Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.
Adaptive Object Code Compression

John Gilbert

A thesis submitted to the University of Dublin, Trinity College
in fulfilment of the requirements for the degree of
Doctor of Philosophy

2009
Declaration

I, John Gilbert, declare that this thesis has not been submitted as an exercise for a degree at this or any other university, and is entirely my own work. I agree that the Library may lend or copy this thesis upon request.
Summary

Previous object code compression schemes have employed static and semi-adaptive compression algorithms to reduce the size of instruction memory in embedded systems. The suggestion by a number of researchers that adaptive compression techniques are unlikely to yield satisfactory results for code compression has resulted in virtually no investigation of their application to that domain. This suggestion has arisen due to erroneous claims that decompression at run-time from arbitrary branch targets or from arbitrary instruction cache line boundaries in the code is required since control passes from one part of a program's code to another during execution. This thesis investigates adaptive approaches to code compression which operate at the granularity of a program's basic blocks and cache lines, where the context for compression is determined by an analysis of control flow in the code being compressed.

We begin by developing an adaptive technique for compressing small quantities of text which are organized as a rooted directed graph. The technique we present, called Graph LZW, determines the set of nodes which are guaranteed to be encountered before reaching a node \( x \) while traversing any valid path in the graph, and uses them as a basis for conditioning an LZW dictionary for the compression/expansion of the data in \( x \). When compared with the LZW technique our Graph LZW algorithm trades additional analysis undertaken at compression-time for improved compression ratios and maintains the dictionary in a stack like fashion. Despite giving superior compression, the modifications to a standard LZW decompressor to support our approach are relatively minor and do not contribute any significant time overhead to the decoding routine.

An approach similar to the Graph LZW algorithm can be adopted for compressing a program's object code. One view of a program's code is as a control flow graph (CFG). Nodes in this graph are called basic blocks and contain maximal sequences of consecutive machine instructions in which flow of control enters at the beginning and only leaves at the end. An edge \( (x,y) \) in the graph is used to indicate that execution of
block \( y \) can immediately follow that of block \( x \) in some execution sequence. Using the CFG as a program representation we modify our Graph LZW technique and consider its suitability as an approach to compressing object code. We find that although the additional context employed by the algorithm greatly improves the compression ratio achieved over compressing the blocks individually using LZW, code size reductions are not achieved. This is due to encoding overheads associated with combining the underlying processor’s requirement that branch targets be aligned on 32-bit boundaries and compressing variable-length blocks of program code.

By moving our unit of compression from variable-length basic blocks to fixed-length instruction sequences found in cache lines these encoding overheads which result in poor compression can be minimized. To this end we develop a novel data structure termed the compulsory miss tree which encodes for each cache line \( c \) in a program which other cache lines are guaranteed to have caused their compulsory miss prior to the first miss for \( c \). Based on this program representation we present a variation on our adapted Graph LZW algorithm, which statically allocates space in the LZW dictionary to those cache lines on the path to the largest number of other cache lines—essentially partitioning the dictionary into variable sized divisions. Using the compulsory miss tree based technique, elimination of 7%, 14%, 19% and 19% of the original code is achieved for cache lines of length 16, 32, 64 and 128 bytes respectively. Motivated by these promising code size reduction results we design a pipelined decompression unit in hardware using Verilog. By considering the speed of a commercial flash memory and that at which a direct synthesis of our pipeline can be clocked at we find that the overhead associated with our code compression system does not incur an unreasonable performance penalty.

Ultimately these techniques allow us to conclude that the pervasively claimed random access decompression requirement for object code compression [LHJC05, LHWO0], MS05, EEF+97, XWL01b, LHC+03, ABFG+03, LSSC03, NMC03, OW02] is unnecessarily restrictive. Furthermore we refute the claims made by other researchers that adaptive compression models are inappropriate, impractical or impossible to use for object code compression since they do not allow for random access or work well for small blocks of data [XWL03, KMH+98, WC92, KW94, LW99b, CAP99, Luc00, LW98, LW99a, LDK95, LDK99].
I would like to express my thanks to my supervisor, David M Abrahamson, for his support and encouragement throughout my time working on this thesis. Edsko de Vries has my gratitude for many enjoyable discussions related to my work and for providing critical feedback on various papers I wrote during my time at Trinity. He vetted the compiler design lectures I gave in 2006 and improved both the quality of the course—by helping to refine my choice of content—and my presentation skills. His comments and suggestions after reading a draft of this thesis have dramatically enhanced its overall structure and readability. The core ideas presented in this document arose while reading papers related to compressing object code, a topic first mentioned to me by David Gregg—who took time out on a number of occasions to answer my questions. I am indebted to the developers of the Diablo link-time optimizer at Ghent, without which I could not have obtained the experimental results for my ideas. I greatly appreciate the time both Gordon Syme and Marc O’Morain took to proofread the final draft of this document. Finally I would like to thank my family and friends for their support.
Contents

1 Introduction .................................................. 1
  1.1 Overview .................................................. 1
  1.2 Contributions of thesis ................................. 3
  1.3 Thesis outline ............................................. 5

2 Background ................................................... 7
  2.1 Review of memory types ................................. 7
  2.2 Advantages of memory size reduction ............... 9
    2.2.1 Improvements in battery life ..................... 9
    2.2.2 Increases in performance ......................... 10
    2.2.3 Decreases in manufacturing costs ............... 10
    2.2.4 Reductions in software development costs ....... 11
  2.3 Instruction set architecture design ................. 12
  2.4 Data compression ....................................... 14
    2.4.1 Entropy, instantaneous codes and the Kraft inequality 15
    2.4.2 Classification of compression algorithms ....... 16
    2.4.3 Huffman coding ..................................... 18
    2.4.4 Canonical Huffman codes ......................... 19
    2.4.5 Tunstall coding ..................................... 20
    2.4.6 Arithmetic coding .................................. 21
    2.4.7 LZ family of algorithms ......................... 23
  2.5 Compilation and execution of embedded software .... 28
  2.6 Related work ............................................ 29
    2.6.1 Wolfe’s Compressed Code RISC Processor ....... 31
    2.6.2 Smith—LAT removal ............................... 32
    2.6.3 Yoshida .............................................. 33
    2.6.4 Fraser—cross-jumping/procedure abstraction .... 34
CONTENTS

5 LZW Based Cache Line Compression 97
  5.1 Individual cache lines ......................................................... 97
  5.2 Compulsory miss tree .......................................................... 98
  5.3 Compulsory miss tree based compression .............................. 102
  5.4 Hardware decompression .................................................... 104
  5.5 Results ................................................................................. 107
    5.5.1 Compression results ..................................................... 107
    5.5.2 Performance evaluation ................................................ 110

6 Conclusions 119
  6.1 Future work .......................................................................... 119
    6.1.1 Immediate dominator DAG / compulsory miss DAG .......... 119
    6.1.2 Scratchpad memories .................................................... 120
    6.1.3 Low-entropy code generation ....................................... 121
  6.2 Summary .............................................................................. 122

A Example Graph Structured Text Application 125
  A.1 client.html .......................................................................... 125
  A.2 server.php .......................................................................... 128
  A.3 compressed.txt .................................................................. 128

B Pipeline Implementation 131
  B.1 Verilog listing ...................................................................... 131

C Detailed Compression Ratio Results 139

Bibliography 144
List of Figures

2.1 Entropy for the binary random variable with probabilities p and q = 1 - p 15
2.2 Classification of codes .......................... 17
2.3 Example codes for the source symbols A, B, C, D and encoding alphabet αβ .......................... 17
2.4 Example of a Huffman code for the symbols A-G. Leaves are labeled with the symbol frequency, symbol and the resulting Huffman code assignment ........................................ 19
2.5 Canonical Huffman code associated with example shown in Figure 2.4 20
2.6 Construction of a 3-bit Tunstall code .......................... 21
2.7 Operation of an arithmetic coder .......................... 23
2.8 Example of the LZ77 encoding of a string .......................... 25
2.9 Example of the LZ78 encoding of a string .......................... 27
2.10 LZW dictionary after recovery of the string ααβααβ .......................... 28
2.11 Architecture of Wolfe's Compressed Code RISC Processor .......................... 32
2.12 Example of Yoshida et. al compression technique .......................... 33

3.1 Example of graph structured text with fixed entry point at Tourist Office 49
3.2 Immediate dominator tree associated with Figure 3.1 .......................... 49
3.3 A rooted directed graph and its associated immediate dominator tree .......................... 50
3.4 Algorithm for partitioning a graph G into its intervals .......................... 52
3.5 Partitioning of a graph G into intervals .......................... 53
3.6 Computation of dominators within an interval .......................... 54
3.7 Iterative algorithm for computing dominator sets .......................... 55
3.8 Dominator sets as computed using the iterative algorithm for the graph G^l from Figure 3.5 .......................... 55
3.9 Computing immediate dominators given sets of dominators .......................... 56
3.10 Compression of data using dictionary pre-initialized by immediate dominator tree .......................................................... 58
3.11 Graph LZW encoded data for each node in 3.3 (i) ................. 58
3.12 Initial structure of dictionary during Graph LZW decompression ................................................................. 59
3.13 Dictionary structure after traversing nodes A, C and D ............ 59
3.14 Connectivity graph extracted from our snapshot of wap.sciam.com (The Scientific American website, mobile edition) ......... 60
3.15 Immediate dominator tree derived from the graph shown in Figure 3.14 .............................................................. 60
3.16 PDA website benchmark data ............................................... 61
3.17 Compression ratios (compressed size/uncompressed size) achieved at various dictionary sizes for both regular LZW and Graph LZW applied to six PDA websites .................................................. 61
3.18 Compression ratios achieved when LZW/Graph LZW pointers are represented using variable-length canonical Huffman codes 63
4.1 Example showing assembly source and its binary encoding, basic blocks and branch blocks are also identified ................. 68
4.2 Control Flow Graph for code shown in Figure 4.1 ................... 68
4.3 Extended Basic Blocks associated with CFG shown in Figure 4.2 ................................................................. 68
4.4 Immediate Dominator Tree associated with Control Flow Graph in Figure 4.2 .............................................................. 70
4.5 Mapping from original address space to the compressed address space ................................................................. 71
4.6 Encoding format for a basic block which contains no control flow instructions (entry), and one which does (join) ................... 72
4.7 High level description of the process of fetching an instruction when requested by the processor .............................. 74
4.8 If the processor next issues a request for address 0x1C, should the fall through block be used or the compressed block located at 0x1C? .................. 75
4.9 Appropriate program layout to avoid the problem in Figure 4.8 ................................................................. 75
4.10 Possible positions for decompression engine .......................... 76
4.11 An example showing why our basic block oriented schemes must be considered as post-cache decompression algorithms 78
4.12 Basic and Branch block size distributions ............................ 85
4.13 CRC immediate dominator tree (compiled with -Os and link-time optimized with Diablo) .............................................. 87
4.14 Context available at each block granularity ........................... 88
4.15 Compression ratio for 9-bit LZW applied at various variable-length block granularities ................................. 88
4.16 Conditioned 16x4 Markov model for crc32 benchmark (optimized with -oO), p(0) is shown as dotted lines, p(1) as solid lines. ................ 91
4.17 Entropy results for cjpeg ............................................................... 92
4.18 Entropy results for djpeg ............................................................... 92
4.19 Entropy results for crc ................................................................. 92
4.20 Entropy results for ispell ............................................................... 92
4.21 Entropy results for rawaudio ......................................................... 93
4.22 Entropy results for rawdaudio ......................................................... 93
4.23 Entropy results for rijndael ............................................................. 93
4.24 Entropy results for toast ............................................................... 93

5.1 Definition of touches-cl and touches-bb, relating basic blocks and cache lines to each-other .................................................. 98
5.2 Compulsory miss tree construction algorithm ...................................... 99
5.3 View of object code from Figure 4.1 showing cache lines (cl), basic blocks (bb) and the touches-cl/touches-bb relations ............................ 100
5.4 Compulsory miss tree ................................................................. 101
5.5 Initial dictionary; dictionary after compressing line 0; and structure of dictionary after five lines lines from the compulsory miss tree have been compressed .............................................................. 103
5.6 Encoding format for Line 0 ........................................................... 104
5.7 Block level diagram of LZW decoder pipeline ...................................... 108
5.8 Extracting a 9-bit stream from a byte stream ...................................... 109
5.9 Compression ratio for 9-bit LZW applied at various cache line lengths 110
5.10 Compression ratio for 9-bit LZW applied at various cache line lengths over a compulsory miss tree .................................................. 111
5.11 cjpeg ............................................................................................. 112
5.12 crc ................................................................................................. 113
5.13 djpeg ............................................................................................. 113
5.14 ispell .............................................................................................. 114
5.15 rawaudio ....................................................................................... 114
5.16 rawdaudio ...................................................................................... 115
5.17 rijndael .......................................................................................... 115
5.18 toast ............................................................................................... 116
### LIST OF FIGURES

6.1  Interprocedural control flow graph (i) dominator tree (ii) and dominator DAG (iii) .......................................................... 119

C.1  cjpeg ........................................................................................................... 140
C.2  crc .................................................................................................................. 140
C.3  djpeg ............................................................................................................. 141
C.4  ispell .............................................................................................................. 141
C.5  rawcaudio ...................................................................................................... 142
C.6  rawudio .......................................................................................................... 142
C.7  rijndael .......................................................................................................... 143
C.8  toast ............................................................................................................... 143
Chapter 1

Introduction

1.1 Overview

Reducing the size required to store a program's object code has always been an important issue for embedded systems. Reduced program size decreases the memory requirements of an embedded device, resulting in decreased hardware production costs and lower power consumption amongst other benefits (section 2.2). In his approach to showing the incompleteness of a formal axiomatic system, Chaitin has presented a rather disconcerting result concerning the size of the smallest program for producing a given result [Cha99]:

Let's call a program **elegant** if no smaller program written in the same programming language has the same output. ... Okay, there have got to be a lot of elegant programs out there, infinitely many. Why? Because for any computational task, for any specific output, there must be at least one elegant program. But what if you want to exhibit an elegant program? What if you want to prove that a specific program is elegant, that no smaller program has the same output? Well, it turns out that you can't, it's impossible!

Despite our inability to *prove* an arbitrary program is the smallest capable of producing a specific result, for pragmatic reasons substantial effort is nevertheless invested in producing dense code for embedded applications through a combination of carefully crafted code on the part of developers, innovative compilation techniques and novel micro-architecture designs. In the early 90s the verbosity of RISC code when compared with its CISC counterpart motivated a number of the mainstream embedded
processor manufacturers (Arm, IBM, MIPS) to develop commercial solutions to help improve the situation (sections 2.3 and 2.6.9). More recently, the ubiquitous deployment of VLIW and DSP architectures in a plethora of embedded devices has rekindled academic interest in the topic. Code compression is one technique which emerged to tackle the situation and has already reached a level of maturity where it is considered appropriate for inclusion in senior undergraduate/recent graduate level textbooks and their associated courses [Wol06, FFY04].

For applications running on bare metal (ie. without an operating system or monitor), compression is not generally applied to a program’s code as a whole. If it were we would need to decompress the entire program at load-time resulting in no memory size saving. For this reason, and given that control passes from one part of a program’s code to another during execution, previous work has allowed decompression at runtime from arbitrary branch targets or from arbitrary instruction cache line boundaries in the code [ABFG+03].

When compressing object code it is imperative that the original data can be recovered without loss, a requirement also shared by text compression. Many object code compression schemes have borrowed methodology from text compression algorithms using a variety of statistical and dictionary-based models [BCW90, BWC89]. An important facet used to characterize a data compression algorithm is its adaptivity, for which there are three classifications: semi-adaptive, static and adaptive. A technique’s adaptivity determines whether the model used for compression is built before compression begins (semi-adaptive and static), or whether its construction is deferred to occur dynamically during compression (adaptive). With regard to adaptivity the focus of object code compression algorithms has remained primarily on static and semi-adaptive methods, and little attention has been paid to the class of adaptive schemes. This emphasis of attention is a direct result of allowing decompression to commence at arbitrary branch targets or instruction cache line boundaries, as justified in [LHJC05, LHW00, MS05, EEF+97, XWL01b, LHC+03, ABFG+03, LSSC03, NMC03, OW02] due to the presence of control flow instructions in the object code being compressed; for example in [LHC+03] the authors note

Code compression mandates random access i.e. the ability to start decompressing the code at any location that might be the target of a jump/call/branch instruction. Only if small parts of the code are decompressed at a time, the memory usage of a system can be kept low. In contrary, file compression/decompression (e.g. images) do not need random access.
1.2. CONTRIBUTIONS OF THESIS

With such a restriction imposed on code compression it is not surprising that many existing compression algorithms cannot be employed directly when compressing object code as noted by Xie and Netto [XLW01, NACA04]. In [XWL03, KMH+98] the authors explicitly identify adaptive compression models as being inappropriate for code compression since they do not allow for random access and they depend on a history of the preceding symbols in a sequential file. The decompression of small blocks of object code from arbitrary positions can only be achieved by an adaptive model that is reset before compressing/decompressing each block. However, given the short length of these blocks, the model could not adapt sufficiently to give rise to compression—something which justified the avoidance of LZ style algorithms in [WC92, KW94]:

Unfortunately, the techniques used in compress are best suited to blocks of data much larger than a cache line so it is not a practical CCRP method\(^1\).

