Instruction-Stream Compression

A Thesis presented

by

Christian James Carrillo

to

Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 2, 2001
Abstract

This thesis presents formal elements of instruction-stream compression. We introduce notions of instruction representations, compressors and the general “patternization” function for representations to sequences. We further introduce the Lua-ISC language, an implementation of these elements. Instruction-stream compression algorithms are expressed, independently of the target architecture, in Lua-ISC. The language itself handles instruction decoding and encoding, patternization and compression; programs within it are compact and readable.

We perform experiments in instruction representation using Lua-ISC. Our results indicate that the choice of representation and patternization method affect compressor performance, and suggest that current design methodologies may overlook opportunities in lower-level representations.

Finally, we discuss four instruction-stream compression algorithms and their expressions in Lua-ISC, two of which are our own. The first exploits inter-program redundancy due to static compilation; the second allows state-based compression techniques to function in a random-access environment by compressing instructions as sets of blocks.
Acknowledgments

First and foremost, I would like to thank my advisor, Professor Norman Ramsey. His generous gifts of time and talent have enriched this work well beyond the highest point of my abilities. In no small way, his insight, clarity and inspiration have been a guiding motivation through many sincere hours of effort. I only regret that I have not captured his brilliance within these pages.

I also wish to thank my readers, Professors Margo Seltzer, Mike Smith and Norman Ramsey. I am indebted to Glenn Holloway and Mike Smith for the use of their Alpha machines, and to my roommate, Jon Cahill, for his crash course on spreadsheets and statistics.
Acknowledgments
Contents

Abstract ............................................................ iii
Acknowledgments .................................................. v

1 Introduction ................................................... 1

2 Elements of Sequence Compression ....................... 3
  2.1 Tools of Description ........................................ 3
  2.2 Symbols and Sequences ...................................... 5
  2.3 Compressors ................................................ 8

3 Representations of Machine Instructions ............... 13
  3.1 Two Views of the Instruction .............................. 14
  3.2 Trees as Sequences ........................................ 16
  3.3 A Formal Description of Patternization ................ 17
  3.4 Reversibility of Patternization .......................... 22
  3.5 Skeleton Trees ............................................. 30
  3.6 Extensions ................................................ 32

4 A Language for Compression: Lua-ISC .................. 35
  4.1 An Introduction to Lua .................................... 35
  4.2 K-Symbols and K-Sequences .............................. 36
  4.3 Compressors ................................................. 37
  4.4 Patternization .............................................. 40

5 Experiments in Representation ............................ 43
  5.1 Test Corpus ................................................ 43
  5.2 Experimental Set-up ....................................... 44
  5.3 Patternization Methods ................................... 46
  5.4 Results .................................................... 52

6 Applications .................................................. 63
  6.1 Supporting Elements in Lua-ISC .......................... 63
  6.2 Wire Form Compression .................................... 66
  6.3 Random-Access Algorithms .............................. 68
7 Future Directions 73
   7.1 Application Environments .......................... 74

8 Conclusion 77

A Extensions to Patternization 79
   A.1 Evicting Entire Trees–Naming the Root ............ 79
   A.2 Step Patternization ................................. 79

B Additional Elements of Compression 83
   B.1 Suffix Trees ........................................... 83
   B.2 ELF Files .............................................. 84
   B.3 Profiling Data ......................................... 86
   B.4 Shuffling K-Sequences ............................... 87

C Supporting Code 89
   C.1 Example Code ......................................... 89
   C.2 Representation Experiments ........................ 91
   C.3 Algorithms ............................................ 92

Bibliography 99
Chapter 1

Introduction

Computer programs are often so large that transmitting, storing and running them can be costly. Compression is an immediate solution; general text compression algorithms are widely used today to compact programs for network transfer, or to archive them for storage. These algorithms are successful in significantly reducing program size, but they miss additional opportunities for compression that are specific to computer programs. Further, they are only appropriate for “wire” forms; programs must be decompressed all at once, and therefore cannot be run, partially compressed, in memory. As hand-held and embedded devices proliferate, it is increasingly important to address program size at every stage of a program’s life, from installation to storage to execution. Text compression methods leave room for improvement in each stage.

Instruction-stream compression, in general, attempts to realize this improvement by specializing algorithms to instruction streams, the machine-specific commands that make up a computer program. Algorithms are designed for specific applications; some try to compact programs further to produce smaller wire forms, while others run compressed programs in memory by decompressing instructions only as they are needed. Techniques span a wide range from dictionary compressors [Lefurgy, Bird, Chen, and Mudge 1997; Lucco 2000] to compact intermediary representations [Franz and Kistler 1997; Ernst, Evans, Fraser, Lucco, and Proebsting 1997] to custom instruction sets [Fraser and Proebsting 1995].

The tight coupling between algorithm and instruction facilitates better overall compression, but often creates machine-specific results. This thesis presents a formal algebra for algorithms in instruction-stream compression, and a programming language, Lua-ISC, that implements it. The symbol is the atomic unit of data; compressors are expressed as operations over sequences of symbols. Instructions are represented as trees. Two “natural” views of instruction trees are supported: constructor trees for which the root is an instruction and the children are its operands; and field trees for which the root is the instruction format and the children are fields. Instruction-stream compression is largely a problem of converting the structured data in an instruction into flat streams for a compressor. We capture this problem in a single operation, called patternization, and express it as a higher-order
function. Patternization serves as the backbone of some algorithms; our formalization allows these to be expressed succinctly and precisely.

Expressions in the Lua-ISC language are machine-independent; the language supplies all the infrastructure for building from bit patterns to instruction trees. This independence does not come at the expense of compression performance or specificity; through patternization and the instruction tree representations, algorithms within the language have access to all the information in the original instruction stream. Lua-ISC programs are applied to instruction streams directly, without requiring any external information. The language is primarily intended as a flexible, general platform for expressing and executing algorithms. It is not built with run-time performance in mind, and is not used to evaluate algorithm speed. Execution speed experiments are, by their nature, implementation-specific.

Using our infrastructure, we conduct experiments over different architectures, patternization methods and instruction representations to measure the effects of these variables on compression performance. Our work shows that strong candidates in one environment can perform very poorly in others, particularly when crossing from RISC to CISC processors. Our work further suggests that the starting point for many algorithms, assembly-language AST’s, is not as advantageous as “flat” field tree representations.

We additionally present two algorithms from the field-a wire form from Ernst et al. [Ernst, Evans, Fraser, Lucco, and Proebsting 1997], and split-stream dictionary compression [Lucco 2000]—in Lua-ISC; and discuss two new algorithms for instruction-stream compression. The first, inter-program compression, exploits sequences shared among different applications, often the result of static linking. The second, a random-access block compressor, uses profiling information to group instructions based on frequency of use. The block compressor simulates individual instruction decoding while allowing state-based compression techniques to exploit the instruction stream more completely than is possible at the single-instruction level.
Chapter 2

Elements of Sequence Compression

This chapter introduces the abstract data types symbol and sequence, over which compression functions are defined. Subsequent chapters represent instruction streams as sets of sequences. Compressors themselves are built from smaller, composable units called compressor components, also described below.

2.1 Tools of Description

Symbols, sequences and compressors are presented using the following four methods.

- **Informal descriptions** provide a conceptual foundation for each element and its functions.
- **Types** formally describe the set of allowable operations on a given element or set of elements.
- **Algebraic laws** formally present the result of an operation on elements, and provide a rudimentary machinery for reasoning about sequences of operations.
- **Examples** of operations over elements and their results reinforce the preceding formal and informal descriptions.

2.1.1 Types and Kinds

Types are used by programming languages to characterize legal operations over data. The int and float types are familiar from C. Functions have types defined by their return values and parameters; the addition function over floats in C, for example, has type float * float → float. More generally, types are defined as follows.
basic ::= \text{int} | \text{float} | \text{bool} \quad (2.1)

type ::= \text{basic} | \text{type} \times \text{type} | \text{type} \rightarrow \text{type} \quad (2.2)

A type can be a \text{basic type} (an \text{int}, \text{bool} or \text{float}); a tuple (a pair of types); or a function (sometimes called an “arrow type”). The recursive nature of the type definition can lead to more complicated forms. The function \text{complicated} below exercises the tuple and function types.

\text{complicated} : \text{int} \ast (\text{int} \rightarrow \text{bool}) \rightarrow \text{int} \rightarrow (\text{int} \rightarrow \text{bool}) \quad (2.3)

The asterisk has precedence over the arrow; \text{complicated} will take an integer and a function from integers to bools, and produce a function from integers to functions from integers to bools.

Noticeably missing from the definition of types are elements like lists, arrays and pointers. Many languages support the type \text{int list}, for example. But if \text{int} is a type and \text{int list} is a type, what is \text{list}?

\text{list} (or \text{array} or \text{pointer}) is a \text{type constructor}. Just as data have types, type constructors have \text{kinds} that describe their operation. Kinds are defined below.

\text{kind} ::= \Omega | \# | \text{kind} \rightarrow \text{kind} \quad (2.4)

A kind is either a type (\Omega, read “type”), a number (#) or a function from kinds to kinds. The kind of \text{list}, then, is “\Omega \rightarrow \Omega,” the list type constructor takes a type and produces a type.

We rely upon the type system to distinguish between symbols of different widths. The \text{symbol} type constructor has kind “\# \rightarrow \Omega,” introducing types like “\#3 \text{symbol},” a 3-bit symbol. Sequences are also differentiated based on the width of the symbols they contain; the \text{sequence} type constructor has kind “\# \rightarrow \Omega” as well. Symbols and sequences are defined below, in Section 2.2.

Functions may have \text{polymorphic types}. The length function over lists, for example, can take any type of list as its parameter and will produce an integer. The type of the length function for lists is shown in Equation 2.5; a similar function over sequences is shown in Equation 2.6. Polymorphic types match a particular type constructor, as opposed to a particular type.

\text{length} : \alpha \text{list} \rightarrow \text{int} \quad (2.5)

\text{length} : \#n \text{sequence} \rightarrow \text{int} \quad (2.6)
2.1.2 Classes of Functions

Introducing a new data type $T$ requires new functions for creating, observing and modifying values of that type. These functions can be categorized as follows.\footnote{This categorization is taken directly from [Kamin, Ramsey, and Cox 2000].}

- **Constructors** produce new values of type $T$. There are two classes of constructors. **Primitive constructors** take arguments that are not of type $T$ and produce a new value of type $T$. They can also take no arguments; these constructors are called **nullary constructors**. Non-primitive constructors take arguments that are of type $T$, and possibly additional arguments.

- **Observers** take an argument of type $T$ and produce a new value, which may or may not be of type $T$. An observer “looks inside” its argument and produces some constituent value or dependent fact.

By definition, constructors and observers have no side effects. Operations that do have side effects are called **mutators**. Although conceptually straightforward, mutation is very difficult to specify using simple algebraic laws.

In this chapter, we describe the operations over symbols and sequences, called **terms**, algebraically. Terms are constructors or observers; they have no side effects. In our language implementation, however, their **function** representations are sometimes implemented as mutators. The terms are intended to describe the semantics of operations over the data type, to which the function implementations adhere. The functions appear in Chapter 4.

2.2 Symbols and Sequences

Compression is often considered in the context of text applications. A text compressor, like gzip or pkzip, can be applied to any type of file; it has no **a priori** knowledge of its input’s internal structure, treating it simply as a stream of bytes. A byte is a natural unit of data for a text file, where the information is presented in a single stream and each character is represented with a byte. For binary files that also store their information in byte-aligned quantities, text compression is also effective; even though the basic data elements may cover two or more bytes, any reasonable text compressor will exploit the repetition of these sequences. But for files where data are represented by sub-byte or unaligned quantities, these methods are much less effective.

We therefore more generally classify compressors as functions over **symbols**, rather than over bytes. A symbol is a representation of a quantity, a string of bits with a specific width. A **sequence** is a list of symbols, each of the same width. The following sections define symbols and sequences more completely, and provide constructors and observers for them.
\[
\text{new} : \quad \text{number} \times \text{number} \rightarrow \#n \text{ symbol} \\
\text{k} : \quad \#n \text{ symbol} \rightarrow \text{number} \\
\text{value} : \quad \#n \text{ symbol} \rightarrow \text{number} \\
\text{minWidth} : \quad \#n \text{ symbol} \rightarrow \text{number} \\
\text{setK} : \quad \#n \text{ symbol} \times \text{number} \rightarrow \#m \text{ symbol}
\]

\[
k(\text{new}(x, y)) = x \\
\text{value}(\text{new}(x, y)) = y \\
\text{minWidth}(\text{new}(x, y)) = \begin{cases} 
\lceil \log_2(y + 1) \rceil & \text{if } y > 0 \\
1 & \text{if } y = 0 
\end{cases} \\
\text{setK}(\text{new}(x, y), k) = \text{new}(k, y)
\]

*Figure 2.1: Terms over symbols and their laws.*

### 2.2.1 Symbols

**Definition 1 (Symbol)** A symbol is a string of \( k \) bits. The width of a symbol is the number of bits it contains, \( k \); the value of a symbol is the numerical value of its bit string.

A symbol contains both a value and a width. Two symbols are equivalent if and only if their values and widths are equal. A symbol is intended to represent a bit string of a given non-zero length; in practice, a symbol could be a byte, a machine word, one-and-a-half machine words, truly any number of bits. The terms over symbols are show in Figure 2.1.

The algebra of symbol terms is straightforward; the width of a symbol is equivalent to the k-value supplied to the \( \text{new} \) constructor, and similarly the symbol’s value matches the supplied value. \( \text{minWidth} \) returns the fewest number of bits required to represent the given symbol. The laws in Figure 2.1 assume that \( \text{new}(x, y) \) is defined if and only if \( x > 0, y \geq 0, x \geq \lceil \log_2(y + 1) \rceil \).

### 2.2.2 Sequences

**Definition 2 (Sequence)** A sequence is a list of zero or more symbols, each with the same width.

The sequence type constructor, like that of symbols, has kind “\( # \rightarrow \Omega \).” Figure 2.2 presents some sequence constructor and observer functions. \( \text{new} \) and \( \text{add} \) are constructors for sequences; \( \text{new} \) takes as its parameter the width for the new sequence, and produces an empty sequence of the appropriate type. \( \text{add} \) will add a symbol to the end of the sequence; this is the opposite of the
Chapter 2: Elements of Sequence Compression

\[
\begin{align*}
\text{new} : & \quad \text{number} \rightarrow \#n \text{ sequence} \\
\text{add} : & \quad \#n \text{ sequence} \ast \#n \text{ symbol} \rightarrow \#n \text{ sequence} \\
\text{length} : & \quad \#n \text{ sequence} \rightarrow \text{number} \\
\text{k} : & \quad \#n \text{ sequence} \rightarrow \text{number} \\
\text{nth} : & \quad \#n \text{ sequence} \ast \text{number} \rightarrow \#n \text{ symbol} \\
\end{align*}
\]

\[
\begin{align*}
\text{length}(\text{new}(n)) &= 0 \\
\text{length}(\text{add}(s, v)) &= \text{length}(s) + 1 \\
\text{k}(\text{new}(n)) &= n \\
\text{k}(\text{add}(s, v)) &= \text{k}(s) \\
\text{nth}(\text{add}(s, v), i) &= v \quad \text{if } i = \text{length}(s) \\
\text{nth}(\text{add}(s, v), i) &= \text{nth}(s, i) \quad \text{if } i < \text{length}(s) \\
\end{align*}
\]

Figure 2.2: Simple constructors and observers for sequences.

familiar “cons” operation on lists. The type rules require the symbol and sequence to have the same width. \text{length} and \text{k} return the length and width of the sequence, respectively; and \text{nth} will retrieve the symbol at the given position, starting from the beginning.

Figure 2.4 shows more terms for sequences. The \text{skrink} and \text{setK} functions effectively translate the sequence from one type to another. \text{skrink} reduces the width as much as possible without violating any of the contained symbols’ representations. \text{setK} is more general, adding or removing the appropriate number of zero bits to each member symbol to reach the specified width. \text{setK}(s, k) is not defined for \( k < \text{minWidth}(s) \).

\text{cast} is more complicated; it reinterprets the raw bitstream of the symbols with the new specified k-value. This is somewhat analogous to casting a pointer in C, for example, except that the sequence \text{cast} can handle any new k-value and must handle cases where the old and new k-values are not evenly divisible. In these cases, \text{cast} will pad the last element with zeros. For the sake of simplicity, \text{cast} is underspecified in Figure 2.4, showing only the behavior for cases where zero bits need not be added. A more complete specification comes at a high notational cost, and is therefore omitted. Figure 2.3 provides an example of this operation graphically.

\text{shift} removes the specified number of symbols from the head of the se-
sequence; \textit{truncate} removes symbols from the tail to match the specified sequence length. It is important to note that the parameter to \textit{shift} is the number of symbols to be removed, whereas the parameter to \textit{truncate} is the final size of the sequence.

\textit{split} is the left-inverse of \textit{combine}; the latter takes two sequences, not necessarily of the same type, and combines them into a single sequence. \textit{split} will produce the inputs to \textit{combine} exactly, but is undefined for sequences generated in other ways.

Figure 2.5 provides an example of sequence functions in use. The list notation on the second line is meant to abbreviate a string of \textit{add} operations in order.

\section*{2.3 Compressors}

In practice, lossless compression programs combine smaller algorithms. Gzip, for example, uses Lempel-Ziv ’77 (sliding window) to first replace repeated subsequences with pointer-length pairs; it then compresses literals and pointers with one Huffman tree, and lengths with another [Gailly and Adler 2001]. A graphical representation of gzip’s method is shown in Figure 2.6.

To model these practical forms of compression construction, we introduce \textit{comressor components} (or \textit{components}) as the opaque building blocks of compressors.

