives recognized by the code generator. The larger tables may take longer to access during code generation.

Sometimes binding and set restrictions must be combined to recognize an idiom. For example, the VAX-11 long integer increment instruction, **incl**, is an idiom for the 3-operand long integer addition instruction, **addl3**:

```
incl    dest        ⇔    addl3    1,dest,dest
```

when one of the sources is the same as the destination and the other source is an immediate (or literal) constant 1.

3.4. Composite Idioms

A *composite idiom* is an instruction that performs the actions of a sequence of instructions. For example, the VAX-11's "subtract one and branch if greater than zero" instruction, **sobgtr**, is an idiom for separate instructions for subtracting, for comparing against zero, and for branching (among other sequences). It may be necessary to check binding and set restrictions to recognize the idiom, for example to see that:

```
sobgtr    counter,label    ; counter := counter − 1
                           ; if (counter > 0) then goto label
```

is an idiom for:

```
decl      counter          ; counter := counter − 1
cmpl      counter,0        ; if (counter − 0 ...
jgtr      label            ; ... > 0) then goto label
```

Composite idioms are characterized by having multiple effects. Every instruction that sets condition codes as a "side-effect" is a candidate composite idiom for a sequence consisting of the instruction followed by a test instruction to (redundantly) set the condition codes. Unless the code generator can track these multiple effects it will be difficult to exploit composite instructions on the target machine. Note that the separate effects may result from separate statements in the source, so a code generator that operates on one statement at a time may miss opportunities to use composite instructions. Some composite instructions are difficult to incorporate into a table-driven code generator because of the buffering they require. For example the **sobgtr** instruction shown above is

an idiom for "**decl; cmpl; jgtr**" only when the comparison is against zero. If the comparison were against anything else, the code generator must generate code for the subtraction. The buffering can be simulated at code generator construction time to avoid dynamic back up at code generation time [Henry 84].

I do not include in composite instructions any instructions with control flow internal to the instruction (*e.g.*, conditional execution, or loops). More extensive analysis is required to use such "complex" instructions (See [Morgan and Rowe 82]).

3.5. Addressing Mode Idioms

The idioms above are discovered by analyzing the computations performed by the instructions of the target machine. On many target machines, addressing modes provide alternative implementations for operations of instructions. Examination of the addressing modes for a target machine may reveal opportunities to relocate operations from instructions into addressing modes.

On machines with complex addressing modes, *e.g.*, automatic scaling of the contents of an index register, or addition of an offset to a base register, a complex instruction sequence using simple addressing modes may be equivalent to a simpler instruction sequence using more complex addressing modes. The computations of the two sequences must be equivalent, but the choice of how to implement the computation might make a difference in the cost of the two sequences. For example, on the VAX-11, both sequences below store the sum of 4 and the contents of 'r0' into 'dest':

addl3	4,r0,dest	⇔	**moval**	4(r0),dest

The sequence on the left performs the addition explicitly with an addition instruction, the sequence on the right folds the addition into an addressing mode that references the address of an offset from a base register. If one implementation of a computation is cheaper than alternative equivalent implementations, the cheapest sequence should be used in place of the more expensive alternatives.

On machines with addressing modes with side-effects, explicit arithmetic instructions can occasionally be subsumed into side-effects. Again, this transformation relocates a computation from an instruction to an addressing mode changed to effect the computation. For example, on the VAX-11, the sequences

subl2	4,sp	⇔	**movl**	src,−(sp)
movl	src,(sp)			

both push 'src' onto a downward growing stack.[3] In the left hand sequence the decrement of the stack pointer is explicit, where in the right hand sequence the decrement is the side-effect of the reference to the stack pointer.

3.6. Idioms Not Searched For (and Not Found)

This section presents four classes of idioms for which my prototype idiom discoverer does not search. These idioms require knowledge beyond that available to my idiom applier, and thus cannot be specified by my idiom discoverer. More research is needed to determine the appropriate retargetable compiler phase in which to perform each of these transformations.

3.6.1. Binary Image Idioms

The idiom applier transforms assembler source code. As such there are idioms on the target machine the idiom applier does not or can not perform. These idioms depend on the binary image of the program, which is not available before assembly. An example is the transformation of operands to use the instruction stream as a source of constants for operands. For instance, the transformation:

```
op      6,6(r)      ⇔    op      (pc),6(r)
```

replaces one copy of an inline constant with a reference to another copy of the constant in the instruction, using the program counter, 'pc', as a pointer [Wulf et al. 75]. This technique can be extended to search the binary image of the program for the desired value. In general, if a constant is found that can be referenced in fewer bytes than the size of the constant, *e.g.*, by a short offset from the program counter, this transformation saves space. The constants need not be visible in the assembler source, as they could include the binary encodings of the instructions as possible sources of data. It is my untested conjecture that such transformations are applicable only rarely.

Another class of binary image transformations not attempted by my idiom applier involve span dependent instructions. These transformations can be handled with existing algorithms in the assembler [Szymanski 78].

An idiom applier cannot transform self-modifying code, since transformations based on program-dependent conditions are evaluated during compilation.

[3] This idiom is incorrect if 'src' uses the stack pointer to reference its operand. See Section 5.7.

3.6.2. Program Flow Graph Idioms

The idiom applier examines assembler source one basic block [Backus et al. 57] at a time. It is therefore unable to recognize several idioms that occur at the boundaries between basic blocks. *Branch chaining* transforms a branch to a branch to transfer directly to the final destination, thus saving a branch at execution time. Since branches end basic blocks, the idiom applier never sees the relationship of one branch to another and thus misses this opportunity. I conjecture that branches to branches are often visible in earlier representations of the program. Where that is the case, the transformation should be performed in the earlier representation.

Other idioms between basic blocks are not visible until code generation is complete. *Cross-jumping* [Wulf et al. 75] merges two code sequences leading to a common point to save space (possibly at the expense of an extra jump during execution). Cross-jumping involves searching the program text for a copy of an instruction, as a previous transformation searched for a copy of a datum. The condition for performing cross-jumping is syntactic and machine independent except for the identification of jumps and labels. Transformations involving branches and their destinations do not fit the simple attributed pattern match and replacement scheme used by my idiom applier.

3.6.3. Available Expression Idioms

Binding and set idioms are predicated on specific values or correspondences of operand attributes in the program being transformed. The predicates for these idioms can be relaxed to check for equivalence, rather than strict equality of attribute values. For example, strict set idiom discovery will identify an "increment by one" instruction as an idiom for an unrestricted addition instruction when one of the source operands of the addition is a constant 1. The predicate on the source operand can be relaxed to specify that the operand must have the value 1 at execution time, rather than requiring that the operand be a constant in the program source code. Such relaxation requires that available expressions [Cocke and Schwartz 70] can be identified at idiom application time [Giegerich 83]. The cost of such relaxed predicates is the cost of computing available expressions and the increased cost of identifying instances of the more general patterns in the program during transformation. More elaborate data flow analysis in the idiom discoverer may uncover more opportunities to apply idioms predicated on available expressions. My prototype idiom applier does not gather available expression data flow information, so my prototype idiom discoverer specifies predicates on operand attributes as equalities rather than equivalences. Note that reusing an available expression reduces the use of the operand that is replaced, possibly freeing resources.

Another form of available expression idiom is demonstrated by the source statements:

```
if (x < y) then goto lessthan
else if (x = y) then goto equalto
else if (x > y) then goto greaterthan
```

A straightforward code generator for a target machine with condition codes might generate:

```
compare   x,y        ; if (x - y ...
jless     lessthan   ; ... < 0) then goto lessthan
compare   x,y        ; if (x - y ...
jequal    equalto    ; ... = 0) then goto equalto
compare   x,y        ; if (x - y ...
jgreater  greaterthan; ... > 0) then goto greaterthan
```

not recognizing that the condition code settings of the first **compare** instruction are common subexpressions redundantly computed by the other **compare** instructions. The prototype idiom applier does not look for transformations based on available expressions. The claim could be made that the condition code settings are not visible until code is generated. Alternatively, the use of condition codes could be exposed as a machine dependency before resource allocation, allowing condition code settings to be optimized by the resource manager in the code generator.

3.6.4. Tree Substitution Idioms

Another class of idioms deliberately not searched for in my idiom discoverer involves tree substitution. A naive code generator may generate instructions to move operands to temporary locations, operate on those temporaries, and then store the result. By substituting the original operands in place of the temporaries (assuming their addressing modes are acceptable) instructions to load the temporaries may be eliminated. This transformation is another form of available expression. Similarly, if the operation can be targeted at the true destination of the computation rather than a compiler temporary, the instruction to store the temporary may be saved. This transformation is called *target path discovery* [Leverett 79]. Both these optimizations require the identification of compiler temporaries.

Copy propagation substitutes the source of a **move** instruction in place of the use of the destination of the **move** instruction. Target path discovery substitutes the destination of a **move** instruction in place of the definition of the source of the **move** instruction. Both these transformations are special cases (for **move** instructions) of embedding the computation of one instruction into another instruction. Consider three

instructions "**foo** src,dest", "**bar** src,dest", and "**foobar** src,dest" in which **foo** and **bar** perform distinct operations and **foobar** performs the composition of those operations. The transformation from:

```
foo     src,temp        ; temp := foo(src)
bar     temp,dest       ; dest := bar(temp)
```

into:

```
foobar  src,dest        ; dest := bar(foo(src))
```

could be performed if one knew that 'temp' were a compiler temporary and that its use in the **bar** instruction was its only use (which annotation would have to be provided by the code generator or resource manager). Copy propagation and target path discovery can be modeled (and are so modeled in the idiom discoverer in [Kessler 84], and the idiom applier in [Davidson and Fraser 80]) by substituting the computation tree for the **foo** instruction in place of the source operand of the **bar** instruction and searching for an instruction that performs this combined computation. Extending this idea, one must analyze sequences in which each operation is performed by a separate instruction. Thus,

```
foobar  src,dest        ; dest := bar(foo(src))
```

might be implemented by:

```
foo     src,temp        ; temp := foo(src)
bar     temp,dest       ; dest := bar(temp)
```

or, to expose the implicit assignments, as:

```
move    src,temp1       ; temp1 := src
foo     temp1,temp2     ; temp2 := foo(temp1)
bar     temp2,temp3     ; temp3 := bar(temp2)
move    temp3,dest      ; dest := temp3
```

This fragmentation of the instructions reduces the program to register transfers. Recognition of more complex instruction patterns (and address modes, which can also be composed of several operations) from these register transfers is exactly code generation. Our Graham-Glanville-Henry code generator is an excellent pattern matcher, therefore this kind of decomposition and regeneration of code is not part of the prototype idiom recognizer. In contrast, Davidson and Fraser have a very naive code generator that produces register transfers. These register transfers are then combined via peephole optimization into addressing modes and instructions (see Section 2.5).

Another problem with tree substitution transformations is the allocation and deallocation of compiler temporaries. If an idiom uses different resources than the code it replaces, then the idiom applier should request those resources from, or return them to, the resource manager [McKusick 84]. If an idiom uses fewer registers than the code it

replaces, the idiom recognizer does not have the data flow information necessary to reuse those registers and, in the pathological case, to remove any stores generated to use those registers. Machine-specific improvements are often applied after code generation and register allocation. The notable exception is Davidson and Fraser's PO, which selects instructions before register allocation. There are many opportunities during compilation for both machine-specific and machine-independent transformations of a program before resource allocation. An area of future research is the automated generation of such transformations. None of the idioms discovered by my prototype idiom discoverer frees any compiler temporaries. Any idiom that would free temporaries should be applied in time for the resource manager to take advantage of the transformation.

Recall the transformation "**foo; bar**" ⇔ "**foobar**" above. The instructions are related by a compiler temporary variable. Therefore, at some stage of the translation, the **foo** and **bar** operations were "adjacent" and the compiler decided to implement them with separate instructions defining and using the compiler temporary. A Graham-Glanville-Henry code generator would not miss this chance to use the **foobar** instruction to implement adjacent **foo** and **bar** operations, so this restriction on the prototype idiom discoverer has not been a problem in practice. A glaring counterexample, however, is the use of an **exchange** instruction when the code generator emits three **move** instructions to perform the exchange (presumably because the exchange operation was not explicitly identified, machine dependently, in the input to the code generator). Thus one might want to recognize that:

```
move    x,temp          ⇔   exchange   x,y
move    y,x
move    temp,y
```

is an idiom on machines with an **exchange** instruction. I believe that since 'temp' is a compiler temporary, the intent of the transfers is apparent in the input to the code generator, and should be recognized as an **exchange** operation when code is generated.

3.6.4.1. Duals to Binding and Set Idioms

Tree substitution can also be used to discover the dual idioms to binding and set idioms. With binding and set idioms, my idiom discoverer constructs predicates on the program that must hold in order to replace a general instruction with a special purpose instruction. Tree substitution can be used to construct idioms in which the special case instruction should be replaced by the more general instruction. Consider the binding idiom:

```
op3     src1,src2,dest     ⇔   op2     src1,dest
```

The predicate here is "src2 ≡ dest", which will be satisfied if the **op2** instruction is preceded by an assignment of 'src2' to 'dest', thus:

```
op3     src1,src2,dest     ⇔   move    src2,dest
                               op2     src1,dest
```

Whereas the original "**op2**" was preferable to "**op3**" because it required one fewer operand reference, "**op3**" is now preferable to "**move**; **op2**" for the same reason. This example may not seem like tree substitution, but consider how this idiom might be discovered. That is, where binding idiom discovery would have extracted the predicate "src2 ≡ dest", tree substitution could ensure the predicate by substituting the assignment of 'src2' to 'dest'.

There are several reasons why tree substitution is not performed in the prototype idiom discoverer. Chief among these reasons is that the Graham-Glanville-Henry code generator does not produce the load–operate–store sequences whose improvement is the main purpose of tree substitution idioms. Another observation is that substitution of assignments is a potentially unsafe transformation, since it relies on (or may change) the order of evaluation of operands. Even the apparently safe transformation of:

```
op3     src1,src2,dest     ⇔   move    src2,dest
                               op2     src1,dest
```

is incorrect if 'src1' and 'dest' refer to the same storage location. Since my idiom applier does not have the data flow information to preclude this aliasing, my idiom discoverer may not specify a transformation predicated on such a restriction. Since the Graham-Glanville-Henry code generator can not verify the constraints for binding idioms, that code generator would never generate the **op2** in the sequence, preferring to use the more general **op3**. Therefore the only reason to generate the **move** instruction is if the assignment is explicit in the input to the code generator. Available expression elimination, a machine-independent tree transformation implementing a subset of tree substitutions, could remove the assignment, in that case. Machine-independent transformations are to be preferred to (even automatically generated) machine-dependent transformations. Tree substitution idioms are better performed by a tree transformation phase than by the limited pattern matcher in an idiom applier.

3.7. Idiom Discovery

The previous section described the idioms discovered by my prototype; this section discusses the techniques used to discover idioms from the target machine description. Several alternative methods of searching for idioms are examined. This section describes

an idiom discoverer, not an idiom discoverer generator. The idioms discovered on a target machine are those for which I have designed the idiom discoverer to search. On a radically different architecture with radically different idioms, I would not expect the prototype idiom discoverer to discover many idioms. My research separates idiom recognition into description driven idiom discovery and table driven idiom application. The separation of an idiom discoverer into architecture driven meta-idiom discovery and table driven idiom discovery is beyond the scope of this dissertation. The idiom discoverer does not compete with assembler source programmers, who use axioms outside those available to the idiom discoverer to employ obscure instructions in devious ways. On the other hand, the idiom discoverer is consistent about the idioms discovered from one target machine to the next, and the idiom applier is more consistent about applying those idioms than most assembler source programmers.