If the full context were available when compressing a program using an LZ-based algorithm then compression might arise, but the random access requirement would be violated [LW99b, CAP99]. It is for these reasons that the LZ algorithms have been classified elsewhere as either impractical or impossible to employ for use in object code compression [Luc00, LW98, LW99a, LDK95, LDK99].

1.2 Contributions of thesis

The work presented in this thesis exploits the observation that, despite the claims of other researchers, decompression from arbitrary branch targets is not required at runtime—it is sufficient to be able to start decompression from each branch target as it is encountered during execution of the program. By a careful analysis of the code which is to be compressed we identify additional context, beyond the boundary of a basic block or cache line, which can be used to condition an adaptive compression model. We then demonstrate how this context can be used when compressing object code adaptively. For one of our schemes we show that the overheads associated with an implementation of the technique do not necessarily cause an excessive slowdown in execution speed—in fact in some cases a speedup can result. As such, we show that the pervasively claimed random access decompression requirement for

\(^1\)The Unix compress program is based on the LZC compression method, which is an LZW implementation using variable-length pointers
1.2. CONTRIBUTIONS OF THESIS

Object code compression [LHJC05, LHW00, MS05, EEF'97, XWL01b, LHC'03, ABFG'03, LSSC03, NMC03, OW02] is unnecessarily restrictive; furthermore we refute the claims made by other researchers that adaptive compression models are inappropriate, impractical or impossible to use for object code compression since they do not allow for random access or fail to work well for small blocks of data [XWL03, KMH+98, WC92, KW94, LW99b, CAP99, Luc00, LW98, LW99a, LDK95, LDK99]. While we see these results as the ultimate contributions of this thesis, our other contributions are as follows:

We present an adaptive technique called Graph LZW for compressing small quantities of text which are organized as a rooted directed graph. We impose a constraint on the technique such that data encountered during a traversal of any valid path through the graph must be recoverable without requiring the decompression of data that is not on the path in question. The technique we present determines the set of nodes $y$ which are guaranteed to be encountered before reaching node $x$ while traversing any valid path in the graph, and uses them as a basis for conditioning an LZW dictionary for the compression/decompression of the data in node $x$. The technique trades additional analysis undertaken at compression time for improved compression ratios over regular LZW. Despite our superior compression results the modifications to a standard LZW decompressor to support our Graph LZW algorithm are minor and do not contribute any significant time overhead to the decoding routine. To the best of our knowledge we are the first to propose compressing the content of nodes in a graph by employing an adaptive model which has been conditioned using inter-node context.

Based on our Graph LZW algorithm we develop a new adaptive object code compression algorithm which is targeted at compressing variable-length program basic blocks in RISC object code, where the decompression unit is located between the instruction cache and the CPU. While a previous attempt was made at using LZW for compressing variable-length blocks of straight line code for VLIW architectures (section 2.6.14), we show that their scheme is a special case of our more general algorithm and uses only a subset of the context we identify; furthermore we demonstrate that their technique would not be suitable for the smaller sized blocks of code typically generated for RISC architectures (section 4.3.3).

We report an alternative technique for managing an LZW dictionary which is appropriate for decompressing fixed-length instruction cache lines between the program memory and the instruction cache, where space throughout the dictionary is statically allocated to blocks during compression and guaranteed to be initialized before the en-
tries are referenced at run-time. This technique addresses a number of shortcomings associated with our variable-length block scheme. To accomplish this we introduce a novel data structure termed the *compulsory miss tree* which characterizes the partial order in which compulsory misses for a program's code are guaranteed to occur at run-time in an instruction cache. This technique reduces the overheads which resulted in poor performance by the algorithm targeted at variable-length basic blocks and achieves excellent compression results. We show that the overheads associated with an implementation of the technique do not necessarily cause an excessive slowdown in execution speed. In summary it can be viewed as a new practical object code compression technique with a number of nice properties lacked by some of the static and semi-adaptive algorithms previously proposed.

### 1.3 Thesis outline

The remainder of this thesis is structured as follows: in Chapter 2 we review the advantages of instruction memory size reduction; instruction set architecture considerations for code size; general data compression background; and specific work on code size reduction techniques. Our Graph LZW algorithm is introduced in Chapter 3 before we consider how it might be used for compressing object code adaptively in Chapter 4. Chapter 5 describes our compulsory miss tree representation of a program's object code along with an algorithm for its construction, and then describes how it can be used as the basis for a practical code compression technique. Our conclusions and future work are the subject of Chapter 6.
Chapter 2

Background

In this chapter we present the context in which the contributions of our thesis should be considered. We give a short review of the common types of memory found in embedded systems and popular architectures for embedded code storage in section 2.1. This review allows us discuss the advantages associated with reducing the quantity of memory required for code storage in section 2.2. There are two ways in which we can reduce instruction memory requirements so that the advantages associated with reduced memory size might be realized: (i) by modifying the instruction set or (ii) by compressing the object code. In section 2.3 we discuss the design of instruction sets from the perspective of reducing static code size. This is followed in sections 2.4 and 2.5 by a non-exhaustive introduction to the topic of data compression and an overview of the compilation/execution of an embedded application. This material is introduced in order to provide the prerequisite background material needed to discuss existing object code compression techniques in section 2.6.

2.1 Review of memory types

Computer memory is typically characterized as being either volatile random access memory (RAM) or non-volatile read only memory (ROM). RAM comes in two forms, either static or dynamic. Static RAM (SRAM) is built from an array of latch/flip flop cells capable of storing a single logic value, generally requiring a total of 6 transistors per storage element. Dynamic RAM (DRAM) is constructed from an array of cells, each of which is capable of storing a single logic value, and is constructed from a capacitor and a single transistor. The main distinction between SRAM and DRAM is that once a value is stored in an SRAM it will remain there indefinitely as long as
2.1. REVIEW OF MEMORY TYPES

Power is supplied to the circuit, while the values stored in a DRAM will begin to decay over a period of time and require regular refreshing to ensure that they remain error-free. A more detailed introduction to their construction and operation can be found in [MK97]. In typical desktop machines the denser capacity cheaper DRAM is the type used for the computer's main memory while the faster SRAM is typically employed in the implementation of caches.

The simplest type of ROM is implemented using tiny fuses which can selectively be either blown or left intact in a memory circuit to represent the two binary states. Once a fuse is blown it cannot be repaired, so programming such ROMs is a one-off occurrence. Not all writing techniques for ROMs are as destructive. If a transistor latch which can be reset only with significant effort is employed a cross between a ROM and RAM can be built—the EPROM (erasable programmable ROM). EPROMs come in two basic types, electrically erasable (EEPROM) and UV-erasable (UV/EPROM).

Flash memory is a non-volatile rewriteable memory generally considered to be a solid state storage device. It stores information in an array of floating gate transistor cells. There are two types: NOR based and NAND based. A NOR flash cell looks like a normal metal-oxide-semiconductor field-effect transistor (MOSFET) except it has two gates instead of one—the first is the control gate (like a normal MOSFET gate) while the second is a floating gate. NOR based flash has a long erase and write time, but has an interface that allows random access to any location making it suitable for storage of program code. NAND flash on the other hand has faster erase and write times, higher density and lower cost per bit than NOR flash. Its I/O interface only allows sequential access to data making it useful for mass storage devices, but less appropriate as a regular computer memory.

Within the embedded market code can be stored and executed using a wide variety of architectures. It might be stored in ROMs, EEPROMs or NOR based flash memory as established in the implementation of automotive systems [Ban]. It may be fetched directly for execution from such memories (a practice referred to as Execute In Place), or alternatively have its code and data image stored in NAND flash memory and loaded into RAM for execution when the device boots (referred to as a Fully Shadowed system). Another option is to use a demand paging system as is employed in most modern operating systems, or employ some variation or combination of these systems [Int06].
2.2 Advantages of memory size reduction

In this section we describe four advantages associated with code size reductions which allow for reduced memory requirements. We will discuss improvements in battery life due to decreased energy consumption, increases in performance due to reduced memory access time, reductions in production costs due to increased yield and reductions in software development/maintenance costs due to effective memory size increases.

2.2.1 Improvements in battery life

Power is typically defined as a measure of the rate at which work is done by a system and energy as the total work done over a period of time. In the context of computing systems work generally refers to activities associated with the execution of a program's instructions, while power and energy consumption are the rate and total amount of electrical energy consumed by the system [VF05]. There are two contributors to the total power consumption of an electronic circuit: \textit{dynamic power} and \textit{static power}. The dynamic power consumed is that caused due to switching activity within the circuit (when a value changes from 0 \textendash; 1 or vice-versa). Static power on the other hand is drawn by the circuit irrespective of whether the circuit is switching or idle and is caused by imperfections in the construction of transistors. While dynamic power is the main source of power dissipation in CMOS devices, the contribution of static power is becoming increasingly significant as circuit implementation technology scales down in size.

Since DRAM requires frequent refreshing of the charge stored in its capacitors it consumes more power than an SRAM equivalent. By reducing the size of a DRAM— for example, by using a code size reduction technique in a system which executes instructions directly from DRAM—the dynamic power dissipation of the circuit can be reduced due to the reduction in the switching activity of the module during these refresh cycles.

Leakage power, one component of static power consumption in a chip, is proportional to the leakage dissipated in all of a circuits transistors regardless of the memory type being employed [VF05]. By reducing the capacity of a transistor based memory due to decreased code size we can help remove the impact of static power dissipation.

Techniques which reduce power consumption do not necessarily reduce energy consumption. Nevertheless by reducing the power consumption of a system due to decreased memory requirements without affecting execution of the application-level
program, the energy consumption of an embedded device can be reduced thus increasing battery life.

### 2.2.2 Increases in performance

Smaller memories can generally be read faster than their larger equivalents, for example in 2001 a flash memory of 16Mb capacity had an access time of 65ns compared with an access time of 150ns for a memory of 128Mb [PH90]. A combined code/data memory that has been decreased in size may result in performance increases for the system due to reduced memory access time.

While such a performance increase can be achieved as a result of any technique which results in reduced memory requirements, previous work investigating object code compression has shown that performance can further increase in the presence of a code compression technique when the backing memory is relatively slow. This happens due to the reduction in time spent accessing memory since less data must be fetched, and was observed in [WC92] when high instruction cache miss rates were encountered during program execution.

### 2.2.3 Decreases in manufacturing costs

In many control oriented embedded systems program memory size overshadows the size and cost of the processor. In a high end hard disk drive an embedded processor might occupy an area of 6mm² while the program memory for the processor requires 20 – 40mm² [KMH+98].

When manufacturing an integrated circuit, a circular silicon wafer is tested and chopped into dies that are packaged. The cost of the final circuit can be estimated once the number of dies that will fit on a wafer and the percentage of those that are not defective (referred to as die yield) have been determined.

\[
\text{Cost of die} = \frac{\text{Cost of wafer}}{\text{Dies per wafer} \times \text{Die yield}} \quad (2.1)
\]

The number of square dies that will fit on a wafer can be estimated approximately as the wafer area divided by the area of an individual die. More accurately the number of dies per wafer is described by
2.2. ADVANTAGES OF MEMORY SIZE REDUCTION

\[
\text{Dies per wafer} = \frac{\pi \left( \frac{\text{Wafer diameter}}{2} \right)^2}{\text{Die area}} - \frac{\pi \times \text{Wafer diameter}}{\sqrt{2} \times \text{Die area}} \quad (2.2)
\]

where the first term is the ratio of wafer area to die area and the second compensates for the problem of rectangular dies near the edges of round wafers. The following formula is an empirical model developed after investigating the yield of many manufacturing lines:

\[
\text{Die yield} = \text{Wafer yield} \times \left( 1 + \frac{\text{Defects per unit area} \times \text{Die area}}{\alpha} \right)^{-\alpha} \quad (2.3)
\]

Defects per unit area is a measure of the defects that occur at random during manufacturing. In 2006 it was typically 0.4 defects per square cm in a 90nm process. The parameter \( \alpha \) corresponds to the number of critical masking levels—a measure of manufacturing complexity. In a multilevel metal CMOS process in 2006 a reasonable estimate was \( \alpha = 4.0 \).

The manufacturing process dictates the wafer cost, wafer yield and defects per unit area, leaving the die area as the sole parameter under control of a computer architect. In practice the number of good dies per wafer, and hence the cost per die, grows roughly as the square of the die area [PH90]. Employing a code compression technique to reduce the size of physical memory required to store program code in a system on a chip embedded design offers the possibility to reduce overall die size, increasing die yield and thus decreasing manufacturing costs.

2.2.4 Reductions in software development costs

Rather than considering the reduction in code size as a technique to shrink the chip size, it can be considered as an effective enlargement of its memory capacity. Increasingly the development of embedded software is being undertaken in high level languages which are compiled into machine code. While this results in cheaper and faster software development and maintenance, compilers still typically produce code that is more verbose than its hand-coded counterpart and requires care by the programmer in preparation of the input [Bun08], [Gan04, Chapter 18—Understanding Your C Compiler: How to Minimize Code Size]. This shift can be attributed in part to the fact that total software cost in the design of an embedded system-on-a-chip is greater than
the total hardware design cost for design processes under 130nm [IBS02]. Having a larger effective memory can justify the use of high level languages and cut the cost of programmer productivity without exacerbating the problem of overall chip size and manufacturing cost.

2.3 Instruction set architecture design

Having reviewed the main advantages associated with reducing instruction memory size, we now turn our attention to a review of techniques which allow for such size reductions. In this section we discuss the design of instruction sets from the perspective of keeping static code size small and continue the review in section 2.6 with a discussion of code compaction and compression after prerequisite information on data compression and compilation are presented in sections 2.4 and 2.5.

In the early 90’s the verbosity of RISC code when compared to its CISC counterpart was considered a problem for the manufactures of embedded systems. This resulted in a trend from the manufacturers of embedded processors towards developing a reduced-width encoding of the most frequently used instructions in their fixed-width 32-bit architectures [Sch98]. This was undertaken in the hope that RISC code would become competitive with CISC code in terms of the demands placed on instruction memory storage requirements. With increasingly diverse processor microarchitectures finding their way into embedded devices, such as Very Long Instruction Word (VLIW) and Digital Signal Processor (DSP) based cores, the design of instruction sets from the perspective of keeping static code size small has become common practice. In this section we review the problems and associated solutions related to reducing static code size which have been adopted by instruction set architecture (ISA) designers for RISC, CISC and VLIW/DSP architectures.

The best known examples of reduced-width RISC encodings were developed by ARM and MIPS and are termed Thumb [Sea00] and MIPS16e [Inc05]. Recently ARM introduced Thumb2 [Phe03], an extension to Thumb which adds 16 and 32-bit instructions to benefit code density and performance. While these reduced instructions require less space than the native ISA encoding, more of them must usually be executed as the reduced ISA has access to only a limited range of registers and other machine resources. A special branch instruction is used to move the processor between a mode executing 16-bit instructions and 32-bit instructions. Originally performed by hand, tool support has been developed to automatically generate mixed-width code at the
function level based on profiled executions [KG03]. Where performance is critical the base ISA is employed, while code that is executed relatively infrequently is encoded using the Thumb/MIPS16 ISA thus balancing performance against reduced code size. Using code compression techniques (which will be described later in this chapter) does not increase the number of actual machine instructions executed, unlike MIPS16e and Thumb/Thumb2, nor does it restrict the number of registers, operations and modes made available from the underlying processor.

CISC code is generally considered more dense than RISC code as the ISA contains instructions capable of performing more complex operations and is encoded using variable-length instruction codewords. For example, Intel’s x86 architecture instructions can be encoded using anywhere between one and fifteen bytes and contains individual instructions capable of operations such as comparing entire strings in memory for equivalence. While the ISA can be used to write fairly dense code this comes at the expense of additional hardware overhead for both decoding and executing instructions. Recent microarchitectural approaches to implementing CISC instruction sets decode individual CISC instructions into RISC instructions and execute them on a RISC processor subsystem [BYTJ05, SS95, SGC01]; hence, depending on the microarchitecture, CISC processors might be considered to be compressed code processors.

With the ubiquitous use of mobile phones and digital cameras, custom DSP architectures based on the VLIW approach to processor design emerged in the late 90s. The VLIW approach to processor design—like modern superscalar architectures—executes multiple instructions in parallel to increase performance. In contrast to superscalar architectures which rely on hardware to identify which instructions can safely be scheduled for simultaneous execution, it is the responsibility of a VLIW compiler to statically schedule the code and explicitly indicate which instructions are to be executed in parallel. To this end a VLIW compiler inserts multiple operations, each of which will be dispatched to a different function unit in parallel, into a single fixed-length very long instruction which is issued to the processor at run-time. Due to the difficulty of scheduling useful operations for each function unit on every cycle ‘do nothing’ operations (NOPs) are inserted into a VLIW long word when no other operation can be issued in parallel with those already scheduled for execution during a given cycle. These NOPs consume a large portion of the code space required statically. They can be classified as either horizontal (multiple NOPs issued in one instruction due to the compiler’s inability to use all function units in a given cycle) or vertical (a series of NOPs issued to all function units on successive cycles due to exposed latencies). Verti-
2.4 DATA COMPRESSION

cal NOPs can be removed by the introduction of a multi-cycle NOP instruction while the removal of Horizontal NOPs require a little more effort, as their removal impacts the processor decode logic. In the TMS320C6x range of DSP processors the removal of horizontal NOPs is accomplished by drawing a distinction between fetch-packets and execute-packets so that NOP instructions are not stored in the program code explicitly and are inserted during instruction issue as required [Ins00]. This is accomplished by explicitly indicating in the instruction stream which instructions should be issued in parallel with subsequent instructions, and a single bit is reserved in all instructions as an indicator of where execute packets terminate. Each instruction encodes which function unit it should be executed on, and NOPs are issued to any unit which will not be issued an instruction from the instruction stream during a given cycle. Nevertheless in order to take advantages of instruction level parallelism, compilers for VLIW architectures typically apply a number of code transformations which increase the size of the object code (such as procedure inlining and loop unrolling), a problem not addressed by clever ISA design.