\textbf{Definition 3 (Compressor Components)} A \textit{compressor component} is a pair of functions

\begin{align*}
  f &: \alpha \rightarrow \beta \\
  f^{-1} &: \beta \rightarrow \alpha
\end{align*}

such that $f^{-1}(f(x)) = x$ for all $x$. $f$ is called the “forward function;” $f^{-1}$ is called the “reverse function.”

A component is a function and its left-inverse. The forward function for a particular component is polymorphic in its return type; it will often return a single
sequence, a set of sequences, or a dictionary or some other type of hierarchical structure in practice. While compressor components will nearly always take sequences as input, they are left polymorphic in their input as well; it is often useful to combine sequences, for example, or use other data types like dictionaries. The gzip method can be constructed from Lempel-Ziv '77 (LZ77) and Huffman components. LZ77's forward function returns two streams: a literals and pointers stream, and a lengths stream. Each is passed to a different Huffman component, which produces a stream and a dictionary. Figure 2.6 mirrors this component construction. The "LZ 77," "Huffman" and "combine" boxes are components by the definition (we presume they have inverses); and so the entire "Gzip" box is also a component.

A compressor is informally defined as a compressor component or the composition of compressor components. The composition of two components \( a \) and \( b \) is defined as the pair of functions \( a . \text{forward} \circ b . \text{forward}, b . \text{reverse} \circ a . \text{reverse} \) where \( a . \text{forward} \) refers to the forward function of component \( a \) and so forth. The composition shown in Figure 2.6 is therefore not technically legal; we must compose both Huffman boxes with the LZ 77 box in one step. This is accomplished by grouping the Huffman boxes together into a single box, represented by the dotted rectangle around them. The composition rules are not meant to be pedantic; they are relied upon heavily in Chapter 4.

It can be trivially shown that the composition of any components is also a component; a compressor therefore always contains a forward function and its left-inverse. Necessarily, the composition of any two compressors is also a compressor; it is equivalent to the composition of their constituent components.

A fairly complicated example of component composition is shown in Figure 2.7. The dictionary replacement component produces both a dictionary and a stream; the stream is passed to a move-to-front with dictionary (MTFD) component, which in turn produces a stream and a dictionary. The stream is compressed using the Burrows-Wheeler transform (BWT); the dictionaries are both compressed using gzip. Finally, everything is combined into a single sequence. As presented, the construction is not a legal composition of components. In this and subsequent figures, we will not rearrange the diagram to fit the rules of composition; the figure is meant to approximate the formal description, and can easily be modified into a composable form.
\textit{shrink}: \quad \text{\#n sequence} \rightarrow \text{\#m sequence}

\textit{setK}: \quad \text{\#n sequence} \ast \text{number} \rightarrow \text{\#m sequence}

\textit{cast}: \quad \text{\#n sequence} \ast \text{number} \rightarrow \text{\#m sequence}

\textit{shift}: \quad \text{\#n sequence} \ast \text{number} \rightarrow \text{\#n sequence}

\textit{truncate}: \quad \text{\#n sequence} \ast \text{number} \rightarrow \text{\#n sequence}

\textit{combine}: \quad \text{\#n sequence} \ast \text{\#m sequence} \rightarrow \text{\#o sequence}

\textit{split}: \quad \text{\#o sequence} \rightarrow \text{\#n sequence} \ast \text{\#m sequence}

\begin{align*}
\text{minWidth(new}(k)) &= 0 \\
\text{minWidth(add}(s, v)) &= \begin{cases} 
\text{minWidth}(v) & \text{if } \text{minWidth}(v) \geq \text{minWidth}(s) \\
\text{minWidth}(s) & \text{otherwise}
\end{cases} \\
\text{setK(add}(s, v), k) &= \text{add(setK}(s), \text{setK}(v, k)) \\
\text{setK}(\text{new}(k'), k) &= \text{new}(k) \\
\text{shrink}(s) &= \text{setK}(s, \text{minWidth}(s)) \\
\text{split(combine}(s, t)) &= (s, t) \\
\text{shift(add}(s, v), i) &= \begin{cases} 
\text{add(new}(k(s)), v) & \text{if } \text{length}(s) = i \\
\text{add(shift}(s, i), v) & \text{if } \text{length}(s) > i
\end{cases} \\
\text{shift}(\text{new}(k), i) &= \text{new}(k) \\
\text{truncate(add}(s, v), i) &= \begin{cases} 
\text{add}(s, v) & \text{if } i \geq \text{length}(s) + 1 \\
\text{truncate}(s, i) & \text{otherwise}
\end{cases} \\
\text{cast}(\text{cast}(s, k), k') &= s \text{ if } k \mod k' = 0
\end{align*}

\textit{Figure 2.4: More terms over sequences, and their laws.}
\[ s = \text{new}(16) \]
\[ s = \text{add}(s, [5, 124, 24, 62, 72, 57, 8]) \]

\[
\begin{align*}
\text{length}(s) & \quad [= 7] \\
\text{nth}(s, 4) & \quad [= 72] \\

t & = \text{shrink}(s) \\
k(t) & \quad [= 7] \\
u & = \text{cast}(s, 8) \\
k(u) & \quad [= 8] \\
\text{length}(u) & \quad [= 14] \\
\text{nth}(u, 0) & \quad [= 5] \\
\text{nth}(u, 1) & \quad [= 0] \\
\text{nth}(u, 2) & \quad [= 124] \\
\text{length}(& \text{shift}(u, 4)) & \quad [= 10] \\
\text{length}(& \text{truncate}(u, 5)) & \quad [= 5] \\
\text{length}(& \text{truncate}(u, 50)) & \quad [= 14] \\
\end{align*}
\]

*Figure 2.5: Sequence example.*

*Figure 2.6: Gzip’s algorithm.*
Chapter 2: Elements of Sequence Compression

Figure 2.7: A compressor and its constituent components.
Chapter 3

Representations of Machine Instructions

A machine instruction is interpreted by the processor as a sequence of fields. Fields contain the opcode, register names, immediate values and other parameters used by the instruction. Fields may be as short as a single bit, or could span multiple bytes. Because machines are byte-addressed, the full instruction itself must span some integer number of bytes. In some cases, additional unused bits are added into the instruction to maintain this property.

Figure 3.1 shows two instructions from the Sparc instruction set. As illustrated, the fields within the instructions do not fit well into bytes. The first byte contains two fields in fact; the second contains one, and a portion of another. The `simm13` field spans more than one byte. This makes standard text compression, which treats each byte as a symbol, less effective on instruction streams than on streams where every data element is an integer number of bytes. A text compressor will not be able to detect like `rd` fields when followed by unlike `rs1` fields; the byte that contains the `rd` field will have a different value.

In this chapter, we present methods for converting the instruction stream into multiple sequences of symbols. Symbols represent atomic pieces of the instruction; there are no implicit boundaries within symbols. The sequence compression elements in Chapter 2 are thereby made applicable to instruction streams.

<table>
<thead>
<tr>
<th>op</th>
<th>op3</th>
<th>rd</th>
<th>rs1</th>
<th>i</th>
<th>simm13</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>op3</td>
<td>rd</td>
<td>rs1</td>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: The Sparc ADD and LD instructions, respectively.
3.1 Two Views of the Instruction

As mentioned earlier, the instruction appears to the processor as a sequence of fields containing opcode and data values. To model this view, we introduce the field tree representation of machine instructions. The Sparc ADD instruction is shown in Figure 3.2.

Informally, a field tree is an ordered tree with a dummy “root” node and a child node for each field, in order. Each node contains the name of the field, the value of the field and its width. From this information the binary form can be unambiguously generated; it is the concatenation of the binary representation of the node values, with the specified width.

From an assembly language perspective, on the other hand, an instruction is a function applied to operands for its side-effect on the machine’s registers and memory. The below assembly code example adds the numbers between one and ten, and stores the result at the memory location in register 4.

```
(assembly code 14)≡
XOR(r1, r1)
XOR(r2, r2)
INC(r2)

loop: TEST(r2, 11)
JE(end)
ADD(r1, r1, r2)
INC(r2)
J(loop)

end: MOV([r4], r1)
HALT()
```

To model the assembly language view, we introduce constructor trees. The Sparc LDSB instruction is shown, in constructor tree form, in Figure 3.3.
Chapter 3: Representations of Machine Instructions

A constructor tree for an instruction \( I \) is patterned after the abstract syntax tree for \( I \)'s assembly language expression. A constructor tree, informally, is an ordered tree with the constructor name at the root, and each of its operands, in order, as its children. As in the \( \text{LDSB} \) example, the children themselves may or may not be leaves; memory addresses, for example, are the result of applying another constructor to a register and an immediate value.

The primary reason we have chosen to represent sequences of fields as “flat” trees, as opposed to lists of fields, is that we wish to introduce a single basic form for instruction representation that can capture both views. The problem of instruction stream compression, under either model of instruction representation, becomes a problem of compressing trees.

In general, each leaf node in a constructor tree is represented by the same node in a field tree. The \( \text{ADD} \ \text{rs1} \) parameter, for example, is placed in a node named “\( \text{rs1} \)” with the same value and width in each tree. Internal nodes in a constructor tree do not have symbols associated with them, and do not bear the name of the field from which they came; in field trees, however, this information is stored directly in a node. Children of internal nodes in constructor trees are often renamed; the field name for the offset of a branch is “\( \text{disp22} \)” on the Sparc, but the constructor tree node is named “target” for this and “\( \text{disp30} \)” fields. This is an important difference between field and constructor trees, because it can cause different types of fields (with different widths) to appear with the same name. Constructor trees may also present values in a different order; the \( \text{rs2} \) parameter for an \( \text{ADD} \) comes at the very end of the instruction on the Sparc, but will come immediately after the \( \text{rs1} \) parameter at the beginning of its constructor tree.

\footnote{The name information is actually superfluous to the binary form reconstruction, although it is very important for subsequent procedures over field trees.}

\footnote{The term ‘constructor’ is borrowed from [Ramsey and Fernández 1997]; we have adopted their view of expressions like \( \text{ADD}(r1, r1, r2) \) evaluating to the binary representation for that instruction. In this sense, \( \text{ADD} \) is a constructor for binary machine instructions.}

Figure 3.3: The constructor tree for the Sparc \( \text{LDSB} \) instruction. This instruction loads a signed byte into register 1 from the location specified by the addition of register 2 and the value 7.
3.2 Trees as Sequences

We compress instruction trees by splitting them into sequences, which are themselves accessible to compressors as we have defined them. The term for this general tree-splitting operation is ‘patternization,’ borrowed from Ernst et al. [Ernst, Evans, Fraser, Lucco, and Proebsting 1997]. To describe patternization, we employ two tools familiar from Chapter 2—informal descriptions and graphical examples—in addition to a formal, code-form expression of the algorithm. This section presents patternization informally; the more complicated formal description is deferred to Section 3.3.

Patternization, in general, evicts some leaf nodes from the instruction tree, placing them in their own sequences. What remains after eviction is called a skeleton tree; it is then placed in its own sequence. The original tree can be reconstructed by putting the evicted nodes back in the skeleton tree. Figure 3.4 gives a conceptual rendering of patternization.

The figure shows two of LDSB’s three leaf nodes having been evicted; patternization allows some or all nodes to remain in the tree if desired. The information about which nodes are to be evicted is contained in a function supplied to patternization. The function is folded over the tree’s leaf nodes in pre-order; if it returns a name for the node, the node is then evicted to the sequence of that name.\(^3\) Hence, it is referred to as the “naming function.” In Figure 3.4, the naming function could have returned “o-stream 1” or “o-stream 2” for the rd node, and similarly for the rs1 node; it could not have returned any name for simm13.

Patternization is reversed by inserting the evicted nodes back into the instruction tree. The location of each evicted node is found by again applying the naming function; when applied to an empty branch (a dotted line in Figure 3.4)

\(^3\)Because we fold over leaf nodes only, internal nodes (and therefore elements with structure) cannot be evicted. This algorithm can be generalized to allow internal-node eviction, but the notational costs are extremely high. The practical consequences do not outweigh them.
where a node has been evicted, the naming function must return the same name that it produced when first invoked on the original node. The difficulty is in supplying the naming function with its original inputs, using only the skeleton tree. To see how this is accomplished, we must look more deeply at the structure of instruction and skeleton trees. Figure 3.5 revises the procedure in Figure 3.4 to show the instruction and skeleton trees as they appear to the naming function.

As shown, only the value at each node is actually evicted. The node itself, along with its name and width, remain within the skeleton tree. This allows the naming function to depend on any information in the original tree except the node values.

3.3 A Formal Description of Patternization

3.3.1 Elements

As in preceding chapters, we must first introduce some basic formal elements that will be used to describe larger formal concepts. The first is a notion of k-symbols, shown below.

\[ (k\text{-symbol form } 17) \equiv \]

\[ \text{ksym} ::= \{\text{width : int, value : int option}\} \]
This notation describes a structure with two members, named “width” and “value” having type \texttt{int} and \texttt{int option}, respectively. This fits the definition of symbols presented in Chapter 2 (page 6); a k-symbol is made up of a string of bits (its value) and a width. The value is optional in the above form because patternization must be able to evict it. The details of this procedure are presented in later sections.

The k-symbol form allows the expression of trees, shown below.

\begin{verbatim}
18a \langle tree form 18a⟩≡
  tree ::= {name : string, data : NONE, children : tree list}
       | {name : string, data : ksym, children : []}
\end{verbatim}

A tree can have one of two forms. An internal or root node has a name, a non-empty child list and no data member. A leaf node has a name and a full k-symbol associated with it. This form is general enough to capture all the information in an instruction tree; it associates a name, a width and a value with each leaf node, and a name with each non-leaf node.

Finally, the data that will be available to the naming function is contained in the following \texttt{nodedata} structure.

\begin{verbatim}
18b \langle nodedata form 18b⟩≡
  nodedata ::= {name : string,
               parentname : string,
               childnum : int,
               width : int}
\end{verbatim}

The name and width of the node appear within the \texttt{nodedata} structure, as well as information pertaining to the node’s parent: the parent’s name, and the index number of the child (the node itself). The naming function has type \texttt{nodedata → string option}.

We also employ a table of sequences, accessed through the \texttt{addTable} and \texttt{getTable} functions. These are described formally in Section 3.4.2; informally, they represent the interface to a table of FIFO buffers. \texttt{addTable} takes a name, indicating the particular buffer, and a value, inserting it into the named buffer. \texttt{getTable} also takes a name, and will remove the first value from the named buffer. The central measure of the \texttt{getTable} and \texttt{addTable} functions is that invoking \texttt{getTable} in sequence will return the values added with \texttt{addTable} in order. This property is relied upon in subsequent sections.

The notation we will use for code-form expressions is similar to the syntax of ML [Harper, Milner, and Tofte 1990]. Below is a short example.

\begin{verbatim}
18c \langle notation examples 18c⟩≡
  fun add(a, b) = (a + b)
  fun curriedadd a b = (a + b)
  fun listlen [x | xs] = 1 + (listlen xs)
  | listlen [] = 0
\end{verbatim}
The second function, \texttt{curriedadd}, is the curried form of the first add function.\footnote{For a good treatment of currying, please refer to \cite{KaminRamseyCox2000}, Chapter 2.} The \texttt{listlen} function exhibits the notation for pattern-matching; if presented with a non-empty list it will execute the first form, and if presented with an empty list \texttt{[]} it will execute the second. ‘\texttt{xs}’ is itself a list, whereas ‘\texttt{x}’ is an element in the list; they correspond to the \texttt{car} and \texttt{cddr} of the list. The vertical bar notation is borrowed from Prolog.

19a \begin{verbatim}
<notation examples 18c>+
\texttt{fun comparelen list1 list2 =}
  \texttt{let}
  \texttt{len1 = listlen list1
  len2 = listlen list2}
  \texttt{in}
  \texttt{if len1 > len2 then 1
  elseif len1 < len2 then -1
  else}
  \texttt{0}
\texttt{end}
\end{verbatim}

This function illustrates local environments within functions. The \texttt{let} binding allows the expressions \texttt{listlen list1} and \texttt{listlen list2} to be bound to the variables \texttt{len1} and \texttt{len2} within the expression enclosed by \texttt{in...end}.

19b \begin{verbatim}
<notation examples 18c>+
\texttt{fun newtree name symbol kids = TREE\{name=name, data=symbol, children=kids\}
\texttt{fun treename t = t.name}
\texttt{fun alsonetree name {name=nn, data=dt, children=kids} = nn}

\texttt{fun printvalue1 SOME(x) = print(x)
  | NONE = print("-")}
\texttt{fun printvalue2 symbol = case (symbol.value) of
  SOME x => print(x)
  | NONE => print("-")}
\end{verbatim}

Finally, these functions illustrate how datatypes can be created and observed. \texttt{newtree} is equivalent to the \texttt{TREE} constructor for trees; the constructor for each datatype shares its name, in all-capsitals.

A member of a datatype is indicated by the C ‘dot’ notation for structures. One can also use pattern-matching as in \texttt{alsonetree} to extract a member. The \texttt{case} construction creates a switch statement over the case of the input. An \texttt{option} kind can be either \texttt{SOME} of something or \texttt{NONE}; \texttt{printvalue1} uses pattern-matching to distinguish between the two cases, and \texttt{printvalue2} uses an explicit case statement.
3.3.2 A Description of Patternization

Below is the code-form representation of the patternization algorithm.5

\begin{verbatim}
fun patternize naming_func parentname idnum table node =
  case (node.children) of
  | [first | others] => dontevict naming_func table node
  | [] => maybeevict naming_func parentname
  | idnum table node
\end{verbatim}

The patternization function takes a naming function (\texttt{naming\_func}), data needed to construct the nodedata structure (\texttt{parentname} and \texttt{idnum}), the table of sequences to which evicted data will be added (\texttt{table}), and the leaf node to be processed (\texttt{node}). It first determines whether or not the node is a candidate for eviction. Internal nodes cannot be evicted; in the case that the node’s child list matches a non-empty list (line 4), the \texttt{dontevict} function is called. If the node has no children, it may be evicted by \texttt{maybeevict} (line 5), shown below.

\begin{verbatim}
fun maybeevict naming_func parentname idnum table node =
  let
    nodedata = NODEDATA(name = node.name,
                         parentname = parentname,
                         idnum = idnum,
                         width = node.value.width)
    nodename = naming_func(nodedata)
  in
  case nodename of
  | SOME(entry) => (TREE{name = node.name,
                         value = KSYM{width = node.value.width,
                                       value = NONE},
                         children = []},
                         addTable table entry node.value.value)
  | NONE => dontevict naming_func table node
  end
\end{verbatim}

The \texttt{node} input to this function is a leaf, and may need to be evicted. \texttt{maybeevict} first constructs the nodedata for the node, and calls the naming function on it (lines 3-7). It must then switch to handle the case where the node is to be evicted (line 11) or to remain (line 17).