3.7.1. Instruction Comparison

The main activity of idiom discovery is the matching of one sequence of instructions, called the *subject*, against another sequence of instructions, called the *pattern*, to see if they perform the same computation. If the two sequences match, the subject is an idiom for the pattern.[4] The most basic decision in designing the idiom discoverer is how to construct the sequences of instructions to compare.

3.7.1.1. Composition versus Decomposition

Previous optimizer generator systems [Davidson 81] [Kessler 81] have used brute force *composition* to construct sequences of instructions. Using composition, a pair of instructions are combined to form the pattern, which is matched against a succession of subjects (which in those systems are single instructions). The matching identifies any possible subjects that perform the computation represented by the pair of instructions in the pattern. The cheapest subject is selected as the replacement for the pattern, and the pattern and replacement are entered into a table for the idiom applier. After a pattern has been matched (successfully or otherwise) against all subjects, another pair of instructions is composed to form a new pattern that is then compared against each subject. Intuitively, composition generates sequences of code (the patterns) and then searches the target machine for a subject that performs that composite computation. For example, a decrement instruction and a conditional branch might be composed into a pattern and compared successfully to a subject consisting of an "subtract one and branch if less than zero" instruction.

[4] The idiom may not be an improvement, but that can be determined from cost information.

For n instructions, there are n^2 pairs of instructions that must be composed and matched against each subject. (If one also wishes to consider single instruction patterns, then there are an additional n patterns to be matched, which does not change the complexity of the composition process.) It is obvious how to extend this algorithm to consider patterns composed of triples of instructions. For a target machine with a moderate number of instructions it is probably impractical to consider composing triples.

Instead of composing patterns as the inner loop of idiom discovery, my idiom discoverer *decomposes* the subject. The subject sequence is compared against a single instruction as the pattern. The comparison either mismatches, partially matches, or completely matches. If the match is complete, an idiom has been identified in which the subject can replace the pattern. If the pattern mismatches it is discarded. A *partial match* is defined as one in which the pattern matches a contiguous portion of the subject, but part of the subject sequence remains unmatched. For example, a pattern consisting of a decrement instruction partially matches a subject "subtract one and branch if less than zero" instruction, leaving only the conditional branch unmatched. In this case, the pattern is extended by an additional instruction, and the comparison against the unmatched portion of the subject continues. In the example above, an appropriate conditional branch instruction can be used to complete the partial match of the decrement instruction above. Since I am only interested in partial matches that can be extended to complete matches, and I am not interested in permutations in the order of matching portions of the subject, I restrict partial matching to the beginning or end of the pattern sequence. The choice of from which end to start is discussed in Section 3.7.1.2. Intuitively, decomposition breaks down the subject into a sequence of instructions. The *length of a (partial) match* is the number of instructions that form the (partially) matching pattern.

Only the successful partial matches of length 1 are extended into patterns with pairs of instructions. This algorithm has a natural extension to (and beyond) patterns with triples of instructions, since partially matching pairs can be extended to triples in the attempt to completely match the subject. Decomposition derives all patterns that perform the same computation as the subject. Each pattern that is more costly than the subject is entered into the table for the transformer, with the subject as its replacement.

To contrast decomposition to composition, I will contrast the number of instruction comparisons needed to discover idioms of a given subject. Let n be the number of instructions on a target machine. Composition always uses n^2 comparisons, but only discovers matches of exactly length 2. Let p_l, $l \geq 0$ be the number of partial matches to the subject of length l. By definition, p_0, the number of partial matches of length 0, is 1, since any subject is partially matched by the empty instruction sequence.

Lemma 1: There are $n \times \sum_{l=0}^{k-1} p_l$ comparisons required to find all complete matches of length k, and all partial matches of up to length k.

The proof is by induction on k.

Basis: $k = 1$. Each of the n instructions is matched against the subject, requiring n comparisons. Any complete matches are of length 1, and n comparisons were used to discover them. In addition, p_1 partial matches of length 1 have been identified. So the basis of the induction is established.

Induction Step: assume $n \times \sum_{l=0}^{k-1} p_k$, comparisons are required to find all complete and partial matches of up to length k. Each of the n instructions is matched against the p_k partial matches of length k. Any complete matches or partial matches are of length $k+1$. $n \times p_k$ comparisons are needed to discover these matches, in addition to the $n \times \sum_{l=0}^{k-1} p_k$ comparisons to find the p_k partial matches. The total number of comparisons used to find all the complete and partial matches of up to length k is

$$n \times p_k + n \times \sum_{l=0}^{k-1} p_k = n \times \sum_{l=0}^{k} p_l$$

which extends the induction. □

Therefore, if the longest match needed to decompose a subject is of length m, $n \times \sum_{l=0}^{m-1} p_l$ comparisons are used to decompose that subject.

Lemma 2: $p_l \leq n^l$.

The proof is by induction on l.

Basis: $l=0$. p_0 is 1 by definition, and n^0 is 1, so the basis of the induction is established.

Induction Step: assume $p_l \leq n^l$. There are n instructions to extend each of the p_l partial matches of length l. These extensions generate at most $n \times p_l$ partial matches of length $l+1$. Since $p_l \leq n^l$ by the induction hypothesis,

$$\begin{aligned} p_{l+1} &\leq n \times p_l \\ &\leq n \times n^l \\ &\leq n^{l+1} \end{aligned}$$

which extends the induction. □

The above two lemmas prove the following theorem.

Theorem: To discover patterns that are pairs of instructions, decomposition performs n^2 comparisons in the worst case. For patterns that are extended to triples, the number of comparisons is n^3 in the worst case, and so on.

The degree of the polynomial for the number of comparisons is a function of the complexity of the subject instruction sequence. This is as it should be: more complex subjects require more examination. The ability to extend patterns beyond pairs of instructions is a significant improvement over the brute force composition methods. In the best case, exactly one instruction extends each partial match, and the number of comparisons needed to decompose a subject to a complete match of length k is $n \times k$.

The worst case number of comparisons is extremely unlikely: every instruction would have to partially match at each extension. In practice, few instructions successfully extend the match, since architectures provide only a few implementations for each operation. If the subject sequence is limited to single instructions, as it is in my prototype, the maximum length of the complete matches is also small in practice, since instructions are not very complex. In addition, the extended pattern need not be compared against the full subject, since it is known that the majority of the pattern already matches. Rather, only the unmatched portion of the subject need be compared against the proposed extension to the pattern. In this way the common parts of extended patterns are compared only once.

It has been suggested that idioms discovered early in a target machine analysis could be used to reduce the number of comparisons needed to find later idioms. For example, once equivalent instruction sequences have been found for a subject, any patterns that are successfully extended by that subject are also successfully extended by the instruction sequences equivalent to that subject. Obviously, an ordering problem is imposed on which idioms to discover first to speed up subsequent idiom discovery. Such an ordering cannot be imposed *a priori*. One argument against implementing a caching scheme is that idiom discovery, even for a complex machine, does not take an unreasonable amount of time (see Section 5.9). Also, it is not clear that the space required to cache the idioms, and the time required to look them up during matching and to unify their restrictions with the restrictions imposed on a partial match are worth the time that might be saved by caching. Caching is not useful when analyzing small subsets (*e.g.*, single subjects) of a target machine.

3.7.1.2. Forward versus Backward Comparison

The direction in which pattern instruction sequences are extended is another choice in the design of the idiom discoverer. Pattern sequences may be extended at the end, and compared to the beginning of the subject sequence; or, they may be extended at the

beginning, and compared to the end of the subject sequence. Appending extensions and comparing "forward" is more natural: such extension is how code generation proceeds. Prepending extensions and comparing "backward" has several advantages for idiom discovery.

To see how the two extension techniques work in idiom discovery, consider a machine with three instructions: **op_cc**, that performs some computation and also sets the condition codes; **op_nocc**, that performs the same computation but does not set the condition codes; and a **test** instruction, that sets condition codes. With **op_cc** as the subject instruction, forward decomposition matches the pattern instruction **op_nocc** and discovers a partial match. The unmatched remainder of the subject is the setting of the condition codes. The forward technique extends the end of the pattern with a **test** instruction, which completes the match. Thus the composite idiom:

op_cc	src,dst	⇔	**op_nocc**	src,dst
			test	dst

is discovered. (A critical reader may argue that "**op_nocc; test**" only sets condition codes based on the value of the result, where "**op_cc**" may set condition codes based on exceptions, *e.g.*, overflows or carries, during the computation. The idiom is conditional on the values of such extra condition codes being extraneous, that is, dead. See Section 3.7.2.)

The backward decomposition technique begins with the **test** instruction as the pattern and matches the **test** against the end of the subject **op_cc** instruction, discovering a partial match. The unmatched remainder of the subject is the main computation. The backward technique extends the beginning of the pattern with an **op_nocc** instruction to arrive at the complete match. The same idiom is discovered by both techniques.

Consider, however, the idiom discussed earlier:

sobgtr	counter,label	⇔	**decl**	counter
			cmpl	counter,0
			jgtr	label

The **decl** instruction sets the condition codes to the values used by the **jgtr** instruction. These condition code values are redundantly computed by the **cmpl** instruction. A forward idiom discoverer will begin by matching the pattern **decl** instruction against the subject **sobgtr** instruction. The remaining unmatched subject will be just the conditional jump instruction. The **cmpl** instruction will not be considered by the forward algorithm as an extension to the pattern sequence. The backward strategy begins by matching the conditional branch instruction against the subject. The remaining unmatched subject is the setting of the conditions codes for the branch and the increment of the counter. The

backward idiom discoverer will prepend a **cmpl** instruction to the pattern to extend the sequence. Finally, the **decl** instruction is prepended, completing the match and discovering the idiom. (An alternative extension to the **cmpl** is just a **decl**, which extension will also be tried, leading to the discovery of a different idiom, "**sobgtr**" ⇔ "**decl; jgtr**".) The point of the backward technique is that the subject is decomposed into, among others, pattern sequences with as many useful instructions as possible. In this way the pattern sequence mimics the breaks between statements that may cause the code generator to produce separate instructions. An implementation of backward decomposition is described in Chapter 5.

Backward decomposition is similar to the bottom-up pass of a dynamic programming code generation algorithm [Ripken 77][Aho and Johnson 76]. However, where dynamic programming is concerned with choosing an optimal decomposition of a sequence, the idiom discoverer is interested in all possible decompositions of the subject sequence.

3.7.2. Data Flow Context Information

The effect of an instruction varies depending on the data flow context in which the instruction appears. That is, if an instruction defines a value for a variable, but the instruction appears in a context in which that value is unused, then that effect of the instruction can be ignored. The data flow context can be used in two ways during idiom discovery. Computations may be ignored if it is known during idiom discovery that the values of those computations are never used. Alternatively, idioms may be predicated on the condition that certain values are dead. Data flow analysis identifies "live" variables, whose values are used in subsequent computations, and "faint" variables (Robert Giegerich's strengthening of "dead" variables), whose values are not used, or are used only to compute values for other faint variables. Data flow context information is kept as an attribute of the partial match.

Live and dead variable analysis is commonly performed via a backwards pass [Allen and Cocke 76]. Prepending instructions and matching backwards during idiom discovery allows live and dead variable information to be maintained for partially matching patterns. Data flow information can be updated as assignments are encountered during matching. The idiom discoverer maintains live and dead variable information based on the instruction descriptions, without the program dependent details of the operands. For this reason data flow predicates refer only to processor state locations, whose definitions and uses are visible in the instruction descriptions.

Recall the idiom "**sobgtr**" for "**decl; cmpl; jgtr**". The **jgtr** instruction uses several condition codes, so the values of those condition codes become live. The **cmpl** instruction sets those condition codes, so their values become dead. Since the values of

the condition codes are dead by the time the decl is prepended to the pattern, the condition code settings of the decl may be ignored during matching. This data flow context information may be necessary for extending matches.

Data flow context information effectively changes instruction descriptions by ignoring assignments to variables whose values are dead. Consider the "decl" instruction as it is prepended to "cmpl; jgtr" in the attempt to decompose the "sobgtr" subject. The "decl" instruction itself is a sequence of a decrement and assignment to its operand, and the assignments to the condition codes reflecting the outcome of the decrement. As the values of the condition codes are dead when the "decl" is prepended to the "cmpl; jgtr" pattern, the assignments to the condition codes can be removed before matching proceeds. Without this pruning of dead assignments the "decl" would fail to match the remainder of the "sobgtr". An alternative method of comparing assignments is to iterate through all possible subsets of a forest of assignments. Even for modest sized forests such iteration causes an unacceptable increase in the time required for matching. Maintaining data flow context information and pruning effectively generates the maximal subset of each instruction that extends the match.

Alternatively, consider an attempted match of assignments that fails because of a mismatch in the source expressions of the assignments, though the destinations of the assignments match. If the value of the destination can be declared dead, the source expressions can be ignored, and the match will succeed. Since data flow context information is one of the attributes of the partial matches, it can be manipulated by the idiom discoverer to record this condition on the match (see Section 5.6.4). The resulting idiom is conditional on the data flow of the program being transformed. The idiom applier must verify this data flow context condition before it can apply the idiom.

The idiom applier gathers data flow for a particular program with a backwards pass as idioms are applied (see Section 6.1). The idiom applier needs to maintain live and dead variable information only for the variables that can be specified in predicates of idioms. The idiom applier gathers data flow information for a program using summary data flow information for instructions. This summary data flow information lists those processor state variables used before being defined, those defined before being used, and those killed by each instruction. The summary data flow information is developed during the analysis of the target machine at idiom discovery time. .

CHAPTER 4

Machine Descriptions

The machine description used by the idiom discoverer is designed to be easily constructed by the description writer, while providing the information necessary to identify idioms in the target machine. The interesting part of a machine description is a characterization of every instruction and addressing mode of the target machine. In addition, the description includes cost information about instructions and addressing modes, data types defined by the target architecture, and state variables maintained by the target processor.

The machine description is factored in several ways to reduce the size of the description and the tedium of constructing a complete description of the target machine. The following sections explain the components of a description, how a description is factored, and how the components are related.

4.1. Trees

The effects of instructions and addressing modes are described by trees. An alternative would be a procedural description, like ISP [Bell and Newell 71]. Procedural descriptions are not used because the main activity of idiom discovery is comparing instructions. Comparing procedural descriptions might require that one solve the "program equivalence" problem, which is undecidable in general. In contrast, trees can be compared by a simple tree walk in time proportional to the number of nodes in the trees. Trees lend themselves to substitution, *e.g.*, replacing a reference to an operand with the tree describing an addressing mode, and to pruning, *e.g.*, of dead branches.

The form of the trees used in machine descriptions is a parenthesized pre-order traversal of the trees. The trees must be linearized in the machine description, and pre-order traversal is a convenient linearization for both the description writer and the idiom discoverer. The use of parentheses avoids having to specify the arity of the operators of the trees. Tree operators may be made up as needed by the description writer to represent operations of the target machine. Made-up operators have meaning only to the description writer, except that they are assumed to represent some function of their subtree arguments. These operators may take any number of arguments, and may be

polymorphic: taking arguments and yielding results of arbitrary type.