2.4 Data compression

Data compression (also known as source coding) is the process of encoding a body of data $D$ into a smaller body of data $\Delta(D)$ to be decoded back to $D$ or some acceptable approximation to $D$ at a later time. In general it is applied to data to remove all of its natural redundancy, and the encoded data is subsequently coded using an appropriate channel code designed to match the characteristics of a noisy communication medium thus ensuring the data can be recovered without error. Not all data can be compressed [Sto88]. Both source and channel coding are the fundamental concerns of the field of Information Theory, whose foundations were laid in Claude Shannon’s seminal paper *A Mathematical Theory of Communication* [Sha48]. In this thesis we concern ourselves only with the process of source coding and we review the most important foundations of this topic as they apply to the field of data compression in the following section. After this background material is introduced we describe the classification of practical compression algorithms in section 2.4.2 and then review a number of popular schemes in sections 2.4.3–2.4.7.
2.4. DATA COMPRESSION

2.4.1 Entropy, instantaneous codes and the Kraft inequality

In his model of a communication system Shannon describes a *discrete* information source as generating a message symbol by symbol choosing successive symbols according to certain probabilities. Given an information source characterized by an alphabet $\Sigma_{\text{source}}$ of $n$ symbols occurring with associated probabilities $p_1, p_2, \ldots, p_n$ lets assume we can measure how much uncertainty there is associated with the selection of a particular symbol and denote it $H(p_1, p_2, \ldots, p_n)$. Shannon defines three properties that would seem appropriate for such a measure ($H$ is continuous in the $p_i$, if all $p_i$ are equal then $H$ should be a monotonic increasing function of $n$, and finally if a choice is broken down into two successive choices the original $H$ should be the weighted sum of the individual values of $H$), and demonstrates that the only function $H$ satisfying the requirements is

$$H = -K \sum_{i=1}^{n} p_i \log(p_i)$$

This value $H$ or the *entropy* of a set of probabilities has a number of properties that substantiate it as a reasonable measure of information; for example $H = 0$ only if all the $p_i$ but one are zero, thus when we are certain of the outcome $H$ vanishes. For a given $n$, $H$ is a maximum and equal to $\log(n)$ when all the $p_i$ are equal, which is also the most intuitively uncertain situation. Taking $K$ as 1 and logarithms to the base 2 gives us the measure in bits. An example is shown in Figure 2.1.
We now turn our attention to the practical constraints associated with assigning codewords to symbols generated by an information source. A code $C : \Sigma_{\text{source}} \rightarrow \Sigma_{\text{code}}^*$ is non-singular if each symbol from $\Sigma_{\text{source}}$ is mapped to a different non-empty string over the code alphabet $\Sigma_{\text{code}}$: $x \neq x' \Rightarrow C(x) \neq C(x')$. It is uniquely decodable if, given a string over the source alphabet, the concatenation of the encoded symbols is non-singular; in other words, given an encoded message there is at most one source message which will generate that message. Note that the entire string may need to be processed before even the first symbol in the source string can be determined. A code is called a prefix code or instantaneous if no codeword is a prefix of another codeword. An instantaneous code can be decoded without reference to future codewords. The relationship between the sets of code types and an example of each type is given in Figures 2.2 and 2.3 respectively. In this thesis we will be interested in uniquely decodable codes, in particular those which are prefix free.

The Kraft inequality [Kra49] states that for any instantaneous code of $k$ codewords over an alphabet of size $D$ the codeword lengths $l_1, l_2, \ldots, l_k$ must satisfy the following inequality:

$$\sum_{k} \frac{1}{D^{l_k}} \leq 1$$

The proof is rather straightforward and is based on constructing a $D$-ary tree and considering the constraints on the number of leaf nodes at depth $\max(l_k)$ by classifying the nodes at that depth as either codes, descendants of codes, or neither. What is less obvious is that this same inequality also gives necessary and sufficient constraints on the codeword lengths required for any uniquely decodable code, as proven by McMillan [McM56]. The result is generally referred to as the Kraft-McMillan inequality.

An elegant proof that Shannon's entropy $H$ gives a lower bound on the average message length attainable for uniquely decodable messages follows from this inequality and is given in [Rez61] based on the observations in [McM56].

### 2.4.2 Classification of compression algorithms

Data compression techniques may be classified according to a number of facets which include their fidelity, model type and adaptivity. The fidelity of a scheme is either lossless or lossy. Lossless algorithms allow the original data be reconstructed perfectly after being compressed and subsequently decompressed. This contrasts with lossy algorithms which allow for some loss of precision or entire omission of fine detail in
2.4. DATA COMPRESSION

Figure 2.2: Classification of codes

1. Not non-singular \( \{ A \rightarrow \alpha, \; B \rightarrow \alpha, \; C \rightarrow \alpha\beta, \; D \rightarrow \beta \} \)
2. Non-singular, but not uniquely decodable \( \{ A \rightarrow \alpha, \; B \rightarrow \beta, \; C \rightarrow \alpha\beta, \; D \rightarrow \beta\alpha \} \)
3. Uniquely decodable, but not instantaneous \( \{ A \rightarrow \beta\alpha, \; B \rightarrow \alpha\alpha, \; C \rightarrow \beta\beta, \; D \rightarrow \beta\beta\alpha \} \)
4. Instantaneous \( \{ A \rightarrow \alpha, \; B \rightarrow \beta\alpha, \; C \rightarrow \beta\beta\alpha, \; D \rightarrow \beta\beta\beta\alpha \} \)

Figure 2.3: Example codes for the source symbols \( A, B, C, D \) and encoding alphabet \( \alpha\beta \)

the input as part of the compression process in return for improved size reduction.

Models are described as either statistical or dictionary-based. In a statistical model the probability of each input symbol occurring is determined. This allows an allocation of variable-length binary codes to each symbol in the input, such that symbols with a high probability of occurrence are given short codes while less likely symbols are given longer ones, subject to the constraints imposed by the Kraft-McMillan inequality. Dictionary-based models operate by moving commonly occurring sequences of symbols from the input to a dictionary (table). Each occurrence of one of these sequences may then be replaced by a shorter codeword indexing the appropriate dictionary entry.

There are three adaptivity classifications: semi-adaptive, static and adaptive. In a semi-adaptive scheme the data to be compressed are first analyzed in their entirety, an appropriate model is then built, and finally the data is encoded. The model is stored as part of the encoded data, as it is required by the decompressor to reverse the encoding. Static schemes are similar to this, but a representative selection of data is used to build a fixed model which is hard-coded into compressors and decompressors. This has the advantage that no model must be explicitly stored with the compressed data, but the
disadvantage that poor compression will result if the model is not representative of data presented for compression. Finally, adaptive techniques commence coding with an empty (or statically determined) initial model, and update this model as the coding of each successive input symbol occurs. When decoding compressed data, the same initial model used for compression is constructed and is appropriately updated as each symbol is recovered. This not only removes the need for the model to be explicitly transmitted or stored with the encoded data, but also facilitates tailoring the model to the particular data being compressed.

In the following sections we review a number of well known lossless general-purpose data compression techniques in preparation for discussing existing work on the compression of program object code later in this chapter.

2.4.3 Huffman coding

Huffman codes [Huf52] are optimal codes when the output of an information source is coded one symbol at a time. To construct a Huffman code the data to be compressed are analyzed and a count of the frequency of occurrence of each symbol in the input alphabet \( \Sigma \) (for example, each byte or word) is determined. Using this information, variable-length binary codes are assigned to input symbols so that those appearing frequently are given short codes while those appearing infrequently are allocated longer ones [BCW90]. These variable-length codes are constructed so that the prefix-property holds, that is, no code is a prefix of any other code. When compressing data the original symbols are replaced by their corresponding Huffman codes and stored in the output. The output also includes a table containing the mapping from Huffman codes to the input alphabet symbols for use by the decoding algorithm. Hence Huffman coding is an example of a lossless, statistical, semi-adaptive technique. Decoding occurs by sequentially matching Huffman codes to the encoded data and outputting the corresponding symbols.

To construct a Huffman code we begin with a set of nodes \( q \), each of which represents a single source symbol and its associated frequency of occurrence. This set is used to build a binary tree where each node becomes a leaf in the tree. The algorithm operates by removing the two nodes \( a \) and \( b \) with the lowest frequency of occurrence from \( q \) and inserting a new node with \( a \) and \( b \) as children and a frequency computed by summing the children's frequencies into \( q \). This process is repeated until the set of nodes contains only one element. The resulting node is the root of a binary tree in which the most frequently occurring source symbols are stored close to the root, and
2.4. DATA COMPRESSION

Figure 2.4: Example of a Huffman code for the symbols A-G. Leaves are labeled with the symbol frequency, symbol and the resulting Huffman code assignment.

the least frequently occurring ones are stored at the greatest depth in the tree. The Huffman code is then assigned by considering each left branch from an internal tree node to generate a 0 while the right branch generates a 1. The code for a symbol $X$ is determined by reading the sequence of 0s and 1s required to form a path from the root of the tree to the node representing the symbol. A simple example of Huffman coding is presented in Figure 2.4.

2.4.4 Canonical Huffman codes

While textbooks either omit a discussion of the decoding procedure (a correct decoding procedure is obvious), or describe it as a bit-by-bit process where you look at a bit and either go left or right in a reconstruction of the tree which was built for the code, this is slow. Given the lengths of codes for a Huffman code, we can build a tree with leaves at the same depths but with a different overall structure. This structure leads to some nice mathematical properties which can be exploited to reduce the overhead of storing and transmitting the encoding tree (or rebuilding it), and improve the decoding implementation. This restructuring gives rise to Canonical Huffman codes which were introduced by Schwartz and Kallick [SK64] and Connell [Con73].

The canonical Huffman code associated with the Huffman example we introduced earlier is shown in Figure 2.5. Note that we build the tree by filling the tree from the left to the right with the smaller to bigger codewords. The binary assignment of codes to symbols will generally not arise from the normal Huffman procedure (so the name
2.4. DATA COMPRESSION

2.4.5 Tunstall coding

While Huffman codes map fixed symbols to variable-length codes, Tunstall codes [Tun67, Say96] instead take a variable-length sequence of source symbols and encode the group using a fixed-length code; hence it is a variable-to-fixed code. The design of a variable-to-fixed code should satisfy the following two conditions: (i) we should be able to parse any source output sequence into a sequence of symbols that appear in the codebook and (ii) we should maximize the average number of source symbols represented by each codeword.

Given a set of symbols \( \{s_1, \ldots, s_n\} \) and associated probabilities \( x_i = p(s_i) \) a static \( n \)-bit Tunstall code (in which there are \( 2^n \) codewords) can be built as follows. We initialize a codebook to contain each of the \( N \) source symbols. Next we remove the entry with the highest probability \( \max(x_i) \) and insert into the codebook a sequence of new entries constructed by appending to this entry each source symbol. Each of the new entries will now have an associated probability which is the product of the probabilities of the letters concatenated to form the new entry. This process is repeated while the number

Figure 2.5: Canonical Huffman code associated with example shown in Figure 2.4

canonical Huffman codes is a misnomer). Codes of the same length increase by a value of 1, and the first code of length \( l_i \) can be determined from the last code of length \( l_{i-1} \) by adding one and multiplying by two. It is this property that leads to an efficient transmission of the coding table/coding tree, and the efficient decoding procedure. More recently canonical Huffman codes were reconsidered by Hirschberg and Lelewer and the interested reader should read [HL90, Method B] for implementation details.
2.4. DATA COMPRESSION

Sequence Probability Code

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\alpha)</td>
<td>0.4</td>
</tr>
<tr>
<td>(\beta)</td>
<td>0.5</td>
</tr>
<tr>
<td>(\gamma)</td>
<td>0.1</td>
</tr>
</tbody>
</table>

\[
P_0 = 0.1 \quad 000
\]

\[
P_1 = 0.2 \quad 001
\]

\[
P_2 = 0.25 \quad 010
\]

\[
P_3 = 0.05 \quad 011
\]

\[
P_4 = 0.16 \quad 100
\]

\[
P_5 = 0.2 \quad 101
\]

\[
P_6 = 0.04 \quad 110
\]

Figure 2.6: Construction of a 3-bit Tunstall code

of codewords in the codebook is \(\leq 2^n\). When we have finished inserting new entries into the codebook, we can arbitrarily assign each entry a binary code of \(n\)-bits.

For example, consider building a 3-bit Tunstall code for the source alphabet \(\alpha, \beta, \gamma\) with probabilities 0.4, 0.5 and 0.1 respectively as is demonstrated in Figure 2.6. Construction of the codebook terminates after two iterations, as a third iteration would generate more entries than the code can accommodate. An arbitrary assignment of 3-bit codes is made and shown in Figure 2.6 (iii).

2.4.6 Arithmetic coding

Unlike Huffman coding which separates the input message to be coded into its individual symbols and replaces each symbol with a codeword, arithmetic coding [WNC87] encodes the entire message into a single number \(n\) such that \(0 < n < 1.0\). Huffman coding is optimal in the sense that the codewords it assigns are the shortest codes possible when the encoding is performed symbol by symbol. Since it assigns integral length codewords to each symbol the redundancy of the Huffman code can be substantial when compared to the entropy of a source in the presence of highly skewed probabilities; consider an alphabet with one symbol whose probability of occurrence is very close to 1. Such a symbol transmits little information but the Huffman code will need to transmit one bit per occurrence of this symbol.

The advantage of arithmetic coding is that the entire message is encoded in a single codeword, and integral length codes for each constituent symbol in the message need not be employed. The code generated for a given message is optimal in theory, and very nearly optimal in practice (it requires \(H + 2/m\) bits, where \(m\) is the number of source symbols). Furthermore, when multiple probability distributions are used to describe the source (such as pairs of data which are generated with separate probabilities), Huffman coding requires the generation of two separate codes to take advantage
of these statistical differences. Arithmetic coding on the other hand is ideally suited
to such situations as there is a very distinct and clean interface between a probability
model and the arithmetic coder which generates the code for an input. An example of
an arithmetic code which corresponds directly to a Huffman code is employed in Lang-
don’s tutorial on arithmetic coding [GGL84], and can be a useful aid in understanding
the algorithm.

The implementation of an arithmetic coder is a rather involved process, and is one
factor contributing to the ongoing popularity of Huffman’s algorithm which is rela­tively simple to implement. In the remainder of this section we give an overview of
the operation of an arithmetic coder, assuming the code is implemented using infinite
precision fixed point arithmetic. Proofs of the uniqueness and optimality of the tech­nique are given in [Say96, Section 4.4.1]. A practical implementation written in the
C programming language, which uses fixed precision integer arithmetic is provided in
[BCW90]. While it provides an implementation that is safe, the arithmetic operations
for encoding and decoding can be slow and Howard and Vitter address these issues
by pre-computing tables which allow the replacement of arithmetic operations with
simple table lookups, giving rise to an approximate arithmetic coder with only mini­nal loss of compression efficiency [HV91, HV94]. Such implementation details are
beyond the scope of our review here and the interested reader should read about the
techniques in the appropriate literature.

Given an alphabet of symbols $\Sigma = \{\alpha, \beta, \gamma, \delta\}$ with corresponding probabilities
$p(\alpha) = 0.3, p(\beta) = 0.3, p(\gamma) = 0.2, p(\delta) = 0.2$ a message $\alpha \beta \gamma \delta$ is encoded using an
arithmetic code as follows. The half-open interval $[\text{low}, \text{high})$ is initialized to $[0.0, 1.0)$,
and then partitioned into $n = |\Sigma|$ subintervals such that the size of each subinterval
corresponding to a symbol is proportional to the probability of occurrence of that sym­bol. This is easily accomplished using the cumulative probabilities of the symbols to
determine these subinterval bounds. The subinterval which corresponds to the first
symbol to be encoded is selected, and the bounds of that subinterval becomes the new
$[\text{low}, \text{high})$. The process is repeated for each symbol in the input until the entire mes­sage has been encoded, and a binary code is generated by taking the midpoint of the final
$[\text{low}, \text{high})$ interval and truncating it to \[\left\lfloor \log_2 \frac{1}{p(\text{message})} \right\rfloor + 1\] bits. The encoding
process is shown in Figure 2.7.