---

5 “Beware of bugs in the above code; I have only proved it correct, not tried it.” From ‘Notes on the van Emde Boas construction of priority dequesues: An instructive use of recursion,” Don Knuth, in a note sent to Peter van Emde Boas on March 29, 1977.
In the former case, patternization wishes to remove the value from the node and add it to the appropriate sequence. This is accomplished in lines 11-15; the function is returning a tuple of the new node (with no value, line 13) and the new table, created by adding the node’s value to the old table (line 15).

In the case where the node is to remain (line 17), the aforementioned dontevict function is called. We have taken some liberty in the maybeevict function: node.value.width and node.value.value are undefined if node.value is NONE. We assert that this cannot be the case for leaf nodes; the overhead for handling these cases is unenlightening.

dontevict is shown below.

21a (patternize algorithm 21a)≡
1 fun dontevict naming_func table node =
2   let
3     (fintable, newkids) = patmap (patternize naming_func node.name) 0
4     table node.children
5     in
6     (TREE{name = node.name,
7       value = node.value,
8       children = newkids},
9       fintable)
10   end

When a node is not evicted, its children are still candidates for eviction. The patmap function maps the patternization function over each of node’s children, and returns the updated sequence table and a list of the new children. The new node is constructed (lines 6-9) from the original data and the new, potentially “empty” children; and the updated sequence table is returned along with it (line 9).

The call to patmap is taking advantage of the curried form of patternization by adding the naming function and parent name parameters in before passing patternization on. The naming function and parent name do not change over the various children. The patmap function is shown below:

21b (patmap function 21b)≡
1   fun patmap func count table [x | xs] =
2     let
3     (newnode, newtable) = func count table x
4     (fintable, rest) = patmap func (count + 1) newtable xs
5     in
6     (fintable, [newnode | rest])
7     end
8   | patmap func count table [] = (table, [])
This function maps patternization over a list of trees. It takes the patternization function with the naming function and parent name already composed in (func), the index of the current child (count), the current sequence table (table) and the list of children ([x l xs]). It first performs patternization on the first child (line 3), storing the new node and new sequence table that are returned. It next recursively applies patternization to the rest of the children (line 4), getting the final table and all the other children as a result. It then returns the final table and all the children (line 6).

The following function completes the definition of the patternization algorithm by specifying the behavior for the root node.

22a \(<patternization \text{ algorithm } 20a>\) +≡

\[
1 \quad \text{fun patternizetree naming_func table root =}
2 \quad \text{donteict naming_func table root}
\]

3.4 Reversibility of Patternization

The original tree is recovered from the skeleton tree and sequence table by inserting node values back into the tree where they were first evicted. This reverse-patternization algorithm folds the naming function over the skeleton tree; each time it returns a name, the first value in the sequence of that name is inserted into the visited node, and removed from the head of the sequence. The subsequent sections present a formal description of reverse-patternization, and prove that it is the left-inverse of patternization.

3.4.1 A Formal Description

22b \(<reverse-patternization \text{ algorithm } 22b>\)≡

\[
1 \quad \text{fun revpatternize naming_func parentname idnum table node =}
2 \quad \text{case (node.children) of}
3 \quad \text{[first | others] => dontinsert naming_func table node}
4 \quad \text{[] => maybeinsert naming_func parentname}
5 \quad \text{idnum table node}
\]
The reverse-patternization function takes the same parameters as patternization: a naming function, the parent name and identification of the node, and the table of sequences with evicted values. Internal nodes cannot have been evicted; \texttt{dcontinsert} is called for any nodes with children (line 4). Others may have themselves been evicted; \texttt{maybeinsert} may replace their values (line 5).

\texttt{(patternization algorithm 20a) +≡}

\begin{verbatim}
1 fun maybeinsert naming_func parentname idnum table node =
2     let
3         nodedata = NODEDATA{name = node.name,}
4             parentname = parentname,
5             idnum = idnum,
6             width = node.value.width}
7     inodenename = naming_func(nododata)
8     case nodename of
9         SOME(entry) =>
10            let
11                (newtable, value) = getTable table entry
12                in
13                   (TREE(name = node.name,  
14                       value = KSYM{width = node.value.width,  
15                       value = value},  
16                       children = []},
17                       newtable)
18            end
19     end
20     end
21 | NONE => dcontinsert naming_func table node
22 end
\end{verbatim}

The value insertion function mirrors the structure of \texttt{maybeevict} shown above. It constructs the nodedata structure for the node (lines 3-7), and either replaces the node’s value (lines 12-20) or leaves it unmodified (line 22). The interesting behavior is in the former case; the value corresponding to the name is “popped” out of the dictionary (line 13) and added back into the tree (line 17). \texttt{getTable} returns a new table that is passed on (line 19), the most recent value having been removed from the stream associated with \texttt{entry}.
Finally, for completeness, we present the `dontinsert` and `revpatternizetree` functions. They have the same form as their predecessors.

24a \(<\text{reverse-patternize algorithm 24a}>\) ≡

\begin{verbatim}
1   fun dontinsert naming_func table node =
2       let
3           (fintable, newkids) = patmap (patternize naming_func node.name) 0
4               table node.children
5       in
6           (TREE{name = node.name,
7               value = node.value,
8               children = newkids},
9               fintable)
10       end
\end{verbatim}

24b \(<\text{reverse-patternization algorithm 22b}>\) +≡

\begin{verbatim}
1   fun revpatternizetree naming_func (root, table) =
2       dontinsert naming_func table root
\end{verbatim}
3.4.2 Proof of the Left-Inverse Property

We begin by introducing some notation on sequence tables to make the following proofs more readable.

- If $T$ is the result of adding name-symbol pairs $(k_1, v_1)\ldots(k_i, v_i)$ to the empty sequence table, $S + T$ is the result of adding the same name-symbol pairs $(k_1, v_1)\ldots(k_i, v_i)$ to $S$.

- If $T$ is a list of names $[k_1, k_2, \ldots k_i]$ and $S$ is the result of adding name-symbol pairs $(k_1, v_1)\ldots(k_i, v_i)\ldots(k_n, v_n)$ to the empty sequence table, $S - T$ is the result of adding the name-symbol pairs $(k_{i+1}, v_{i+1})\ldots(k_n, v_n)$ to the empty sequence table. This is simply a shorthand; it allows us to pull out items from the sequence table by referring to them in another sequence table.

- If $S$ is the result of adding name-symbol pairs $(k_1, v_1)\ldots(k_n, v_n)$ to the empty sequence table, `getTable S k1 = (v1, T)` where $T$ is the result of adding name-symbol pairs $(k_2, v_2)\ldots(k_n, v_n)$ to the empty sequence table.

"Adding name-symbol pairs" is meant to indicate the result of applying the `addTable` function. The `getTable` rule dictates the sequence table's FIFO behavior.

Next, we prove a Lemma on the reversibility of chained patternizations. The key to this proof is tracking the sequence table through `patmap`.

**Lemma 4** Let

$$(T, \text{newkids}) =$$

```
patmap (patternize nfunc tree.name) i table tree.children
```

```
(table’, newkids’) =
```

```
patmap (revpatternize nfunc tree.name) i T’ newkids
```

where $T’ = (T - \text{table}) + S$ for some $S$. Given that Theorem 1 holds for $\text{tree}$, $\text{table’} = S$ and newkids’ = tree.children.

Informally, Lemma 4 says that mapping patternization over a list of children produces a new table $T$; we can perform as many subsequent patternizations as we want ($S$); and when reverse-patternizing, if we have reverse-patternized the nodes that preceded these children ($\text{table}$), applying `revpatternize` to each of the skeleton children produces the original children, and the appropriate sequence table ($S$) for the remaining skeleton children. This lemma is used within the induction step of Theorem 1, and thus can depend on its induction hypothesis. Theorem 1
makes the same statement as the lemma, except for patternization of a single tree. Please refer to page 1 for the theorem’s statement.

We will prove the Lemma by induction on the length of \((\text{#children tree})\).

**Basis Step:** The length of \((\text{tree}.\text{children})\) is 1. By the definition of \text{patmap},

\[
\text{patmap (patternize nfunc tree.name) i table tree.children} = \\
(\text{table}', [\text{tree}'])
\]

where

\[
(\text{tree}', \text{table}') = \text{patternize nfunc tree.name i table tree.children[0]}
\]

We next need to apply to Theorem 1. Let \(T = \text{table}' + S\) for some \(S\), and let \(T' = T - \text{table}\). By the definition of \text{patmap},

\[
\text{patmap (revpatternize nfunc tree.name) i T' tree.children} = \\
([\text{tree}''], \text{table}'')
\]

where

\[
(\text{tree}'', \text{table}'') = \text{revpatternize nfunc tree.name i T' tree'}
\]

Then by Theorem 1, \(\text{tree}'' = \text{tree}.\text{children[0]}\) and \(\text{table}'' = S\).

**Induction Hypothesis:** The length of \((\text{tree}.\text{children})\) is \(N\). Lemma 4 holds.

**Induction Step:** The length of \((\text{tree}.\text{children})\) is \(N + 1\). Let

\[
[\text{firstkid}, \text{otherkids}] = \text{tree.children} \\
(\text{firstkid}_p, \text{table}_p) = \text{patternize nfunc tree.name i table firstkid} \\
(\text{table}_pp, \text{otherkids}_pp) = \\
\text{patmap (patternize nfunc tree.name) (i + 1) table}_p \text{ otherkids}
\]

\(\text{firstkid}_p\) is meant to refer to the first patternized child in a list; \(\text{table}_p\) refers to the resultant sequence table. \(\text{table}_pp\) is refers to the sequence table that is the result of all subsequent patternizations in the list of children; \(\text{otherkids}_pp\) is the list of skeleton children. Note that \(\text{table}_pp - \text{table}_p\) is defined; the final table contains all the sequence additions in the previous table. It is important to be comfortable with the \(p\) and \(pp\) semantics; these tables are used heavily below. \(\text{table}\) is the input table for all the patternizations in the list; \(\text{table}_p\) is an intermediary table, produced after one patternization; and \(\text{table}_pp\) is the final table produced after all the patternizations in the list.

By the definition of \text{patmap},

\[
\text{patmap (patternize nfunc tree.name) i table tree.children} = \\
(\text{table}_pp, [\text{firstkid}_p, \text{otherkids}_pp])
\]

Let \(T = \text{table}_pp + S\) for some \(S\), and let \(T' = T - \text{table}\). We need to show that
Chapter 3: Representations of Machine Instructions

\[
\text{patmap (revpatternize nfunc tree.name) i } T' \ [\text{firstkid}_p \mid \text{otherkids}_{pp}] = (\text{tree.children, } S)
\]

First, let

\[(\text{firstkid}_r, U) = \text{revpatternize nfunc tree.name i } T' \text{ firstkid}_p\]

\text{firstkid}_r is meant to refer to the result of reverse-patternizing the first child that was patterned, \text{firstkid}_p. We observe that

\[
T' = T - \text{table} \quad \text{(Definition of } T')
\]
\[
= \text{table}_{pp} + S - \text{table} \quad \text{(Definition of } T)
\]
\[
= (\text{table}_p + S') + S - \text{table} \quad \text{(Substitution)}
\]
\[
= \text{table}_p - \text{table} + S' + S \quad \text{(Order operations)}
\]
\[
= \text{table}_p - \text{table} + K \quad \text{(Substitution)}
\]

for some \(K\). The difficult step is the “order operations,” fourth from the top. Since we know \(\text{table}_p - \text{table}\) is defined, we can add elements to \(\text{table}_p\) whenever we like without affecting the removal of elements it contained originally (elements are always removed from the front).

Using the above relation, we can apply Theorem 1 to conclude,

\[
\text{firstkid}_r = \text{firstkid}
\]
\[
U = K = S' + S
\]

We note that

\[
U = \text{table}_{pp} - \text{table}_p + S
\]

and we can therefore apply to the Induction Hypothesis, concluding that

\[
\text{patmap (revpatternize nfunc tree.name) (i + 1) U otherkids}_{pp} = [\text{otherkids}, S]
\]

Appealing finally to the definition of \text{patmap}, we ultimately conclude that

\[
\text{patmap (revpatternize nfunc tree.name) i } T' \ [\text{firstkid}_p \mid \text{otherkids}_{pp}] = ([\text{firstkid} \mid \text{otherkids}], S)
\]
\[
= (\text{tree.children}, S)
\]

Next, we prove the left-inverse relationship between \text{patternize} and \text{revpatternize}, using Lemma 4.

**Theorem 1 (Reversibility of Patternization)**

Let

\[
(\text{tree}', T) = \text{patternize nfunc pname id table tree}
\]
\[
(\text{tree}'', \text{table}') = \text{revpatternize nfunc pname id } T' \text{ tree}'
\]
where \( T' = T + S - \text{table} \) for some \( S \). Then \( \text{tree}' = \text{tree} \), and \( \text{table}' = \text{table} \).

Theorem 1's form is familiar from Lemma 4. We will prove Theorem 1 by induction on the height of \( \text{tree} \).

**Basis Step:** The height of \( \text{tree} \) is 1. \( \text{tree.children} = [] \); by its definition,

\[
\begin{align*}
\text{patternize func pname id table tree} &= \text{maybeevict nfunc pname id table tree} \\
\text{maybeevict constructs a nodedata object data from its parameters, and calls nfunc, the naming function, on it. There are two possible outcomes: nfunc can return a string, or nothing.}
\end{align*}
\]

- **Case 1.** \( \text{nfunc(data)} = \text{s} \). By the definition of \text{maybeevict},

\[
\begin{align*}
\text{maybeevict nfunc pname id table tree} &= \text{TREE}{\text{name} = \text{tree.name},} \\
&\quad {\text{value} = \text{KSYM}\{\text{width} = \text{tree.value.width},} \\
&\quad \quad {\text{value} = \text{NONE}}{\},} \\
&\quad {\text{children} = [{}]}{,} \\
&\quad \text{addTable table s tree.value.value})
\end{align*}
\]

Let

\[(\text{tree}', \text{table}') = \text{patternize nfunc pname id table tree}\]

From the above, we conclude that \( \text{table}' = \text{table} + V \), where \( V \) is the sequence table created by adding \((s, \text{tree.value.value})\) to the empty sequence table. Let \( T = \text{table}' + S \) for some \( S \), and let \( T' = (T - \text{table}) = V + S \). Further,

\[
\text{revpatternize nfunc pname id T' tree'} = \text{maybeinsert nfunc pname id T' tree'}
\]

by the definition of \text{revpatternize. maybeinsert constructs a nodedata object data'} from its parameters. From the above, \text{tree'.name} = \text{tree.name},

\[
\text{tree'.value.width} = \text{tree.value.width}; \text{therefore}, \text{data'} = \text{data}, \text{and nfunc(data')} = \text{s}. \text{Let} (\text{newtable, value}) = \text{getTable table'}' s. \text{By the rules on getTable and addTable, newtable} = \text{S and value} = \text{tree.value.value}. \text{By the definition of maybeinsert,}
\]

\[
\begin{align*}
\text{maybeinsert nfunc pname id T' tree'} &= \text{TREE}{\text{name} = \text{tree'.name},} \\
&\quad {\text{value} = \text{value},} \\
&\quad {\text{children} = [{}]}{,} \\
&\quad \text{newtable})
\end{align*}
\]
Chapter 3: Representations of Machine Instructions

\[
S = \text{TREE} \{ \text{name} = \text{tree.name}, \\
                  \text{value} = \text{tree.value}, \\
                  \text{children} = \text{tree.children} \}, \\
\text{S}
\]
\[
= (\text{tree, S})
\]

- Case 2. \text{nfunc(data)} returns no string. By the definitions of \text{maybeevict}, \text{donteveict} and \text{patmap},

\[
\text{maybeevict nfunc pname id table tree} = \\
\quad \text{donteveict nfunc table tree} \\
= \text{TREE} \{ \text{name} = \text{tree.name}, \\
                  \text{value} = \text{tree.value}, \\
                  \text{children} = [], \}
                  \text{table}
\]

Let

\[
(\text{tree'}, \text{table'}) = \text{patternize nfunc pname id table tree}
\]

The above shows that \text{table'} = \text{table}. Let \text{T} = \text{table'} + \text{S} for some \text{S}, and let \text{T'} = \text{T} - \text{table} = \text{S}. Further,

\[
\text{revpatternize nfunc pname id T' tree'} = \\
\quad \text{maybeinsert nfunc pname id T' tree'} \\
= \text{dointinsert nfunc T' tree'} \\
= \text{T'} \\
= (\text{tree, S})
\]
**Induction Hypothesis:** tree has height $N$; Theorem 1 holds.