All of the built-in polymorphic operators represent operations on homogeneous arguments yielding results of the same type as the arguments. A convention I have adopted is that polymorphic operators take the type of their operation as a first argument. This convention must be adopted by other description writers when using the built-in operators in my system. Alternative notations include type-crossing the operators (cf. [Henry 84]), that is, creating a new operator for each type qualification of an operator; or using the type as a unary operator qualifying a subtree (cf. [Crawford 82]). The type-crossing alternative is rejected because it becomes difficult to separate the type qualifier from the operator. When gathering data flow information, for example, one needs to identify assignment operators. Having a different assignment operator for each type complicates this identification. Similarly, type-crossing makes it harder to compare attributes of the types of operations without also considering the operations. Using a type as a unary operator separates the type from the operator, but inconveniently. Subtrees become rooted by their type qualification, rather than the "real" operator, making identification of the "real" operators more complex, *e.g.*, for axiomatic transformations. The convention of specifying a type qualifier as an operand to an operator keeps the type separate from the operator, yet makes the type conveniently part of the tree rooted at the operator. Since polymorphic operators that are type qualified must always have the type qualification, the arity of these operators is uniformly changed.

The prototype idiom discoverer is implemented in LISP, so it is convenient to write machine descriptions as LISP *s-expressions*. Much of the "syntactic sugar" in the machine descriptions is based on LISP. LISP notation is particularly convenient for describing trees and lists. Additionally, LISP provides an environment that facilitates writing and maintaining the descriptions. Part of the LISP environment is macro expansion, allowing macros, possibly defined by the description writer, to simplify the construction of the machine description.

4.2. Properties of Tree Operators

The syntactic descriptions of instructions are unlikely to be sufficient for discovering idioms, since it is unlikely that a target machine will have two instructions with syntactically identical descriptions. Therefore, certain tree operators have semantic properties that are known to the idiom discoverer. For example, there is an operator to represent assignment (←) to permit data flow analysis. A complete list of the built-in operators and leaves appears in Figure 4.1. The semantics of built-in operators cannot be changed easily by the description writer.

Built-in Operators	*Interpretation*
(→ type src dest)	*assignment of src to dest.*
(**operand** name)	*all attributes of name.*
(**parallel** tree ...)	*unordered forest of trees.*
(**sequential** tree ...)	*ordered forest of trees.*
(**cond** (tree tree) ...)	*alternative trees.*
(**jump** label)	*transfer of control to label.*
(**constant** type value)	*a constant.*
(**constant-list** type value element element ...)	*a constant in {element,element,...}*
(**constant-range** type value lower upper)	*a constant in [lower,...,upper]*
(**type-size** type)	*size attribute of type.*

Built-in leaves	
type	*factored type.*
offset	*offset of an addressing mode.*
basereg	*base register of an addressing mode.*
indexreg	*index register of an addressing mode.*
unique	*unmatchable tree.*

Figure 4.1. Built-in Operators and Leaves.

Most instructions compute more than a single result: *e.g.*, computing, say, an addition and setting condition codes based on the result of that addition. Several effects of an instruction might be considered "side-effects", but there is no such distinction in the machine descriptions. Multiple effects of an instruction are modeled by forests of trees. Often the trees of a forest must be ordered by data flow constraints: *e.g.*, in the previous example, the addition must be performed before the condition codes can be set. The ordering of trees of a forest is represented explicitly in instruction descriptions. There is an operator, **sequential**, to indicate sequential computation of subtrees. Not all trees need to be strictly ordered, a partial ordering is sufficient. For example, assignments to several condition codes need not be ordered. An operator, **parallel**, can be used to root subtrees that can be examined in any order. Parallel subtrees form an equivalence class in the ordering of trees. An unordered forest may not be decomposed into a sequence of trees, since to do so would impose an ordering on the members of the equivalence class. Finally, a forest of trees may represent alternatives. A forest may be collected as subtrees of a **cond** operator in pairs representing guarding predicates and guarded computations.

Operators made up by the description writer have no semantics imposed on them by the idiom discoverer. These operators are matched syntactically. To allow for axioms on these operators in the implementation of the idiom discoverer, the description writer may provide a function for each operator, to be called each time the operator is being matched. The function maps a single tree to a list of trees, each of which will be matched in place of the original. For example, the function for a commutative operator may return a list of the original tree and a copy of the original with the subtrees commuted. This mechanism is the only semantic escape for operators made up by the description writer.

4.3. Type Descriptions

Type descriptions serve two purposes. The first purpose is to provide a list of all type symbols so they can be distinguished during matching. The second purpose is to provide a set of fixed attributes for each type. Type symbols have two attributes: one to map a type to a set of equivalent types, and the other to specify the size of the type (a unitless measure). Logical types may be depicted in addition to the hardware types available on the target machine. For example, it is convenient to specify a type, *type-address*, for qualifying addressing mode arithmetic. Type type-address could then be defined as equivalent to the hardware type for long integers, *type-long*, so that an operation on addresses would be recognized as equivalent to the same operation on long integers.

Types appear in instructions and addressing mode descriptions as qualifiers on operations. References to the size of a type may also appear in description trees. The built-in operator **type-size** is polymorphic, taking a type as its only argument.

For example, the description for the type of an address is given by:

```
(define-type type-address
  (make-type
   name        "address"      ; for diagnostics.
   size        4              ; size of the type.
   equivalents '(type-long))) ; list of equivalent types.
```

where the first two lines define the symbol *type-address* as a type, and each of the remaining lines gives an attribute of the type.

4.4. Processor State Variables

Changes to the processor state are an important part of instruction execution. (In fact, for test instructions that set condition codes, changes to the processor state are the only effect of execution.) These changes must be preserved during idiom application, and must be compared during idiom discovery. These changes are represented in the

description trees as assignments to distinguished processor state variables, *e.g.*, condition codes, general registers, *etc.* The program counter is not considered part of the processor state. Though the program counter is affected by instruction execution and operand evaluation, the program counter is not a part of the description writer's virtual machine: *e.g.*, it is not affected explicitly in instruction descriptions.

Processor state variables appear as leaves in the trees describing instructions and addressing modes. The idiom discoverer matches them syntactically; in this respect they are fairly uninteresting. Data flow information is maintained for the processor state variables, both during idiom discovery and idiom application. Thus these variables are by definition interesting, as they complicate matching and transformation (see Section 5.6.3, Section 5.3, and Section 6.1). Data flow analysis assumes that processor state variables are unique names by which the processor state is changed. If processor state variables have aliases, some technique like that described by Robert Giegerich [Giegerich 83] would be needed to gather data flow information at idiom discovery time.

4.4.1. Limitations on Flow Analysis

There are two limiting factors on data flow analysis. First, many of the operators in the description trees have no associated semantics. Thus operations in the destination subtree of an assignment have unknown effects. For example, the destination tree may represent indirection through an address, which does not assign to the address itself. Definition points of variable contents can only be noted when the variable is the immediate destination of an assignment. Any other references must be treated as uses of the variable. Better data flow analysis would require the description writer to distinguish trees (as opposed to leaves) to be tracked. Such specification is not unreasonable, but it has not proved necessary. Second, some processor state variables may be referred to in particular programs by being specified as operands of instructions. Thus summary information describing the effect of instructions on these variables may be gathered by the idiom discoverer, but that information must be qualified by the idiom applier when a particular program is transformed. For example, the contents of a register may be changed explicitly in an instruction description, or implicitly by specifying the register as an operand of an instruction.

To alleviate the restrictions on data flow analysis during idiom discovery, processor state variables are separated into two groups: those variables that can appear in addressing modes of instruction operands, and those variables that can only appear explicitly in instruction descriptions. It may help to think that the latter processor state variables are *addressable* as instruction operands, while the former processor state variables are unaddressable, explicit, *locations*. For example, condition codes that may be

set or tested only explicitly in instructions would be locations, whereas general registers might be addresses of processor state variables.

The VAX-11 description contains the following processor state variables:

```
(define-processor-state-locations '(cc-n cc-z cc-v cc-c))
(define-processor-state-addresses '(r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 ap fp sp pc))
```

declaring the condition codes as locations and the general registers as addresses.

4.5. Cost Information

Both the instruction descriptions and addressing mode descriptions specify costs associated with their use. Space costs, in units of the instruction stream, and time costs, in processor time, may be given. In the prototype idiom applier, costs for a collection of addressing modes and instructions are assumed to be additive. It would be better to have the description writer provide functions to sum and compare costs. For example, the cost of an instruction may be a function of the addressing modes of its arguments. Such functions could manipulate structured costs, if necessary.

4.6. Addressing Modes

Addressing mode descriptions factor the trees for operands from the description of the instructions. Without this separation, each instruction would have to be described with each possible addressing tree for each of its operands. This repetition is tedious and impractical for machines with many addressing modes and multiple operand instructions. Addressing mode factoring allows the common part of an instruction to be specified once, with operands as parameters, and conversely allows the operand descriptors for addressing modes common to several instructions to be described once.

Operand references in instructions may refer to the effective address of the operand or may refer to the contents of the operand. Usually the contents of the operand is the contents of the effective address. This is not always the case, and an addressing mode need not allow both references. For example, on the VAX-11, register-direct addressing mode does not have an effective address. Trees describing each reference supported by an addressing mode are included in the description of the addressing mode.

Addressing mode descriptions are parameterized by an offset, a base register, and an index register. References to these parameters may appear as leaves in the trees describing the addressing mode. The actual values for the parameters of an addressing mode are attributes of operands. In general, these actual parameters are not known during idiom discovery. Binding these parameters to a set of values, or binding the parameters of one operand to the parameters of another operand is one source of idioms

on the target machine. Address mode descriptions are also parameterized by types. References to the type of an operand can qualify the operations in trees describing the addressing mode. References to attributes of the type of an operand (*e.g.*, the size of the type) may also appear as operations in description trees.

Address mode descriptions include cost information. Time costs are specified both for accessing the effective address of the operand and for accessing the contents of the operand. Space costs for both accesses are assumed to be the same, and are given by a single specification. The cost functions are parameterized by type, since, for example, a reference to a byte operand may cost less than a reference to a long operand. The cost functions are not parameterized by the values of other operand attributes. If, for example, byte offsets are cheaper than long offsets, separate addressing mode descriptions are used.

An assumption implicit in address mode factoring is that references to operands in instruction descriptions are free of side-effects. Many target machines have addressing modes with side-effects, but these effects occur during operand evaluation, not instruction execution. If an addressing mode has side-effects during evaluation, these effects also appear in the description. Side-effects are classified as pre-effects if they occur before the reference to the operand, or as post-effects if they occur after the reference to the operand.

For example, autoincrement addressing mode on the VAX-11 is described as shown in Figure 4.2, except that the space and time costs have been omitted for brevity. In this example, the pre-effect tree is empty, the post-effect tree portrays the increment to the base register parameterized by the type of the operand. The "classes" field is described below.

4.7. Address Mode Classes

Address modes are not referenced directly by operands of instructions. Usually an operand is not limited to a single addressing mode. Some notation is needed for sets of addressing modes. Orthogonal architectures define a reasonably small number of distinct sets of address modes: *e.g.*, readable modes, including constants and registers, and writable modes, including registers but not constants. To follow this regularity, the machine description provides address mode classes. Each required set of addressing modes defines a new address mode class. A reference to an operand is interpreted as a reference to all the addressing mode trees of the addressing mode class of the operand. The members of the addressing mode class are not enumerated explicitly; each addressing mode declares the classes to which it belongs, and the classes are constructed from those declarations.

```
(define-mode postinc-mode
  (make-mode
   ;                    for debugging.
   name                 "postinc-mode"
   ;                    mode classes to which addressing mode belongs.
   classes              '(mode-class-r mode-class-float-r
                          mode-class-w mode-class-m
                          mode-class-a mode-class-fr
                          mode-class-fw mode-class-fm)
   ;                    types that addressing mode can access.
   types                '(type-byte type-word type-long type-quad type-octa
                          type-ffloat type-gfloat type-dfloat type-hfloat)
   ;                    there is no pre-effect.
   pre-effect           (empty-tree)
   ;                    effective address is the contents of the base register.
   address-tree         (new-tree
                         '(contents type-address basereg))
   ;                    operand is the contents of the effective address.
   contents-tree        (new-tree
                         '(contents type
                                    (contents type-address basereg)))
   ;                    post effect tree: basereg := basereg + type-size(type)
   post-effect          (new-tree
                         '(→ type-address
                             (+ type-address
                                (contents type-address basereg)
                                (constant type-address (type-size type)))
                             (contents type-address basereg)))))
```

Figure 4.2. Post-Increment Addressing Mode.

4.8. Instruction Descriptions

Instruction descriptions record information about the instructions of the target machine. The interesting part of each description is the tree giving the actions performed by the instruction. The usual instruction tree is a sequence of operations on the operands and assignments to processor state variables. This sequencing is the source of the idiom discoverer's decomposition into simpler instructions. In addition, the operands to the instruction are described and cost information noted.

Each operand is specified by a formal name, a type, an access kind, and an address mode class. Formal names are used to represent the operand in the tree for the

instruction. (The type, access kind, and address mode class of the operands are inherited, on the machines examined, from the instruction. On a different architecture, these attributes might be properties of the operands themselves, and thus more dynamic than the static attributes described here.) For each operand it is specified whether the instruction accesses the effective address or the contents of the operand. This attribute is used to select the appropriate reference tree from the address mode descriptions. The type of the operand is used to qualify the operations in the address mode trees.

All instructions in the target machine are enumerated. Each instruction need not be described in great detail, or may be described as performing a unique operation, *e.g.*, for particularly complex instructions. However, at least the effect of the instruction on the processor state variables must be described, to permit data flow information to be gathered across an instance of the instruction.

Address mode factoring exploits the regularity of instruction sets with respect to addressing modes for operands. A similar regularity exists with respect to the types of operands. One might find "move" instructions, for example, for each of the data types on the target machine. The trees describing those instructions are all identical except for the type qualifiers of the operations. Similar consistency can be seen among descriptions for arithmetic operations. Therefore instruction descriptions parameterized by type are significantly less tedious to construct, maintain, and examine than when the types are instantiated in the descriptions. Type factoring significantly cuts down on the size of the instruction descriptions. It has the unfortunate effect that the description writer is no longer describing the instructions of the target machine (*e.g.*, **subb2**, **subb3**, **subw2**, **subw3**, and so on) but type parameterized instruction families. (*e.g.*, **sub[bwl]2**, **sub[bwl]3**). The cost for instructions is also parameterized by the type of the instruction. I did not explore factoring instruction descriptions by operator, *e.g.*, to take advantage of the structural similarity between addition instructions and subtraction instructions.

For example, the description of the VAX-11's **sub[bwl]3** instruction family (three operand byte-, word-, and long-integer subtraction) is shown in Figure 4.3. Three instructions are described, named **subb3**, **subw3**, and **subl3**. The type and costs for each instruction are represented in the corresponding positions of the the appropriate fields of the description. The operand field describes three operands, with formal names, types (here, parametric), accesses, and addressing mode classes. Finally, the description tree depicts an ordered forest containing the subtraction and the setting of the condition codes.