The code $c$ generated to represent $\alpha \beta \gamma \delta$ is the rational number 0.1413. To decode
this back into the original message we intiailize the range $[\text{low}, \text{high})$ to be the first
interval used during the encoding process, ie. $[0,1)$. This interval is then partitioned
into the same subintervals used when encoding the first symbol of the message. We locate the interval in which the number $c$ falls [0, 0.3), and see that it corresponds to the symbol $\alpha$. This symbol is output, and now the interval in which it was contained is partitioned appropriately based on the symbol probabilities. To recover the second symbol we note that the codeword $c$ falls into the second interval $[0.09, 0.18)$ corresponding to $\beta$ and output that symbol, and continue in this fashion until the entire message has been recovered.

### 2.4.7 LZ family of algorithms

The algorithms developed by Ziv and Lempel are adaptive dictionary based data compression techniques. The basic idea exploited by the algorithms is to replace a repeated string by a reference to a previous occurrence of itself, rather than directly attempt to derive codes for each symbol in the source based on Shannon's entropy (referred to as entropy coding). They form the basis for a rather large family of data compression techniques which broadly can be classified as being derived from either the LZ77
2.4. DATA COMPRESSION

[ZL77] or the LZ78 [ZL78] algorithms. The original papers describing the LZ algorithms are rather theoretical and disguise simple and elegant compression techniques in the form of mathematical papers. A more accessible description of the two algorithms, on which the following two sections are based, is presented by [BCW90].

LZ77

The LZ77 technique uses a sliding window of \( N \) characters over a string which is to be encoded. Within the window the first \( N - F \) characters have already been processed, and the remaining \( F \) characters represent the next part of the data to be encoded and will be referred to as the lookahead. A match of maximal length from the data in the lookahead to any position within the entire buffer which starts in the first \( N \) positions is found at position \( i \). A pointer consisting of the triple \( \langle i, j, k \rangle \) is output where \( i \) is the index of the start of the match in the buffer, \( j \) is the length of the match and \( k \) is the next character in the lookahead which was not matched into the buffer. The buffer moves forward \( j + 1 \) positions and the process is repeated until the entire string has been encoded. The original string can be recovered by interpreting each \( \langle i, j, k \rangle \) triple in sequence and re-expanding the data. Each pointer is encoded using \( \lceil \log(N - F) \rceil + \lceil \log(F) \rceil + \lceil \log(|\Sigma|) \rceil \) bits. Ziv and Lempel showed that this technique performs at least as well as any semi-adaptive dictionary designed specifically for compressing the string being encoded, if \( N \) is sufficiently large.

We show the process of encoding a string over the alphabet \( \Sigma = \{ \alpha, \beta, \gamma \} \) in Figure 2.8. The sliding window is indicated using the \([\ ]\) brackets and the distinction between the data already encoded and the lookahead is indicated using a \( \mid \). A \( - \) is used to represent a null character, which initially fills the section of the sliding window dedicated to data already processed.

By varying the choice of window size and the technique employed for encoding the components of the LZ77 pointers (using either variable-length encodings of the integers, or Huffman encoding the pointer components), and indeed allowing a mix of arbitrary characters and pointers, a number of variations on the LZ77 technique have been derived, such as LZR, LZSS, LZB and LZH [BCW90]. In principle these algorithms are the same as LZ77; the differences are in their implementation details.

LZ78

A piece of LZ78-coded data is comprised of a sequence of \( \langle i, j \rangle \) pairs, where \( i \) is a fixed-length \( n \)-bit index into a dictionary of \( 2^n \) phrases, and \( j \) is a single character.
2.4. DATA COMPRESSION

\[ \alpha \beta \alpha \beta \gamma \beta \gamma \alpha \]
\[ \langle 0, 0, \alpha \rangle \]
\[ \langle 0, 0, \beta \rangle \]
\[ \langle 0, 2, \gamma \rangle \]
\[ \langle 0, 2, \alpha \rangle \]

Figure 2.8: Example of the LZ77 encoding of a string

from the input alphabet. Initially the dictionary is empty and location 0 is initialized to be the empty string \( e \). The dictionary is then searched for the longest phrase which is a prefix of the input data. The index \( i \) for this phrase is output in the compressed encoding along with the next character \( j \) from the input which was not matched to the dictionary and a new entry, consisting of the phrase just matched concatenated with \( j \), is inserted into the dictionary. Coding continues in this fashion, restarting at the next unprocessed symbol in the input. When the entire dictionary is full no more entries can be added and the remaining input data is coded using the dynamically-constructed dictionary.

Decompression of LZ78 coded data starts with the same initial dictionary that was used for compression. The dictionary index is extracted from the codeword pair \( \langle i, j \rangle \) and is used as an index to the dictionary and its corresponding phrase is output to the decompressed stream, with the symbol \( j \) appended. This new phrase just output is also inserted into the dictionary at the next available location to ensure the dictionary is kept in synchronization with that generated during compression. The dynamically constructed dictionary encodes a history of previously encountered phrases in the input stream and gives rise to compression when a code (dictionary index) is output in place of multiple symbols from the input stream during coding.

We show an example of encoding a string over the alphabet \( \Sigma = \{ \alpha, \beta, \gamma, \delta \} \) in Figure 2.9. At each step we indicate the position from which coding will occur using a \( | \), show the dictionary before it is updated at this step, and the resulting sequence of LZ78 pointers which encode the string after this step is finished.

LZ78 will code an indefinitely long string in the minimum size dictated by the entropy of the source. Note however that most sequences of data are substantially shorter than an infinity. A big benefit of the LZ78 technique is that some variants lead themselves to highly efficient implementations. By varying the way in which entries
are generated for insertion into the dictionary and for adaptively removing those which are not being used frequently gives rise to a number of variations on the algorithm such as LZC, LZT, LZMW, LZJ, LZFG and LZW. In the next section we describe the LZW variant, which we used as a representative adaptive dictionary based algorithm in developing our techniques later in this thesis.

**LZW**

LZW [Wel84] is a popular variant of LZ78 which eliminates the inclusion of an explicit character in the output after each phrase, which is often wasteful. It is classified as lossless, dictionary-based and adaptive. It eliminates the need to output an explicit character after each phrase by initializing the dictionary to contain the set of all symbols from the source alphabet $\Sigma$. During encoding the character component of each LZ78 phrase is encoded as the first character of the next LZW phrase. This is possible since the dictionary already contains all characters from the source alphabet, and so a mechanism for introducing characters to the dictionary which have not previously been encountered is unnecessary.

Decompression of LZW coded data starts with the same initial dictionary that was used for compression. Each codeword to be decompressed is used as an index to the dictionary and its corresponding phrase is output to the decompressed stream. Then the first symbol from the phrase in the dictionary indexed by the next codeword in the compressed stream is appended to the phrase just decompressed, and this sequence is then inserted as a new entry in the dictionary. In this way, the decompressor maintains the same dictionary as that generated during compression.

As an example assume $\Sigma = \{\alpha, \beta\}$ and let $n = 3$. Then, after recovering the compressed version of the string $\alpha\alpha\beta\alpha\alpha\beta$ (which consists of the sequence of LZW indices 0,0,1,2,1), the LZW dictionary will appear as shown in Figure 2.10. Our dictionary representation consists of two arrays, one of pointers back into the dictionary ($\text{Parent}$), and one of symbols taken from $\Sigma$ ($\text{Suffix}$). The first section is the initial dictionary, containing entries for each symbol in $\Sigma$. The second section contains the adaptations which are entered as part of the standard LZW encoding/decoding procedure. The final section consists of unused dictionary space. A phrase in the dictionary located at index $w$ consists of the phrase in the dictionary located at $\text{Parent}[w]$, concatenated with $\text{Suffix}[w]$.$^1$

A minor complication occurs when processing a string of the form $\alpha\beta\alpha\beta\alpha$ when

---

$^1$The phrase represented at index ‘-1’ in the dictionary is $\varepsilon$, the empty string.
Figure 2.9: Example of the LZ78 encoding of a string
$\alpha \beta$ is already present in the dictionary. The compressor will parse $\alpha \beta$ and output the code for this string (say $g$), then it inserts a new entry containing $\alpha \beta \alpha$ into the dictionary at index $i$. In the next encoding step, this entry at location $i$ matches the remainder of the input and so $i$ will be output in the compressed encoding. The difficulty occurs at the point when $i$ is encountered in the compressed encoding during decode-time, as no entry has been inserted into the dictionary at location $i$ yet. This occurs because only after $i$ is decoded can the new dictionary entry arising from processing the previous codeword $g$ be inserted into the dictionary! This is the only situation when a referenced dictionary entry will be uninitialized, and can be resolved simply by appending the first character of the string decoded from $g$ to the string represented by $g$ to produce the entry for $i$.

### 2.5 Compilation and execution of embedded software

Before it is possible to execute a computer program it must typically be compiled from its high level source programming language (such as C, C++ or Fortran) into an assembly language representation appropriate for a given computer architecture (Intel x86, Sparc, MIPS etc.). The assembly language code is then assembled into a binary object code file, after which it will generally be linked [Lev99] with appropriate libraries containing support code (such as the C run-time library, or the object code for other source level files). Once in an executable form the code can be directly executed on the underlying processor (referred to as running the program on the bare metal, which is quite typical for some low end embedded processors) or executed under the governance of an operating system which will coordinate loading the program, scheduling
its execution and providing an appropriate execution environment [SGG01]. While this is not the only approach that can be used for executing computer programs (intermediate representations of byte codes which are either interpreted or JIT compiled, such as Java, are quite popular), it is the typical view we will take of the process in this thesis.

Conceptually compilers can be thought of as being constructed from three parts – the front and back ends and an optimizer which operates between these two [ASU86, Muc97]. The front end processes the source code, constructs an abstract representation of the code tree referred to as an Abstract Syntax Tree and performs semantic analysis (a process which is independent of the target architecture). The back end takes the internal representation and performs three target-specific tasks: instruction selection, register allocation and instruction scheduling. The optimizer is the location within a compiler where language/target independent optimizations such as common subexpression elimination, dead code elimination etc. are performed.

2.6 Related work

In the work presented in this thesis we will primarily be interested in techniques which reduce the static size of a program while it is in memory, in an effort to reduce the die size or increase the effective size of available memory. Some of the existing work on code compression is not targeted specifically at the static footprint of the application, but instead concerns itself with reducing the dynamic overhead of transferring data from memory to the processor in an effort to reduce energy consumption for an embedded device [MBP02, Chapter 6—Application specific code compression].

Previous work on static code size reduction has focused on two general approaches termed code compaction and code compression. Code compaction applies transformations to the input program creating an output program which is semantically identical to the input, but which requires less space for instruction storage. Binaries resulting from code compaction are directly executable on their target architectures and no decompression step is required. The transformations employed for code compaction can be applied at both compile-time and link-time. Compiler/linker techniques for producing compact code include

- careful activation record layout to take advantage of auto-increment/decrement addressing modes in some processors (as suggested by Liao in [Lia96] with the Simple Offset Assignment and General Offset Assignment problems)
• tail merging and procedural abstraction as advocated by Fraser [FMW84] where a suffix tree is used to identify repeated instructions which are then candidates for cross jumping or turning into small subroutines

• software pipelining to schedule loop level parallelism rather than using loop unrolling to exploit instruction level parallelism

• and traditional compiler techniques such as redundant and dead code elimination applied in an aggressive interprocedural fashion, combined with procedural abstraction and cross jumping [CM99, DEMS00].

The overheads associated with separate compilation (limited analysis scope, unknown addresses and duplicated code) can be addressed by an optimizing linker [SBB05].

Code compression techniques on the other hand recode the input binary in a form which is not directly executable on the target processor, requiring an intermediate decompression step before execution can occur. Such techniques are based on coding and information theory concepts and are orthogonal to the code compaction methods mentioned above. Various code representations have been the target of compression, including the program’s source code [Eva98], its abstract syntax tree [FK97], a byte code representation of the program targeting a virtual machine [EF03], and indeed the compiled binary code. The granularities which compression has been applied at vary; the entire program [Eva98, FK97], the program’s functions and procedures [DE02, DE03, KKMS97], branch blocks [LXW04], sequences of instructions which do not extend beyond a basic block [LBCM97], individual cache lines [KMH+98, WC92] and even individual instructions [YSO+97]. In the following sections we review a cross-section of these techniques which will provide the appropriate context in which to consider the contributions of this thesis. A full review, or even a comprehensive listing, of related work is beyond the scope of this chapter and the interested reader is directed to the code compression bibliography [vdWL04].

Code compression for RISC architectures was first proposed by Wolfe, who described a compressed code RISC processor which stored Huffman coded cache lines in memory that were decompressed upon a cache miss. We describe Wolfe’s work in section 2.6.1 which is the framework in which we set our contributions of Chapter 5. Section 2.6.2 discusses a technique for removing an overhead associated with Wolfe’s work called the LAT, which gives inspiration for our solution to a problem presented in section 4.2.1. A number of semi-adaptive dictionary based code compression techniques have been proposed, many of which can be seen as a variation on the technique.
described by Yoshida that we will describe in section 2.6.3. Fraser’s work on code size reduction by generating compact code is well known to members of the compiler design community, and as we draw a comparison between improvements based on his work and some of our future work (section 6.1.3) we include a brief overview of his approach in section 2.6.4. An alternative semi-adaptive dictionary based compression technique which is similar in principle to Fraser’s work is described in section 2.6.5 to give a better idea of how semi-adaptive dictionary based compression has been applied in the context of compressing object code. The work of Ernst introduced a stream-splitting technique that has become popular for code compression and is discussed in section 2.6.6. Section 2.6.7 describes techniques which operate at the granularity of entire program procedures and are implemented purely in software. We mention Franz’s work in section 2.6.8 as it uses a form of LZW (which is the representative adaptive algorithm we employ throughout our thesis), although in practice is it not directly comparable to our work since program ASTs are the target of compression and the work does not target static memory size reduction for embedded systems. IBM developed a technique based on Wolfe’s architecture in contrast to Arm and MIPS’s who adopted reduced-width ISA encodings for reducing static code size. As the only commercial object code compression technique we are aware of it is discussed in section 2.6.9. In sections 2.6.10, 2.6.11 and 2.6.12 we discuss the work of Lefurgy, Lekatsas and Xie, each of whom undertook investigations of object code compression as the subject of their doctoral research. An adaptive approach to code compression based on the LZ77 approach to data compression which is exposed at the instruction set level is described in section 2.6.13. The most closely related work to that presented in this thesis was performed by Lin and is the topic of section 2.6.14.

### 2.6.1 Wolfe’s Compressed Code RISC Processor

Wolfe introduced the *Compressed Code RISC Processor* (CCRP) [WC92, KW94], an architecture where a standard RISC core is augmented with a code-expanding instruction cache (see Figure 2.11). Individual fixed-length cache lines are compressed to variable-length blocks using Huffman coding and stored in main memory. On a cache miss, the compressed cache line is loaded from memory, decompressed and placed in the cache by custom cache refill logic. As the location used by the processor for the requested cache line will differ from its compressed-form location in memory, due to the difference in size between the compressed and uncompressed forms of the program, the compressed cache line address in memory is looked up in a so called *line address*...
2.6. RELATED WORK

Figure 2.11: Architecture of Wolfe’s Compressed Code RISC Processor

table. Once the cache line has been placed in the cache, the requested instruction is extracted and returned to the processor as if the data had never been compressed. Thus the decompression mechanism is entirely transparent to the processor.

The line address table (LAT) may be naively encoded using one entry for every cache line in the program. The entry contains the location, in main memory, where the compressed version of each cache line is located. In a system where cache lines are 32 bytes long and addresses are 32 bits long, 32 bits of storage for every 32 bytes of instruction data are required—that is, an increase of 12.5% over the original program size (assuming compressed cache lines are byte-aligned). A better encoding can be derived by allowing a single LAT entry represent the mapping for a number of cache lines using a simple delta coding scheme. Using a delta coding approach, a base address is stored for the first line and zero or more offsets are added to this to obtain the address for contiguous lines with higher addresses in compressed memory. Each offset stored is shorter than a full address, and contains enough bits to specify the length of any compressed cache line. For the same setup as before, Wolfe’s encoding requires a LAT storage overhead of about 3.5% over the original program’s code size when eight cache lines are represented by each LAT entry.

2.6.2 Smith—LAT removal

Breternitz and Smith [JS97] investigated a technique which could be used to remove the requirement of Wolfe’s Line Address Table, by laying the uncompressed code out in memory such that there is a trivial mapping from a cache miss address to its compressed location in memory. Using this approach they remove the 3.5% overhead of
2.6. RELATED WORK

storing the LAT, drop the requirement of using a *Translation Lookaside Buffer* style structure called the CLB to store recently computed address mappings, and drop the penalty of performing a lookup in the LAT entirely. They do encounter a performance overhead with implementations of this technique which arises due to fallthrough addresses from one cache line to the next, where the fallthrough address does not necessarily preserve the required mapping. They suggest two techniques to address the problem:

- Avoidance – the architecture is aware of the issue and reserves the final instruction in the cache line for an unconditional jump to the appropriate address for the fallthrough address.

- Correction – the CPU is involved directly in identifying and correcting the situation.