**Induction Step:** tree has height $N + 1$. By the definitions of patternize and dontevict, we conclude

```haskell
  patternize nfunc pname id table tree =
    dontevict nfunc table tree
  = (TREE{name = tree.name,
         value = tree.value,
         children = newkids},
     fintable)
```

where

```haskell
  (fintable, newkids) = patmap (patternize nfunc tree.name) 0 table tree.children
```

Let tree' refer to the tree returned by patternize. We know that tree.children is non-nil, and that the height of each child is $N$ or fewer; we can therefore apply Lemma 4, at least in principle, to the output of patmap. First, we introduce the familiar references. Let $T = \text{fintable} + S$ for some $S$, and let $T' = T - \text{table}$. By the definitions of revpatternize and dontinsert,

```haskell
  revpatternize nfunc pname id T' tree' =
    dontinsert nfunc T' tree'
  = (TREE{name = tree.name,
         value = tree.value,
         children = newkids'},
     fintable')
```

where

```haskell
  (fintable', newkids') = patmap (revpatternize nfunc tree.name) 0 T' tree'.children
```

We now apply Lemma 4 to conclude that,

\[
\text{fintable'} = S \\
\text{newkids'} = \text{tree.children} \\
\text{revpatternize nfunc pname id T' tree'} = (\text{tree}, S)
\]

\[\boxed{}

### 3.5 Skeleton Trees

The above patternization function updates a table of evicted values, and creates a skeleton tree from the original instruction tree. The sequence table itself is ready for compression; the skeleton tree, on the other hand, may be no closer to
compression than it was before. If no nodes were evicted, for example, the skeleton tree is identical to the original instruction tree; in all other cases, the full structure of the tree and all but the evicted node values are present. We must therefore define a method for representing skeleton trees as symbols, so that sequence compression can be applied.

We have chosen to represent skeleton trees using the binary machine instructions that they describe. A particular skeleton tree can describe more than one machine instruction, if any of its values have been evicted. We pick one such instruction and use it to represent each occurrence of that particular skeleton tree. This is where compression improvement comes from; when values are evicted, the same binary form is used to represent what may have been different instructions in the original stream.

### 3.5.1 Solving the Original Problem

The beginning portion of this chapter illustrated the problem of sub-byte and unaligned fields within machine instructions. Skeleton tree representations still have the same fields in the same positions, but fields that are evicted can no longer be a source of variation between similar instructions. Figure 3.6 shows two Sparc ADD instructions that differ only in their rs1 fields. The second and third bytes do not match before patternization; after patternization evicts the rs1 values, the skeleton trees of the two instructions match, and both are represented by the same bits.

Earlier tree figures are misleading in their presentation of the amount of information removed through node eviction. Although all node information remains except the values, this information is not contained explicitly in the instruction but is deduced by the machine's decoder. The skeleton tree structure, as well as the
Figure 3.7: A bit view of a fully-patternized instruction. Shaded bits are unspecified, and therefore take the same values for all matching skeleton trees.

node names and widths can all be inferred from the instruction's opcode. Figure 3.7 shows how many bits can be ignored by the compressor in a fully-patternized Sparc ADD instruction (i.e. when all the leaf nodes have been evicted).

3.6 Extensions

The following extensions are added to the formal definition presented above.

- Patternization can evict the root through a postprocess addition to the algorithm. This mechanism allows like instructions to be grouped together in their own streams. The details of root eviction are presented in Appendix A.

- Unused bits in instructions are handled so that the stream after patternization has been applied and reversed is identical to the original. The method is described in Section 3.6.1.

- Raw data in the instruction stream is stored in the skeleton tree stream without modification. Reverse patternization, when trying to interpret it as a tree, again recognizes it as raw data and outputs it unmodified.

- Redundancy in field tree streams, specifically evicted opcodes and the skeleton tree stream, is eliminated by discarding the full skeleton tree stream. The algorithm presented in Section 3.6.2 illustrates how the skeleton tree stream is reconstructed.

3.6.1 Handling Unused Bits

Some instruction bit-patterns contain unused bits. The Sparc LD instruction shown at the beginning of this chapter (fig. 3.1, page 13) contains an eight-bit space between the i and rs2 fields. Although unused bits are not used in decoding the instruction and do not affect its meaning to the processor, they may affect the semantics of the overall program if it reads from its instruction stream. We have chosen to store unused bits to ensure that program semantics cannot be affected by
patternization.⁶

In the case of field trees, we insert “bits” nodes where appropriate to capture holes in the bit pattern. The ADD example is shown in Figure 3.8; a “bits” node of width 8 is inserted between i and rs2. The transformation from field trees to instruction bitstream simply packs the stream with the bits in the nodes it encounters as it reads left-to-right. The addition of “bits” nodes therefore does not affect the instruction encoder.

Constructor trees are more complicated. There is no expression in the specification language that will yield a particular value for unused bits. We therefore check unused bits when creating skeleton tree representations for constructor trees. Two skeleton trees that match and have matching unused bits will be represented by the same form, which necessarily contains the unused bits.

3.6.2 Retrieving Field Trees

For constructor trees, the skeleton tree stream effectively serves as a list of root and internal nodes. The root, in constructor trees, unambiguously specifies the names of its child trees, and similarly for all internal nodes. From these, the original tree structure can be completely reconstructed. In field trees, however, the root serves only to tie fields together. It contains no information about the fields themselves; this information is contained in the opcode nodes. Patternization on field trees therefore duplicates information when opcode nodes are evicted; the skeleton tree stream still has enough information to reconstruct the original tree forms (an invariant of our algorithm), but the opcode streams contain this information as well.

To eliminate this redundancy, we introduce an algorithm for reconstructing trees without the skeleton tree stream when opcodes have been evicted. This algorithm is sketched in Figure 3.9.

The first step is to retrieve the opcode field or fields; these are known to be present in all field trees. A “fake” skeleton tree is constructed with these nodes; inverse patternization then fills them with their original values. From this information, the skeleton tree can be either partially or completely reconstructed.

⁶The most immediate practical implication of this decision is that patternization’s correctness can be more easily verified: a file that is patternized and then rebuilt will match the original file, bit for bit.
from the architecture specification. In some cases, other fields need to be used in determining the full instruction format. Step 2 illustrates this procedure. It is important to note that the fields must be retrieved in order. We cannot skip the \texttt{rd} and \texttt{rs1} fields in retrieving the value for \texttt{i} in the above example because they may reside in the same stream. Retrieving fields in order is a condition of Theorem 1.

There could be any number of intermediary “Step 2’s” before the tree is fully reconstructed. Step 3 indicates the final result of the algorithm. The skeleton tree stream may also contain raw data symbols, which must be preserved by the algorithm. Raw data is extracted from the skeleton tree stream and placed in its own “raw data” stream; the skeleton tree stream is replaced by a sequence of indices, serving as placeholders for raw data. This is a practical use of root eviction, described in Appendix A.
Chapter 4

A Language for Compression: Lua-ISC

The elements of stream compression introduced in Chapter 2—symbols, sequences and compressors—along with the patternization method and supporting tree structures presented in Chapter 3, constitute a basis for constructing compression algorithms over instruction streams. This chapter presents a tool for computing over those constructions, the Lua-ISC\(^1\) language. Lua-ISC is the result of extending the Lua language over stream and instruction compression elements.

4.1 An Introduction to Lua

Lua was designed by researchers in the Computer Graphics Technology lab at the Pontifícia Universidade Católica de Rio de Janeiro (PUC-Rio). Its original use was as an embedded configuration language for large applications; it exports a C interface for creating new data types and adding functions to the language. Lua is a safe, dynamically typed language with first-class functions. Its syntax is similar to Pascal; Lua code is generally imperative in style. Lua’s semantics can be extended dynamically through a method called fallbacks, which provide support for operator overloading. Developers outside PUC-Rio have contributed extension modules for a wide range of different tasks, from socket programming to HTML parsing.

The basic types in Lua are number, string, function, table and nil. Numbers in Lua are 32-bit floats. Basic arithmetic operations are defined over number, as are tests for equality and order. A string is as one would expect, representing zero or more characters; concatenation is defined over strings. Functions, both written in Lua and as extensions, have the function type. A table serves as both a dictionary and a list. It is a set of name-value pairs, where the name can have type string or number, and the values can have any (not necessarily unifiable) types. Elements can be added to and removed from a table. The contents of tables can be enumerated in Lua through a next function.

\(^{1}\)For Lua Instruction-Stream Compression. Pronounced “LEW-isk;” the 'a' is elided.
Tables are used extensively in Lua, not only in programming but in its implementation; environments are implemented directly as Lua tables. This allows, among other things, the garbage collector to be at least partially implemented in Lua. Finally, the nil type means "error," "false" or "none" in Lua. Setting an table element to nil effectively removes that element; trying to retrieve a non-existent element will result in nil. Functions can simulate option types by returning either data of a certain type (like string or number), or nil in the case of no data.

Appendix C.1.1 (page 89) contains an example of Lua code, an insertion sort function.

## 4.2 K-Symbols and K-Sequences

```
new : number * number -> #n symbol
k : #n symbol -> number
value : #n symbol -> number

k(new(x, y)) = x
value(new(x, y)) = y
```

*Figure 4.1: Lua-ISC functions over k-symbols.*

Lua-ISC supports symbols and sequences as they appear in Chapter 2, called "k-symbols" and "k-sequences" in Lua-ISC respectively.

Because Lua can only express 32-bit numbers, the value of a k-symbol created in Lua-ISC is artificially bound as its constructor takes its value parameter as a number. Further, k-symbols created outside Lua-ISC (in the underlying C implementation of patternization, for example) may exceed Lua-ISC’s number representation limit and are therefore not observable in the language. The value function, provided for observing the value of a k-symbol, will cause a run-time error for any symbol value greater than 32 bits in width. The Lua-ISC functions on k-symbols are shown in Figure 4.1.

In practice, this restriction is not limiting. Compressors are usually composed of built-in components, themselves immune. Higher-precision values are used exclusively to store instructions for the variable-width Pentium set; with the exception of Pentium skeleton trees, Lua’s number type is sufficient.

K-sequences are supported by Lua-ISC as they appear in Chapter 2, except that the add operation is a mutator in Lua-ISC. K-sequence functions are accessible through a KSequence table in Lua-ISC; the path to the new function for k-sequences, for example, is KSequence.new. Similarly, k-symbol functions are stored in the KSymbol table. Figure 4.2 displays some k-sequence Lua-ISC code.
(k-sequence example 37)≡

\[
s = \text{KSequence.new}(16)
\]

\[
\begin{align*}
\text{KSequence.add}(s, \text{KSymbol.new}(16, 5)) \\
\text{KSequence.add}(s, \text{KSymbol.new}(16, 124)) \\
\text{KSequence.add}(s, \text{KSymbol.new}(16, 24)) \\
\text{KSequence.add}(s, \text{KSymbol.new}(16, 62)) \\
\text{KSequence.add}(s, \text{KSymbol.new}(16, 72)) \\
\text{KSequence.add}(s, \text{KSymbol.new}(16, 57)) \\
\text{KSequence.add}(s, \text{KSymbol.new}(16, 8))
\end{align*}
\]

\[
\begin{align*}
\text{KSequence.length}(s) & \quad \text{--} \quad = \quad 7 \\
\text{KSequence.nth}(s, 4) & \quad \text{--} \quad = \quad 72 : \text{ksymbol}
\end{align*}
\]

\[
\begin{align*}
t & = \text{KSequence.shrink}(s) \\
\text{KSequence.k(t)} & \quad \text{--} \quad = \quad 7
\end{align*}
\]

\[
\begin{align*}
u & = \text{KSequence.cast}(s, 8); \text{KSequence.k(u)} \quad \text{--} \quad = \quad 8 \\
\text{KSequence.length}(u) & \quad \text{--} \quad = \quad 14 \\
\text{KSequence.nth}(u, 0) & \quad \text{--} \quad = \quad 5 : \text{ksymbol} \\
\text{KSequence.nth}(u, 1) & \quad \text{--} \quad = \quad 0 : \text{ksymbol} \\
\text{KSequence.nth}(u, 2) & \quad \text{--} \quad = \quad 124 : \text{ksymbol}
\end{align*}
\]

\[
\begin{align*}
\text{KSequence.length}(\text{KSequence.shift}(u, 4)) & \quad \text{--} \quad = \quad 10 \\
\text{KSequence.length}(\text{KSequence.truncate}(u, 5)) & \quad \text{--} \quad = \quad 5 \\
\text{KSequence.length}(\text{KSequence.truncate}(u, 50)) & \quad \text{--} \quad = \quad 14
\end{align*}
\]

*Figure 4.2: K-sequence example in Lua-ISC. Translated from Figure 2.5.*

### 4.3 Compressors

Abstractly, compressor components in Lua-ISC are opaque blocks that can be applied, reverse-applied and composed. They are implemented as tables of functions, supporting forward, reverse and composition operations. It is not necessary to maintain (or construct) the inverse for a particular component: composition maintains the properties of components. The compressor component functions are shown in Figure 4.3.

\[
\begin{align*}
\text{apply} & \quad : \quad 'a \rightarrow 'b \\
\text{revapply} & \quad : \quad 'b \rightarrow 'a \\
\text{compose} & \quad : \quad 'a \ 'b \ \text{component} \times \ 'b \ 'c \ \text{component} \rightarrow \ 'a \ 'c \ \text{component}
\end{align*}
\]

\[
\text{revapply(apply(s))} = s
\]

*Figure 4.3: Compressor components in Lua-ISC. The tick notation is meant to indicate a Greek letter; ’a is read ‘alpha.’*

The apply and revapply functions are accessed using Lua’s colon no-
tation for tables:² “gzip:apply(s),” for example, applies gzip to k-sequence s. Lua’s string concatenation operator ’..’ has been overridden to handle component composition in Lua-ISC.

New components can be introduced using the new function, shown below.

```
new : ('a -> 'b) * ('b -> 'a) -> 'a 'b component
```

new takes apply and revapply functions, and constructs a compressor object out of them. It is necessary that revapply be the left-inverse of apply in order to meet the definition of components. Although the type system will allow any type-legal application of new to two functions, we consider only expressions where the second function is the left-inverse of the first to result in components.

Lua-ISC offers the following basic compressor components.

38 (primitive compressors 38)≡

```
arithmetic : #n sequence -> 8 sequence
huffman : #n sequence -> 8 sequence * #n sequence
mtf : 8 sequence -> 8 sequence
mtfd : #n sequence -> #n sequence * #n sequence
dictionary : #n sequence -> #n sequence * #n sequence
rle : #n sequence -> #n sequence
bwt : #n sequence -> #n sequence
gzip : 8 sequence -> 8 sequence
```

These components are accessible through the Compressor table in Lua-ISC. The arithmetic component performs arithmetic coding on its input, representing symbols with as fewer bits. The huffman component accomplishes a similar task, with a different underlying algorithm (Huffman also creates an explicit dictionary). These components, called entropy coders, do not strictly fall within our definition of components; their output is not a sequence of like-width k-symbols, but a sequence of k-symbols of varying width. In the case of arithmetic coders, the widths are not even integral. We have artificially expressed the output of entropy coders as a sequence of 8-symbols. Because symbol boundaries are not respected in the output of entropy coders, they should not be applied before other coders. [Bell, Cleary, and Witten 1990] describe entropy coders and the other algorithms in detail.

²The component table structure contains information about the compression procedure; this allows components to be composed and maintain the expected properties. The apply function therefore needs to know the identity of its component; the colon silently inserts a self parameter to the function, referencing the table.
mtf and mtfd do not reduce the size of their inputs, but can make them more amenable to subsequent compressors. mtf implements a simple Move-To-Front (MTF) coder; it begins with a dictionary of 256 symbols, sorted. mtfd is more general; it builds a dictionary up as it processes its input, and can therefore handle wider sequences (MTF needs to manage a table of all possible symbols). dictionary is a simpler form of mtfd; it traverses its input, creating a dictionary entry for each new symbol it sees and replacing each symbol with its dictionary index. dictionary may compact its input somewhat; “sparse” k-sequences, having large widths but using few distinct symbols in that space, will be represented by a thinner indices stream and a short dictionary.

rle is a run-length coder; it effectively replaces subsequences of two or more like symbols with a symbol and the length of the run.

but implements the Burrows-Wheeler transform [Burrows and Wheeler 1994]. This component alone does not reduce the size of its input, but is likely to group similar symbols together. gzip represents the gzip algorithm, described by [Gailly and Adler 2001].
Below is an example of component creation and composition using these elements. It creates a full Burrows-Wheeler compressor as described by [Burrows and Wheeler 1994]. From the above types, we can infer that this composition has type 8 sequence -> 8 sequence; the mtf and arithmetic coders restrict the preceding ones to 8-bit sequences. This can be circumvented by inserting a translating component between rle and mtf. A translating component converts an arbitrary k-sequence to a particular width; Appendix C.1.2 presents a very simple one.

\begin{verbatim}
40a (compression example 40a)≡
    BWT_pipeline = (Compressor.bwt .. Compressor.rle ..
                      Compressor.mtf .. Compressor.arithmetic)

Finally, we build a compressor that performs dictionary replacement on its input, and then compresses its dictionary using the BWT pipeline.

40b (compression example 40a)+≡
    function app(datastream, dictionary)
      return BWT_pipeline:apply(datastream), dictionary
    end

    function revapp(datastream_c, dictionary_c)
      return BWT_pipeline:revapply(datastream_c), dictionary_c
    end

    full_compressor = Compressor.dictionary .. Compressor.new(app, revapp)
\end{verbatim}

\textit{Figure 4.4: Building a compressor from components in Lua-ISC.}

4.4 Patternization

The interface to patternization in Lua-ISC is slightly different from Chapter 3’s treatment of it. Chapter 3 presents a two-step procedure, translating from a bitstream to an instruction tree list, and then applying patternization to convert to a set of sequences. In Lua-ISC, the entire procedure is represented by one function, \texttt{patternize}, which takes the instruction tree translation mechanism as a parameter. This hides the tree representation from Lua-ISC, requiring that the programmer use the nodedata type to access it.

The nodedata type is presented in Lua-ISC as it was in Chapter 3, with a few additions. It is named \texttt{NodeData} in Lua-ISC; its functions are repeated below for convenience.