The subtraction shown is the subtraction of the first operand from the second operand, and the assignment of that result to the third operand. Note that most of the

```
(define-instruction i-sub3_bwl
  (make-idesc
   ;              names, types and costs, factored by type
   name           '("subb3"      "subw3"      "subl3")
   types          '(type-byte    type-word    type-long)
   space          '(1            1            1)
   time           '(1            1            1)
   ;              operand descriptions
   operands       `(,(make-op-attr
                      name        'arg1
                      type        'type
                      access      'access-contents
                      mode-class  'mode-class-r)
                    ,(make-op-attr
                      name        'arg2
                      type        'type
                      access      'access-contents
                      mode-class  'mode-class-r)
                    ,(make-op-attr
                      name        'arg3
                      type        'type
                      access      'access-contents
                      mode-class  'mode-class-w))
   description                          ; tree describing the computation
   '(sequential
     (→ type                                        ; arg3 := arg2 - arg1
        (- type (operand arg2) (operand arg1))
        (operand arg3))
     (parallel
      (→ type-boolean                               ; cc-n := arg3 < 0
         (lss type (operand arg3) (constant type 0))
         cc-n)
      (→ type-boolean                               ; cc-z := arg3 = 0
         (eql type (operand arg3) (constant type 0))
         cc-z)
      (→ type-boolean                               ; cc-v := overflow(arg3)
         (overflow type (operand arg3))
         cc-v)
      (→ type-boolean                               ; cc-c := carry(arg3)
         (carry type (operand arg3))
         cc-c)))))
```

Figure 4.3 Description of sub[bwl]3.

operations in the description tree are qualified by the type parameter to the instruction family. The idiom discoverer attaches no meaning to the subtraction operator (−). The condition code settings are described by an unordered forest of assignments to processor state locations. These assignments will be noted during data flow analysis. As with the subtraction operator, the source expressions for these assignments have no meaning to the idiom discoverer. Thus these expression trees can be described in more or less detail, as needed. The role of the description writer is to describe identical operations identically, so that those operations can be matched by the idiom discoverer.

4.9. Experience with Machine Descriptions

Only two machines have been described for analysis by the idiom discoverer. The machine descriptions are large, in part because they are descriptions of every instruction and addressing mode of the target machine to some level of detail. The VAX-11 description is 6800 lines long. That describes 323 instructions, type-factored into 238 instruction descriptions, and all 22 addressing modes. Macros could have been used to reduce the size of the description by factoring common elements (*e.g.*, condition code settings), but as the VAX-11 was the first machine described I preferred clarity over size. The MC68000 description was written in just over 2 days, by someone who had never studied the manual [Motorola 82] closely. The MC68000 description is 5600 lines long. This description makes use of macros for condition code settings, which reduces the number of lines somewhat. The MC68000 description includes 275 instructions, type-factored into 192 instruction descriptions. More instruction descriptions were needed than one would assume from the formal descriptions in the user's manual, since many special cases are noted only in the English language descriptions. Rather than the 12 addressing modes described in the MC68000 manual, there are 28 addressing modes described, again due to special cases noted only in the prose descriptions.

CHAPTER 5

An Implementation of Idiom Discovery

Idioms are discovered by backward decomposition. The algorithm described in this section decomposes an instruction sequence called the *subject* into a sequence of instructions called the *pattern* such that the pattern *covers*, that is, performs the same computation as, the subject. The terminology is borrowed from pattern matching [Hoffmann and O'Donnell 82], which idiom discovery resembles. The covering is confirmed by comparing the trees depicting the computation of instructions from the target machine description. Rather than being satisfied by a single covering, the algorithm finds all reasonable instruction sequences on the target machine that cover the subject sequence. It is unlikely that two instruction descriptions will match without some restrictions, since target architectures rarely include two instructions to perform the same computation, and even if such instructions exist, the correspondence of operands would have to be noted. The goal of instruction decomposition is not so much to discover equivalences between instruction sequences as it is to discover constraints that make instruction sequences equivalent. After the *idiom discoverer* identifies a cover of one sequence by another, the *idiom applier* may replace an instance of the pattern by a suitably constrained instantiation of the subject and be assured that the computations performed are equivalent. This separation of idiom discovery from idiom application is a key idea of this dissertation.

The prototype idiom discoverer identifies three kinds of idioms during decomposition: binding idioms, set idioms, and composite idioms. Since a combination of these idiomatic properties is usually required to find a cover, it should not be surprising that they are identified by this one strategy. Binding idiom discovery is the identification of the correspondence between two (or more) operands of the pattern and an operand of the subject. Set idiom discovery is the identification of the correspondence between an operand of the pattern and a set of values from the subject. Composite idiom discovery is the identification of the correspondence between the partial order of computations in the pattern and the partial order of computations in the subject. If the correspondences are made, and the resulting constraints admit a solution, then the pattern covers the subject and an idiom has been identified.

First I describe the data structures used to perform idiom discovery, followed by a description of the covering routines, and finally a description of the tree matching routines with all the special cases of matching. Before the algorithm shown here can be called, summary information for all instructions and addressing modes is gathered and unique instructions and test instructions are identified.

5.1. Data Structures for Decomposition

The major data structure used to characterize a cover is a pair of instruction sequences (the subject and the pattern), and a set of constraints under which the pattern covers the subject. The constraints are represented as bindings on the parameters to the instructions: the operands and the data flow context[1]. The bindings may be specific, *e.g.*, that the base register of a particular operand must be the stack pointer; or more general, *e.g.*, that the offset of an operand must be in the set {1,...,8}; or the binding may be only a correspondence, *e.g.*, that an operand of a pattern instruction have the same addressing mode, offset, base register, and index register as an operand of the subject. It is convenient to separate the constraints into constraints on the subject and constraints on the pattern and attribute them to the subject and to the pattern rather than to the subject and pattern pair. This separation permits routines to be written that manipulate instruction sequences regardless of whether or where they are paired in a subject-pattern cover. During the decomposition process, both of the instruction sequences are modeled by the trees representing their computations from the target machine instruction descriptions. Thus *attributed instruction sequences* (or just *attributed instructions*) may be referred to almost interchangeably with *attributed trees*. Also during decomposition, subjects and patterns are paired before it is known whether they represent (even with constraints) equivalent computations. These tentative pairs are called *matches*. These pairs are the input to the matching function, which returns only pairs where the pattern (at least partially) covers the subject. Only those matches returned from the matching functions actually represent partial matches. Only those matches extended to complete matches represent equivalent computations.

5.1.1. Operand Attributes

Instructions are obviously parameterized by their operands. Operands may have fixed attributes that they inherit from the instruction: their type, access, and address mode class. Additional attributes of an operand can limit its addressing mode and constrain or bind the offset, base register, and index register of the operand. These last

[1] As an artifact of my implementation, types are also parameters of instructions, see Section 5.6.5.

three operand attributes may be specific or they may represent bindings to attributes of other operands. The addressing mode of an operand can be bound to any subset of the addressing modes in its address mode class. Binding to attributes of other operands need not bind to the corresponding attribute: *e.g.*, the base register of one operand may be bound to the index register of another operand. It is also convenient to have an operand attribute to indicate that the addressing mode, offset, base register, and index register attributes must correspond to those attributes of another operand.

The attributes for each operand are grouped together in a table of all the operands of an instruction. As instructions are assembled into instruction sequences, additional operand attributes are added to the table. As entries are added to the table, unique names are constructed and uniformly substituted for each reference to the operand in the instruction description tree. This mechanism allows the same formal names to be used in several instruction description trees, and allows the same instruction to be represented more than once in an instruction sequence[2].

5.1.2. Data Flow Information Attributes

The effect of an instruction may be different in different data flow contexts. These differences can be exploited when searching for decompositions. For example, if some locations set by an instruction are dead, those assignments need not be performed and the remaining computations may be coverable, where the original (entire) instruction was not coverable. Thus instruction sequences are parameterized by live and dead variable information. To make these parameters explicit, attributed instruction sequences include representations of data flow context information. Since data flow information is only tracked for the relatively small number of processor state variables (as defined in the target machine description), live and dead variables can be conveniently stored as small sets.

In order to propagate the data flow context and gather summary data flow information, three additional sets of variables are maintained as attributes of instruction sequences. One set records variables used before being defined in the computation of an instruction or sequence. A second set records variables defined before being used. The third set records variables killed by the execution of the instruction.

The inherited data flow context attributes (live and dead variable sets) are independent of the actions of instructions and may be constrained to obtain a match. The

[2] This might be confusing during debugging of machine descriptions, but it is less confusing than not renaming operands. I could have saved the original names in the table, but I didn't. The formal/index names are irrelevant to anything except indexing the operand attribute table.

other data flow attributes are functions of the computation tree in the inherited context, and can not be constrained by the process of decomposing instructions.

5.2. An Overview of Decomposition

When matching one attributed instruction against another, five results are possible, as shown in Section 3.7.1.1. Recall that instruction description trees can represent several computations partially ordered by data flow dependencies: say, an arithmetic operation and the setting of condition codes; or an increment, a test, and a conditional jump. Thus instruction description trees are actually partially ordered forests of computation trees. The most likely outcome is that the instructions will not match each other. Another possibility is that the instructions match completely, possibly by requiring some constraints on either or both of the instructions. (That is, the instructions as originally presented need not match: a restriction of the attributes of the instructions or a rewriting of the description trees may be required to have the instructions match completely.) For instructions consisting a single computation tree, those two choices are the only possibilities. A third (and by symmetry, a fourth) possible outcome exists when matching forests of trees. One instruction may match some but not all of the trees in the forest of the other instruction. The fifth case has some of the trees of one forest matching some of the trees of the other forest, with the trees of neither forest completely matched. These five cases are examined in more detail below.

One tree may fail to match another because the syntax of the trees mismatches. A syntactic mismatch would occur, for example, from trying to match a tree rooted with one operator against a tree rooted with some other operator, barring axioms to transform the syntax of one tree to correspond to the other tree. During syntactic matching, certain operators in the tree trigger examination of attributes from the attributed trees. Attribute comparison may narrow the constraints on either tree to achieve constraints that can be satisfied by both trees. If the constraints do not admit any solution (*e.g.*, the domain of some attribute becomes empty), the trees fail to match. The matching criterion is applied recursively to subtrees. An example of a semantic restriction mismatch would occur if two trees were matched representing operands, one restricted to some addressing mode and the other restricted to exclude that addressing mode.

Complete matching is possible when comparing single trees or forests of trees. Complete matches occur when one tree, or each of the trees in a forest of trees, syntactically matches another tree, or the corresponding trees in another forest, and the semantic binding of attributes can be satisfied. Complete matching is the absence of mismatching. Since the trees of a forest are partially ordered by data flow dependencies, the trees of a forest must be compared in an order permissible under that ordering. A

complete match implies that the partial orders on two forests are equivalent.

Partial matching can occur only when the subject is a forest of trees. In a partial match, all of the pattern covers part of the subject, but some of the subject remains uncovered. As for complete matches, trees from the pattern are compared against the subject in an order permissible by the partial ordering. Since I am only concerned with partial matches at one end of the partial ordering, the remaining unmatched trees of the subject are a contiguous "tail" of the partial ordering: itself a partial ordering of trees. This is the basis from which a partial match may be extended to a complete match. Equivalence classes in the partial ordering may not be decomposed, since to do so would impose an ordering on the members of the equivalence class.

The symmetric case to partial matching, in which the subject is completely covered by the pattern, but some of the pattern remains unused in that cover, is considered a mismatch. Subjects are not extended to be covered by the pattern. Only patterns are extended to cover subjects. The extension of subjects to discover an equivalence with single instruction patterns is redundant, since every single instruction will be examined as a subject for decomposition.

The fifth case for matching, in which some (but not all) trees of the subject forest are matched by some (but not all) trees of a pattern forest, is considered a mismatch. Because there is a partial ordering imposed on the forests, each least tree of the ordering on the subject must be matched by a least tree from the ordering on the pattern, and so on through the partial ordering of trees. If at any point a least tree from the ordering on the pattern does not match a least tree from the ordering on the subject, that pattern tree can not be matched against any greater trees in the subject ordering since that would violate the partial ordering. Because of the completeness with which the patterns are chosen and extended, extensions of the subject to cover unmatched pattern trees need not be considered by the idiom discoverer.

5.3. Data Flow Analysis

Several steps in idiom discovery depend on the data flow context following an instruction or tree. The data flow gathering routines are passed a tree attributed with information summarizing the data flow context following the tree. These routines traverse the tree qualifying the data flow information inherited from the following context. Data flow for subtrees is gathered as needed by calling the data flow routines recursively. Inherited context information is recorded in two sets: one listing variables that are *live* (that is, whose values are used) in the following context and one listing variables that are *dead* (that is, whose values are not used) in that context. As trees and instructions are examined and added to the following context (*e.g.*, in an instruction sequence, or the

decomposition of a forest of trees) the inherited context information is qualified by two sets: one listing variables that are used before being defined in the trees added to the following context, and one listing variables defined before being used in those additional trees. Inherited data flow information is not changed by data flow gathering. Qualifications are recorded only for variables in the inherited context. Data flow information is augmented in a backward pass over the trees of a forest. The backward direction of this pass is a function of the information required and is not related to the backward direction of the decomposition algorithm.

Keeping data flow context as inherited information and qualifications to the inherited information complicates certain data flow predicates, but allows other predicates to be written. For example, a variable is live if the value of the variable is used before being defined or if the value of the variable is live and not defined before being used. If only live and dead variable sets are kept, and modified by the data flow gathering routines, the live variable predicate is a simple membership test, but it is not possible to distinguish variables that are live because of inherited constraints and variables that are live because of a use in the current instruction sequence.

The qualifications on the inherited data flow context are discovered in a single pass over the nodes of a tree. As a member of the live or dead variable sets is seen as the destination of an assignment operator, that variable is removed from the set of variables that are used before being defined (if the variable was in that set) and added to the set of variables that are defined before being used. All other references to variables are uses, so these references remove the variable from the set of variables that are defined before being used, and add the variable to the set of variables that are used before being defined. Assignments and sequentially ordered forests of trees are the only special cases during data flow analysis. Since data flow gathering examines every tree node, this analysis is linear in time to the size of the tree.

In addition to the two qualification sets gathered, the data flow routines also record all variables from the inherited context that are *killed* (that is, defined) by a computation. The set of killed variables accumulates variables as they are seen as the destinations of assignments. Data flow information is gathered once for each instruction description tree, and the qualification sets and the killed set are recorded as summary data flow information for the instruction.

Summary information should be gathered for addressing modes, too, but the prototype idiom discoverer does not gather this information. The assumption here is that everything either used or modified by an addressing mode will be a parameter to that addressing mode. A counterexample would be an addressing mode that refers to and affects (without parameters), say, the stack pointer.

5.4. Instruction Classification

Before the idiom discovery algorithm is run, the instructions of the target machine are classified. The idiom discoverer uses the identification of instructions that compute unique functions and instructions that only set processor state locations (those locations liable to be declared dead during assignment matching). In addition, the idiom applier uses a list of the instructions that terminate basic blocks, so instructions that transfer program control are also distinguished. The classification of instruction is recorded by a set of flags attributed to each instruction description.

Unique instruction identification is accomplished by building an index from operators to those instructions whose description trees contain those operators. Any operator found in the description of only one instruction distinguishes that instruction as computing a unique function. Unique instruction classification visits each node of each instruction description tree once, and so takes time proportional to the number of nodes in the target machine instruction description trees. Constructing the index requires a table lookup for each operator, which can be performed in time proportional to $\log_2 o$ for o distinct operators in the best case. Therefore unique instruction identification takes time proportional to $n \times s \times \log_2 o$ for n instructions whose average description trees contain s nodes representing o distinct operators.