While the technique demonstrates a way to eliminate the LAT overhead, in practice it has not been adopted. We are aware of no subsequent work in the CCRP framework which has employed the elimination of a LAT in this way.

### 2.6.3 Yoshida

An obvious semi-adaptive approach to code compression [YSO+97] takes a list of all *n m*-bit instructions used in a given program’s object code and extracts a list of

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0</td>
</tr>
<tr>
<td>0x08</td>
<td>4</td>
</tr>
<tr>
<td>0x04</td>
<td>2</td>
</tr>
<tr>
<td>0x01</td>
<td>1</td>
</tr>
<tr>
<td>0xC0</td>
<td>6</td>
</tr>
<tr>
<td>0x30</td>
<td>5</td>
</tr>
<tr>
<td>0xFF</td>
<td>7</td>
</tr>
<tr>
<td>0x05</td>
<td></td>
</tr>
<tr>
<td>0xC0</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0xC0</td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td></td>
</tr>
<tr>
<td>0xFF</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x00</td>
</tr>
<tr>
<td>1</td>
<td>0x01</td>
</tr>
<tr>
<td>2</td>
<td>0x04</td>
</tr>
<tr>
<td>3</td>
<td>0x05</td>
</tr>
<tr>
<td>4</td>
<td>0x08</td>
</tr>
<tr>
<td>5</td>
<td>0x30</td>
</tr>
<tr>
<td>6</td>
<td>0xC0</td>
</tr>
<tr>
<td>7</td>
<td>0xFF</td>
</tr>
</tbody>
</table>

Figure 2.12: Example of Yoshida et. al compression technique
the unique binary encodings to form a dictionary containing $p$ entries. The program is then recoded, replacing each instruction with a pointer to its original encoding in a $\lfloor \log_2(p) \rfloor$-bit wide memory unit. Decoding is implemented on-chip via a dictionary lookup. Locating compressed code in memory is straightforward since both the uncompressed program and its compressed version use an identical memory address space. Thus decoding can begin at any instruction in the stream and is not restricted to start at branch targets. The original program requires $n \times m$ bits for storage, while the recoded version uses $m \times p + \lceil m \times \log_2(p) \rceil$.

For example, given a hypothetical ISA which encodes each of its instructions with a single byte (8 bits) then the program of 14 instructions shown on the left of Figure 2.12 can be recoded as the sequence of 14 indices into a dictionary containing the unique list of 8 instructions, as shown on the right. The original program requires $112 = 14 \times 8$ bits of storage, while the recoded version uses only $106 = 8 \times 8 + \lceil 14 \times \log_2(8) \rceil$ bits.

2.6.4 Fraser—cross-jumping/procedure abstraction

A suffix tree is a data structure that presents each suffix of a string as exactly one path from the root of the tree to the tree’s leaves, where edges are labeled with non-empty strings and leaves indicate the position of the suffix in the original string. The path from the root of the tree to an internal node with $m$ descendant leaves represents a common prefix for the $m$ suffixes. Fraser [FMW84] builds a suffix tree of the generated assembly code for a program and uses it as the basis for identifying repeated sequences of instructions in the code generated by a compiler. After the suffix tree is built sequences of repeated instructions are evaluated for their suitability for replacement via cross-jumping/tail merging (where identical epilogues of procedures occur in different procedures, one is retained and the others are removed and replaced with unconditional jumps to the remaining one) and for abstraction into procedures which can then be replaced with a simple call instruction (without the overhead of a full procedure call backing up state onto the stack etc. and restoring it upon returning). Their approach unifies the two techniques which previously had been implemented as separate code size optimizations. The technique forms the basis for that subsequently developed by Cooper and McIntosh [CM99], where the author’s main improvement is obtained by abstracting over register names in the search for repeated sequence of instructions followed by live-range recoloring.
2.6. RELATED WORK

2.6.5 Liao

Liao uses Storer’s external pointer macro compression model [SS82] in developing his code compression techniques [Lia96, LDK95, LDK99]. The first is a purely software technique where common sequences of instructions are extracted to form a dictionary of what Liao calls mini-subroutines and are then replaced by calls to the dictionary. Any sequence which is also a suffix of some dictionary entry may be replaced by a call to that entry with the appropriate starting point. Hence this scheme is similar, in concept at least, to the procedure abstraction proposed by Fraser.

The second technique is somewhat more flexible in its use of the dictionary, which is seen as a single large entry rather than a sequence of entries. Any substring of instructions in the dictionary can be replaced by an appropriate call to the dictionary. This technique requires additional hardware support in the form of a call dictionary instruction CALLD, which takes as operands the location in the dictionary to begin executing instructions and also how many instructions to execute before implicitly returning. A restriction on this implementation is that conditional control flow within the dictionary entry must be excluded due to the potential presence of a different number of instructions executing on different paths taken dynamically at run-time.

Potential dictionary entries are identified using a brute force comparison of basic blocks against each other. A greedy approach is then taken to select which of the potential entries will reside in the dictionaries in [LDK95] while subsequent work considers a set-covering based heuristic which improves the compression results substantially [LDK99].

2.6.6 Ernst—Wire code and interpretable code

Ernst et al. [EEF97] describe a wire code as a code that needs not be interpreted directly but can be at least partly decompressed into an interpretable form or even compiled before they are used. Using the LCC compiler [FH95] they develop a scheme where a program’s AST is linearized and split into a number of streams for the AST’s operators and each of its operand types. These streams are then coded using variable-length Huffman codes and subsequently each stream is gzipped. As the program must be decompressed and compiled before execution this is not a representation that can be used for reducing the memory required by an embedded system running a single program throughout the duration of its lifetime.

The second technique presented in the paper is described as an interpretable code,
which targets the Omniware Virtual Machine. The VM has a RISC like instruction set of 224 instructions, which is greedily extended with macro instructions containing specializations of the instruction set containing predetermined operands, and fused instructions where common code generation idioms (like \texttt{LD} followed by \texttt{MOV} instructions) are exploited to build a parameterized-dictionary style compression technique. This essentially gives rise to superoperations in the VM, which previously have been shown to not only reduce the static size of code for interpretation but to also speed up the interpreter [Pro95]².

2.6.7 Kirovski and Debray's procedure compression

In [KKMS97] Kirovski describes a technique which requires little or no hardware support, where machine code is compressed at the granularity of a procedure and decompressed at run-time into a procedure cache in RAM. Intraprocedural control flow (that is, control flow in which the source and target are the same procedure) in the decompressed code is handled by relative branches and jumps, while interprocedural control flow (where flow may pass from one procedure to another) is managed by indirect procedure invocations via a directory service. Most of the paper is concerned with management of the procedure cache, and little discussion is devoted to the actual compression technique.

Debray and Evans [DE02, DE03] use a similar technique to reduce the memory footprint of software using their \textit{squash} tool, but rather than compressing all procedures they selectively set a threshold so that only infrequently-executed procedures are compressed; thus taking the overhead of decompression off the path of execution for the most frequently executed code. They use a technique inspired by the wire code described by Ernst [EEF+97]. The streams are taken from machine instructions operand/opcode fields rather than linearizations of the compiler's AST, and employ a canonical Huffman code for each individual stream to facilitate fast decompression of streams' data at run-time. The individual streams are interleaved, with the opcode first followed by its encoded operands using the appropriate canonical Huffman codes—which can be deterministically identified at run-time based on the decoded opcode type.

²In that work however the target is stack code rather than the implied register-code of the OmniVM
2.6.8 Franz's Slim Binaries

Franz developed the Slim Binary format [FK97] where the AST of a program is compressed using an LZW like technique with heuristic updates of the dictionary and lexically-scoped dictionary entry pruning. Code is generated for each module at load-time on the host architecture, thus providing a platform independent representation and the ability to tailor the final code executed to specific processors. The technique of generating code at load-time from a compressed format is directed at alleviating the growing distance between the speed of accessing I/O versus the speed of modern microprocessors.

2.6.9 IBM's CodePack

CodePack [KMH+98] is a commercially deployed code compression system developed by IBM for its PowerPC family of embedded processors. The decompression unit sits on the processor local bus between the system’s memory and the processor. Blocks of code are compressed using the CodePack algorithm and stored in memory, unlike the approach taken by MIPS and ARM where a reduced width instruction set was developed to augment the ISA. When a miss occurs in the processor cache the missing block is located in memory using an index table and decompressed, making the conceptual setup identical to Wolfe's CCRP. The setup can be enabled or disabled on a page-by-page basis as the system contains a memory management unit.

The compression technique used is inspired by Huffman's technique of replacing symbols which occur frequently with shorter encodings and those which occur less frequently with longer encodings. PowerPC instructions are split into two halves (the bit ranges 31:16 and 15:0) and a separate code is used for each half. The code consists of a prefix-free tag which indicates how many subsequent bits should be consumed and used to index a statically built dictionary based on the frequency of occurrence of each 16-bit literal. In a sense this is like the technique of Yoshida, but using shorter indices into the dictionary for the more frequently referenced entries and splitting the instruction into two halves. For example, encoding of the upper half of an instruction contains codewords of the following format:

- 00,nnn (represents the 8 most frequent 16-bit literals which are stored in the dictionary)
- 01,nnnnn (represents the next 32 most frequent 16-bit literals)
- 100,nnnnnn (represents the subsequent 64 most frequent 16-bit literals)
2.6. RELATED WORK

- ...  
- 111,LLLLLLLLLLLLLLLLL (used for those 16-bit literals which occur only once and so are not stored in the dictionary)

The design of CodePack style codes is fully investigated in [OW02], where the authors formulate the design of the code classes (i.e. which instructions should be represented by which code lengths) as an optimization problem and investigate the impact of varying a number of parameters (dictionary size, alphabet etc.) on the results.

2.6.10 Lefurgy

Bird, Chen and Mudge [BM96, CBM97] describe a technique where a binary image is searched for patterns of between 1 and 8 instructions which do not overlap basic block boundaries. The most frequent patterns are then greedily replaced with 1-byte opcodes which are expanded within the CPU into the original set of instructions, stalling the instruction-fetch stage within the processor’s pipeline until the sequence completes. They briefly discuss the organization of the ROM which stores the patterns to be expanded, suggesting a structure that minimizes both the space required and the time necessary to decompress the instructions. The report does not describe the specific implementation of the technique nor the encoding of compressed opcodes vs. uncompressed instructions in sufficient detail to allow a reproduction of the author’s results, nor do they address the alignment requirements required by most RISC ISAs for branch targets.

Lefurgy builds upon the basic scheme outlined and experiments with a number of implementation details [LBCM97] using PowerPC, Arm and x86 as the host architectures. Compressed code branch targets are aligned to 4-bit boundaries and the interpretation of the architectures’ encoded branches are modified to deal with this. In the baseline technique, six PowerPC illegal opcodes are identified in the first byte of the regular 4-byte encodings and used as ‘escapes’ into the dictionary codeword base. Using the remaining two bits in these six encodings allows for 32 escape bytes, each of which can be followed by a second byte indexing the dictionary. Relative branch instructions are not compressed, instead being patched after compression to target their new destinations in compressed memory. It seems appropriate that call instructions cannot be contained within dictionary sequences, as the return destination would not be easily fixed by the processor’s run-time. While some savings are attributed to the shorter codewords used by the authors, they identify frequently oc-
curing single instructions as good candidates for compression, which were missed by the scheme described by Liao where only sequences of 2 or more instructions were placed in the dictionary. The authors also consider using variable-length codewords rather than fixed 2-byte ones, which essentially re-encodes the entire host architecture ISA prefixing each codeword with a 4-bit length tag. One tag is reserved as an escape for regular uncompressed individual instructions, while the remaining tags are appropriately allocated lengths based on the frequency of dictionary entry occurrences.

Lefurgy subsequently undertook an evaluation of IBM’s CodePack algorithm implemented within the SimpleScalar framework [LPM99]. By varying the underlying architecture from issuing between 1 and 8 instructions/cycle (with appropriately sized caches) he observes that a performance benefit over native code can be achieved in systems with a narrow memory bus or long memory latency, and in those systems where CodePack does not perform well the performance loss can be improved by increasing the cache size and reducing cache misses. After introducing some simple optimizations into the decompression architecture to reduce the index fetch and decompression overhead (essentially, a CLB like cache for the index table and a number of decompressors that operate in parallel), combined with the algorithm’s inherent prefetching of compressed instructions, CodePack can fetch a compressed program with fewer main memory accesses and less latency than a native program—often providing a speedup over executing native instructions.

After observing potential speedups when decompression is performed in hardware, Lefurgy goes on to consider the implementation of such techniques in software by introducing an instruction which can directly write decompressed code into a cache line (without having to write it to the data cache, flush the data cache and invalidate the instruction cache), and an exception handler which is called whenever a miss is triggered for a line whose address lies between certain bounds low and high (the original, uncompressed address range). He considers two compression techniques: one which is similar in form to that described by Yoshida, and a software implementation of the CodePack algorithm which achieves better compression but executes a lot slower. Thus this software-decompressor approach trades the speed of hardware for the flexibility of software. Without further optimization, the software approach can take up to 17 times longer to execute than native code. To reduce this overhead Lefurgy considers two approaches: use hybrid programs where code is selectively either compressed or not compressed for an entire procedure based on profiling (but still at the granularity of a cache line), or memoization by caching uncompressed cache lines which are evicted
from the actual cache so as to avoid re-decompressing them on a subsequent miss.

**2.6.11 Lekatsas**

Semi-adaptive dictionary compression (SADC) [Lek00, LW98] is a compression technique developed by Haris Lekatsas which is engineered for a specific instruction set architecture, with the ultimate aim of exploiting repeating patterns of opcodes between adjacent instructions, or frequent opcode-register or opcode-immediate combinations. In a sense this is like building superinstructions. The technique proceeds as follows:

- Split instructions into their constituent fields (opcode, registers, immediate values etc.) forming a number of distinct streams

- Construct a semi-adaptive dictionary containing sequences of frequently occurring adjacent opcodes or opcode/reg or opcode/immediate pairs, as there is often an interdependence between such elements of an instruction.

- After selecting a new dictionary entry rewrite the entire program to make use of this new entry in a greedy fashion. The opcode stream implicitly consumes an appropriate number of entries in the other streams as required, and this functionality is hard-coded into the program decompression unit (it appears some information needs to be encoded along side the dictionaries entries to this end, but neither the paper nor the PhD thesis (section 3.1.1) discuss this in sufficient depth to reimplement the technique).

- Huffman compress the resulting streams after the dictionary has been built and the entire program has been rewritten.

This technique is presented as a pre-cache architecture [LW98], however some details are not discussed. For example, how much space do the Huffman decompression tables take up (assuming a canonical Huffman code is being employed). Also, is padding to byte alignment accounted for? How do the authors deal with the potential for code size expansion in their LAT? The authors propose two methods of writing the dictionary (trading time at decode-time for space), but don’t indicate which they used in their experiments. Whether the Huffman tables, padding to byte alignment, LATs and the semi-adaptive dictionary overheads are accounted for in the authors compression results is not indicated.
Semi-adaptive Markov compression (SAMC) is Lekatsas' second object code compression technique. Here instructions are again split into streams (not necessarily constructed from adjacent bits) with high correlations between their bits so that the upcoming bits can be predicted with a high probability. Each stream is used to train a binary Markov tree; these trees are then used to provide probabilities to a binary arithmetic coder for encoding the program's blocks. As this approach encodes each stream independently of the others, no inter-stream correlations are taken into account. Compression can be improved by connecting the Markov trees between adjacent streams providing some limited memory between streams of the model. Building a single Markov tree for instructions would require the storage of \(2^{32} - 1\) probabilities, which is impractical in practice. Instead, splitting the instruction into separate streams which each have low entropy means the technique can still attain excellent compression.

The work is again initially [LW98] set in the context of Wolfe's CCRP architecture (that is, it's a pre-cache setup). In their later work the authors move from the LAT lookup mechanism employed by Wolfe to using a technique suggested by Lefurgy where branch instructions are themselves not compressed but rather compacted and patched to jump directly to the location of compressed branch targets on a byte aligned boundary [Lek00, section 4.4], [LW99a].

In [LW98] the authors use a straightforward implementation of a binary arithmetic coder with the basic interlinked Markov tree model described above, while their later work [LW99b, LW99a] exploits reduced precision lookup-table based implementations of arithmetic coding [HV91, HV94] combined with more sophisticated Markov models. Their work on decoding programs using the lookup-table based arithmetic coder investigates fast implementations, which essentially trade space for decode time by conceptually integrating the Markov model and arithmetic finite state machine decoder, and unrolling the resulting decoder's states. This was later investigated in the context of VLIW code with Xie [XWL07].

SAMC is a technique which only works effectively for fixed-width instruction sets, since the Markov model wraps back to an initial state at the beginning of each instruction in an attempt to capture repeated occurrences of bits in instructions effectively.

2.6.12 Xie

Xie [Xie02] extends the work of Lekatsas, considering modern VLIW architectures as the target of code compression where operations for a given function unit do not always occupy the same bit positions in a long word. Given the wide instruction words issued
to the processor, a fast mechanism for accessing memory is essential when retrieving
the long words and code compression can again help here.