The \texttt{isLeaf} and \texttt{isRoot} are added to support root eviction. There is no constructor for the \texttt{NodeData} type; it is used only within \texttt{patternize}.

The opaque type \texttt{MachineDescription} contains the information \texttt{patternize} needs to interpret the instruction bitstream. There are no observers for the \texttt{MachineDescription} type in Lua-ISC; its information is used exclusively within \texttt{patternize}. The following machine descriptions are provided:
parentName : NodeData -> string
name : NodeData -> string
isLeaf : NodeData -> number option
isRoot : NodeData -> number option
childNum : NodeData -> number
width : NodeData -> number

sparc.flat
sparc.constructor
alpha.flat
alpha.constructor
mips.flat
mips.constructor
pentium.flat
pentium.constructor

Each machine architecture supports both flat and constructor trees. Finally, the **patternize** and **unpatternize** functions are shown below.³

patternize : ElfData * MachineDescription *
(NodeData -> string option) -> sequence table

unpatternize : sequence table * MachineDescription *
(NodeData -> string option) -> ElfData

unpatternize(patternize(d, m, n), m, n) = d

The forward patternization function, **patternize**, takes an ElfData object, a MachineDescription, and a naming function of type (NodeData arrow string option), and produces a table of k-sequences. The ElfData object supplies the instruction stream, interpreted by the supplied MachineDescription to produce tree representations for each instruction.

### 4.4.1 Patternization Examples

The naming function is the principle element in directing patternization. Below are a few simple examples of naming functions, intended to enforce one’s understanding of the way in which patternization works.

³We use the term “sequence table” to loosely describe tables of any type of sequences. In Lua, the types of elements in a table do not have to unify; a table therefore is never anything more than a table to the type system. In practice, our functions expect that the table contains sequences. This disparity is a shortcoming of the type system, but it offers greater flexibility in creating structures within tables.
41 (single k-sequence 41)≡
  function name(NodeData data)
    return nil   -- "none"
  end

  This very simple name function never evicts values; patternization is then
  the identity function over instruction trees.

42 (k-sequences for widths 42)≡
  function name(NodeData data)
    if NodeData.isLeaf(data) and not NodeData.isRoot(data) then
      return " " .. NodeData.width(data))
    end

    return nil
  end

  This naming function evicts all leaf nodes (but not single roots), and ar-
  ranges them by width. This addresses a recurrent issue in instruction-stream com-
  pression, namely, that values of different widths must be separated if they are to
  avoid being padded. Any k-sequence formed from k-symbols of different width must
  match the largest observed width of the symbols, and pad the other symbols ac-
  cordingly.

  The naming function above can be extended to more fine-grained separa-
  tions; for example, dividing based on parent name as well. These and other types
  of techniques are exercised in Chapter 5, which presents experiments in instruction
  representation.
Chapter 5

Experiments in Representation

This chapter seeks to answer the question, "what is the compression performance of the different tree representations, different patternization methods and different text compressors over the different architectures?" The following are the five dimensions of variation.

- Tree representations. Field and constructor trees.
- Patternization methods. Listed in Section 5.3.
- Text compressors. Gzip and BWT.
- Architectures. Pentium (CISC), Sparc and Alpha (RISC).
- Program. Listed in Section 5.1.

Borrowing statistical terminology, each dimension is called a 'factor.' The performance measure we use is bits-per-bit: the number of bits in the compressed file's text segment divided by the number of bits in the uncompressed text segment. Better values, with this metric, are smaller. The primary goal of this chapter is to exhibit the contribution of each factor to instruction-stream compression.

5.1 Test Corpus

Our primary intention in choosing the members of the test corpus is to generate a representative sampling of instruction sequences for each instruction set. To allow comparison between different platforms, we have restricted the corpus to programs available on all that we support, specifically, Alpha, Sparc and Pentium. Although Lua-ISC contains the machinery for the MIPS architecture, we do not have access to a MIPS machine; these results therefore do not cover that architecture.

The SPEC CPU95 benchmark suite [Reilly 1995] provides the source code for its member applications, allowing them to be compiled on each platform (and potentially with different compilers on any particular one). These programs range
in size from 1,420 to 193,754 lines of code; half are written in C, and half in Fortran. They were originally assembled under the criterion that they must be primarily processor-bound; because it is not clear that this direction will bias their instruction streams, we conclude that they are appropriate for our criteria. Each file is compiled on each target architecture, with the GNU compilers (gcc and g77, respectively) using the -O2 flag. The -O2 optimization setting does not unroll loops or inline functions, but does eliminate common sub-expressions.

The Linux kernel is also included. It represents a particularly pressing opportunity for instruction stream compression; the kernel must obey more rigid size constraints than standard applications. Unlike the SPEC95 benchmarks, the Linux kernel files are taken from different distributions of Linux, and were not compiled by us; they are therefore not an appropriate reference point of comparison among compression on different architectures.

Only the text segment of each file is compressed using the experimental methods, because our instruction-specific methods are not meaningful for raw data. We present the bits per bit measurements for the text segment only; the representation method is not applied to other segments. Figure 5.10 illustrates compression performance for the entire file, where other segments are compressed using standard text compressors.

### 5.2 Experimental Set-up

Standard text compression is the control for these experiments. We apply both gzip and the Burrows-Wheeler pipeline presented in Chapter 4; each treats the instruction stream as a sequence of bytes. The byte compressor code, in Lua-ISC,
is presented in Appendix C.2.1.

The representation experiments run within the following experimental framework.

\[ \text{(experiment shell 45a)} \equiv \]
\[
\text{function compressor(data, machdesc, text_comp)}
\]
\[
\text{local streams = ElfData.patternize(data, machdesc, naming_function)}
\]
\[
\text{-- Compress the data}
\]
\[
\text{(compressor core 45b)}
\]
\[
\text{-- Return the streams}
\]
\[
\text{return streams}
\]

The \text{naming_function} referenced above is defined separately for each experiment. The \text{compressor} function applies patternization, and compresses its streams one by one.\(^1\) The \text{streams} table returned by the \text{patternize} function contains the instruction stream (\text{istream}), a trailing bytes stream (\text{trailing}), and optionally operand streams (\text{ostream n}) and evicted instruction streams (\text{istream n}). Because none of the experiments evict root nodes, the final compression loop will not be exercised; it is provided for completeness only.

The following code implements the compressor core.

\[ \text{(compressor core 45b)} \equiv \]
\[
\text{-- Apply the compressor to the instruction stream}
\]
\[
\text{streams.istream = text_comp:apply(streams.istream)}
\]
\[
\text{-- Compress each of the operand streams}
\]
\[
\text{local id = 1}
\]
\[
\text{local nm = "ostream" .. id}
\]
\[
\text{while streams[nm] do}
\]
\[
\text{streams[nm] = text_comp:apply(streams[nm])}
\]
\[
\text{id = id + 1; nm = "ostream" .. id}
\]
\[
\text{end}
\]
\[
\text{-- Compress each of the instruction streams (if there
\text{-- has been root eviction)}
\]
\[
\text{id = 1}
\]
\[
\text{nm = "istream" .. id}
\]
\[
\text{while streams[nm] do}
\]
\[
\text{streams[nm] = text_comp:apply(streams[nm])}
\]
\[
\text{id = id + 1; nm = "istream" .. id}
\]
\[
\text{end}
\]
\[
\text{-- Compress the trailing bytes}
\]
\[
\text{streams.trailing = text_comp:apply(streams.trailing)}
\]

\(^1\) We might expect that coalescing the streams would result in better compression; this would allow the compressor to exploit patterns among different streams. We do not allow this in our experiments because it forces symbols to be padded, confusing the results.
5.3 Patternization Methods

The following sections present five different methods of patternization. Each is expressed precisely through its Lua naming function; informal descriptions and diagrams are provided to further motivate them.

5.3.1 Homogeneous Sequences

The aim of this representation is to aid the compressor by grouping together values of the same width. Figure 5.2 illustrates this procedure; the rd and rs1 values both have width 5, placed in stream “5;” the simm13 value is also evicted. Below is the Lua-ISC naming function for homogeneous sequences.

```lua
function naming_function(data)
    if NodeData.isLeaf(data) and not NodeData.isRoot(data) then
        return " " .. NodeData.width(data)
    end

    return nil
end
```

The naming function marks each leaf node for eviction; the sequence name is identical to the node’s width.\(^2\)

Constructor trees may have a root but no children (e.g. NOP has no parameters); it is therefore necessary to ensure that a leaf node is not a root before evicting it. No roots are evicted in this experiment.

This method creates the fewest streams of all the methods we test—one for each field width. In general, fewer streams imply longer streams for a particular file, and thus a greater opportunity for compressors to exploit redundancy. There may be more than one field type represented in a sequence, however, as in Figure 5.2. This may make compression more difficult than with a finer-grained naming function, because fields with intrinsically different types of information may be placed in the same stream.

\(^2\)A blank space has been prepended to appease Lua’s type system, however; returning just the width would produce a number instead of a string.
Figure 5.3: “Node name” sequences for the Sparc LDSB instruction.

5.3.2 Node Names

The “Node Names” experiment addresses the potential problems of homogeneous sequences by splitting instruction trees into more streams, based on the names of nodes. All leaf nodes are evicted. The resultant sequences are generally homogeneous (only target addresses are renamed from different-width fields), and will necessarily have grouped related values together. Figure 5.3 exhibits this method on the familiar Sparc LDSB example. The naming function, shown below, returns the name of the visited node for all leaf nodes.

```
(node names 47)≡
  function naming_function(data)
    if NodeData.isLeaf(data) and not NodeData.isRoot(data) then
      return NodeData.name(data)
    end

    return nil
  end
```

Although this method does implicitly transmit more information to the compressor than the above homogeneous sequences method, the split may be too ambitious. In the Sparc instruction set, for example, the fields rs1, rs2 and rd all represent the same type of information but will be placed in different streams. Not only does this prevent the compressor from detecting some patterns in register use, it also produces shorter sequences than a method that groups register numbers together in the same stream. The next experiment, “Node Name Equivalence Classes,” addresses this problem.
5.3.3 Node Name Equivalence Classes

While distinguishing field values based on width may group unlike values, methods based on names may split like values. In particular, the Sparc fields \texttt{cd}, \texttt{rd} and \texttt{rs1} all have the same width, but represent two different types of information; \texttt{cd} is a coprocessor register number, and \texttt{rd} and \texttt{rs1} identify integer registers. \texttt{rd} and \texttt{rs1} themselves contain the same type of information; keeping them in the same sequence may take advantage of register reuse in proximate instructions. The “Node Name Equivalence Class” experiment attempts to balance the features of homogeneous sequences and “node name” sequences against each other by evicting leaf nodes based on equivalence class names. Figure 5.4 illustrates this procedure over the Sparc \texttt{LDC} instruction. In contrast to homogeneous sequences (fig. 5.2, page 46), the \texttt{cd} and \texttt{rs1} values are placed in different streams. \texttt{rs1}, \texttt{rs2} and \texttt{rd} values are stored in the same stream.

The following Lua-ISC function translates from node names to equivalence class names. As shown, the Sparc integer register numbers are grouped into the same stream, and similarly are floating-point register numbers.

```lua
function class_name(node_name)
    -- For the Sparc
    if node_name == "rd" or node_name == "rs1" or node_name == "rs2") then
        return "rd"
    end
    if(node_name == "fs1" or node_name == "fs2" or node_name == "fd") then
        return "fs"
    end
end
```
The Alpha has three register fields of each type as well; they are combined in a similar fashion.

\[
\begin{align*}
\langle \text{equivalence class names} \rangle & + \equiv \\
\langle \text{equivalence class names} \rangle & + \equiv
\end{align*}
\]

\begin{verbatim}
-- For the Alpha
if (nodename == "ra" or nodename == "rb" or
    nodename == "rc") then
    return "ra"
end

if (nodename == "fa" or nodename == "fb" or
    nodename == "fc") then
    return "fa"
end
\end{verbatim}

Finally, the Pentium has three different types of register name fields: one for the 8-, 16- and 32-bit register groups. We combine each of these fields in the same stream.

\[
\begin{align*}
\langle \text{equivalence class names} \rangle & + \equiv \\
\langle \text{equivalence class names} \rangle & + \equiv
\end{align*}
\]

\begin{verbatim}
-- For the Pentium
if (nodename == "r8" or nodename == "r16" or
    nodename == "r32") then
    return "r"
end

return nodename
end
\end{verbatim}

The naming function itself mimics the "node names" experiment, with the \texttt{class.name} translation composed in.

\[
\begin{align*}
\langle \text{class names} \rangle & \equiv \\
\langle \text{class names} \rangle & \equiv
\end{align*}
\]

\begin{verbatim}
function naming_function(data)
    if NodeData.isLeaf(data) and not NodeData.isRoot(data) then
        return class_name(NodeData.name(data))
    end

    return nil
end
\end{verbatim}
5.3.4 Constructor Parameters

The “Constructor Parameters” experiment is the most aggressive in creating output streams. It creates a sequence for each parent-node combination. Although introducing potentially high structural cost for the many sequences it generates, in addition to reducing the length of the inputs to compressors, this experiment will allow a compressor to take advantage of any frequently-occurring instruction-parameter pairs. If load instructions often use a zero immediate offset, for example, the “load immediate” stream will be very amenable to compression.

The following naming function describes the experiment. All leaf nodes are evicted; a node’s parent name is concatenated with its own name. Figure 5.5 shows patternization on the familiar LDSB example. Subsequent non-LDSB instructions with rd, rs1 and simm13 fields will place their values in different streams.

```plaintext
〈parameter names 50〉 =
    function naming_function(data)
        if NodeData.isLeaf(data) and not NodeData.isRoot(data) then
            return(NodeData.parentName(data) .. " " .. NodeData.name(data))
        end
        return nil
    end
```
5.3.5 Parent Names

The “Parent Names” experiment groups all the children of a particular parent together, in their own stream. Figure 5.6 shows patternization using this method. This experiment tries to exploit associations between constructors and the values of their operands; if the ADD instruction prefers certain registers, for example, this method will perform well. Because it depends on root name only, it is not appropriate for field trees where the root is always the same.

This is the only experiment that expands values when placing them in their target sequences; for instructions with a large disparity in field width, such as LDSB, much space is wasted.

The naming function is as one would expect. It names all leaf nodes, returning the name of the parent node.

```
51 (parent names 51) ≡

function naming_function(data)
  if NodeData.isAdminLeaf(data) and not NodeData.isRoot(data) then
    return NodeData.parentName(data)
  end

  return nil
end
```
Table 5.1: Alpha ANOVA

<table>
<thead>
<tr>
<th></th>
<th>DF</th>
<th>Sum of squares</th>
<th>Mean of squares</th>
<th>F value</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeRep</td>
<td>1</td>
<td>0.05821</td>
<td>0.05821</td>
<td>46.8071</td>
<td>5.129*10^{-5}</td>
</tr>
<tr>
<td>Compressor</td>
<td>1</td>
<td>0.01466</td>
<td>0.01466</td>
<td>4.2833</td>
<td>0.04039</td>
</tr>
<tr>
<td>FileName</td>
<td>16</td>
<td>1.67461</td>
<td>0.10466</td>
<td>30.2228</td>
<td>&lt;2.2*10^{-16}</td>
</tr>
<tr>
<td>Residuals</td>
<td>355</td>
<td>1.22938</td>
<td>0.00346</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Pentium ANOVA

<table>
<thead>
<tr>
<th></th>
<th>DF</th>
<th>Sum of squares</th>
<th>Mean of squares</th>
<th>F value</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeRep</td>
<td>1</td>
<td>2.434</td>
<td>2.434</td>
<td>264.977</td>
<td>&lt;2*10^{-16}</td>
</tr>
<tr>
<td>Compressor</td>
<td>1</td>
<td>0.0061</td>
<td>0.0061</td>
<td>0.660</td>
<td>0.417</td>
</tr>
<tr>
<td>FileName</td>
<td>16</td>
<td>2.877</td>
<td>0.180</td>
<td>19.580</td>
<td>&lt;2*10^{-16}</td>
</tr>
<tr>
<td>Residuals</td>
<td>355</td>
<td>3.261</td>
<td>0.0092</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.4 Results

Each experiment is run over five factors: tree representation, patternization method, text compressor, architecture and test program. To aid analysis, we wish to remove any factors that do not contribute to variation in the response (bit-per-bit measurements). For this, we apply an Analysis of Variance (ANOVA) calculation. The ANOVA model presumes that its factors are independent; the null hypothesis is that each factor does not contribute to variation in the response.

The tree representation, patternization method and architecture are dependent factors. The tree generator depends on the architecture type, and the result of patternization itself will vary based on the shape and nature of its input trees. We therefore fix the architecture factor, and compute the ANOVA table for the independent factors and the tree factor only. The results are shown in Tables 5.1, 5.2 and 5.3.

The statistic of primary interest is the p-value of each factor. The p-value indicates the probability of seeing the results we observe given that the null hypothesis is correct. A low p-value rejects the null hypothesis, indicating that its factor contributes to overall compression performance.

For each architecture, the compressor factor does not contribute to variation in the response. The tree representation and file name, however, contribute significantly. Replacing the file name factor with the text segment size for each file, a subsequent ANOVA calculation (not shown) indicates that text segment size is also a source of variation ($F=28.7$, $p<2.2*10^{-16}$).