Test instruction classification identifies all instructions whose only effects are the setting of processor state locations, *e.g.*, condition codes (hence the name "test" instructions). The tree describing each target machine instruction is pruned (see Section 5.6.3) in a data flow context in which all processor state locations are dead. Pruning removes assignments to dead locations. If those effects are the only effects of the instruction, the empty tree is returned by the pruning routines and the instruction is flagged as a "test" instruction. This distinction is used during decomposition to check that assignment matching has not declared dead all the effects of an instruction. This identification could also be used in the idiom applier to check for dead test instructions: that is, test instructions in contexts where all the effects of the instruction are dead. Since the Graham-Glanville-Henry code generators do not emit dead test instructions, checking for dead tests in the idiom applier has not been important. Pruning takes time proportional to the number of operators in the tree being pruned, so test instruction identification takes time proportional to the number of operators in the target machine instruction description trees.

The idiom applier operates on one basic block at a time. Therefore it must be able to distinguish the ends (or beginnings) of basic blocks. For this purpose, the idiom discoverer identifies instructions of the target machine that transfer control, thus ending any basic block in which the instruction appears. Jump instruction identification

examines each instruction description tree for the distinguished operator **jump**, and so runs in time proportional to the size of the target machine instruction description trees. Jump instruction identification has no effect on idiom discovery.

5.5. Decomposition of a Subject Instruction

This section presents the top level of the algorithm to discover idioms for an instruction. The algorithm given here finds decompositions for each of the instructions on the target machine. That is, the prototype idiom discoverer considers only single instructions as subjects.

Matches will be shown in two columns, with the subject on the left and the pattern on the right, related by the operator "$\overset{?}{\simeq}$" to indicate a match that has not been verified. As a running example, the VAX-11's "**incl**" (increment by one) instruction will be the subject decomposed into the instruction sequence "**addl3**; **tstl**" (3-operand addition, followed by a test against zero). The tree describing the **incl** instruction is shown in Figure 5.1. The tree for the **addl3** instruction is shown in Figure 5.2, and the tree for the **tstl** instruction is in Figure 5.3. This match will be shown as:

incl	inc-dest	$\overset{?}{\simeq}$	**addl3**	add-src1,add-src2,add-dest
			tstl	tst-src

Where restrictions on operands are not immediately apparent from the match, they will be given in the text. For example, the above match will require the restrictions "add-src1 $\equiv$ 1" and "inc-dest $\equiv$ add-src2 $\equiv$ add-dest $\equiv$ tst-src"

The outermost loop of the decomposition algorithm is given a list of instructions and data flow context in which to find decompositions for each of those instructions. Instructions are decomposed by consulting the machine descriptions. Each instruction in turn is passed with the data flow context to an *instruction covering* routine that returns a list of covers. These covers are then recorded for the use of the idiom applier. For example, when the "**incl**" instruction is covered, the results include the "**addl3**; **tstl**" cover. Supplying a list of instructions, rather than iterating through all the instructions on the target machine makes it easy to run subsets of the full machine. Analyzing subsets is extremely useful for debugging machine descriptions (or the idiom discoverer), where an interesting case may come up only when decomposing a particular instruction. Since the decompositions of one instruction do not affect the decompositions of another instruction, the idiom discovery for a target machine may be run as several smaller jobs, *e.g.*, when the machine cycles are available, rather than requiring that an entire machine description be analyzed all at one time.

```
(sequential
  (→ type-long
     (+ type-long
        (constant type-long 1)
        (operand incl-dest))
     (operand incl-dest))
  (parallel
   (→ type-boolean
      (lss type-long
           (operand incl-dest)
           (constant type-long 0))
      cc-n)
   (→ type-boolean
      (eql type-long
           (operand incl-dest)
           (constant type-long 0))
      cc-z)
   (→ type-boolean
      (overflow type-long
                (operand incl-dest))
      cc-v)
   (→ type-boolean
      (carry type-long
             (operand incl-dest))
      cc-c)))
```

Figure 5.1. Tree Describing **incl**.

```
(sequential
 (→ type-long
    (+ type-long
       (operand add-src1)
       (operand add-src2))
    (operand add-dest))
 (parallel
  (→ type-boolean
     (lss type-long
          (operand add-dest)
          (constant type-long 0))
     cc-n)
  (→ type-boolean
     (eql type-long
          (operand add-dest)
          (constant type-long 0))
     cc-z)
  (→ type-boolean
     (overflow type-long
               (operand add-dest))
     cc-v)
  (→ type-boolean
     (carry type-long
            (operand add-dest))
     cc-c)))
```

Figure 5.2. Tree Describing **addl3**.

```
(parallel
  (→ type-boolean
     (lss type-long
        (operand tst-src)
        (constant type-long 0))
     cc-n)
  (→ type-boolean
     (eql type-long
        (operand tst-src)
        (constant type-long 0))
     cc-z)
  (→ type-boolean
     (constant type-boolean 0)
     cc-v)
  (→ type-boolean
     (constant type-boolean 0)
     cc-c))
```

Figure 5.3. Tree Describing **tstl**.

If the instruction is a *unique* instruction, no decomposition will succeed and so the empty list of covers is returned. This check avoids attempting matches of the instruction against each of the other instructions on the target machine, some of which may partially match and will need to be extended. For a machine with n instructions, of which u perform unique operations, this saves at least $n \times u$ attempted matches at a cost of identifying the unique instructions. Identifying unique instructions takes time proportional to the size of the descriptions of the n instructions.

If the instruction is not unique, the tree describing its computation is made into an attributed tree with operand attributes inherited from the instruction description and the data flow context information that was passed in to the instruction covering routine. The attributed tree for the subject is pruned (see Section 5.6.3) to remove parts of the tree that are dead in the inherited data flow context. If no computations remain after pruning, the instruction performs no useful computations in this data flow context and so the search for decompositions of this instruction is abandoned and the instruction covering routine returns the empty list of covers. The instruction could be deleted by the idiom applier if found in this data flow context. However, this information is not recorded, since our code generator only generates instructions whose results are used. That is, the idiom applier does not remove dead code.

If some computations remain after pruning, they must be decomposed into a pattern instruction sequence. The attributed tree is packaged as the subject of a match with the empty instruction sequence as the current partial pattern. The pattern sequence has no operand attributes, but inherits the same data flow context information as the subject. This match is handed to a *match covering* routine. In the example of covering an "incl" instruction, the subject of the match will be the entire tree describing the incl instruction (see Figure 5.1), since, as is usual, the inherited context has all the processor state variables live.

The match covering routine performs the actual decomposition of the subject. The match covering routine is always given a partial match that it extends by prepending single instructions. These extensions to the partial match are verified by calling the *tree matching* routine.

The tree matching routine is given an extended partial match and returns a list of altered copies of that match. The tree matching routine deletes the matching portions of the subject and pattern trees, leaving mismatching trees in place for alternative strategies to handle. Tree matching thus computes the "tree difference" between the subject and pattern. Tree matching also notes and records restrictions on attributes necessary to achieve a match.

Tree matching indicates that an instruction tree extends a partial match by returning a match with an empty pattern tree. The remaining subject is passed to a recursive invocation of the match covering routine. When a recursive call of the match covering routine is called with a partial match whose subject tree is empty, a complete match has been identified and the recursion unwinds.

In the example, the top level match covering routine is given the partial match of incl against the empty pattern sequence. This partial match is extended by each instruction on the target machine, including the tstl instruction. This extension (among others) is successful, so a recursive call to the match covering routine is made with the remaining uncovered subject, which is just the assignment of the addition from the incl description. The recursive call extends the match with each instruction of the target machine, including the addl3 instruction. This extension (among others) is also successful, so a third call is made to the match covering routine. This last call is passed a partial match whose subject tree is empty, so it simply returns. As the recursion unwinds, the instructions chosen as the extension at each level is recorded in the pattern of the match.

The only instructions not chosen as possible extensions are those instructions marked as unique instructions by instruction classification (see Section 5.4). Another special case is that the top level match covering routine does not extend the empty pattern with the

subject instruction. This check avoids discovering the identity transformation as an idiom for each instruction.

An instruction may extend a partial match under several different sets of constraints (*e.g.*, due to axioms, bindings of attributes, *etc.*). The tree matching routine therefore returns to the match covering routine a list of partial matches, each of whose attributes records a different set of constraints. The match covering routine iterates over the members of this list making recursive calls for extensions. For example, when extending the running example with an **addl3** instruction, if the commutative axiom for addition has been described, two sets of operand bindings are discovered (see Section 5.6.2). Thus two complete matches are discovered from a single partial match. Note that the common tail of these two idioms, the **tstl** instruction, is matched only once.

5.6. Tree Matching

The covering routine described above chooses decompositions and records successful covers for the idiom applier. The tree matching routines described here discover binding, set, and data flow constraints and record them in the attributes of matches.

The bulk of tree matching is syntactic. However, all the built-in semantics of the instruction description trees must be enforced during tree matching. Especially important during tree matching are the triggers of attribute comparisons and adherence to the partial ordering of trees within forests.

5.6.1. Forest Matching

Two built-in tree operators (**sequential** and **parallel**) represent the partial ordering on trees in a forest. Their semantics influence the order in which their subtrees are compared.

Matching one sequence, *i.e.*, a forest rooted with a **sequential** operator, against another may (if successful) return attributed trees with shortened sequences. Matching a sequence against a tree may shorten the sequence by one subtree. In both cases, resulting sequences of unit length are altered to delete the **sequential** operator, leaving just the component tree. Sequential matching takes time proportional to the length of the shorter sequence being matched.

Sequential matches implement the backward policy for matching forests. Sequences are matched from their "tails" toward their "heads": from lesser members of the partial ordering towards the greater members.

In the example, the first tree match is between the tree describing the **incl** instruction and the tree describing the **tstl** instruction:

```
(sequential
 (→ type-long
    (+ type-long
       (constant type-long 1)
       (operand inc-dest))
    (operand inc-dest))
 (parallel
  (→ type-boolean
     (lss type-long
        (operand inc-dest)
        (constant type-long 0))
     cc-n)
  ...))
```

```
(parallel
 (→ type-boolean
    (lss type-long
       (operand tst-src)
       (constant type-long 0))
    cc-n)
 ...)
```

Since the pattern tree can be constrained to match the last subtree of the subject tree, the subject tree is reduced to the assignment of the addition, and the pattern tree is completely deleted.

Parallel operators root trees whose subtrees form an equivalence class in the partial ordering. As such the subtrees may be matched against the subtrees of a corresponding **parallel** tree in any order. Each subtree of an equivalence class must be completely matched; there is no chance to return a "smaller" equivalence class, since to do so would violate the equivalency of the class. All possible permutations of the subtrees of a **parallel** node are tried in the effort to match all the subtrees of a corresponding **parallel** tree. Therefore, parallel matching may be forced to consider $n!$ possible permutations for equivalence classes of size n. Permutations are generated only as needed. If any subtree fails to match in every position of the permutation, the **parallel** match is abandoned, so often not all permutations are generated and tested. If the machine description writer uses some canonical ordering among the subtrees of **parallel** operators, the matching of parallel trees can be verified in time proportional to the number of subtrees.

5.6.2. Operand Matching

Both binding and set idioms are discovered when matching operands. There are three opportunities to match operands: one case in which both subject and pattern trees are operand references, the case in which the subject tree refers to an operand and the pattern is not an operand reference, and the symmetric case in which the pattern is an operand reference and the subject is not an operand reference. These cases are examined in detail below.

5.6.2.1. Binding Constraint Discovery

The matching of a subject operand reference against a pattern operand reference is the key step in binding idiom discovery. The two operands match if they can be constrained to have the same addressing mode, offset, base register, and index register. There is an attribute of each operand to record a list of other operands to which the operand is bound in this manner. The addressing modes of the two operands are restricted to the intersection of their separate addressing mode sets. If this intersection is empty, the operands can not be bound, and so the operand match is abandoned. In addition to sharing at least one addressing mode, the operand must perform the same access (*i.e.*, effective address or contents) in order to be bindable. Further, the types of the operands must be the same (or equivalent). If the operands are bindable, the name of the subject operand is added to the "bound-to" list of the pattern operand, and the name of the pattern operand is added to the "bound-to" list of the subject operand. The operand reference trees are deleted to indicate that they match, and the match returns successfully. The transitive closure of the "bound-to" lists are used by the idiom applier to verify the correspondence between operands of the pattern and to construct the subject operands.

Operand binding comes up several times when decomposing "**incl**" with "**addl3; tstl**". For example, the match of:

```
(→ type-long                    ≈   (→ type-long
   (lss type-long                       (lss type-long
      (operand inc-dest)                   (operand tst-src)
      (constant type-long 0))              (constant type-long 0))
   cc-n)                                cc-n)
```

eventually matches:

```
(operand inc-dest)      ≈      (operand tst-src)
```

that derives the correspondence between 'inc-dest' and 'tst-src'. Other binding constraints in the example are "inc-dest ≡ add-src2" and "inc-dest ≡ add-dest".

5.6.2.2. Set Constraint Discovery

The match of a subject tree against a pattern operand reference is the key step in set idiom discovery. Since addressing mode descriptions are factored from instruction descriptions, the operand reference stands for the trees of all the addressing modes of the operand. Each of these trees in turn must be compared against the subject tree to see which addressing modes, if any, the subject tree covers. The addressing mode descriptions are parameterized by the access and type of the operand. These parameters are fixed by

attributes of the operand. Addressing mode descriptions are also parameterized by an offset, a base register, and an index register, that are not fixed attributes of the operand. The addressing mode trees for each addressing mode of the pattern operand are instantiated with references to the free attributes of the operand. Each of these trees is paired against the subject tree in a match in which the pattern operand has been restricted to the addressing mode represented by the instantiated tree. These matches are passed to the tree matcher, and any complete matches are returned as successful operand matches.

As references to parameters of the pattern operand appear during the recursive calls to the tree matcher, they will attempt to bind the attributes of the pattern operand. Those bindings, if successful, will be reflected in the matches returned to the operand matcher (and upward) as constraints on the match.

An example of set constraints occurs when matching the addition in the incl description with the addition in the addl3 description. The match is:

```
(→ type-long                    ≊   (→ type-long
   (+ type-long                         (+ type-long
      (operand inc-dest)                   (operand add-src2)
      (constant type-long 1))              (operand add-src1))
   (operand inc-dest))                  (operand add-dest))
```

that eventually matches:

```
(constant type-long 1)          ≊   (operand add-src1)
```

The addressing modes for 'add-src1' include "immediate-constant" mode, which references the value of the offset of the addressing mode. The addressing modes of 'add-src1' will be restricted to immediate-constant mode, and the trees matched are:

```
(constant type-long             ≊   (constant type-long
      1)                                   (offset add-src1))
```

that eventually leads to the match:

```
1                               ≊   (offset add-src1)
```

that binds the offset of 'add-src1' to the value 1.