Xie employs three probability models for the code to be compressed: a static in­
dependent and identically distributed (iid) probability model for the probability of a
0/1 in an instruction, and two forms of Markov model: one with a static iid for each
bit position in the instruction, and a more complex one of \( \text{width} \times \text{height} \) states as sug­
gested by Lekatsas. Even the simplest model gives the surprising result that over six
different architectures the probability of a 0 is much higher than that of a 1, giving the
initial indication that a statistical coding technique might lead to big space savings. The
tradeoff between good compression (using the \( \text{width} \times \text{height} \) Markov models through
the simpler iid ones) and decompression overhead in hardware has been investigated
[XWLO1a] showing that for a sacrifice of 10% compression ratio much smaller de­
coder designs can be realized.

Three different coding techniques are considered in Xie’s work: the standard fixed­
to-variable code compression over a binary alphabet as used by Lekatsas, where Vit­
ter’s reduced precision fast lookup table arithmetic coder is employed, and two forms
of variable-to-fixed coding: one a direct implementation of Tunstall codes (with a vari­
ation added to support Markov models) [XWL02], and the second a form of arithmetic
coding which is designed as a variable-to-fixed technique.

When coding using the binary arithmetic coder, to speed up decompression of bi­
nary arithmetically coded blocks of data, he splits them into sub-blocks which are
coded independently and prefixes a block of 4 compressed units with a tag indicat­
ing where the different blocks begin; thus facilitating the decompression of the four
blocks which constitute a single block in parallel. Like Lekatsas, Xie also investigates
the multi-bit decompression unit by integrating the statistical model with the decoding
finite state machine and ‘unrolling’ [XWL01b].

His techniques are all semi-adaptive, first conditioning the statistical model to be
used and then subsequently encoding the program to be compressed. They are pre­
sented as a pre-cache architecture using Wolfe’s LAT technique to look up compressed
blocks in memory.

The novel work Xie produces is mainly centered about the use of variable-to-fixed
techniques, which allow for parallel indexing and decompression—unlike arithmetic
coding which is a strictly sequential technique. The first technique described is stan­
dard Tunstall coding, while the second is based on arithmetic coding [XWL03]: the
interval \([low, high)\) is recursively split based on the probability of a 0/1 until all inter­
vals are of size 1. Equal codeword lengths are then assigned to these unit intervals. Each unit interval uniquely represents a sequence of input bits and can be recoded as a fixed-length codeword indexing a codebook.

The results of Tunstall coding always outperform Xie’s modified arithmetic coder in their experiments. This seems intuitive since the Tunstall code uses the actual probability distribution for code construction while the modified arithmetic coder only uses approximations to that distribution during the use of the arithmetic coding interval splitting.

When using Tunstall codes with a Markov model, a separate variable-to-fixed codebook is built for each state in the Markov model. This stops the ability to decompress blocks in parallel (since subsequent codes index codebooks conditionally selected based on the previous state), but in practice it leads to building simpler decoders than using the sequential binary arithmetic decoder with a Markov model.

In Chapter 5 of his thesis Xie investigates profile driven code compression with two goals in mind: firstly, by profiling the code to be compressed instruction bus power can be reduced; secondly, the exploration of performance and compression ratio trade-offs can be explored for a post-cache architecture. Rather than compress the most frequently fetched instruction (adding delays when fetching this instruction), it is the least frequently accessed instructions that are compressed. This is in keeping with the ‘make the common case fast’ rule of thumb. A variable threshold is determined which chooses the relative amount of the program to compress while keeping a fixed performance goal met during profiling. The processor pipeline is modified to incorporate N states for decompression of the instruction post-cache. The assumption is that anything not fetched, or the percent of fetched that can be compressed are all compressed using the Markov fixed-to-variable technique. Details such as how the mapping problem is addressed and how to determine which instructions are compressed and which are left uncompressed are not discussed.

Using the Tunstall coding technique complete freedom is available in selecting the actual choice of codewords and these may be selected to minimize Hamming distance between frequently fetched pairs on the bus—reducing switching power. A technique to reduce the power consumed is presented in [XWL02]. While compressing the code will increase the entropy of the sequence of instructions, thus increasing the number of bit toggles on the bus when compressed instructions are moved, fewer bus transactions are required at run-time. Xie shows that bit switching on the bus can be improved by up to 70% with a careful allocation of Tunstall codewords based on profiling.
2.6. RELATED WORK

2.6.13 Echo technology

Fraser introduced an instruction similar to Liao’s CALLD, but rather than extracting repeated phrases to form a dictionary the instruction directly references the instructions to be executed within the regular program memory [Fra06, Fra02]. The technique is described as an instruction for direct interpretation of LZ77 compressed programs, and was subsequently investigated at Intel Labs [WJH*]. The idea is similar to that of using procedure abstraction to obtain code size reductions. Unlike using a normal CALL instruction which mandates that all instructions on the target path be executed until a RETURN instruction is encountered, this LZ77 instruction implicitly returns after a predetermined number of instructions specified at the ‘call site’ have been executed. Brisk investigated instruction selection techniques within a compilation framework to identify sequences of code that can be replaced using such instructions in [BNS04].

While Liao’s CALLD was designed for use in embedded systems which could put the dictionary in ROM—which might be cheaper, faster or more power efficient than the main program memory—this is not the case with Fraser’s LZ77 instruction. As presented, the technique modifies the processor ISA and hence is not transparent to the underlying processor on which the code executes. As such, their approach is orthogonal to our code compression algorithms described in chapters 4 and 5. Furthermore as the LZ77 instructions are permitted to reference sequences of instructions anywhere within the range of their operand’s offset field, compressed programs might exhibit poor cache performance due to poor locality of reference during execution.

2.6.14 Lin

Lin’s work [LXW04] is the most comparable to that which we describe in this thesis. He compressed VLIW code at the granularity of a branch block (a maximal sequence of instructions between two consecutive possible branch targets) using the LZW algorithm (which we describe in more detail in section 4.1 and 4.2.2). To the best of our knowledge this was the first attempt at compressing blocks of program code smaller than a procedure using an adaptive algorithm. His work assumes random access is not needed to all instructions in the program, but need only ensure possible branch targets are accessible. The decompression unit is positioned between the instruction cache and the processor, and compressed data is looked up using a table which maps from uncompressed branch target addresses to their compressed counterparts. The LZW dictionary is reset upon entering each new branch block. A variation on the
basic technique called \textit{minimum code size selective compression} operates as follows: if compressing a branch block does not result in code size reduction, it is left in its uncompressed form and bypasses the decompression unit when fetched from memory. Subsequent work [LXW07] considers compile-time optimizations to improve the compression attainable by appropriate positioning of code in the generated binary program (to maximize branch block size), and proposes an alternative adaptive compression technique for filling a table with previously unseen full instructions which can be referenced to in the dictionary if repeated within the branch block.
Chapter 3

Adaptive Compression of Graph Structured Text

In this chapter we introduce an adaptive technique for compressing small quantities of text which are organized as a rooted directed graph. We impose a constraint on the technique such that data encountered during a traversal of any valid path through the graph must be recoverable without requiring the decompression of data that is not on the path in question. The technique we present determines the set of nodes \( y \) which are guaranteed to be encountered before reaching node \( x \) while traversing any valid path in the graph, and uses them as a basis for conditioning an LZW dictionary for the compression/decompression of the data in \( x \). This technique will form the basis for a family of object code compression algorithms to be introduced later in the thesis.

3.1 Motivation

Consider an on-line tour guide application for a personal digital assistant where users position themselves at a designated starting point and log onto a website to begin a tour from that location. The pages delivered to a PDA during such tours are fairly small—generally containing a couple of paragraphs of text at most. Each page describes an item/location of current interest along with links to other nearby attractions offering users some choice in tailoring their tour. The collection of pages and links form a directed graph, similar to pages on the world-wide-web, however they remain static once authored and they have a fixed entry point (Figure 3.1).

The route of attractions a user visits during their tour can be visualized as a path through the graph. Any route which leads to *The Quays* in our example must start
at the Tourist Office and directly proceed to Trinity College Dublin optionally visiting The Book of Kells en-route to O’Connell Bridge before reaching the destination. The Tourist Office is guaranteed to be on any path reaching Trinity College, which itself is guaranteed to be on any path reaching either the Book of Kells or O’Connell Bridge. Finally, O’Connell Bridge is guaranteed to be on any path reaching The Quays. This relation, where one node in a graph is guaranteed to be present on any path reaching another is called dominance, and its transitive reduction gives rise to a tree structure called an immediate dominator tree (Figure 3.2). While compression can be applied to the text for each page individually using well known techniques, we propose exploiting the inter-page context identified by dominance to improve results when using adaptive dictionary-based compression methods.

The remainder of the chapter is structured as follows. First we give formal definitions of a rooted directed graph, the dominance and immediate dominance relations, and we discuss the relationship between an immediate dominator tree and the graph from which it was derived. Existing algorithms for computing these relations are then reviewed. With the appropriate background material covered, section 3.3 describes our new approach to compression in detail. Results and related work (to set the contributions of this chapter in context, but which does not necessarily relate to our main thesis topic of object code compression) are presented in sections 3.4 and 3.5 respectively, before some conclusions are made in section 3.6.

### 3.2 Immediate dominance

Let \( G = (V, E, root) \) be a directed rooted graph where \( V \) is the set of vertices/nodes, \( E \subseteq V \times V \) is the set of edges, and \( root \in V \) is a distinguished node. Given \( x, y \in V \) we say \( x \) dominates \( y \) (\( x \text{ dom} y \)) if every path from \( root \) to \( y \) in \( G \) passes through \( x \). Every node dominates itself and every node is dominated by \( root \). Node \( x \) strictly dominates \( y \) (\( x \text{ sdom} y \)) if \( x \text{ dom} y \) and \( x \neq y \). Node \( x \) immediately dominates \( y \) (\( x \text{ idom} y \)) if \( x \text{ sdom} y \) and \( x \) does not dominate any other dominator of \( y \). Every node other than \( root \) has a unique immediate dominator. This relation may be represented by a dominator tree stemming from \( root \) with edges representing the idom relation between nodes.

Intuitively, the nodes \( y \) which a node \( x \) dominates are those that are no longer reachable from the root of \( G \) after \( x \) has been removed; the immediate dominator of a node \( y \) is the most recent node \( x \) which, regardless of the path taken, is guaranteed
3.2. IMMEDIATE DOMINANCE

Figure 3.1: Example of graph structured text with fixed entry point at Tourist Office

Figure 3.2: Immediate dominator tree associated with Figure 3.1
3.2. IMMEDIATE DOMINANCE

Figure 3.3: A rooted directed graph and its associated immediate dominator tree
to have been encountered before reaching \( y \) while traversing \( G \) from root\(^1\). Figure 3.3 shows an example graph (i) composed of five nodes labeled \( A \) to \( E \) and its associated immediate dominator tree (ii).

Based on these definitions we make the important observation that arrival at a node \( p \) while traversing \( G \) starting from root occurs in the following way with respect to the immediate dominator tree: (i) it arrives directly from \( p \)'s immediate dominator; or (ii) it arrives directly from some descendent of \( p \)'s immediate dominator. The justification for this is as follows: a traversal may not pass directly from a node at level \( i \) in the tree to node \( p \) at level \( i+2 \), as doing so would imply that \( p \)'s immediate dominator on level \( i+1 \) did not dominate node \( p \). Furthermore, a traversal may only pass across a tree from node \( q \) to node \( p \) if their common ancestor in the tree is \( p \)'s immediate dominator (otherwise it would again bypass traversing the immediate dominator of \( p \)). It is easy to verify that traversal may pass from a node to one of its ancestors in the tree due to the existence of back edges in \( G \). The importance of this relationship between the flow of execution through a control flow graph and its associated immediate dominator tree will become evident in section 3.3.

3.2.1 Computing dominators and immediate dominators

A good review of the history of algorithms for computing the dominance relation is presented in [CHK01]. An intuitive algorithm for computing dominators is given in [AU72, Algorithm 11.5]. The basic idea is that we remove each node \( x_i \) from the graph and determine those nodes \( y_j \) which are no longer reachable from the root while \( x_i \) is

\(^1\)In these presence of infeasible paths, the notion of 'most recent' can no longer be used as a basis for defining immediate dominance. This issue will be revisited in section 6.1.1.
missing. The set of unreachable nodes are all dominated by \( x \). With an additional step integrated, it is possible to directly construct the immediate dominator tree during the execution of this algorithm.

Purdom and Moore [PWPM72] describe a similar algorithm for constructing the immediate dominator tree which runs in \( O(nm) \) (\( n \) is the number of nodes in the graph and \( m \) the number of edges). The technique builds a spanning tree of the graph with the root of the tree the same as the root of the graph. Each node \( x \) is evaluated to determine whether it might be the immediate dominator for any of its descendents \( y_1, y_2, \ldots, y_n \) in the tree, by performing a search to find if there is a path in the graph to any \( y_i \) from nodes which are not descendents of \( x \) in the tree. If no such path exists, then \( x \) is a dominator of \( y_i \) and tentatively we set the immediate dominator of \( y_i \) to \( x \). By starting at the root of the tree and progressing to the leaves in a preorder fashion, the last dominator we tentatively set as the immediate dominator of a node will be the node’s immediate dominator.

A fast algorithm \( O(m \log(n)) \) for computing immediate dominators exists [LT79]; although it has been noted that for pragmatic reasons an iterative algorithm for dataflow analysis (a technique used for optimization in modern compilers) is simpler, easier to understand, easier to implement and faster in practice and should be the technique of choice for computing dominators [CHK01]. In the following subsections we review the framing of computing dominance as a dataflow analysis problem on a directed graph as originally described by Allen using interval analysis, and look at a simpler and more commonly used iterative algorithm. Both dataflow approaches compute the dominators for each node in the graph, and from these sets the transitive reduction must be computed to give the immediate dominator tree described above. The algorithm to accomplish this is given in the final subsection.

**Interval analysis**

Allen formulated the problem of computing the dominators in a graph as a global dataflow analysis problem which could be solved using *Interval Analysis* [All70, AC76]. Given a node \( h \) in a directed graph an interval \( I(h) \) is the maximal single entry subgraph for which \( h \) is the entry node and in which all closed paths contain \( h \). A graph may be partitioned into a unique set of disjoint intervals \( \mathcal{S} = \{I(h_1), I(h_2), \ldots, I(h_n)\} \) as given by the algorithm in Figure 3.4.

The intervals associated with graph \( G' \) in Figure 3.5 are indicated to the right of the graph in \( I' \) of the same figure. A second order graph \( G^2 \) is derived from the first
3.2. IMMEDIATE DOMINANCE

1. Establish a set \( H \) for header nodes and initialize it with \( n_0 \), the unique entry node for the graph.

2. For \( h \in H \) find \( I(h) \) as follows:
   (a) Put \( h \) in \( I(h) \) as the first element of \( I(h) \).
   (b) Add to \( I(h) \) any node all of whose immediate predecessors are already in \( I(h) \).
   (c) Repeat 2 (b) until no more nodes can be added to \( I(h) \).

3. Add to \( H \) all nodes in \( G \) which are not already in \( H \) and which are not in \( I(h) \) but which have immediate predecessors in \( I(h) \).

4. Add \( I(h) \) to a set \( \mathcal{H} \) of intervals being developed.

5. Select the next unprocessed node in \( H \) and repeat steps 2, 3, 4 and 5. When there are no more unprocessed nodes in \( H \), the procedure terminates.

Figure 3.4: Algorithm for partitioning a graph \( G \) into its intervals

order graph \( G^1 \) and its associated intervals \( I^1 \) by making each first order interval into a node. The immediate predecessors of a node representing the interval with header \( h \) from \( G^1 \) in \( G^2 \) are those nodes representing intervals from \( G^1 \) in \( G^2 \) which contained immediate predecessors of \( h \) (but were not members of the interval defined by \( h \)), and the immediate successors of this new node are all of the nodes in \( G^2 \) that contained immediate successors of the exit nodes from the interval with header \( h \) in \( G^1 \) (but again which were not members of the interval defined by \( h \)). Second order intervals are the intervals in the second order graph. Successively higher order graphs can be derived until the \( k^{th} \) order graph consists of a single node. This process is shown in Figure 3.5.

Given an interval, Allen computes the dominators for each node in the interval using her Procedure B, shown in Figure 3.6.

Combining this procedure with the hierarchy of graphs \( G^1 \ldots G^k \) we can compute the dominators for each node in the graph \( G^1 \). For the single node in graph \( G^k \) initialize the dominator set to \( \{ \} \). Move to the order \( k - 1 \) graph, and associate the dominator information for the corresponding node in \( G^k \) with the header of the interval in \( G^{k-1} \) and apply Procedure B to each interval, starting with step 2. Repeat this process, moving to lower-order graphs until the final graph \( G^1 \) has been processed and the dominator information is associated with each node.

Unfortunately interval analysis cannot always reduce the input graph to a single node as we have assumed. In situations where this is not possible, the graph can be
3.2. IMMEDIATE DOMINANCE

Figure 3.5: Partitioning of a graph $G$ into intervals
3.2. IMMEDIATE DOMINANCE

1. Assign the interval head an empty dominator list.
2. For the next node $b_j$ in the interval list, form

$$Dom(b_j) = \bigcap_{i \in \text{imm.\_pred}(b_j)} (b_i \cup Dom(b_i))$$

3. Repeat 2 until all nodes in the interval have been processed.

Figure 3.6: Computation of dominators within an interval

transformed into another graph which is reducible, but such a transformation requires
duplication of nodes in the original graph. As our goal in this chapter is to develop a
compression method for the text at each node in a graph, duplicating nodes is not an ap­
pealing idea. In the next subsection we look at a simpler dataflow analysis framework
which does not suffer from this problem.