5.4.1 Compression Per Architecture

Figure 5.7 (page 57) shows the bit-per-bit compression results for the Alpha, Sparc and Pentium. Better compression performance is indicated by lower bars. Corpus files have been sorted left-to-right by text segment size (on the Pentium). The “compress95” test has been omitted from the graphs; its compression
Table 5.3: Sparc ANOVA

<table>
<thead>
<tr>
<th>Experiment Name</th>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte compressor</td>
<td>gzip</td>
<td>Gzip applies directly to the instruction stream. This experiment is not</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dependent on tree representation or patternization.</td>
</tr>
<tr>
<td>Parent names</td>
<td>cname</td>
<td>Stores all the operands for a particular constructor in a single stream.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This experiment is not performed on field trees; because they all have the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>same “dummy” constructor.</td>
</tr>
<tr>
<td>Constructor parameters</td>
<td>cparam</td>
<td>Creates a stream for every constructor-operand name combination. For field</td>
</tr>
<tr>
<td></td>
<td></td>
<td>trees, this is equivalent to the node names experiment.</td>
</tr>
<tr>
<td>Equivalence class</td>
<td>eqclass</td>
<td>Creates a stream for each node equivalence class. For the Pentium, this is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>equivalent to the node names experiment.</td>
</tr>
<tr>
<td>Homogeneous sequences</td>
<td>homogeneous</td>
<td>Creates a stream for each field (or operand) name.</td>
</tr>
<tr>
<td>Node names</td>
<td>nodename</td>
<td>Creates a stream for each field (or operand) name.</td>
</tr>
</tbody>
</table>

Table 5.4: Table of Experiments

performance is very poor due to its small size (approximately 5 kilobytes). Bits per bit measurements relate to the text segment only; end-to-end compression results are presented separately later in the chapter. All measurements have been made both with gzip and our implementation of the Burrows-Wheeler transform; their averages are presented. Table 5.4 lists the experiments. Figure 5.8 presents these results again, but groups results by file rather than by experiment.

In each figure, blue bars indicate measurements for field trees, and green bars compressor trees. Shorter bars are shown in the foreground, and in the case of a tie (the gzip experiments only, which are independent of tree representation) the blue bar is shown in front. The “cname” experiment is not applicable to field trees; data for “cname” field trees is not shown.

5.4.2 Fields and Constructor Trees

Field trees beat compressor trees heavily on the Pentium; on the Sparc and Alpha, with a few exceptions, field trees also perform better. One potential source of this disparity is the representation of skeleton trees. While field trees with
evicted opcodes can effectively toss out the skeleton tree stream, constructor trees must store the full tree around, including unused bits. Although the Sparc has 252 constructors, its skeleton tree representation consumes 32 bits.

To test the influence of skeleton trees, we run a second batch of experiments in which the constructor skeleton trees are replaced with integer identifiers. The results are shown in Figure 5.9. Not surprisingly, constructor trees perform better under these conditions. Some tests on the Pentium reduce compressed text segment size by as much as 0.08 bits per bit; on the Alpha and Sparc the change is not as significant, but on the Alpha it is enough to generally outperform field trees in the “nodename” and “homogeneous” experiments, and for larger files in “cparam.” These results are closer to the true potential of constructor trees; the skeleton trees can be safely thrown out, because the dictionary of potential skeleton trees (when all leaf nodes are evicted) does not change per file, only per architecture. This information, therefore, does not need to be encoded in each file. In general, however, one cannot tell a priori what the behavior of the naming function will be; for this reason, we do not consider the results shown in Figure 5.9 to be the official performance of constructor trees, but an estimate of the potential performance.

On the Pentium in particular, and to some degree on the other architectures the field trees still outperform the constructor trees. This tends to imply that more fine-grained division of the instruction leads to greater compression; constructor trees coalesce these fields implicitly within the root node, keeping them away from patternization. Referring to Figures 5.11, 5.13 and 5.12, we note that a significant percentage of fields not included in the constructor trees (3 of 13 on the Pentium; 6 of 21 on the Alpha; and 7 of 21 on the Sparc). But contrary to our expectation, the Pentium has the fewest implicit fields but the greatest distance between constructor and field trees. Discussion of this difficulty is deferred to our consideration of homogeneous sequences on the Pentium.

5.4.3 Equivalence Class and Homogeneous Sequences

On the Sparc and Alpha architectures, equivalence class performance beats homogeneous sequence and node name performance for both constructor and field trees. This confirms our expectation; equivalence classes merge like information, reducing the number of streams while still segmenting unlike information.

The parent names (“cparam”) and node names experiments produce the same results for field trees. This is because the parent name of any node in a field tree is always “root;” prepending it in the naming function does not change the mapping from nodes to streams. The parent name method does not compete (for constructor trees) on the Pentium or Sparc; the high number of streams, especially for the Pentium, incurs a considerable cost. On the Alpha, however, “cparam” performance is better than “nodename” performance for larger files. And in general, we observe more competitive numbers for “cparam” on the Pentium and Sparc as files become larger. This tends to indicate that the stream-splitting overhead is the primary contributor to performance degradation, not the nature of the algorithm
itself. The superior performance on Alpha indicates that a coupling exists between instructions and register values, at least on Alpha, that the compressor can exploit despite the higher number of streams. It would be interesting to test this theory on files of significantly greater size.

On the Pentium, the homogeneous sequences experiment with field trees outperforms the others for field trees, including the “eqclass” experiment. But unlike on the Alpha and Sparc, where constructor tree performance follows field tree performance, the Pentium constructor trees for homogeneous sequences perform less well than the equivalence class and node name experiments. To understand this behavior, we refer to the Pentium field figure (fig. 5.11, page 61). The performance degradation for constructor trees implies that the combination performed in creating homogeneous sequences is not effective for those fields that appear in the constructor trees. These are the r8, r16 and r32 fields, the register specifiers for the 8-bit, 16-bit and 32-bit register banks respectively. This is bolstered by the results for the field tree “eqclass” experiment. The only difference between the “eqclass” and “nodenames” experiments is that the former combines the aforementioned register fields; we note that “eqclass” performance is worse than “nodenames” for fields. We conclude that it is not meaningful to treat the 8-, 16- and 32-bit registers as similar in the compressor; this is in contrast to the register fields on the Alpha and Sparc, which when combined produce better compression.

The surprising result is that homogeneous sequences improve performance for field trees on the Pentium. This improvement must necessarily come from combining fields that appear in the field tree, but not in the constructor tree. Figure 5.11 shows the 3-bit values reg/opcode, r/m, index and base, and the 2-bit values mod and ss as candidates. The values of the 16-bit registers base and offset are summed to create a 32-bit address; frequently-occurring combinations of register pairs will lead to better compression. The reg/opcode and r/m fields are also combined by the homogeneous sequences method; they are used together in specifying 16- or 32-bit addresses. Again, if combinations of registers are used frequently for addressing calculations, combining these fields will aid the compressor.

This may begin to explain the disparity between compressor and field trees on the Pentium. The only data available to the latter, but unavailable to the former, are the values in the mod and r/m fields (as well as the opcode, of course). If these fields do indeed represent a significant opportunity for compression, constructor trees will, on the whole, perform worse than field trees.

5.4.4 Constructor Names and Padded Bits

The results of the “cname” experiment make an interesting statement about wasted bits in streams. This constructor name experiment combines all the children of a constructor into a single stream; each constructor has its own stream. For field trees with wide fields (a Pentium instruction with a 32-bit immediate for example), a significant amount of space is wasted; “thinner” values are padded to fit into the same stream. On the Sparc, the “cname” experiment beats the parent
names ("cparam") experiment, and on the Pentium it performs comparably. This indicates either that the extra bits are handled well by the compressor, or that the "cename" method effectively offsets the overhead by providing more opportunity for the compressor. The latter case would imply frequent association between a particular constructor and operand value. From our data we are not able to conclude that either is the case. The work done in replacing skeleton trees with indices tends to indicate that unused bits are not handled well by the compressor, at least in the instruction stream.

5.4.5 The Influence of Raw Data

The bits-per-bit devoted to "illegal instructions" on the Alpha is shown within Figures 5.7 and 5.8 in an attempt to explain the consistent "skyscraper" patterns across experiments. These patterns tend to imply that something about the files themselves affects compression more than the compression method. Illegal instructions, a piece of raw data in the instruction stream that cannot be interpreted as an instruction, are a potential source because they are unaffected by the representation method. The small graphs show that the illegal instructions cannot create the skyscraper patterns we observe; the patterns do not match, and the magnitude of the illegal instruction frequency is not sufficient.
Figure 5.7: Compression performance on each architecture. Blue bars indicate field tree numbers; green are compressor trees. Files in the corpus are sorted left-to-right based on text segment size. "compress95" is omitted.
Figure 5.8: Compression performance on each architecture. Blue bars indicate field tree numbers; green are compressor trees. Files in the corpus are sorted left-to-right based on text segment size; the compressors appear in the order, [gzip, cname, cparam, homogeneous, eqclass, nodename]. “compress95” is omitted.
Figure 5.9: Compression performance without skeleton trees in constructor tree experiments. Files in the corpus are sorted left-to-right based on text segment size. “compress95” is omitted.
Figure 5.10: End-to-end compression results. “apsi” is off the scale; it compresses to 1% of its original size. Blue bars indicate field tree numbers; green are compressor trees. “compress95” is omitted.
Figure 5.11: Pentium instruction fields. Fields that do not appear in constructor trees are shaded.

Figure 5.12: Alpha instruction fields. Fields that do not appear in constructor trees are shaded.
Figure 5.13: Sparc instruction fields. Fields that do not appear in constructor trees are shaded.
Chapter 6

Applications

In practice, instruction-stream compression algorithms must adhere to the constraints of a particular application. Some systems keep instructions compressed, in memory, until they are needed by the processor [Lucco 2000; Fraser and Proebsting 1995]; in this case, the methods we discussed in Chapter 5 do not apply, because they require that the full instruction stream be decoded at once. Others strive to maximize compression at the expense of decompression speed, to produce compact “wire” formats that can be easily transmitted and stored [Ernst, Evans, Fraser, Lucco, and Proebsting 1997].

In this chapter, we discuss these two poles in instruction-stream compression applications. We present an existing and a new algorithm in each area, and introduce additions to Lua-ISC that support them. This chapter is meant to illustrate the use of Lua-ISC in more varied compression contexts, and to introduce preliminary work in designing and evaluating new algorithms.

6.1 Supporting Elements in Lua-ISC

To facilitate the algorithms below, support for ELF files, suffix trees, and flat profiles is added to Lua-ISC. Each datatype is described in detail in Appendix B. One algorithm requires basic-block information as well; the outline for this feature appears in Chapter 7.

Of primary importance to the below, and a number of additional compression algorithms, is the step form of patternization. The step form provides an additional interface to patternization, through which each instruction is patternized one-at-a-time. In Chapter 3 (page 25), we proved that patternizing a single instruction is reversible, given some constraints on the state of the sequence tables. These constraints must be respected when using the step form of patternization, available in Lua-ISC as the patternize_start, patternize_step and patternize_end functions. A detailed description of these functions appears in Appendix B. The step functions allow programs in Lua-ISC to observe and modify the sequence tables during patternization. They are often used to associate evicted data with the
instruction from which it was evicted. For example, if an algorithm wishes to inject modified branch targets into an instruction stream at run-time (many compression algorithms move instructions around), it can use step patternization to store the location of branches and their values for later processing. The below code is a simple example of this.

\begin{verbatim}
64a \langle get branch targets 64a\rangle \equiv
    function naming_function(nodedata)
        if NodeData.name(nodedata) == "target" then
            return "target"
        end
        return nil
    end

64b \langle get branch targets 64a\rangle \equiv
    function getbranchtargets(elfdata, machdesc)
        local branchtargets = {}
        local seetable = {}
        local nbt = 0
        local pos = 0
        ElfData.patternize_start(elfdata, machdesc, naming_function)
\end{verbatim}

The naming function is presented for completeness. It returns the string “target” for any node of the same name. This function will not work appropriately for field trees as defined; only constructor trees exhibit that node name. We ignore this and other implementation issues to simplify this example.
The first step is to initialize patternization with the naming function, machine description and ELF data segment. This information is kept within the underlying machinery; the only data that can be modified during patternization is the sequence table.

```plaintext
|get branch targets 64a|+≡<64b
while pos do
    pos, seqtable = ElfData.patternize_step(seqtable)
    -- Any targets are stored in the stream "target" by
    -- the naming function
    if seqtable.target and KSequence.length(seqtable.target) > 0 then
        -- Store the branch target in its own table
        branchtargets[nbt] = {}
        branchtargets[nbt].loc = pos
        branchtargets[nbt].val = KSequence.nth(seqtable.target, 0)
        nbt = nbt + 1
    end
end

return branchtargets
end
```

The rest of the function builds a table (branchtargets) from all the “target” nodes encountered. Each entry is comprised of a location and a k-symbol (loc and val). The Appendix provides more information on patternize_step and the patternize_end function (not shown).
6.2 Wire Form Compression

The term “wire form” is adopted from Ernst et al. [Ernst, Evans, Fraser, Lucco, and Proebsting 1997]; a wire form is intended for transmission over a network, or potentially for storage in an archive. The decompressor is given a relatively large amount of freedom. Decompression may be a preprocess to execution—the operating system’s loader might decompress the file into memory. Or it might be an independent procedure altogether—a computer user might decompress the file after he has downloaded it from the Internet.

We consider two wire form algorithms in this chapter. The first is taken from Ernst et al. [Ernst, Evans, Fraser, Lucco, and Proebsting 1997]; it uses patternization to aid a simple dictionary, replacing common 2- to 4-word subsequences in the instruction stream. The second is our own method for detecting shared sequences among different programs.

6.2.1 Ernst Wire Form

The Ernst wire form attempts to achieve better compression than previous techniques. It uses patternization to remove operand values from the instruction stream (representing instructions as constructor trees), and then compresses the resultant streams. In this sense it is similar to our experimental compressors in Chapter 5. The general algorithm is outlined below; the Lua-ISC expression is sketched in Figure 6.1.

The algorithm first performs patternization to remove operand values. Its patternization procedure is nearly identical to our “nodenames” experiment on constructor trees, except that Ernst uses the lcc intermediate representation as the source of his trees.\(^1\) Values are evicted into streams based on their names in the tree only. Subsequently, each resultant stream (data streams and the skeleton tree stream) is pushed through a move-to-front coder (with an explicit dictionary); the MTF indices are encoded using Huffman trees, but the MTF dictionaries are not modified. Finally, everything is passed through gzip. The Huffman operation in Figure 6.1 is starred because, in Ernst’s algorithm, it expands its output so that it

\(^1\) A full description of the lcc IR can be found in [Fraser and Hanson 1995].
will be accessible by gzip. In general, it is not entirely sensible to apply an entropy coder before the final step in the procedure; because it outputs symbols of different widths, subsequent compressors have a very difficult time interpreting the data. Ernst tries to get around this by padding Huffman symbols to fill full bytes. It is not clear from [Ernst, Evans, Fraser, Lucco, and Proebsting 1997] why the Huffman step is included.

The Lua-ISC expression appears in Appendix C.3.1. It relies on basic components we’ve seen before; an MTF coder, Huffman trees, and gzip.

6.2.2 Inter-Program Sequences

Applications using Microsoft’s Foundation Classes (MFC) for user-interface (and sometimes even system) functions must resolve MFC references either statically or dynamically. The later versions of their Windows operating system ship with compatible libraries; some older versions, and some stripped-down installations, do not. It is common practice in publication to statically link these libraries instead; programs will necessarily resolve all their references, but pay an overhead of over 100 kilobytes, sometimes significantly more. Static linking is used on other platforms as well for the same reason.

The statically-linked portions of the instruction stream are mostly identical from one application to another, excepting jump targets and potentially related fields that can be removed using patternization. This offers an opportunity to remove the static-link overhead; representing these sequences once over many programs or potentially an entire system could have a large impact not only in network transfer times, but in hard disk consumption. In practice, eliminating static linkage overhead may be a per-computer operation; the system can detect common sequences over all the local programs, and the loader can patch them when the program is executed.

The general algorithm for inter-program sequence detection is divided into two main phases. The first, sequence generation, creates candidate sequences for the final sequence dictionary. Sequence generation requires $O(n)$ space and time in the length of the input applications. The second, sequence refinement, shortens candidate sequences or removes them from consideration. It is also in $O(n)$.

Sequence generation begins with a pair of instruction streams. Each stream is preprocessed to remove application-specific modifications to the statically-linked code. The Ukkonen suffix tree [Ukkonen 1995] of the first is constructed; the second stream is traversed, finding longest-matching sequences in the first. This procedure is nearly identical to Lempel-Ziv ’78 [Ziv and Lempel 1978], which matches symbols in a sequence with others in a sliding-window buffer. If a match is longer than the candidate length cut-off, the match is stored as a candidate and the transversal jumps past the match in the second sequence. This avoids needless duplication of candidate sequences.

2This measurement is taken from Microsoft’s VC6 product, compiling some simple MFC programs.
The suffix tree is built in \( O(n) \) time and consumes \( O(n) \) space (where \( n \) is the length of the first stream); the traversal is accomplished in \( O(m \times p) \) time, where \( m \) is the length of the other sequence and \( p \) is the length of the longest match. The real time consumption is expected to be much less; ignoring skips at candidate matches, the expected time consumption is in \( O(m \times a) \), where \( a \) is the average match length over all the symbols in the second sequence.

All the matches with length above a given threshold are retained from the sequence generation step. The sequence refinement step uses new input files to test these sequences. It first creates a suffix tree from its input refinement instruction stream, also patternized. The refinement instruction stream can come from any program except those used in the generation step. The suffix tree is queried for each candidate sequence; the longest subsequence of the candidate sequence matching part (or all) of the refinement instruction stream is passed on to the next round. The cost analysis for the refinement step is more complicated than illuminating; we conjecture that, in practice, this operation will be fast because sequences will probably match, certainly after initial rounds and partly because generation gives them a good chance.

The Lua-ISC implementation of this algorithm appears in Appendix C.3.2.