5.6.2.3. Binding Subject Operands

The third case is symmetric to the previous case: matching a subject operand reference to a pattern tree. The same strategy could be employed: iterating through the addressing modes trees of the subject and matching them against the pattern. This is not ordinarily done in the idiom discoverer. Expansion of a subject operand can be attempted in an effort to fold the operation of pattern instructions into the addressing mode

arithmetic of subject operands. For example, consider trying to constrain an **addl3** instruction so that instruction covers a move-effective-address instruction, **moval**, using an "offset from base register" addressing mode. The match between:

moval mova-src,mova-dst $\overset{\cdot}{\approx}$ **addl3** add-src1,add-src2,add-dest

matches the trees:

```
(→ type-address                 ≈.   (→ type-long
   (operand mova-src)                   (+ type-long
   (operand mova-dest)                     (operand add-src2)
                                           (operand add-src1))
                                        (operand add-dest))
```

Since one of the addressing modes of 'mova-src' includes "offset from base register" address mode, the substitution of addressing mode trees for 'mova-src' will match:

```
(+ type-address                 ≈.   (+ type-long
   (offset mova-src)                    (operand add-src2)
   (basereg mova-src))                  (operand add-src1))
```

that will match:

```
(offset mova-src)               ≈.   (operand add-src2)
```

and

```
(basereg mova-src)              ≈.   (operand add-src1)
```

These last two matches will succeed if 'add-src2' can be bound to an addressing mode representing the offset of 'mova-src' (as immediate-constant mode from the previous example) and 'add-src1' can be bound to an addressing mode representing the base register of 'mova-src' (as register-direct mode does). The cover will represent the folding of the addition into the addressing mode of the 'mova-src' operand. The Graham-Glanville-Henry code generator leaves almost no opportunities to apply such idioms, so it has not been important to detect such idioms. In the prototype idiom discoverer the search for special cases of subject operands is optional.

5.6.3. Pruning

Pruning is used to discard irrelevant (*i.e.*, dead) portions of a tree before matching. The pruning routine is passed a tree attributed with data flow context information, and it returns that attributed tree with the tree altered to remove computations whose values are not used in that context. For example, when the **addl3** instruction is tried as an extension during the decomposition of **incl**, the values of the condition codes are all dead, and so the tree returned from the pruning routine is just the assignment of the addition. The pruning routine needs exact, rather than summary, data flow information, *e.g.*, about variables used in subtrees. In practice, the pruning routine calls the data flow analysis

routine (see Section 5.3) as needed, and returns a pruned tree with the correct data flow attributes.

The pruning routine is called for each subject and for each pattern extension considered during matching. In addition, this routine is called once for each instruction on the target machine during "test" instruction classification. Pruning examines each node of a tree once, and so runs in time proportional to the number of nodes in the tree.

5.6.4. Assignment Matching

Data flow attributes of a match can be manipulated to achieve covers. The key idea is that changes to data flow context can specify conditions under which mismatching trees are irrelevant and need not prevent matching. Data flow attribute manipulation is like pruning during tree matching. Data flow attributes are changed by the tree matching routine that matches assignments.

In the easy case, the assignment matching routine discovers that all the subtrees of the assignment (type, source, and destination) match, and just returns the resulting list of matches without altering the data flow attributes. Ordinarily, if any of the subtrees mismatches, the assignment match is abandoned. The prototype idiom discoverer manipulates data flow attributes in the special case when the destination and types of the assignments match but the source expression trees do not match. If the destination were dead, the sources of the assignment (and the assignment itself) would be immaterial to the computation being performed. By contrast, if the destination had been marked as dead before matching, the assignment would have been pruned and the assignment match would never be considered, much less fail. Clearly the destination is not dead if tree matching gets as far as matching the assignment. If the data flow context can mark the destination dead, the assignment matcher can ignore the mismatch of the sources.

For example, in the matching of **incl** against **tstl**, two of the condition code assignments mismatch in their source expression trees. One of these is the match:

```
(→ type-boolean                    ≅   (→ type-boolean
   (carry type-long                        (constant type-boolean 0)
          (operand inc-dest))              cc-c)
   cc-c)
```

Here the mismatch of the sources causes the destination 'cc-c' to be declared dead by the tree matcher. Similarly, 'cc-v' is declared dead. Thus the idiom "**incl**" ⇔ "**addl3; tstl**" is conditional on 'cc-v' and 'cc-c' being dead. This condition must be checked by the idiom applier before applying the idiom.

Not every destination can be declared dead. In particular, a destination can not be declared dead if the destination is used later in the instruction sequence. This is one reason for maintaining the set of variables used before being defined. The data flow analysis performed by my prototype is severely limited, which limits the destinations that can be declared dead. The list of variables that might be declared dead is conveniently the set of live variables maintained in the data flow attributes. The assignment matcher can declare dead any destination noted as live and not used before being defined later in the sequences. The declaration is recorded by the assignment matcher by removing the variable from the live variable list and adding the variable to the dead variable list. The assignment matcher returns the altered match indicating success.

One drawback to this technique is that the destinations and types must match for the assignment matcher to handle the mismatch of the sources. For example, individual assignments that do not match can not be selectively removed from enclosing trees (say, from sequences) to allow the enclosing trees to match. Dynamic programming could be used here to solve the "string editing" [Wagner and Fischer 74] problem for the enclosing trees, but that approach is not used in the prototype idiom discoverer. Instead, the description writer inserts identity assignments where these assignments are needed. Consider two instructions: **op_cc**, which performs some operation and sets condition codes; and **op_nocc**, which performs the same operation but does not set the condition codes. Clearly if the condition codes are dead, one instruction is an idiom for the other. The description of **op_nocc** would have to contain the identity assignment of the condition codes for the idiom to be discovered. This has not been a problem in practice, because common condition code settings are described only once, as macros, and then expanded wherever they are needed.

5.6.5. Type Unification

The target machine descriptions may be type-factored into instruction families, representing, for example, byte-, word-, and long-addition. This is a convenient facility for the description writer. Type factoring is also convenient for the idiom discoverer, since the time complexity of the target machine analysis is polynomial in the number of descriptions examined. Therefore it is useful to maintain the type-factoring during analysis, rather than macro-expanding the type-factored instruction families supplied by the description writer. Type factoring was removed from the running example for clarity. In practice, the matching described above discovers three idioms (for byte-, word-, and long operations), since the VAX-11's instruction set is type-orthogonal for the instructions examined.

Just as instruction descriptions portray a type-factored family of target machine instructions, attributed matches portray type-factored families of matches. Type-factoring does not persist into the idiom applier. To improve the speed and simplicity of the idiom applier, factored types are distributed through covers as the tables for the idiom applier are produced.

Maintaining type-factoring in the idiom discoverer means that type attributes are not simple type names, but rather sets of type names representing the domain of the type parameter. References to type parameters may appear in the trees, as may explicit types, yielding four cases of matching types. Matching one specific type against another is almost syntactic: the types match if they are the same types, or if one type is on the list of types equivalent to the other type in the descriptions of the types. Matching a specific type against a reference to a type parameter restricts the domain of the type parameter to the types equivalent to the specific type. If that domain becomes empty, the match fails. The interesting case is the matching of one type parameter against another. The type attributes must be bound to indicate that they must have the same value and their domains must be restricted to the intersection of their previous domains. Both these constraints are stored in a type attribute by storing the domain of the cross-product of the subject and pattern type domains. The cross-product domain is stored in the type attribute of both trees, for symmetry. To restrict the pattern domain to a particular type, the type match routine restricts the cross-product to those pairs whose pattern element is equal to (or equivalent to) that particular type. Similarly restricting a subject type restricts the cross-product to those pairs whose subject element is equivalent to the particular type. The binding of a subject type to a pattern type is accomplished by restricting the cross-product domain to pairs in which the subject element is equal to (or equivalent to) the pattern element.

Since the type domains are needed for each of the members of an instruction sequence, the cross product is actually the cross product of type domain sequences. Since type domains tend to be small (since target machines are only type orthogonal for small sets of types), the cross product is not unmanageably large. Type-factoring was added to the prototype late, and this was a convenient representation for the constraints on types. If I had to implement type-factoring again, I would keep the bindings and domains of types as separate attributes.

5.8.6. Axioms of Tree Operators

The target machine description consists of trees in a single static form. The distinguishing operators of these trees have meaning only to the description writer. The idiom discoverer imposes no semantics on these operators. Often, however, these

operators have algebraic properties (*e.g.*, commutativity) that could be exploited to achieve covers. Properties of these operators are part of the target machine description, and as such they are the responsibility of the description writer. Clearly the idiom discoverer must provide a mechanism for specifying properties of operators.

The prototype idiom discoverer allows the description writer to designate a very limited form of tree transformation. The description writer can indicate, for any operator, an operator that is equivalent when the subtrees of the original operator are reversed. For example, the reverse operator for '+' (representing addition) is '+', the reverse operator for '<' (less than) is '>' (greater than), *etc.* This mechanism is sufficient to indicate commutativity, a useful tree axiom for achieving matches. When a pattern tree rooted with a reversible operator is matched, the idiom discoverer also constructs the reversed tree and attempts to match that tree.

This method of specifying axioms is unsatisfactory. The main failing of this method is that the idiom discoverer provides a very limited tree transformation system that cannot be extended by the description writer. That is, this method is both a mechanism (tree transformation) and a policy (only reversing operators). What is needed is a separation of the mechanism from the policy. A better mechanism for letting the description writer specify axioms is for the idiom discoverer to call tree transformation routines written by the description writer. A tree transformation routine could be associated with any operator. A tree transformation routine would be passed the pattern tree each time the designated operator was being matched and would return a list of trees, each of which would be matched in place of the original tree. Axiomatic transformation of description trees thus becomes part of the target machine description and part of the responsibility of the description writer. For example, the description writer could include machine-specific axioms, such as changing additions of negative quantities to subtractions, *etc.* The idiom discoverer could include a library of simple tree transformations that the description writer could use or augment. A change to this style of axiomatic transformation is planned for the near future of the idiom discoverer.

Allowing the description writer to specify axioms is a clean solution to the tree transformation problem, particularly from the point of view of the idiom discoverer. This solution is not without its drawbacks, however. If the tree transformations are expensive to compute, or generate many candidate trees to be matched, the time required for analyzing a target machine may be adversely affected. I assume such direct feedback will be a sufficient constraint on overzealous description writers. For example, the idiom discoverer is probably not the place to search for identity transformations on operators (additions of zero, *etc.*). Instead, identity operations should be removed by machine-independent transformations before code is generated.

5.7. Addressing Mode Side-Effect Idiom Discovery

In addition to the idiom transformations discussed above to decompose one instruction sequence into another, the idiom discoverer also discovers idioms that transform an instruction sequence into an addressing mode side-effect. First, addressing modes are classified to determine which, if any, have side-effects. Next, addressing modes are compared to find equivalences among their references. Finally, the decomposition algorithm is used to find instruction sequences that cover addressing mode side-effects.

Separating addressing modes into those addressing modes with side-effects and those modes without side-effects is easy, since the addressing mode descriptions explicitly include descriptions of any side-effects. Addressing mode side-effects are distinguished in the descriptions as either pre-effects, *i.e.*, taking place before reference to the operand, or post-effects, *i.e.*, taking place after reference to the operand. Pre-effects can be used to replace instruction sequences preceding an operand; post-effects can be used to replace instruction sequences following an operand. Two pieces of information from the machine description are needed to replace an instruction sequence with an addressing mode side-effect. The idiom applier needs to know, first, which addressing modes without side-effects can be replaced by the addressing mode with side-effects and continue to reference the same object; and second, what instruction sequences cover the side-effect.

The idiom discoverer determines, for each addressing mode with side-effects, which addressing modes without side-effects reference the same object, given the same operand parameters, *e.g.*, type, offset, base register, and index register. Since each addressing mode may access either the effective address or contents of its operand, two lists are derived for each addressing mode with side-effects: one list for each possible access to the operand. The comparison of addressing modes takes time proportional to $s \times p$, where s and p are the sizes of the description trees for addressing modes with and without side-effects. The lists of equivalences are recorded as attributes of each addressing mode.

Instruction sequences that cover the side-effect of an addressing mode are determined by the same covering algorithm used for discovering idioms of instructions. The tree describing the side-effect of an addressing mode is made the subject in a match against an initially empty pattern instruction sequence. This match is passed to the tree covering routine, which returns a list of covers. The time required to find covering sequences is, again, polynomial in the number of instructions on the target machine, with the degree of the polynomial varying with the length of the covering sequences, *i.e.*, the complexity of the side-effect. The addressing mode and the instruction sequences covering the side-effect of that addressing mode are recorded in a table for use by the idiom applier.

5.8. Artifacts of My Implementation

5.8.1. Trees versus Instruction Sequences

In the discussion of decomposition above, it may appear that both the instruction sequence and parts of the trees representing those instructions exist in the cover (or match) simultaneously. In fact, in my implementation only the tree or the instruction sequence is needed at any one time, and so instruction sequences and trees are variants of a data structure rather than members of a more general structure. If I had to do this work again, I would include them both in the structure and selectively ignore one or the other when I didn't need it.

5.8.2. Allowing Side-Effects on Addressing Modes

Addressing modes with side-effects are disallowed during decomposition and added to operand attributes after decomposition is complete. Side-effects are disallowed because the operand matching routines cannot constrain the rest of the decomposition algorithm to account for the side-effects of operands. In addition, the data flow information required is not gathered by the prototype idiom discoverer. Part of the problem is that my machine descriptions do not include operand evaluation order. Different architectures may evaluate addressing modes and their side-effects in different orders, *e.g.*, in parallel, or sequentially either left-to-right or right-to-left. The solution outlined below is conservative, and works with either parallel or sequential left-to-right evaluation orders. In practice, our Graham-Glanville-Henry code generator rarely uses addressing mode side-effects. However, the idiom discoverer does identify opportunities to fold instructions into addressing modes, so the idiom applier might construct addressing modes with side-effects.

At first glance, it might appear that side-effects of operand addressing modes in a subject could extend a pattern instruction sequence by covering larger patterns. Instead, I chose to have the idiom discoverer find covered sequences for side-effects once for each side-effect, rather than once each time the side-effect might appear as an addressing mode in a subject. As a consequence, the idiom applier folds instructions into side-effects as a separate pass before applying other transformations.

The output of the covering routine is a list of covers. In these covers, each operand is attributed with a list of addressing modes. These lists are originally constructed from the lists of addressing modes in the addressing mode class of the operand. Addressing mode attributes are restricted, before matching, to only those modes without side-effects. Addressing mode attributes may be further restricted during matching, *e.g.*, to the intersection with some other addressing mode attribute during binding, or to a particular addressing mode during set idiom discovery. After decomposition, side-effect addressing

modes are added to addressing mode attributes in a very conservative fashion, discussed below.

A first observation is that a side-effect addressing mode can be added to the addressing mode attribute of an operand only if the address mode class of the operand includes that addressing mode. The second observation is that if a side-effect is added to the operand of a pattern, that side-effect must also be added to an operand of the subject. A third observation is that a side-effect may change the effective address to which the operand refers. For this last reason, pre-effects are added only to the first operand of an instruction sequence, and post-effects are added only to the last operand of an instruction sequence. Combining these observations leads to the conclusion that the first operand of a pattern can have a pre-effect only if that operand is bound to the first operand of the subject, and the last operand of a pattern can have a post-effect only if that operand is bound to the last operand of the subject. For example, it is acceptable to transform:

add3 src1,src2,dest ; *dest := src2 + src1*

into:

add3 –(src1),src2,dest ; *dest := src2 + (src1 := src1 − 1)*

(with the '–(src1)' indicating a pre-effect on 'src1'), since any correspondence between the operand referred to by 'src2' and 'dest' (and between 'src2' and 'src1', for that matter) will be maintained. In contrast, the transformation to:

add3 src1,–(src2),dest ; *dest := (src2 := src2 − 1) + src1*

is disallowed since the side-effect on 'src2' may alter a component common to the 'src1' and 'dest' operands in the original instruction. Consider the cover:

add2 src,dst ⇔ **add3** dst,src,dst

(with bindings indicated by identical operand names) and a potential transformation to allow side-effects on operands into:

add2 –(src),dst ⇔ **add3** dst,–(src),dst

This transformation is disallowed since the side-effect of '–(src)' may change a component of 'dst' such that with sequential evaluation of operands the two 'dst's of the **add3** yield different operands.