Iterative dataflow analysis

A second approach to computing dominators associates a set of dominators $Dom()$
with each node in the graph and solves the equations

$$Dom(root) = root$$

$$Dom(n) = \left( \bigcap_{p \in \text{preds}(n)} Dom(p) \right) \cup \{n\}$$

using the algorithm given in Figure 3.7 which iterates until the $Dom()$ sets stop chang­
ing (ie. a fixed point for the equations is determined). The algorithm requires that the
$Dom()$ sets be initialized to contain all nodes from the graph, or alternatively exclude
those sets which have not yet been initialized from the intersection operation. This
algorithm will always terminate and does not encounter any problems with computing
dominators for irreducible graphs as was possible with the interval analysis approach.

Consider applying this procedure to the graph $G^1$ in Figure 3.5. After lines 4 and
5 have executed the $Dom()$ sets are as shown on line (a) in Figure 3.8. The variable
changed is set to true and the algorithm begins to iterate. We assume in line 11 that
the nodes $n$ are selected in the order 1, 2, ...., 8. After the first iteration, the $Dom()$
sets will be given as in line (b), and since at least one of the $Dom()$ sets changed during
the first iteration, the algorithm will execute a second pass over lines 9–15. In this second
3.2. IMMEDIATE DOMINANCE

```pseudo
procedure Dom_Comp(N): Node -> set<Node> is
    N: set<Node>

    for each Node n € N do:
        Dom(n) := {1,...,n}
        changed := true

    while(changed) do:
        changed := false
        for each n € N do:
            temp := (∩p∈preds(n) Dom(p)) ∪ {n}
            if temp ≠ Dom(n) do:
                Dom(n) := temp
                changed := true

    return Dom
```

Figure 3.7: Iterative algorithm for computing dominator sets

<table>
<thead>
<tr>
<th>Dom(1)</th>
<th>Dom(2)</th>
<th>Dom(3)</th>
<th>Dom(4)</th>
<th>Dom(5)</th>
<th>Dom(6)</th>
<th>Dom(7)</th>
<th>Dom(8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) {1..n}</td>
<td>{1..n}</td>
<td>{1..n}</td>
<td>{1..n}</td>
<td>{1..n}</td>
<td>{1..n}</td>
<td>{1..n}</td>
<td>{1..n}</td>
</tr>
<tr>
<td>(b) {}</td>
<td>{1,2}</td>
<td>{1,2,3}</td>
<td>{1,2,3,4}</td>
<td>{1,2,3,5}</td>
<td>{1,2,3,6}</td>
<td>{1,2,7}</td>
<td>{1,2,8}</td>
</tr>
<tr>
<td>(c) {}</td>
<td>{1,2}</td>
<td>{1,2,3}</td>
<td>{1,2,3,4}</td>
<td>{1,2,3,5}</td>
<td>{1,2,3,6}</td>
<td>{1,2,7}</td>
<td>{1,2,8}</td>
</tr>
</tbody>
</table>

Figure 3.8: Dominator sets as computed using the iterative algorithm for the graph $G^1$ from Figure 3.5

pass there are no changes to the $\text{Dom}(\cdot)$ sets (row (c)), so the algorithm terminates. It should be noted that the order in which nodes are selected in line 11 can have a large impact on the number of iterations required over the loop.

When this algorithm terminates we need to compute the immediate dominators for each node. This is accomplished using the algorithm described in the next section.

**Computing immediate dominators from dominator sets**

Muchnick gives an algorithm to determine the immediate dominator tree when given the sets of dominators for each node in a graph [Muc97, Section 7.3, Figure 7.15], adapted and listed in Figure 3.9. The basic idea used is that for a node $n$ we check the list of nodes $s$ which strictly dominate $n$ and check if any other node $t$ that strictly dominates $n$ also strictly dominates $s$. If it does, then $t$ definitely is not the immediate dominator of $n$ so we eliminate it as a candidate for $n$'s immediate dominator. The process is repeated for each node in the graph resulting in the identification of each
3.3. GRAPH LZW

The key insight we make in this chapter is that the nodes which dominate a given node \( x \) constitute the set from which we can obtain textual-context for the adaptive compression of the contents of \( x \); furthermore, the structure imposed on these dominators by the immediate dominator tree indicates the order in which they will have been encountered on all paths leading to \( x \). Based on these observations we will present an adaptive

---

```
procedure Idom_Comp(N, Dom, root): Node -> Node is
    N: set<Node>
    Dom: Node -> set<Node>
    root: Node in N
    n, s, t: Node
    for each n in N - {root} do:
        Tmp(n) := Dom(n) - {n}
    for each n in N - {r} do:
        for each s in Tmp(n) do:
            for each t in Tmp(n) - {s} do:
                if t in Tmp(s) then:
                    Tmp(n) -= {t}
    for each n in N - {r} do:
        Idom(n) := first element in Tmp(n)
return Idom
```

Figure 3.9: Computing immediate dominators given sets of dominators

node’s immediate dominator.

For example, given the dominator sets in Figure 3.8 assume we would like to compute the immediate dominator of node 4. The strict dominators of node 4 are \( Dom(4) = \{4\} = \{1, 2, 3\} \). Taking \( t = 1 \) we let \( s = 2 \) or \( s = 3 \) in line 11. On line 13 we see that \( t \) strictly dominates \( s \) (for either value that \( s \) assumes), and therefore in line 14 we remove \( t \) as a candidate for the immediate dominator of node 4 since it dominates another of 4’s dominators, and hence it cannot be the ‘closest’ dominator to node 4.

In practice we have found that combining the iterative algorithm for computing dominators in a graph with this algorithm to compute the transitive reduction of that relation operates quickly.

3.3 Graph LZW

The key insight we make in this chapter is that the nodes which dominate a given node \( x \) constitute the set from which we can obtain textual-context for the adaptive compression of the contents of \( x \); furthermore, the structure imposed on these dominators by the immediate dominator tree indicates the order in which they will have been encountered on all paths leading to \( x \). Based on these observations we will present an adaptive
3.3. GRAPH LZW

dictionary-based algorithm for compressing nodes using inter-node context-based on the LZW algorithm. However, since the dictionary used during encoding/decoding will be conditioned by the contents of only those nodes guaranteed to have been encountered previously on all paths reaching the node in question, any attempt to start decoding from some arbitrary position in the graph will therefore fail.

Before encoding the contents of the root node in the immediate dominator tree the dictionary is pre-initialized to contain entries for the alphabet $\Sigma$. The tree is then traversed in pre-order and the contents of each node are compressed using the dictionary which resulted from coding its parent. The deeper a node is in the tree, the more context we have available for compression. In Figure 3.10 we show the dictionaries that result after each node from Figure 3.3 has been processed.

For each node $x$, before decoding starts we must ensure the dictionary contains only those entries added up to the point where $x$'s immediate dominator made its final adaptation. In the previous section we established that flow passes through a graph from node to node such that we move down the corresponding immediate dominator tree one level at a time, move from the current node to one of its ancestors' children or move from the current node to one of its ancestors. Hence the LZW dictionary may be maintained appropriately at decode-time as follows.

We introduce an additional array, end_of_adapt, with capacity for storing $k$ $n$-bit entries where $k$ is the maximum immediate dominator tree depth to be supported. end_of_adapt[0] is initialized to $|\Sigma|$. The next available dictionary index, after node $x$ at depth $i$ in the immediate dominator tree has been decoded, is recorded in end_of_adapt[i]. Before the next node $y$ at depth $j$ in the dominator tree is decoded, all entries in the dictionary from end_of_adapt[j - 1] onwards are removed and then the node $y$ is decoded, adapting the dictionary while the compressed data is expanded and finally setting end_of_adapt[j] to the next available dictionary index. Thus we need to store the depth of each node in the immediate dominator tree along with its encoded data. Since dictionary maintenance must occur prior to decoding each node, this is stored as the first index in the encoded block.

Returning to our example, the compressed encoding for each node in Figure 3.10 is shown in Figure 3.11. The first value in the encoding is the $\log_2 k$-bit depth of the node in the immediate dominator tree, while the remaining values are $n$-bit LZW dictionary indices for the compressed data. Consider traversing the path $A \rightarrow C \rightarrow D \rightarrow C \rightarrow \ldots$. We begin with the initial dictionary shown in Figure 3.12. After visiting nodes $A$, $C$ and $D$ the dictionary will be as presented in Figure 3.13 where each successive
Figure 3.10: Compression of data using dictionary pre-initialized by immediate dominator tree

Figure 3.11: Graph LZW encoded data for each node in 3.3 (i)
3.4 Results

We applied our technique to a collection of PDA websites downloaded from the web in July and September 2007 using the wget program (generally invoked as wget -r -l 1000 http://homepage-address). The full list of websites we used is given in Figure 3.16, where summary information characterizing the websites’ respective file sizes is also included. Once downloaded we extracted the connectivity graph for each website using a collection of perl scripts to identify the edges exiting each file and targeting another file within that website. This is the input that was given to our prototype.
3.4. RESULTS

Figure 3.14: Connectivity graph extracted from our snapshot of wap.sciam.com (The Scientific American website, mobile edition)

Figure 3.15: Immediate dominator tree derived from the graph shown in Figure 3.14

compression implementation. In Figures 3.14 and 3.15 we show the connectivity graph and the associated immediate dominator tree for one of the websites.

We applied regular LZW to each of the files individually, and Graph LZW to the entire set of files for each website, the results of which are given in Figure 3.17. For each dictionary size ranging from $2^9$ up to $2^{16}$ entries we show the compression ratio, defined as $cr = \frac{\text{Size(compressed)}}{\text{Size(original)}}$ which shows the relative size of the compressed data compared with the original encoding of that data. Numbers below 1.0 represent compression, while numbers above 1.0 show an expansion of the original data.

While regular LZW gives its best compression when using 10 and 11-bit dictionary indices, our Graph LZW technique doesn’t achieve its best compression until larger dictionaries requiring 12 and 13-bit indices are employed. This is because while regular LZW begins compressing each node with an initial dictionary, which it needs to incrementally build before long character strings in the source can be replaced by single indices in the encoding, our graph LZW dictionary is preconditioned with plenty of content for those nodes which occur deeper in the immediate dominator tree (the most common place for a node in our benchmarks) giving rise to longer entries in the dictionary being used in the encoding earlier during compression (giving rise to compression), and hence justifying the slightly larger dictionary indices required for encoding. This clearly demonstrates that exploiting inter-page context is of great benefit to compressing graph-structured text. The summary figures account for all overheads in the compressed data encodings.

gzip [IGA], a popular compression program, employs a variant of LZ77 combined with Huffman encoding to attain excellent compression results. It is used in the widely
3.4. RESULTS

<table>
<thead>
<tr>
<th>PDA Website URI</th>
<th>#Files</th>
<th>Corpus Size (kB)</th>
<th>Average File Size (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>wap.sciam.com</td>
<td>158</td>
<td>528.78</td>
<td>3.35</td>
</tr>
<tr>
<td>news.bbc.co.uk/low/english/pda</td>
<td>151</td>
<td>882.18</td>
<td>5.84</td>
</tr>
<tr>
<td>news.bbc.co.uk/low/english/pda_sport</td>
<td>182</td>
<td>1098.16</td>
<td>6.03</td>
</tr>
<tr>
<td><a href="http://www.rte.ie/pda/entertainment">www.rte.ie/pda/entertainment</a></td>
<td>4150</td>
<td>17376.19</td>
<td>4.19</td>
</tr>
</tbody>
</table>

Figure 3.16: PDA website benchmark data

Figure 3.17: Compression ratios (compressed size/uncompressed size) achieved at various dictionary sizes for both regular LZW and Graph LZW applied to six PDA websites
deployed *modgzip* [mgz] extension available for the Apache webserver which compresses data before sending it via HTTP to clients capable of expanding data prior to rendering. We have included the average compression ratio achieved by *gzip* on our data in Figure 3.17. Whereas *gzip* is a carefully engineered compression program, our Graph LZW and the results given here are simply the kernel of an idea. Nevertheless we achieve competitive compression results (especially for wap.sciam.com and news.com.au) and the decompression routine for Graph LZW is substantially less involved than that of *gzip*. In our prototype implementation the decoder required the following simple alterations to the regular LZW decoding algorithm

- introduction of an additional array, `end_of_adapt`
- one read from `end_of_adapt` before decompression of a node to identify the position where adaptations should occur in the dictionary
- one write to `end_of_adapt` after decompression of a node has completed to update the contents of the array

To support our claim that such modifications are trivial, the changes required in a simple prototype client are shown in appendix A (*client.html*) and consist of just 5 lines (lines 10, 11, 90, 91 and 105 highlighted in bold). Later in the thesis we will see that the same changes in a hardware implementation of LZW are also easy to accommodate.

Inspired by *gzip*’s combination of an adaptive dictionary technique with a semi-adaptive code allocation phase, we investigated using a canonical Huffman code to represent the dictionary indices used by our Graph LZW algorithm. The results are shown in Figure 3.18. Using this approach we found that although the compression ratios improved over our fixed-length allocation of dictionary indices, they did not beat *gzip* on average. Careful engineering and an appropriate combination of our technique with a better form of data encoding might yield compression ratios which improve on those achieved by *gzip*, however we believe the overhead required in the decoder would not represent a justifiable penalty for the marginal improvements likely to arise.

### 3.5 Related work

To the best of our knowledge we are the first to propose compressing the content of nodes in a graph by employing an adaptive model which has been conditioned using inter-node context [GA08]. Recent work at Google investigates the use of shared
Figure 3.18: Compression ratios achieved when LZW/Graph LZW pointers are represented using variable-length canonical Huffman codes
3.6. SUMMARY

Static dictionaries for capturing inter-response redundancy when compressing data to be transmitted over HTTP [BLMM08]. While our work makes extensive use of the structure and content of the data to be compressed, the work at Google addresses the compression of dynamically generated content. As such, there is a mismatch between our approach to compression and the goal of Google’s work. All previous work we encountered in the literature which related to compression involving graphs focused on efficiently storing their structure in a more compact manner than that offered by a straightforward representation using adjacency matrices or adjacency lists. Some of the techniques targeted web graphs—which represent the link structure of the world-wide-web [BV04, LF04, MKNG06], while others targeted trees represented by a stationary ergodic source [CR96]. In some cases the authors stored the URI of a node as content along with the node and also required this data to be represented in a space efficient way [SY01]. Typically, to allow direct and efficient access into the data, standard graph data structure operations must be implemented over the compressed encoding. This is in stark contrast to our approach which only requires that valid paths be traversable through the graph. A more restricted view is that presented in [CR96] where LZW is used to compress trees, but decompression of the trees must occur in their entirety due to the breadth-first search parsing of entries in the LZW dictionary. Like our approach, all methods we have seen operating on trees and graphs assume the structure is static and any modification requires a re-computation of the compressed representation.

3.6 Summary

Ziv-Lempel methods produce optimal coding as the size of the input tends to infinity. Our technique increases the effective length of sequences being compressed within a set of textual nodes organized as a directed rooted graph structure, giving a dramatic increase in the compression ratios attained. We do this by exploiting the structural properties of directed graphs which often provide a good abstraction over data encountered in computer systems, such as pages in the world-wide-web or control flow graphs encountered by an optimizing compiler.

When compared with the LZW technique our Graph LZW algorithm trades additional analysis undertaken at compression-time for improved compression ratios. Despite our superior compression, the modifications to a standard LZW decompressor to support our approach are relatively minor and do not contribute any significant time overhead to the decoding routine. Notwithstanding the simplicity of our decoding rou-
tine, the results we have presented are comparable with those achieved by gzip which requires a considerably more complex decoder combining both static Huffman and LZ77 decoding routines.

In the following chapters we will extend the basic ideas developed in this chapter in our search for an adaptive technique for compressing object code. One view of the object code, already alluded to above, is the control flow graph – a data structure used in optimizing compilers. Using the CFG as the target program representation we adapt our graph-based compression technique developed in this chapter and consider its suitability as a technique for compressing object code.
3.6. SUMMARY
Chapter 4

Graph LZW Based Compression of Basic Blocks

In the previous chapter we described a modification of LZW which gave better compression than regular LZW for small blocks of text when organized as a directed rooted graph. Although the technique requires additional analysis at compression-time, little modification is needed to the standard LZW decompression routine. In this chapter we consider how this technique might be used to compress the object code of a statically linked program binary.

Our technique can be considered as employing a generalization of Lin's Branch Blocks (section 2.6.14) which were used in the first work on adaptive object code compression. Lin's branch blocks and the immediate dominator tree described in the previous chapter are members of a family of data structures used in optimizing compilers. In the next section we consider each of these structures in turn and the context they might offer for adaptive compression. The subsequent section then develops an associated compression algorithm for each of these structures and a decompression architecture in which the techniques can execute. There are a number of issues which complicate the correct implementation of the algorithms we describe, and our solutions to these are discussed before we present experimental results in the final section.