6.3 Random-Access Algorithms

Random-access algorithms fall on the opposing side of instruction stream compression; they must generally support decompression of instructions as needed. The delivery mechanism is also more complex. Some methods use virtual machines, decompressing an intermediary machine language [Ernst, Evans, Fraser, Lucco, and Proebsting 1997; Fraser and Proebsting 1995; Hoogerbrugge, Augusteijn, Trum, and Wiel 1999] or other forms [Lucco 2000]; others execute code natively, handling data requirements as the program runs [Kirovski, Kin, and Mangione-Smith 1997; Wolfe and Chanin 1992].

We consider two random-access algorithms. The first, introduced by Lucco [Lucco 2000], creates a dictionary for the instruction stream. Portions of the dictionary are aggressively compressed; given the uncompressed dictionary and the indices stream, original instructions can be decoded in any order. The second algorithm is our own, a block decompressor for instruction streams. It is motivated primarily by work in program profiling; if we can construct blocks of instructions that, with high probability, are either completely used in program execution or not used at all, we do not incur additional run-time overhead in decompression them as single units. At the same time, processing groups of instructions allows state-based compression techniques, which can produce more compact results.

6.3.1 Split-Stream Dictionary Compression

Split-Stream Dictionary Compression or SSD [Lucco 2000], replaces instructions and frequently-used instruction sequences with indices into a dictionary.
The dictionary is created directly from the instruction stream; parts of it are compressed to produce an even more compact form. When the dictionary is not compressed, the instruction stream can be reconstructed one instruction at a time, in any order. A sketch of SSD is shown in Figure 6.2.

SSD first separates branch targets from the instruction stream. PC-relative branch targets are not included in the dictionary; they are supplied manually when the stream is decompressed. We use the step form of patternization to accomplish this; the earlier branch target example illustrates how patternization can be used, in its step form, to accomplish exactly this operation.

After the branch targets have been removed, the remaining stream (identical to the original instructions stream, except with “uninitialized” values in the branch target operands) is used to create two hashes, identified as “H-i” and “H-d” in Figure 6.2. “H-i” contains all the individual instructions observed in the stream; “H-d” contains all the digrams that appear more than once. From these, two dictionaries are created; “D-i” is identical to “H-i,” associating individual instructions with indices. “D-s” contains sequences of length 4 or fewer that are repeated in the stream. Conceptually, the “Make dict” box creates its dictionaries by passing over the instruction stream; each time it sees a repeated instruction or sequence of instructions, it adds it to the dictionary. In SSD’s specification, it is implemented as shown in Figure 6.2, using the hash tables to improve speed performance.
Finally, the dictionaries are reordered to aid compression. The individual instruction dictionary (“D-i”) is sorted based on opcode first; within opcode groups, instructions are sorted based on immediate fields. After the sorting, the instructions are serialized within an opcode group—the immediate fields are listed, then the register fields, and so on. The opcode is emitted only once; it is the same for each opcode group. We accomplish this procedure in Lua-ISC by again appealing to step patternization. Figure 6.3 shows step patternization applied to the “D-i” stream; it splits its input into three streams—for opcode, immediate and everything else—and passes them on to the sort box. The sorting procedure also uses step patternization to pull out each field into its own stream. The sequence dictionary is compressed by merging sequences with like prefixes. [Lucco 2000] provides more detail.

The Lua-ISC implementation, as shown in Figure 6.2, is missing one important piece. SSD does not allow instruction sequences to be replaced when the span basic-block boundaries; this would violate the random-access decompression property of the algorithm. The dictionary construction blocks must have access to basic-block information, which is currently unavailable in Lua-ISC. This extension is mentioned in the Future Directions chapter.

### 6.3.2 Block Decompression

The goal of the block decomposition algorithm is to allow compressors access to reasonably long sequences of symbols, while generally preserving low-cost random-access to the instruction stream. We utilize profiling data to construct blocks of related use frequency; instructions accessed heavily are grouped together so that, when one is accessed and all are decompressed, the others will be used with high probability. Similarly, untouched instructions remain heavily compressed in memory, reducing the footprint of the application.

The heuristic used for assembling instructions into blocks most greatly affects the algorithm’s performance. The implementation framework exhibited here is not dependent on a particular heuristic; rather, it is intended as a general platform for evaluation of potential ones. A naive heuristic is presented in Section 6.3.2.

The compression algorithm has, as its inputs, the original instruction stream and the mapping from stream locations to blocks, provided by some external method. Primitive operations are provided Lua-ISC for converting these objects into a table of instruction blocks; any flavor of compressor can then be applied to each. On the execution side, the JIT translator detects requests for instructions that have not yet been decompressed; it translates the full block in which the de-
sired instruction resides, stores it in a cache, and continues execution until another unseen instruction is required. The structure of the JIT translator is presented in Chapter 7. Figures 6.4 and 6.5 illustrate the block compressor algorithm.

The Lua-ISC code for block decompression is included in Appendix C.3.3.

Assembling Blocks

Block construction is one of the most difficult and interesting aspects of this algorithm. In order to avoid digression, we present only a naïve methodology, one which uses flat profiling information to group functions and their constituent basic blocks together by frequency.

The Lua extension GProfData basic type provides support for flat profiling data (see Appendix B.3). From this basis, profile function entries can be coalesced to form blocks. One could arguably ignore coalescing; each function could represent a single block. The hit-rate for blocks would intuitively be maximal, because the flat profile is too coarse to allow blocks of any finer granularity. Alternatively, one might expect that coalescing functions would impact compression performance (and translation overhead) by reducing the size of the instruction-block mapping, and at least initially the number of compressor invocations. Larger blocks always assist the compressor as well, and the cost of decompressing a single instruction generally goes down as the number of instructions processed at a time goes up. Without attempting any resolution to these issues, we present the infrastructure for function coalescing. In practice it can be used or ignored.

The Appendix contains much supporting code for translating from the GProfData table format to the format expected by the k-sequence shuffle function (and therefore block_compress). Function coalescing is done in the shuffle format, because it supports block descriptions which contain non-contiguous code sequences. The shuffle format is slightly augmented to retain profiling information; this allows the coalescing function the ability to evaluate blocks appropriately. The
shuffle format is also highly amenable to this operation; one must simply equate block identifiers to group blocks together. The below function contains much of the machinery for block formation through coalescing; it performs all possible pairwise checks, applying a heuristic function to decide when to coalesce.

```plaintext
(function coalescing 72) ≡

function coalesce_functions(profiledata, heuristic)
    local i, j
    i = 0
    while profiledata[i] do
        j = i + 1
        while profiledata[j] do

            -- Decide whether or not to coalesce these blocks
            if (heuristic(profiledata[i].calls, profiledata[j].calls)) then

                -- Merge these blocks. Always pick the block id
                -- of the lower item (i).
                profiledata[j].id = profiledata[i].id
            end

        j = j + 1
    end
    i = i + 1
end

return profiledata
end
```
Chapter 7

Future Directions

We are interested primarily in three avenues of future work.

- *Embedding* Lua-ISC in different run-time environments. This will allow Lua-ISC to express JIT translators, for example. We discuss two extensions in Section 7.1.

- *Extending* Lua-ISC over new datatypes. The following are necessary for our near-future experimental work.
  - Basic block information for instruction streams. Required by SSD and other random-access algorithms.
  - COFF support. A significant number of Pentium machines run Windows, which uses Microsoft’s PECOFF format. We would like to provide support for Windows binary compression to sample this segment of computers. Windows applications can be quite large, representing new types of opportunities in compression. Our inter-program sequence algorithm is one example.
  - Finer granularity in profiling. We currently offer support for flat profiles of functions; we would like to offer phase profiles in addition, and wish to support basic block usage statistics as well.

- *Experiments* in applications, made possible by the above items. In the near-term, we are interested in expressing the algorithms in the field within the Lua-ISC framework, and evaluating them over different architectures and instruction representations. This will provide a comprehensive view of compression techniques, long the staple of text compression literature, which is conspicuously lacking from the instruction-stream compression literature. Over a longer term, we are interested in using our experimental knowledge to design better compressors both for wire and random-access applications. The block compressor alone is a potentially deep area of research. There is much work to be done in algorithm design and evaluation.
7.1 Application Environments

The Lua language was designed to allow easy embedding within larger applications. Our infrastructure can be moved, as a single unit, into an enclosing application that could use it to compress or decompress streams, for example. This section presents two such applications: a program loader and a JIT translator. The structure diagrams for the loader and translator are shown in Figure 7.1.

A structure diagram describes both the control and data flow of a particular piece of code. Data flow is indicated by a thick arrow line, and is typed; the type is shown in a tabbed box along the data flow line. Control flow where no data is passed is depicted by a standard arrow line. Control flow from any particular point can be traced by following any control flow lines first; if there are none, control flows along the shortest data flow edge. The environment of a structure within a

---

1The shortest edge is defined as the edge to the node of least depth. The depth of a node is the length of the longest path to that node in the structure diagram.
structure diagram is the set of all data flow edges into and out of the structure. The environment is equivalent to the type of the structure.

### 7.1.1 A Modified Program Loader

Some algorithms are designed to heavily compress programs on disk, but must decompress the programs all at once. When it is executed in memory, the program must be present in its native form. Support for these types of algorithms can be included directly within the operating system by modifying the system’s loader. Figure 7.1 shows a structure diagram for the loader’s new decompression preprocess.

The environment of the Lua-ISC infrastructure is “ELF -> ELF.” This is nearly identical to the type of the experiment functions in Chapter 6; the loader would simply invoke the enclosed Lua-ISC function over `ElfData` objects it supplies.

Lua-ISC’s environment is, in fact, equivalent to the preprocess’ environment. This indicates that the implementation overhead consists of translating from the loader’s data structures into `ElfData` objects, and back.

### 7.1.2 A JIT Translator

The JIT translator’s structure is shown in Figure 7.1; it is certainly more complicated than the loader’s. The structure shown has environment “kseq. dictionary * instr. loc. -> instr.;” presumably, the full translator reads the k-sequence dictionary when the program begins, and uses the displayed structure to get each instruction as it is needed. The Lua box’s environment is general enough to allow for a number of different internal algorithms. The box returns a table of k-sequences and associated locations; it could simply return anything from a singleton sequence and the requested location, to a block of sequences drawn from different parts of the instruction stream. The instructions it returns are stored in the cache at least in part; the Lua box is bypassed if the cache already contains the desired instruction.

The JIT translator would provide opportunities for research in cache management. Certainly the behavior of the Lua box’s algorithm poses new questions to the cache strategy; the relationship between the granularity of decompressed blocks, the size and management of the cache would be interesting.

Although the Lua box structure is not implemented with efficiency in mind, the overall platform can be used to answer speed performance questions. The overhead of the Lua box can be subtracted to leave only emergent cache behavior as an influence on performance. This behavior is comprised of the block granularity, block coalescing heuristic, and specific caching strategy; evaluation of different overall management strategies with respect to speed and memory consumption is a potentially wide avenue of future research.
Chapter 8

Conclusion

As a general tool for instruction-stream compression, our Lua-ISC language makes algorithm specification precise. We have seen that patternization as a single operation is the basis for a number of wire form and related aggressive compression techniques. Additionally, the stepwise form of patternization can be used more generally to remove specific fields or types of fields from an instruction. Algorithms that are built on these pieces may rely on their proven properties. Arguably, the space of potential transforms over streams and trees is more navigable given these tools; Theorem 1’s conditions can be tested on any patternization-based algorithm to verify its reversibility. Our algorithm for reconstructing skeletons for field trees, for example, may not be obviously correct. Applying Theorem 1 is an easy way to deduce its correctness, and also to intuit it.

Implementing an algorithm that compresses instruction trees, beginning with an ELF binary file, is not entirely trivial. One must parse the file format, handle raw data in the text segment, decode and encode instructions for a particular architecture, manage unused bits in the instruction patterns, and implement text compressors on top. Lua-ISC provides this infrastructure with the flexibility of machine-independence. It is our hope that it may be used for future work in the field.

We have also made potentially valuable observations of compression performance over different representations and architectures. It can be argued that constructor trees and related structures are viewed as superior starting points for compression. The more aggressive compressors of which we are aware all use higher-level representations, as opposed to relatively low-level field trees. Our experimental results suggest that field trees are in fact a more useful point of origin, at least for the compression and patternization methods we employ. In the least, our work suggests that the choice of representation and the arrangement of data affects compressor performance, much more so than the choice of back-end text compressor does. It may be worthwhile to reexamine the choices made in algorithms like BRISC, SSD and the Ernst wire form [Ernst, Evans, Fraser, Lucco, and Proebsting 1997; Lucco 2000]; changing the representation domain may improve their performance.

It is ultimately most appropriate to conclude, however, by citing what this
thesis *does not* accomplish. As a result of this work, the potential avenues of research into algorithms, and at a lower level still instruction representation, have expanded beyond our immediate ability to explore them. The seemingly incongruous bits and pieces of infrastructure, most of which having been relegated to the appendices, hint at a larger effort to walk into the space of instruction-stream compression. The images we can see today are puzzling and intriguing. We are very excited to extend this work in the manifold “future directions,” to see what lies beyond them.
Appendix A

Extensions to Patternization

A.1 Evicting Entire Trees–Naming the Root

Up to this point, patternization has been presented as skipping the root node in the preorder walk. It may be useful, however, to allow the naming function to indicate the root node for eviction. We do in fact use this procedure in Chapter 5 to split raw data and skeleton trees into separate streams, to measure the impact of skeleton tree inclusion.

Root eviction is accomplished entirely by a postprocess on the skeleton tree sequence.

Figure A.1 illustrates this procedure. When the naming function indicates that a root node should be evicted, the “instruction stream” output of patternization takes on a new meaning. Rather than containing skeleton trees, the instruction stream contains pointers to the sequences that contain the skeleton trees. The pointers are integers that correspond with the data stream number, shown as (ostream-n).

Because patternization is general with respect to naming functions, it is necessary to support functions which do not choose all root nodes for eviction. If any root node is chosen, it is necessary to switch the meaning of the instruction stream output. In the case where some but not all roots are evicted, the evicted roots are placed in their appropriate streams and the non-evicted roots are stored in a special stream together.

Figure A.2 illustrates the output of patternization where some, but not all, root nodes are evicted.

A.2 Step Patternization

Step patternization gives Lua-ISC programs access to the sequence table during patternization, by processing the instruction stream one instruction at a time. Chapter 6 uses step patternization to store the locations of evicted data. This is a recurring feature in instruction-stream compression algorithms; step patternization ultimately allows individual fields to be accessed in an instruction.
Figure A.1: Splitting the instruction stream into multiple streams based on instruction width.

Figure A.2: Eviction within evicted instructions.

This increased power comes with a caution, however; we must obey the conditions in Theorem 1 (page 28) to maintain the reversibility property of the entire procedure.

The Lua-ISC interface to step patternization is shown below.

```
(patternize_start, patternize_step, patternize_end)

(patternize_start : ElfData * MachineDescription * (NodeData -> string option) -> unit
(patternize_step : number * sequence table -> number * sequence table
(patternize_end : unit -> sequence table

unpatternize_start : sequence table * MachineDescription *
(NodeData -> string option) -> ElfData

unpatternize_step : number * sequence table -> #n symbol * sequence table
```

The forward patternization function, patternize, is split into the patternize_start, patternize_step and patternize_end functions. When patternize_step is first called, it must be given a table that neither it nor its related functions create. One can supply an empty table, but any table of sequences will do. patternize_step processes the instruction at the supplied location, and returns the new k-sequence table with the next instruction location.
The table used and returned by `patternize_step` is not the same in structure as that returned by `patternize`. For the latter, streams are named ordinarily; the actual names returned by the naming function are stripped out. For ease of access, `patternize_step` preserves the names in the table. For example, if a naming function returns the string “rd1,” the resultant stream can be accessed through `table.rd1`, where `table` is the result of `patternize_step`. The `patternize_end` function simply converts the table from the intermediary format, where names are preserved, to the standard `patternize` format. If the sequence tables have not been modified, the result of `patternize_end` can be safely passed to the standard reverse function `unpatternize`.

The reverse patternization step process is somewhat different, highlighting the new ability in step patternization. With this procedure, we can access any part of the instruction stream we wish by supplying the appropriate location to `unpatternize_step`. It is the programmer’s responsibility, however, to supply the appropriate sequence table. Theorem 1 describes the requirements for the sequence table. In short, it must contain the values to be inserted at the front of its sequences.

The `unpatternize_start` function sets up internal state for the computation much in the same way that `patternize_start` does for the forward process.

An example of step patternization is shown in Chapter 6, Section 6.1.
Appendix B

Additional Elements of Compression

B.1 Suffix Trees

Suffix trees support fast searching over strings. The Ukkonen algorithm for on-line suffix tree construction [Ukkonen 1995], in particular, allows suffix trees to be constructed in $O(n)$ time and space, in the order of the length of the string. Searching for a particular substring of length $m$ can be done in time in $O(m)$, that is, independent of the length of the string to be searched. The functions over suffix trees are shown in Figure B.1; their algebraic laws in Figure B.2.

new creates a new suffix tree from a sequence of bytes; all suffix tree functions use 8-sequences. empty takes as its parameter the maximum sequence length; while this buffer preallocation is not required by Ukkonen’s algorithm, it greatly simplifies the underlying code. K-sequences can be appended to suffix trees. The find function supports the suffix tree’s search facility, and will return the location of a full match; longestMatch finds the longest subsequence subsequence of the suffix tree that matches the provided sequence, starting from the beginning. The return values are the length and zero-based position of that match, respectively.

new : 8 sequence -> suffixtree
empty : number -> suffixtree
append : suffixtree * 8 sequence -> unit

find : suffixtree * 8 sequence -> number option
longestMatch : suffixtree * 8 sequence -> number * number

Figure B.1: Suffix tree functions.
new(s) = append(empty(|s|), s)
new(s .. t) = append(new(s), t)

find(s, v) = none if v ∉ s
find(s, v) = i if s[i to (i + |v| - 1)] = v

longestMatch(s, v) =
    (j : ∅ k : (k > j, ∃ i : s[i to (i + k - 1)] = v[0 to k - 1]),
     (i : s[i to (i + j - 1)] = v[0 to k - 1])

Figure B.2: Algebraic laws for suffix tree functions.