If the only bindings on a subject and pattern operand are the binding necessary for allowing a side-effect, the side-effect can be allowed "in place". That is, the addressing mode attributes of the operands are augmented to include the addressing mode with the side-effect. If in addition to the binding necessary for allowing the side-effect, either the subject or pattern operand is bound to another operand, the side-effect must be added to the operands in a copy of the original cover to separate the addressing modes of the

operands from the binding. The original cover is left untouched, representing the cover in the absence of any side-effect addressing modes. In the copy the addressing modes of the first (or last) operands are restricted to the addressing mode with the side-effect, and other operands bound to those two operands are restricted to addressing modes that reference the same object but without any side-effects. All the bound operands continue to share other operand attributes (*e.g.*, offset, base register, index register), but not addressing modes. The correspondence of addressing modes is thus made explicit in the operand attributes. Thus, to extend the previous example, using addressing modes from the VAX-11 (specifically, '(dst)', which refers to the object pointed to by 'dst', and '(dst)+', which references the same object, but then increments 'dst'), the idiom discoverer may copy the general cover:

add2 src,dst ⇔ **add3** dst,src,dst

with multiple addressing modes possible for 'dst', and restrict it to:

add2 src,(dst)+ ⇔ **add3** (dst),src,(dst)+

with '(dst)' and '(dst)+' representing specific addressing modes for those operands.

5.9. Experience with Idiom Discovery

The prototype idiom discoverer runs acceptably fast for use in a phase constructor. The VAX-11 description takes somewhat over 2 cpu hours (on a VAX-11/750) for decomposition. The prototype idiom discoverer is written in LISP, in a style chosen for clarity and debugging, rather than speed, so it is likely that a different implementation could be faster. 1273 idioms are discovered from the VAX-11 description. The number of partial matches found during idiom discovery is also of interest. When decomposing the instructions from the VAX-11, 111 partial matches of length 0 were extended to 53 complete matches and 112 partial matches of length 1. These partial matches were extended to 290 complete matches and 27 partial matches of length 2. The partial matches of length 2 were extended to 138 complete matches of length 3. Note that there are many fewer partial matches of length 0 than instruction descriptions (238). Idiom discovery does not consider decomposing unique instructions, so those instructions do not appear in the partial matches of length 0. Note also that many fewer complete matches (481) are formed than idioms are discovered. The increase is due to the distribution of types into complete matches as the matches are recorded as idioms.

Similar results are found for the MC68000. Decomposition of the MC68000 description requires somewhat under 4 cpu hours (also on a VAX-11/750) to discover 503 idioms. 177 partial matches of length 0 are extended to 50 complete matches and 306 partial matches of length 1. These partial matches are extended to 376 complete matches and 2 partial

matches of length 2. The partial matches of length 2 are extended to 6 complete matches of length 3.

CHAPTER 6

An Implementation of Idiom Application

The previous chapter describes a system for automating the discovery of machine-specific improvements for compiled code. This chapter presents one proposal for applying those improvements: a separate phase of a retargetable compiler that transforms assembler source code. What is discussed below is a simple pattern matching and replacement system. The idiom applier identifies instances of idiom patterns in the input assembler source code. This identification is partly syntactic, for example, comparing instruction names, and partly semantic, for example, checking restrictions on operand attributes. For each identified pattern the idiom applier instantiates the idiom subject using attribute values from the pattern. The attributed subject then replaces the pattern, and the idiom applier continues searching for applicable patterns. The purpose of the prototype idiom applier described here is to verify that the information produced by the idiom discoverer is sufficient to retarget an idiom applier. Little or no attention was given to building a fast idiom applier.

The main advantages of using a separate compiler phase to perform code improvement are that a separate phase is easier to write, test, and evaluate. A disadvantage of a separate phase is that there is no feedback from the idiom applier to the code generator. In practice, the idiom discoverer is constrained to identify only idioms for which feedback is unimportant. If the idiom discoverer identifies tree substitution idioms, for example, the idiom applier would have to be able to return resources to the resource manager, *e.g.*, the compiler temporary variable holding the value of the substituted expression. In fact the idiom applier could benefit from integration with the code generator. For example, idioms may be predicated on data flow context. Therefore the idiom applier must determine the data flow context during pattern matching. If an earlier compiler phase develops data flow information, that information could be used by the idiom applier. As a separate phase, the prototype idiom applier must compute data flow information from summary data flow equations for instructions as that information is needed.

An alternative use of the results of idiom discovery would be an integrated machine-specific code improver that worked in parallel with code generation and resource allocation. For example, the idioms could be used to generate predicates and functions for

an attribute influenced parsing code generator. In this way, idioms (*e.g.*, improved code sequences) would be selected directly by the code generator, as opposed to the current scheme in which generated code is transformed after code selection. An advantage of such integration is that idioms affecting resource allocation (*e.g.*, tree substitutions idioms, which free registers, and available expression idioms, which reuse registers) could be identified and resources requested from or returned to the resource manager. The idiom discoverer would have to be extended to discover idioms exploiting tree substitutions and available expressions. Such a proposal is not unreasonable, but is left for future research.

6.1. Information Required for Idiom Application

The idiom applier does no analysis of the target machine. All machine-specific information needed for idiom application is supplied by the idiom discoverer. In particular, no description of the semantics of instructions or addressing modes is needed by the idiom applier. Any useful information about the semantics of instructions or addressing modes has been incorporated in the discovered subject and pattern pairs that are the major input to the idiom applier. (Recall, also, that the idiom discoverer determines equivalence of instruction sequences with no knowledge about the semantics of most operations performed by instructions.) The idiom applier also uses tables of summary information about instructions and addressing modes.

Space and time costs are not incorporated into the idioms, but are supplied by auxiliary tables. Costs can not be associated with idioms since idioms may specify correspondence of attribute values, rather than particular attribute values. Thus an operand of an idiom pattern may be restricted to a set of addressing modes, but the particular addressing mode found in the program during idiom application will determine the cost of the operand, and thus the benefit that can be derived from applying the idiom. Consider, for example, the idiom on the VAX-11:

moval offset(reg),reg ⇔ **addl2** offset,reg

which, on the left, uses a "move address" instruction and "offset from base register" addressing mode arithmetic to add an offset to a register, and, on the right, uses an explicit 2-operand addition instruction. If the offset is a small integer (in the range [0,...,63]), the **addl2** instruction saves 1 byte over the **moval** instruction. If the offset is not a small integer, the **moval** instruction saves up to 3 bytes over the **addl2** instruction. Therefore, idioms are discovered without regard for costs. The idiom applier calculates costs from tables of instruction and addressing mode costs as idiom instances are found.

The idiom applier must determine the data flow context for instructions in the input source program, since idioms are predicated on particular data flow contexts. The idiom

discoverer derives summary data flow information during its analysis of the target machine. Summary information for each instruction gives three sets of processor state variables, describing how the values of those variables are affected by the execution of the instruction. The sets specify which values are used before being defined by the instruction, which values are defined before being used by the instruction, and which values are killed by the instruction. This summary information is used by the idiom applier to construct data flow contexts without any additional description of the semantics of instructions.

Entire programs need not be examined all at once. The idiom applier is concerned only with a few instructions and their data flow context when identifying any given idiom. Computing correct data flow information would require examining the entire program. If simplifying assumptions can be made at the boundaries between basic blocks, the idiom applier need only examine the program one basic block at a time. These assumptions about data flow between basic blocks can be conservative, *i.e.*, all processor state variables are live at block boundaries, or optimistic, *i.e.*, all processor state variables are dead at block boundaries. A third alternative is to have the compiler writer specify some set of processor state variables that are live across basic block boundaries.

The idiom applier needs a machine-independent technique to identify basic block boundaries. Boundaries can be identified given a list of instructions that transfer program control, thus ending the basic block in which they appear. Such a list is derived by the idiom discoverer from the instruction descriptions (see Section 5.4).

The idiom discoverer identifies addressing modes that have side-effects. For each such addressing mode, the idiom discoverer constructs a list of attributed instructions that perform the same computation as the side-effect, and a list of addressing modes that reference the same operand without the side-effect. Folding arithmetic into addressing mode side-effects involves changing the address modes of instruction operands. While the idiom applier can assume that it is only given legal addressing modes for instruction operands in the input program, to change operand addressing modes, it must be given a table of valid addressing mode classes for each instruction operand, and a table of addressing modes in each class.

6.2. An Implementation of an Idiom Applier

The prototype idiom applier is retargeted by loading tables of idioms, costs, summary data flow information, and instruction and addressing modes classifications. (Unfortunately, in the prototype, the compiler writer also has to supply routines to read and print the assembler source code for the target machine. The construction of these routines is automated by the use of standard scanner and parser generators, but in

another implementation of the idiom applier these routines should be derived from format information in the instruction and addressing mode descriptions.)

Instructions are read from a program until the end of basic block is found. The idiom applier exhausts all possibilities for pattern matching and replacement idioms in a single pass over the program, once data flow information is available. The pattern matching pass can be either forward or backward. Since data flow information is computed by a backward pass over each basic block, it is possible to combine these two operations into a single backward pass. In contrast, applying addressing mode side-effect idioms may require examination of the instruction stream multiple times, and so is implemented as a separate phase of the idiom applier. Finally, the transformed basic block is written out. The transformation of basic blocks continues until the entire program has been examined.

6.2.1. Pattern Matching and Replacement

Pattern matching and replacement of idioms consists of identifying instances of idiom patterns, verifying operand and data flow constraints, and replacing the pattern with an appropriately constrained subject from the idiom. The object of idiom application is to minimize the cost of the code for a basic block.

Matching and replacement of idioms is exhaustive in a single pass over the instructions of a basic block. The proof of this claim relies on the thoroughness of idiom discovery and the application of the most beneficial idiom at each point. There are three sources of counterexamples to this claim: that the replacement subject is a pattern of another idiom all by itself, that the replacement subject is part of another idiom pattern when combined with instructions preceding the subject, or that the replacement subject is part of another idiom pattern when combined with instructions following the subject. The first case, in which the subject is itself another pattern, is ruled out by choosing the most beneficial idiom to apply at each transformation. Clearly the subject is an idiom for the pattern it replaces. If the subject is also a pattern for a second idiom, then the subject of that second idiom is a replacement for the original pattern. The idiom consisting of the original pattern and the second subject will have been considered by the idiom discoverer. That idiom cannot be more beneficial than the chosen idiom, since the most beneficial idiom is chosen at each transformation. A similar argument holds for a transformation placing a subject where it can be extended, in either direction, into a pattern of a second idiom. If the subject can be extended into a longer pattern, that extension will have been considered by the idiom discoverer. The longer idiom cannot be more beneficial than the chosen idiom, or it would have been chosen by the idiom applier. Therefore a single pass over the instructions exhausts all opportunities to apply idioms.

The idioms applied by a forward pass may differ from the idioms applied by a backward pass.

Idiom application depends on having data flow context information available for the evaluation of data flow predicates. Data flow context consists of live and dead variable sets. These sets are computed in a single backward pass from data flow summary information for instructions and the assumptions about data flow between basic blocks. The data flow context following an instruction is not needed until idiom application examines that instruction. Therefore it seems natural to combine data flow computation with pattern matching and replacement.

6.2.2. Identifying Patterns and Forming Replacements

The idiom applier begins transforming an instruction sequence by computing the data flow context of that sequence. The initial instructions of the sequence are examined to select possible idioms based only on the syntax of the instructions, *i.e.*, the names of the instructions. Any of these idioms may have data flow predicates that make the idiom inapplicable in the current data flow context. All variables that must be dead to apply the idiom must be faint (*i.e.*, their values are unused) in the program. Not all variables that are preserved by the idiom need be live in the program. Next, the attributes of operands to the instructions in the program are compared against the restrictions on those attributes in the idioms. For example, if the base register of an operand is restricted to a particular register, that register must appear as the base register of that operand in the program. Idioms may also note required correspondence between operand attributes. The correspondence between the actual operands of the instruction sequence are checked. Idioms whose patterns satisfy all these conditions are applicable to the instruction sequence. Note that more than one idiom may be applicable at any point in a program, and that the patterns of the applicable idioms need not match the same number of instructions.

Operand attributes of an idiom subject may be specified as correspondences to attributes of the idiom pattern. Attributes of the actual program operands are copied into the corresponding attributes of each subject operand. This last step annotates the subject as a replacement sequence for the instructions from the program. Finally, the costs of the attributed subjects and patterns are computed.

The idiom providing the most benefit is selected. The annotated subject of this idiom replaces the instructions that match the pattern. The current implementation computes both space and time costs, and prefers saving time over space. A better way to handle cost information is to have the compiler writer supply routines to compute and compare costs. These routines could incorporate biases or tradeoffs with which the

compiler writer wishes to experiment.

The idiom applier computes the data flow context for the instructions preceding the replacement from the following context of the replacement and the summary information for the instructions of the replacement. If no pattern is applicable at a given instruction, the data flow context preceding that instruction is computed from the following context and the summary information for the instruction.

6.2.3. Addressing Mode Side-Effect Idioms

Folding arithmetic into pre- and post-effects of addressing modes is performed by a two additional passes over each basic block. The transformations made by these passes are different from the pattern match and replacement strategy described for instruction idioms. Instruction idioms replace contiguous instructions; address mode side-effect idioms remove instructions and relocate their computations into addressing modes. The pre- and post-effect passes may examine the instructions of a basic block more than once. In the worst case, each instruction that might be subsumed causes the examination of every other instruction in the basic block.

Side-effect idiom application identifies instances of instructions that can be subsumed into addressing mode side-effects and instances of addressing modes to which the side-effects can be added. Identification of instructions that may be subsumed is comparable to the pattern matching phase of instruction idiom application. Instruction sequences from the program are matched against the table of instruction sequences that can be subsumed into addressing mode side-effects. This matching involves checking restrictions on attributes, including data flow context attributes. Attribute values of the idiom are qualified by the specific values found in the program instructions. A search through the program is begun for an operand to whose addressing mode the equivalent side-effect can be added. Relocating a computation to an addressing mode assumes left-to-right operand evaluation, in the current implementation. If the idiom subsumes instructions into a pre-effect, the search is forward through the operands of following instructions. If the idiom subsumes instructions into a post-effect, the search is backwards through the operands of preceding instructions. The search ends successfully if an operand is found that can be changed to perform the side-effect. The search is abandoned if an operand is encountered that uses any parameter of the instruction being replaced (*e.g.*, a base or index register used by operands of the subsumed instruction). The search must also be abandoned if an instruction is encountered that uses such a parameter implicitly, *e.g.*, as a stack manipulation instruction uses a stack pointer register. This last condition is verified by examining the summary data flow information for the instruction. The search is unsuccessful if the operand found to perform the side-effect can not be changed due to

restrictions on the addressing mode class of that operand.

If the search identifies an operand that can be changed to perform the side-effect, the idiom applier makes that change and deletes the original instruction.