B.2 ELF Files

The ELF file format is simple, supported by Linux and accessible through a number of software packages. The Lua extension offers access to much of the ELF file structure, including sections and data segments. The latter can be converted into k-sequences directly (upon which compression can be applied), or interpreted through patternization (see Section 3.2). The `ElfData` and `ElfFile` basic types have been added to the language. `ElfFile` is supported by the functions,

<table>
<thead>
<tr>
<th>(ElfFile functions 84)≡</th>
</tr>
</thead>
<tbody>
<tr>
<td>open : string → ElfFile</td>
</tr>
<tr>
<td>create : string * ElfHeader → ElfFile</td>
</tr>
<tr>
<td>readHeader : ElfFile → ElfHeader</td>
</tr>
<tr>
<td>readData : ElfFile → ElfData option</td>
</tr>
<tr>
<td>writeData : ElfFile * ElfData → unit</td>
</tr>
<tr>
<td>close : ElfFile → unit</td>
</tr>
<tr>
<td>name : ElfFile → string</td>
</tr>
<tr>
<td>textSeg : ElfFile → number</td>
</tr>
</tbody>
</table>
open will open an ELF file and associate it with the resultant object (or produce a run-time error); create will create a new file, with the ability to write to that file using writeData. An ElfFile object created using open can be used with readHeader and readData, but not writeData; the converse is true for file objects created using create. We have not introduced observers for the ElfHeader data type; it is entirely opaque, because its contents are unnecessary within the context of our experiments. For this reason, it is necessary to create an ELF file by using the header of another; this is generally part of a procedure that either compresses or decompresses the contents of an ELF file, but leaves the header untouched. close is used only with create objects, to signal some underlying cleanup procedure necessary in creating the ELF file’s final form. name simply returns the name of the file associated with the ElfFile object. And the textSeg function returns the zero-based index of the text segment within the ELF file. This function’s implementation falls outside the realm of ELF at a basic level; it has been included here as an aid to instruction stream compressors in particular.

ElfData contains the information describing a data segment in the ELF format, including the section identifier and the raw data itself. The ELF format supports an additional level of indirection which is hidden in the Lua extension—ELF files are two-dimensional, an array of sections, each of which may have one or more data segments. The form of the readData function makes the enumeration of data segments inherently one-dimensional; the ElfData object therefore contains its section identifier so that the full ELF structure can be appropriately reconstructed, although the section identifier is not strictly part of an ELF data segment description. Below are the functions for the ElfData type.

85 (ElfData functions 85) ≡

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>number * 8 sequence -&gt; ElfData</td>
</tr>
<tr>
<td>length</td>
<td>ElfData -&gt; number</td>
</tr>
<tr>
<td>section</td>
<td>ElfData -&gt; number</td>
</tr>
<tr>
<td>data</td>
<td>ElfData -&gt; 8 sequence</td>
</tr>
</tbody>
</table>
In addition to the readData function for ElfFile objects, ElfData objects can be introduced using the new function. new must be supplied with the zero-based section identifier, and an 8-sequence representing the data for that section. length returns the length, in bytes, of the data only; section returns the section identifier; and data returns the 8-sequence containing the data for the ElfData object.

### B.3 Profiling Data

A number of data types have been added to the language to support only very specific calculations. MachineDescription is one of them. This type, actually implemented as a table of functions, is entirely opaque. An object of type MachineDescription is passed to the patternization functions in order to allow them to decode and encode instruction streams; it has no use outside of that context. The NodeData type is very similar; it is used only internally with patternization, and will be described later in Section 3.2.

Similarly, GProfData is used only to interface with gprof-x output, for block coalescing experiments. GProfData allows the Lua environment access to the flat profile of a program, with indices into that program’s instruction stream. It has the following supporting functions.

86a \(\langle GProfData\ functions\ 86a\rangle\equiv\)

\[
\begin{array}{l}
\text{read} : \text{string} \rightarrow \text{GProfData}
\end{array}
\]

A GProfData object is represented by a table, a zero-based array of tables which contain information about each function in the profile. For example, for a profile that begins,\(^1\)

86b \(\langle\text{sample gprof output}\ 86b\rangle\equiv\)

\[
\begin{array}{cccccc}
% & cumulative & self & self & total & name \\
\hline
time & seconds & seconds & calls & ms/call & ms/call & name \\
32.36 & 10.09 & 10.09 & 3473191 & 0.00 & 0.00 & setSequenceSymbol \\
18.57 & 15.88 & 5.79 & 594 & 9.75 & 23.78 & KSequence_castSmall \\
\end{array}
\]

the structure of the associated GProfData object would be,

86c \(\langle\text{sample GProfData values}\ 86c\rangle\equiv\)

\[
\begin{array}{l}
data.textsection = 12
\end{array}
\]

\[
\begin{array}{l}
data[0].start = 2781096 \\
data[0].end = 2782407 \\
data[0].calls = 3473191 \\
data[0].ticks = 1009 \\
data[1].start = 2777268 \\
data[1].end = 2778471 \\
data[1].calls = 594 \\
data[1].ticks = 579 \\
\end{array}
\]

\(^1\)This profile is taken from the Lua interpreter itself, compressing an ELF file using gzip.
Each element in the array is itself a table, with members `start` and `end` (indicating the beginning and end of the function in the instruction stream), `calls` (as in the above profile) and `ticks` (one tick per self-millisecond).

### B.4 Shuffling K-Sequences

A `shuffle` function over k-sequences is also provided by Lua-ISC. The `shuffle` effectively reorders the symbols in a sequence using a mapping table; the `unshuffle` function can be applied to retrieve the original sequence. These functions are used by our block compression algorithm (see Chapter 6) to initially create the blocks, and then to retrieve individual instructions from them.

It is important to understand the representation of `shuffle`'s block mapping table in order to read the block compression code fragments in Appendix C.3.3. The mapping, along with the procedure, is shown in Figure B.3. The block mapping is a sequence of tables; each table contains a block identifier number, and the number of symbols to be added to that block. Block identifier numbers do not need to be consecutive or otherwise related.

The k-sequence shuffle functions are shown below.

```plaintext
shuffle : #n sequence * sequence table -> #n sequence
unshuffle : #n sequence * sequence table -> #n sequence
unshuffle(shuffle(s, f), f) = s
```
Appendix C

Supporting Code

This appendix contains various code expressions in Lua-ISC. Section C.1 provides examples of the language, referenced primarily by Chapter 4. Section C.2 supports the experiments in representation, containing the byte compressor and pieces of the experimental infrastructure. Finally, Section C.3 contains the codeform representations of the algorithms in Chapter 6: Ernst wire form, inter-program compression and block compression. The SSD implementation has been omitted because of its size and complexity.

C.1 Example Code

C.1.1 Lua Example–Insertion Sort

The following code is intended to provide a small picture of the language in use. It defines an insertion sort function.

```
function insertionsort(data, greaterthan)
    local new_data = {} -- create an empty table for output
    local i, v = next(data, nil) -- get the first index and element
    local num = 0 -- Number of elements in the new table
    while i do
        local j, w = next(new_data, nil) -- get the first index and element
        -- of the current output

        -- Skip the elements less than the one to be inserted
        while j and greaterthan(v, w) do
            j, w = next(new_data, j)
        end

        -- Insert the new element over the one in that position
        if j then
            new_data[j] = v -- add in data from input
        end
    end

    return new_data
end
```

89
else
    new_data[num] = v -- add at the end
end
num = num + 1

-- Shift the other elements over one
while j do
    local next_j, next_w = next(new_data, j)

    if next_j then new_data[next_j] = w
    else new_data[j + 1] = w
    end

    j = next_j; w = next_w
end

i, v = next(data, i)
end

return new_data
end

A number of language features appear in this example. First, the syntax of control
flow for statements while and if are shown. Scope of variables is controlled by the
keyword local; variables can also be initialized simply by referencing them without
declaration, but they are treated as global in that case. Functions can return more
than one value; next returns the next index and the value at that index. Table
indexing is accomplished using square-brackets; Lua also contains syntactic sugar
for table indexing, whereby

```
    table.name        <=>  table["name"]
    table:name(x, y)  <=>  table.name(table, x, y)
```

The insertion sort function does not require that the indices be of a par-
ticular type, but its output is always a zero-based array.
C.1.2 Translating Components

It is often necessary to translate from one k-sequence width to another; this allows compressors to be more general in their input types. The below code is a very simple translating component. Its construction illustrates the way in which we build new components in Lua-TSC.

\[
\text{function convert_to_8(kseq)} \\
\quad \text{return KSequence.cast(kseq, 8), KSequence.k(kseq)} \\
\text{end}
\]

\[
\text{function unconvert_from_8(kseq, k)} \\
\quad \text{return KSequence_cast(kseq, k)} \\
\text{end}
\]

```
Compressor.convert_to_8 = 
Compressor.new(convert_to_8, unconvert_from_8)
```

In practice, it is important to pad the k-sequence to bytes before casting it; this way, data is still effectively aligned in the 8 sequence output of the forward function. This is not difficult to do forward, but the reverse is somewhat trickier. We have omitted these details to simplify the example.

C.2 Representation Experiments

C.2.1 Byte Compressor

The byte compressor is straightforward; it does not use patternization. The ElfdData observer function data creates an 8 sequence, to which the text compressor can be directly applied.

```
-- compressor(data, machdesc, text_comp)
function compressor(data, machdesc, text_comp)

-- Compress the data directly
local origdata = ElfdData.data(data)
local compdata = text_comp:apply(origdata)

local table = {}
table.istream = compdata
return table
end
```
C.3 Algorithms

C.3.1 Ernst Wire Form

Finally, the Huffman codes are expanded to fit byte boundaries before the full conglomeration of Huffman-coded MTF indices and the MTF dictionary are pushed through gzip.

Lua-ISC contains the necessary primitive elements for expressing this algorithm. The below code segments present the interesting parts of the implementation; other portions are left to the Appendix.

```lua
function name(NodeData data)
    if NodeData.isLeaf(data) and not NodeData.isRoot(data) then
        return NodeData.parentName(data)
    end

    return nil
end
```

This function, used to direct patternization, differentiates operands with different parents. Patternizing a binary constructor, for example, will create two entries in the same operand stream. As in Ernst [Ernst, Evans, Fraser, Lucco, and Proebsting 1997], internal nodes are never evicted.

The primitive `Compressor.mtfd` produces the index and dictionary streams; the primitive `Compressor.huffman` creates a k-sequence which is then padded to match byte boundaries. The code is shown below. It does not avail itself of the language’s composition facilities, in order that the structure of the operation is made more clear.
function wire_comp(streams)
  -- Compress each of the operand streams
  local id = 1
  local nm = "ostream" .. id
  while streams[nm] do
    local indices, dictionary

    -- Apply MTFD to each of the operand streams
    indices, dictionary = Compressor.mtfd:apply(streams[nm])

    -- Huffman-code the MTFD indices
    local d_indices
    indices, d_indices = Compressor.huffman:apply(indices)

    -- Expand the Huffman codes to byte boundaries
    indices = expand_to_bytes(indices)

    -- Apply gzip to everything
    streams[nm] = KSequence.combine(indices, dictionary)
    streams[nm] = KSequence.combine(streams[nm], d_indices)
    streams[nm] = Compressor.gzip:apply(streams[nm])

    -- Move on to the next stream
    id = id + 1; nm = "ostream" .. id
  end

  -- Compress the instruction stream.
  streams.istream = Compressor.gzip:apply(streams.istream)

  -- Compress the trailing bytes
  streams.trailing = Compressor.gzip:apply(streams.trailing)

  -- And convert the table to a ksequence
  kdata = table_to_ksequence(streams)
This implementation of Ernst’s algorithm differs only in its use of resolved addresses; Ernst creates his wire form from the lcc IR, which contains symbolic names as text.

C.3.2 Inter-Program Sequences

Lua-ISC contains a primitive data type SuffixTree which implements Ukkonen suffix trees, the most complicated part of the overall algorithm. The rest can be fairly concisely expressed in Lua-ISC; the sequence generation and refinement functions are shown below.

```lua
function find_candidates(textseg1, textseg2, cutoff_len)
  -- Convert the first text segment into a suffix tree
  local suffixtree = SuffixTree.new(textseg1)

  local candidate_number = 0
  local candidates = {}

  -- Find matching subsequences
  while(Sequence.length(textseg2) > 0) do
    local match_len, match_pos
    match_len, match_pos = SuffixTree.longestMatch(suffixtree, textseg2)

    if(match_len >= cutoff_len) then
      -- Store the new candidate match
      candidates[candidate_number] = {len=match_len, pos=match_pos}
      candidate_number = candidate_number + 1

      -- Advance the sequence
      textseg2 = Sequence.shift(textseg2, match_len)
    else
      -- Just advance the sequence
      textseg2 = Sequence.shift(textseg2, 1)
    end
  end

  -- Return the candidates
  return (candidates, textseg1)
end
```
The `find_candidates` function is used by both sequence generation and sequence refinement. It takes advantage of the `KSequence.shift` function, which removes `n` elements from the front of the sequence. `get_text_segment` simply returns the contents of the text segment as a k-sequence. Suffix tree operations are straightforward; `longestMatch` returns the length and position of the longest subsequence that matches `textseg2`, starting from its beginning.

The resultant candidate table contains the length and position of matches, indexed in the first input sequence. `find_candidates` also returns the text segment of the first file, as the source of sequence data referenced by the candidate table. Sequence generation, shown below, is trivial given `find_candidates`.

```plaintext
function generate_sequence(file1, file2, cutoff_len)
    -- Get the text segments for the input files
    local textseg1, textseg2
    textseg1 = get_text_segment(file1)
    textseg2 = get_text_segment(file2)

    -- Let find_candidates do the work
    return find_candidates(textseg1, textseg2, cutoff_len)
end
```
The sequence refinement step is shown below. Although less trivial than `generate_sequence`, it relies heavily upon `find_candidates` to refine the candidate pool.

```plaintext
<sequence refinement step 96>≡

function refine_sequence(candidates, seqdata, file, cutoff_len)
  -- Get the text segment for the input file
  local textseg = get_text_segment(file)
  -- Convert it into a suffix tree
  local suffixtree = SuffixTree.new(textseg)
  local candidate_number = 0
  local sequence_location = 0
  local new_candidates = {}
  local new_candidate_number = 0
  -- Store the original sequence data so we can return it
  local orig_seqdata = seqdata
  -- Refine each candidate sequence
  while candidates[candidate_number] do
    -- The candidates are in order of starting position;
    -- shift the sequence data over to begin with the
    -- new candidate, and make sure to truncate it so we don't
    -- get matches beyond the real candidate
    local offset = candidates[candidate_number].pos - sequence_location
    seqdata = KSequence.shift(seqdata, offset)
    local test_seqdata = KSequence.truncate(seqdata,
                               candidates[candidate_number].len)
    sequence_location = sequence_location + offset
    -- Find matches between the candidate and the text segment
    local sub_candidates, sub_candidates_num, s
    sub_candidates, sub_candidates_num, s =
     find_candidates(test_seqdata, textseg, cutoff_len)
    -- Add all the sub-candidates to the new list
    local i = 0
    while sub_candidates[i] do
      new_candidates[new_candidate_number] = sub_candidates[i]
      -- Update indices into the original sequence data
      new_candidates[new_candidate_number].pos =
        new_candidates[new_candidate_number].pos + sequence_location
      i = i + 1; new_candidate_number = new_candidate_number + 1
    end
  end
```
-- Return the candidates
    return (new_candidates, orig_seqdata)
end

The core of this function is an elegant reuse of `find_candidates`. Each candidate
from the previous refinement step is used as a full text segment in `find_candidates`;
the result of the function call will therefore be zero or more subsequences, matching
or exceeding the cutoff length, that appear both in the candidate and the new
refinement stream. The new candidates can be blindly added to the next generation.

The above implementation consumes extra time in its use of `find_candidates`;
because we need the candidate position index in terms of the original candidate (and
thus the original sequence stream), we must supply the candidate as the first argu-
ment to `find_candidates` and the full text segment as the second. The candidate
is transformed into a suffix tree and the full segment is transversed, rather than the
desireable opposite. Because the position translation is difficult to accomplish as
well, we have chosen to sacrifice performance for readability in this implementation.
The algorithm in no way mandates this efficiency sink; other implementations are
free to remove this problem.

C.3.3 Block Compression

```bash
-- block_compress : 'a sequence * 'b sequence table *
    function block_compress(stream, block_mapping, compressor)
        local blocks = KSequence.shuffle(stream, block_mapping)
        local i = 0
        while blocks[i] do
            blocks[i] = compressor:apply(blocks[i])
        end

        return blocks
    end
```
As presented, this function lacks the flexibility for a number of interesting features, including different compressors for different types of blocks (heavily-used blocks might not even be compressed). This and other extensions are discussed in Chapter 7; the above will suffice for this work.

Decompression is even more straightforward. The translation from instruction to block is aided by a compression-side preprocess that inverts the direction of the block mapping; the code for the inversion is somewhat lengthy, and has been omitted.

```plaintext
(Block access 98)≡
function block_access(instr_pos, blocks, instr_mapping, compressor)
    local block_id = apply_mapping(instr_mapping, instr_pos)
    return compressor:revapply(blocks[block_id])
end
```

Cache management, particularly storing “hot” instructions and detecting when a desired instruction is not in the cache, are considered the responsibility of the execution environment and are not presented here. When the environment detects the need for an instruction at a particular address in the original instruction sequence, it can retrieve this instruction (and those in its block) by supplying the block_access function with the position, mapping table, the set of blocks and the compressor. The instruction mapping also contains the requisite information for distributing the instructions within the block over the original instruction sequence space.
Bibliography


99