6.3. Interactions of Idiom Application Phases

Since idioms for instructions are discovered without considering addressing mode side-effect idioms, applying side-effect idioms to a program may create additional opportunities for applying instruction idioms. (The converse is not true, since the search for instruction sequences covered by side-effects discovers all idioms for those instruction sequences.) Therefore, it seems reasonable to apply side-effect idioms before instruction idioms. An unfortunate artifact of my prototype idiom discoverer is that side-effects on operands of instruction idioms patterns are disallowed, except for special cases, due to the inability to prohibit certain operand correspondences. Therefore is also seems reasonable to apply instruction idioms before side-effect idioms. The traditional solution to this "phase ordering" problem is to perform alternate phases until no more idioms are applicable [Wulf et al. 75]. It would appear that a better description of the semantics of operand evaluation, and a stronger language for predicates, *e.g.*, prohibited correspondences between operands, would allow these two phases to be integrated. Note however that instruction idiom application is a linear pass over the instructions of a basic block, examining at most the number of instructions in the longest pattern at each instruction. The number of instructions examined by each attempt to apply a side-effect idiom is based on the particular attribute values found in the program being transformed. For example, if idioms were applied during code generation, a basic block buffering mechanism would be necessary to apply side-effect idioms. Such a buffering mechanism is visible in Ganapathi's code generators [Ganapathi 80]. More study is needed of the effects and alternative implementations of idiom application.

6.4. Experience with Idiom Application

In practice, my prototype idiom applier runs just fast enough to verify the ideas presented in this dissertation. The idiom applier is written in LISP, in a style chosen for ease of debugging rather than speed of execution. The idiom applier spends much more than half of its time reading instructions, constructing the internal representations for them, and writing instructions back out. Instruction idiom application is 7 pages of code. Each of pre- and post-effect idiom application is 5 pages of code, and either pass could be parameterized to perform both passes. The LISP implementation processes 5 instructions per second on a VAX-11/750. Davidson and Fraser quote speeds of over 200 instructions per second for a similar table-driven idiom applier. An alternative to table-driven idiom

application represents idioms by compiled routines to check applicability and perform the transformations [Lamb 81]. Another alternative is the integration of idiom application into the code generation phase. The integrated code generator and idiom applier has the advantage of allowing feedback from idiom applier to the resource manager and code selector.

CHAPTER 7

Summary and Conclusions

The thesis of this dissertation is that target machine descriptions can be analyzed at compiler construction time to discover transformations yielding improved code from a retargetable compiler. Target machine analysis identifies equivalences between instruction sequences. Once two instruction sequences are shown to compute the same results, instances of the more expensive instruction sequence may be replaced by the cheaper instruction sequence. The analysis of the target machine is called *idiom discovery*. The replacement of inefficient code sequences is called *idiom application*.

The idiom discoverer is run once for each target machine at compiler construction time, leaving only program-dependent predicates to be verified during compilation of any particular program. Therefore, more time may be devoted to target machine analysis than would be acceptable during compilation.

Two sets of predicates must be satisfied to identify instruction sequence equivalences: machine-dependent predicates based on the effects of the instructions, and program-dependent predicates based on the operands and data flow contexts found in particular programs. Machine-dependent predicates are tested by examining a description of the target machine instructions and addressing modes. In the proposed idiom discoverer, most of the required testing is syntactic matching. This matching is augmented by a very small set of semantic tests that identify program-dependencies from certain syntactic mismatches. Program-dependent predicates include tests for particular values of operand attributes, or tests of correspondence between two (or more) operands. In addition, program-dependent predicates may test data flow context conditions, *e.g.*, that the values of particular variables are dead.

The traditional view of program dependencies is that they are conditions that must hold in a particular program to assure an equivalence between instruction sequences. The view taken in this dissertation is that program dependencies may be manipulated by the idiom discoverer to achieve an equivalence. That is, the idiom discoverer does not so much discover equivalences between instruction sequences as attempt to constrain instruction sequences to perform equivalent computations.

This dissertation describes a new technique for selecting instruction sequences to compare during idiom discovery. Previous work in automated improvement of generated code [Davidson and Fraser 80][Kessler 84] relies on *composition* of instructions to form sequences. Instructions are composed, either by substituting definitions for uses or by concatenation of effects, and the target architecture is searched for a better implementation of the combined effects of the sequence. Composition takes time polynomial in the number of instructions composed, in both the best and worst cases. Composition is limited, by practical considerations, to composing pairs of instructions, that is, discovering instructions that are equivalent to pairs of instructions. In contrast, this dissertation proposes *decomposition* to discover idioms. The individual effects of complex instructions are matched against simpler instructions until the complex instruction is decomposed into a sequence of simple instructions. Decomposition also takes time polynomial in the length of the decomposed instruction sequence in the worst case, but takes much less time in the average case. Decomposition is not limited to discovering equivalence to pairs of instructions; it may discover that an instruction is equivalent to a longer sequence of instructions.

A prototype idiom discoverer has been implemented that acts as a phase constructor for a retargetable idiom applier. The prototype idiom discoverer examines a target machine description for instances of a small interesting set of idioms and produces tables that retarget a transformer of assembler source code. The idiom discoverer has been used to replace analyses formerly done by the compiler writer and encoded in hand-written routines in an otherwise automatically retargeted code generator [Henry 84]. The idiom applier performs the same transformations as the hand-written routines. Further experimentation should shift more of the target machine analysis from the compiler writer to the idiom discoverer. Such a shift should clarify the uses of machine-dependent information in retargetable compilers.

Critique of the System

The prototype system is not without its problems. The most serious deficiency is that the idiom discoverer finds instances of idioms for which it was designed to search, and not others. New classes of idioms can be identified only by adding code to the idiom discoverer. (Once added, however, those idioms will be identified in each machine description analyzed.) On different classes of architectures, *e.g.*, stack machines, the prototype idiom discoverer will not discover many idioms, since it was designed for multiple-operand architectures.

The machine descriptions used for idiom discovery have proven inadequate in several respects. There is no specification of the order of evaluation of operands, and thus no

ordering on side-effects of those evaluations. Without this specification, the allowing of side-effects on operands to idioms is awkward, and the folding of instructions into addressing mode arithmetic or side-effects is extremely limited. Many interesting cases simply can not be handled by the idiom applier without the ordering information.

The program-dependent predicates manipulated by the idiom discoverer and evaluated by the idiom applier are all positive; that is, they are sets to which operand attributes must belong, or correspondences that must hold between operands. In many cases it would be useful to have predicates specifying negative set membership and negative correspondence between operands. I do not foresee any problems with adding negative predicates, but others trying to build on this research should be reminded of the utility of such specifications.

At several points during this research it appeared that adding temporaries to instruction descriptions might simplify certain aspects of machine description. Temporaries could be treated as additional operands by the idiom discoverer algorithms (that is, they would be subject to constraints and bindings). The idiom applier would then have to verify that the program operands corresponding to instruction temporaries of an idiom subject were, in fact, single use compiler temporaries, since the corresponding instruction temporaries would not appear in the transformed code. It it still unclear to me that adding this mechanism to the idiom discoverer is worth the additional complexity it would require in the idiom applier.

A relatively minor inconvenience with the machine descriptions is that they do not specify the syntax for the instructions and operands. The reading and writing of assembler source in the idiom applier is handled by an auxiliary description of the syntax for the assembler source.

The prototype idiom applier runs entirely too slowly for production use. Davidson and Fraser cite speeds of 200 instructions per second through their similar transformation system [Davidson and Fraser 84], and there is no reason to suspect my idiom applier must be slower than theirs. Furthermore, the code produced by Davidson and Fraser's code generator is naive, and much of it needs to be transformed to be competitive with conventional code generators. In contrast, better code generation techniques make idiom application truly optional, thus restricting the time spent in idiom application to those programs that need it.

Future Research

It seems reasonable to use the idiom discoverer interactively, allowing the compiler writer to suggest subjects for decomposition. Such usage would allow experimentation in two orthogonal directions. The compiler writer could try decompositions of subject

Future Research

instruction sequences longer than those automatically considered by the idiom discoverer. Alternatively, the success or failure of a decomposition could be used by the compiler writer to debug an instruction description, or, by a machine architect, to change an instruction set design. Only minor additions to the present idiom discoverer are needed to allow interactive specification of subjects and to provide reasonable feedback.

The technique of transforming assembler source is limiting, since it is applied after code generation and register allocation. Several classes of idioms are deliberately avoided in the idiom discoverer since they would change the resource demands of programs. There is nothing inherently difficult about discovering these excluded idioms. A more integrated system could have the idiom applier running as a co-routine with code generation and register allocation to allow some feedback on resource usage. A completely different kind of integration would incorporate the results of idiom discovery directly into the code generator, say, as attribute predicates and functions. More research is needed into appropriate uses of the analysis performed by an idiom discoverer.

The allocation of machine-specific transformations to various phases of retargetable compilers is still an open problem. As techniques are developed to automate the analysis of machine descriptions, it will be possible to argue that certain transformation are most appropriately performed by particular compiler phases. This dissertation is one attempt to automate such target machine analysis.

Bibliography

[Aho and Johnson 76]
A. V. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees", *Journal of the ACM 23*, 3 (July 1976), 488-501.

[Allen and Cocke 76]
F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure", *Communications of the ACM 19*, 3 (March 1976), 137-147.

[Backus et al. 57]
J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes and R. Nutt, "The FORTRAN Coding System", *Proceedings of the Western Joint Computer Conference*, Los Angeles, CA, February 1957, 188-198.

[Bell and Newell 71]
C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971.

[Cattell 78]
R. G. Cattell, "Formalization and Automatic Derivation of Code Generators", PhD Dissertation, Technical Report 78-115, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1978.

[Cocke and Schwartz 70]
J. Cocke and J. T. Schwartz, *Programming Languages and Their Compilers*, Courant Institute of Mathemtical Sciences, New York University, April 1970.

[Crawford 82]
J. Crawford, "Engineering a Production Code Generator", *Proceedings of the SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices 17*, 6 (June 1982), 205-215.

[Davidson and Fraser 80]
J. W. Davidson and C. W. Fraser, "The Design and Application of a Retargetable Peephole Optimizer", *ACM Transactions on Programming Languages and Systems 2*, 2 (April 1980), 191-202.

[Davidson 81]
J. W. Davidson, "Simplifying Code Generation Through Peephole Optimization", PhD Dissertation, Technical Report #81-19, Department of Computer Science, University of Arizona, December 1981.

[Davidson and Fraser 84]
J. W. Davidson and C. W. Fraser, "Automatic Generation of Peephole Optimizations", *Proceedings of the SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices 19*, 6 (June 1984).

[Fraser 79]
C. W. Fraser, "A Compact Machine-Independent Peephole Optimizer", *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, San Antonio, TX, January 1979.

[Ganapathi 80]
M. Ganapathi, "Retargetable Code Generation and Optimization Using Attribute Grammars", PhD Dissertation, Technical Report #406, Computer Science Department, University of Wisconsin, Madison, WI, 1980.

[Giegerich 83]
R. Giegerich, "A Formal Framework for the Derivation of Machine-Specific Optimizers", *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1983), 478-498.

[Glanville 77]
R. S. Glanville, "A Machine Independent Algorithm for Code Generation and Its Use In Retargetable Compilers", PhD Dissertation, Technical Report 78-01, Electronics Research Laboratory, EECS, University of California, Berkeley, Berkeley, CA, December 1977.

[Glanville and Graham 78]
R. S. Glanville and S. L. Graham, "A New Method for Compiler Code Generation", *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, Tucson, AZ, January 1978, 509-514.

[Henry 84]
R. R. Henry, "Graham-Glanville Code Generators", PhD Dissertation, Technical Report 84/184, Computer Science Division, EECS, University of California, Berkeley, Berkeley, CA, May 1984.

[Hoffmann and O'Donnell 82]
C. M. Hoffmann and M. J. O'Donnell, "Pattern Matching in Trees", *Journal of the ACM 29*, 1 (January 1982), 68-95.

[Johnson 77]
S. C. Johnson, *A Tour Through the Portable C Compiler*, Bell Laboratories, Murray Hill, NJ, 1977.

[Kessler 81]
R. R. Kessler, "COG: An Architectural Description Driven Compiler Generator", PhD Dissertation, University of Utah, 1981.

[Kessler 84]
R. R. Kessler, "Peep – An Architectural Description Driven Peephole Optimizer", *Proceedings of the SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices 19*, 6 (June 1984), 106-110.

[Knuth 68]
D. E. Knuth, "Semantics of context-free languages", *Mathematics Systems Theory 2*, 2 (1968), 127-145.

[Lamb 81]
D. A. Lamb, "Construction of a Peephole Optimizer", *Software—Practice & Experience 11*, 6 (June 1981), 639-647.

[Leverett 79]
B. Leverett, "Machine Independent Register Allocation in Optimizing Compilers", PhD Dissertation, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1979.

[Leverett et al. 79]
B. Leverett, R. Cattell, S. Hobbs, J. Newcomer, A. Reiner, B. Schatz and W. Wulf, "An Overview of the Production Quality Compiler Compiler Project", Technical Report 79-105, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, February 1979.

[McKeeman 65]
W. M. McKeeman, "Peephole Optimization", *Communications of the ACM 8*, 7 (July 1965), 443-444.

[McKusick 84]
M. K. McKusick, "Register Allocation and Data Conversion in Machine Independent Code Generators", PhD Dissertation, Computer Science Division, EECS, University of California, Berkeley, Berkeley, CA, December 1984.

[Morgan and Rowe 82]
T. M. Morgan and L. A. Rowe, "Analyzing Exotic Instructions for a Retargetable Code Generator", *Proceedings of the SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices 17*, 6 (June 1982), 197-204.

[Motorola 82]
Motorola, *MC68000 16 Bit Microprocessor User's Manual, 3rd Edition*, Prentice Hall, Englewood Cliffs, NJ, 1982.

[Ripken 77]
K. Ripken, "Formale Beschreibung von Maschinen, Implementieurungen und optimierender Maschinencodeerzeugung aus attributierten Programmgraphen", PhD Dissertation, Technische Universität München, Munich, West Germany, July 1977.

[Szymanski 78]
T. B. Szymanski, "Assembling Code for Machines with Span Dependent Instructions", *Communications of the ACM 21*, 4 (1978), 300-308.

[Tanenbaum, van Staveren and Stevenson 82]
A. S. Tanenbaum, H. van Staveren and J. W. Stevenson, "Using Peephole Optimization on Intermediate Code", *ACM Transactions on Programming Languages and Systems 4*, 1 (January 1982), 21-36.

[Wagner and Fischer 74]
R. A. Wagner and M. J. Fischer, "The String-to-String Correction Problem", *Journal of the ACM 21*, 1 (January 1974), 168-173.

[Wulf et al. 75]
W. Wulf, R. Johnsson, C. Weinstock, S. Hobbs and C. Geschke, *The Design of an Optimizing Compiler*, American Elsevier Computer Science Library, 1975.

Every gun that is made, every warship launched, every rocket fired signifies – in the final sense – a theft from those who hunger and are not fed, those who are cold and not clothed.

D. D. Eisenhower,
Peace in the World,
Speech to the American Society of Newspaper Editors,
Washington, D.C., April 16, 1953.

Prepared by the author using the text processing tools *vi*, *spell*, *awk*, *sed*, *vlp*, *m4*, *bib*, *dtbl*, *deqn*, and *dtroff* with the *–me* macro package, coördinated by *csh* and *make*.