4.1 Sources of context

A program's object code consists of encoded machine instructions each of which can be classified as some form of branch or one that does not affect control flow. Execution moves sequentially through a program until a branch instruction is encountered. A pro-
## 4.1. SOURCES OF CONTEXT

<table>
<thead>
<tr>
<th>Label</th>
<th>Object code</th>
<th>Assembly code</th>
<th>Basic blocks</th>
<th>Branch blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>0x8C080200</td>
<td>lw t0,512(zero)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x8C090204</td>
<td>lw t1,516(zero)</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0x10080004</td>
<td>beq zero,t0,&lt;C&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x240A0000</td>
<td>li t2,0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B:</td>
<td>0x01095020</td>
<td>add t2,t0,t1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x10000002</td>
<td>420 &lt;D&gt;</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x014A5020</td>
<td>add t2,t2,t2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C:</td>
<td>0x01205020</td>
<td>add t2,t1,zero</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x01205020</td>
<td>add t2,t2,t2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D:</td>
<td>0x010A0004</td>
<td>beq zero,t2,&lt;F&gt;</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x016A5820</td>
<td>add t3,t3,t2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E:</td>
<td>0x21A50000</td>
<td>addi t2,-1</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x0100FFFF</td>
<td>b &lt;D&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F:</td>
<td>0x0100FFFF</td>
<td>b &lt;F&gt;</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x00000000</td>
<td>nop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.1: Example showing assembly source and its binary encoding, basic blocks and branch blocks are also identified

![Control Flow Graph](image)

Figure 4.2: Control Flow Graph for code shown in Figure 4.1

A program’s object code can be partitioned into a set of basic blocks—maximal sequences of consecutive instructions in which flow of control enters at the beginning and only leaves at the end. Compression schemes targeting object code must allow decompression to commence at the start of potential branch targets. An example program written in MIPS assembly language along with its associated encoding is shown in Figure 4.1. The basic blocks are labeled A through F and each branch instruction has an associated delay slot.

Our work exploits the fact that decompression from arbitrary basic block boundaries is not required. We compress each block $x$ in the context of those blocks which

![Extended Basic Blocks](image)

Figure 4.3: Extended Basic Blocks associated with CFG shown in Figure 4.2
4.1. SOURCES OF CONTEXT

are guaranteed to have been encountered whenever flow of control is transferred to \( x \). Lin et al noticed this opportunity using branch blocks rather than basic blocks as the unit of compression \([LXW04]\). A branch block starts at any target of a branch and runs sequentially through the code until the instruction preceding the next target of a branch is reached. The branch blocks in our sample program are labeled 1 through 4 in Figure 4.1. While Lin identified additional context beyond the boundary of a single basic block, there are several ways to extract further context. With reference to the branch blocks in our example, the fact that basic block \( b \) is only executed after block \( a \) has been encountered is identified, but the fact that block \( c \) too will execute only after block \( a \) has been encountered is missed. To take advantage of such opportunities requires some elementary control flow analysis from compiler design \([ASU86, Muc97]\).

The control flow graph (CFG) of a program is an abstract representation of the code which models, at compile-time, possible flow of control within the program at run-time\(^1\). Its nodes are basic blocks and an edge \( \langle x, y \rangle \) indicates that execution of block \( y \) can immediately follow that of block \( x \) in some execution sequence. The graph has a distinguished node \textit{entry} representing the program’s unique entry point. CFGs are used in many compiler and link-time optimizations and their construction is well described in the literature. The control flow graph from our running example is given in Figure 4.2.

An extended basic block \( \beta \) is a set of basic blocks \( b_1, b_2, ..., b_n \) where \( b_1 \) may have multiple predecessors and every other \( b_i, 2 \leq i \leq n \) has a unique predecessor in \( \beta \). By identifying extended basic blocks in a program rather than branch blocks, additional context will be available at the point of compression for each of the blocks \( b_i \). As a node \( x \) with only one predecessor \( y \) is guaranteed to execute immediately after \( y \) (if it executes at all), the context obtained from compressing/decompressing \( y \) may be used for compressing/expanding \( x \). The extended basic blocks from our running example are shown in Figure 4.3.

The context identified by employing extended basic blocks picks up many opportunities lost by Lin’s scheme, for example basic block \( C \) can use context resulting from encountering \( A \). By studying the control flow graph representation of the program (Figure 4.2) we can observe that basic block \( A \) will definitely have been visited prior to control passing to node \( D \) at run-time. Similarly node \( D \) will have been visited prior to control ever passing to node \( F \) at run-time. Hence context resulting from encountering

\(^1\)The term control flow graph typically refers to the representation of control flow within a single procedure. Here we use the term to mean an interprocedural control flow graph of the supergraph form \([Mye81]\).
4.2 Compression techniques

In the previous section we described a variety of sources of context which previously have not been considered for use in implementing an adaptive code compression scheme. The following sections outline how context identified earlier may be used to build an LZW-based technique for code compression at the level of a basic block, branch block, extended basic block and immediate dominator tree. As each of the increases in context from basic block through dominator tree contains all opportunities identified by its predecessors, the dominator tree-based scheme should be considered the ultimate result. In all but the basic block scheme it is not possible to commence decoding at any arbitrary basic block boundary because the dictionary used to encode/decode blocks is based on those guaranteed to have been encountered previously on any valid execution path leading to a given block.

4.2.1 Basic block

Before coding each basic block the LZW dictionary is reset. At decompression-time the dictionary must be reset prior to decoding the first instruction in a block. The compression scheme is transparent to the processor from which instruction memory requests originate, hence the processor operates as if it is executing a normal uncompressed program. To enable this charade the decompression engine must reconstruct the uncompressed memory address space and its contents as requests for memory lo-
Block A

Block B

Block C

Regular address space
(as seen by processor)

Compressed address space
(as seen by memory subsystem)

Figure 4.5: Mapping from original address space to the compressed address space

Applications are sent from the processor (which is using the original address space) to the memory subsystem which stores basic blocks at different locations (due to their change in size) when they are compressed as demonstrated in Figure 4.5.

To locate a block of compressed code in memory we adopt the solution proposed in [LBCM97] where a post-compression pass rewrites program jump tables and branch instructions to point to the destination block in compressed memory essentially doing a form of address relocation [Lev99] albeit at compression-time rather than link or load-time.

Although jump tables can easily be rewritten to point to the new location of a block in compressed memory, patching branch instructions which have been LZW encoded requires them to be re-compressed possibly altering the encoded block size, causing movement in the location of other compressed blocks and hence requiring other branch instructions to be rewritten. It is not clear what requirements exist to achieve a fixed-point solution to this problem, so individual branch instructions which require updating are left uncompressed in the instruction stream. In this way patching instructions after compression requires a single pass which effects neither the dictionary nor the size of compressed blocks. To achieve this for LZW we reserve a single dictionary index...
4.2. COMPRESSION TECHNIQUES

<table>
<thead>
<tr>
<th>Label</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry:</td>
<td>00</td>
<td>(\text{inst}_1)</td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>(\text{inst}_2)</td>
</tr>
<tr>
<td></td>
<td>08</td>
<td>(\text{inst}_3)</td>
</tr>
<tr>
<td></td>
<td>0C</td>
<td>(\text{inst}_4)</td>
</tr>
<tr>
<td>join:</td>
<td>10</td>
<td>(\text{inst}_5)</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>(\text{branch}(-8))</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>(\text{inst}_7)</td>
</tr>
</tbody>
</table>

Figure 4.6: Encoding format for a basic block which contains no control flow instructions (entry), and one which does (join)

When we encode blocks using LZW we need a mechanism to indicate to the decoder at which points it should reset the dictionary so that decompression can safely commence at branch targets. For this we reserve the pointer consisting of all 1s and place it at the end of each encoded block. This reset code may be omitted if a basic block contains a relative branch instruction, since the branch terminates the basic block one instruction later and can be trivially identified during decompression in hardware. Often this encoding will result in a non-word aligned block. We pad these blocks with additional 1s until they become 4-byte aligned to ensure transparency of the compression scheme to the underlying MIPS processor which requires all branch targets be aligned to these boundaries. Figure 4.6 shows the form of a simple program composed of two basic blocks and demonstrates the encoding format for each block.

Architecture

Code compression schemes at the granularity we work with are typically classified as either pre-cache or post-cache techniques, as determined by the location in the memory hierarchy at which decompression occurs (Figure 4.10). In our basic block based techniques (those presented in this chapter) we position the decompression engine between the cache and the processor; the reason for not positioning it between memory and the instruction cache will be discussed in the next section.

Initially the processor will request an address that has never been seen before, which is the first instruction in a block. The decompression logic will expand that block from memory into a temporary buffer and return the requested instruction to the processor. Each time an instruction is executed the next one in sequence is requested...
from memory unless the last instruction was the final instruction in a block—in which case the destination of a branch may be requested. When the last instruction in a block is requested the processor will either issue a request for the fall through block (which will be one address greater than the previous instruction accessed) or it will request a branch destination (which we previously patched to jump directly to the compressed location in memory). This is shown in our pseudo-hardware description language of Figure 4.7.

A few issues remain: if a block ends on a conditional branch to a (compressed) location X when the fall through address of the current uncompressed block is also X, we cannot determine what code should be used to continue execution of the program if the processor next issues a request for X (this is depicted in Figure 4.8). This situation should be detected at compression-time and avoided by manipulating the program layout in compressed memory by introducing appropriate padding between blocks (Figure 4.9), or by having the compiler lay out the basic blocks in a different linear order.

So far, all branches target locations in compressed memory. We can determine fall through blocks and continue decoding them by storing appropriate state during decoding in the decompression engine as outlined above. Therefore, as far as the processor is concerned, a fall through is a real fall through (it just continues to increase its program counter). Since a block can be located at different addresses in uncompressed memory (one via a branch to its compressed location and another via a fall through from a previous block), it would be impossible to correctly patch PC relative branch instructions with this setup.

For example, consider again the code in Figure 4.9. Basic block B can be reached by falling through from basic block A when the BEQ in the instruction buffer is not taken, giving B an address (as seen by the processor) of 0x1C, while any code which directly jumps to execute block B will use its compressed address 0x12. If we assume Block B contains a relative jump instruction then in that block’s encoding, the jump will have been left uncompressed and the immediate value for the jump patched to jump to the compressed location of its destination. Clearly such a jump cannot be patched to jump to the appropriate location in the compressed memory since, at runtime, the processor can potentially see the block via two different addresses.

Our solution to this problem is as follows: automatically generate an unconditional branch to the compressed location of a fall through for each block and position it after the last instruction in the decoded block during decoding. This will force the processor to use the same address for every block, and allow relative branch instructions
4.2. COMPRESSION TECHNIQUES

```c
-- buffer used to store decoded instructions.
address low = 0;
address high = 0;
instruction ibuffer[SIZE] = {};

-- state used by decompress() to identify the next
-- address it should request from main memory during decoding.
address next_request = 0;

-- fetch takes an address issued from the processor and
-- returns the instruction at that location. If the
-- requested address is not already in the instruction
-- buffer the basic block starting at that location is
-- first decompressed.
instruction fetch(address program_counter)
{
    if(low <= program_counter < high)
    {
        -- instruction is already available in ibuffer.
    } else if(program_counter == high) {
        -- we’ve just fallen through to the next basic block
        -- we want the block located at ‘next_request’.
    } else {
        next_request = program_counter;
        low = program_counter;
        high = program_counter + (decompress() << 2);
    }

    return ibuffer[program_counter - low];
}

-- decompress decodes a single basic block of instructions
-- starting at next_request. The end of the block is determined
-- by either an ‘end of basic block’ code, or by identifying a branch
-- instruction in the decoded stream. The return value is
-- the number of instructions inserted into the instruction buffer.
int decompress()
{
    while(!end of basic block)
    {
        ...
        word tmp = memory_read(next_request);
        next_request++;
        ...
        ibuffer[...] = ...
        ...
    }
    ...
}
```

Figure 4.7: High level description of the process of fetching an instruction when requested by the processor
4.2. COMPRESSION TECHNIQUES

Figure 4.8: If the processor next issues a request for address 0x1C, should the fall through block be used or the compressed block located at 0x1C?

Figure 4.9: Appropriate program layout to avoid the problem in Figure 4.8
4.2. COMPRESSION TECHNIQUES

(i) Pre-cache decompression logic (decompress only on a miss)

(ii) Post-cache decompression logic (increase effective size of cache)

Figure 4.10: Possible positions for decompression engine

be appropriately patched. Our inspiration for this approach came from the work by Smith [JS97] on the removal of LATs, where the final 4-bytes of an uncompressed line are reserved during compression to store a similar unconditional branch which is automatically generated during decompression.

A final issue with making this scheme transparent to the underlying processor is its use of call/jal and return instructions, which implicitly store the return address (as seen by the processor in uncompressed space) on the stack only to be directly jumped to at some point later in the programs execution. If our decompression engine ignores such technicalities it will assume the incoming address is that of a compressed basic block in memory, and attempt to begin decoding there. Instead what is needed is to maintain a stack which maps from expected return addresses to their compressed locations, which is inspected by the decompressor whenever an address is presented to it. A new entry is pushed when a call instruction is issued to the processor, and it is popped when the return address is presented. This solution assumes the program is well behaved with regards to its calling structure and that it does not use any tricks like that employed on the x86 to obtain the current program counter into a register by exploiting the call instruction; requires conditional (predicated) procedure calls to be disallowed, and does not allow for any out-of-order of speculative issuing of call instructions (ie. a call must become a barrier in the instruction fetch).

This general architecture is assumed throughout the remainder of this chapter—only the decompression engines maintenance of state need change as we progress from basic blocks through to immediate dominator trees.
4.2. COMPRESSION TECHNIQUES

Positioning of decompression unit

In the previous section we stated that the decompression unit for all our basic block oriented schemes was to be positioned between the instruction cache and the processor. In Figure 4.11 we show two blocks from a control flow graph (Block A and Block B) where it is assumed execution of the program begins at the first instruction in Block A. For simplicity we assume a system which uses a 64-byte direct mapped cache, with lines of 8 bytes. The low three bits of an address requested from the cache determine the offset of data in a line to be returned, the next three indicate the index in the cache that a given address will be stored at, while the remaining upper bits of the address form the tag which is used to determine whether a hit or miss occurs in the cache when it is accessed.

Assuming the path through the control flow graph executes Block A, Block B and then takes the back edge to Block A once again we now demonstrate why the decompression unit could not be placed between the memory storing instruction data and the instruction cache. The processor begins executing and requests Instruction 1 from the cache. As no data has previously been loaded into the cache there is a miss for this data stored at address 00000000002, and so a decompression unit is invoked which decompresses the basic block at this address and places it into the instruction cache filling lines 0002 and 0012. Instruction 1 is returned to the processor, and the remaining instructions are executed in order before control passes to Block B. When Instruction 5 is requested there is again a miss in the cache, as the data stored in line 0012 is that of Instruction 3 and Instruction 4. In response the decompression engine will locate the compressed Block B in memory based on the requested address 10000010002, decompress it and insert it into the cache — evicting Instruction 3 and Instruction 4 from line 0012. Upon flow returning to Block A for a second time, the first two instructions will be found in the cache before we encounter a problem. Execution of Block B evicted some trailing instructions belonging to Block A, and when a miss in the cache occurs for these instructions it is non-trivial to fetch them from compressed memory since the mapping stored only translates from uncompressed basic block addresses to their location in compressed memory.

A second problem associated with positioning the decompression unit between memory and the instruction cache occurs when the code for multiple basic blocks occupies a single cache line. When a single given block is decompressed and inserted into the instruction cache some of the data in the lines inserted may be uninitialized as it should be taken from a preceding or succeeding basic block in the uncompressed
4.2. COMPRESSION TECHNIQUES

Figure 4.11: An example showing why our basic block oriented schemes must be considered as post-cache decompression algorithms version of the program.

To address these issues would require significant re-engineering of the instruction cache, greatly complicating the implementation of such compression techniques.

4.2.2 Branch block

Our branch block compression scheme is similar to that described by Lin [LXW04], apart from the technique employed for resolving the uncompressed/compressed address space. While Lin uses a lookup table which maps from uncompressed addresses to their compressed counterparts (requiring storage of two addresses per branch block), we employ the patching technique described in the previous section. Rather than resetting the dictionary at each basic block boundary, it is reset after decompressing each branch block (for which the all 1’s LZW index is again reserved). Hence an encoded branch block consists of a sequence of LZW encoded machine instructions mixed with escape indices immediately followed by branch instructions which are finally terminated by the all 1’s index before being padded with additional 1’s to give word alignment.

4.2.3 Extended basic block

Our extended basic block compression scheme encodes each basic block in the extended basic block individually. Compression of each block begins with the model which resulted from compressing its parent in the extended basic block and the dictio-
nary is reset only upon entering a new extended basic block at run-time. Each basic block contains an end of block indicator (the all 1’s index) so that the decompressor knows at what point to cease decoding data from memory (lest it allow the model fall out of step with that used during compression). As with the individual basic block scheme this indicator may be omitted if the block ends on a branch instruction which is easy to identify in hardware. An additional LZW pointer is reserved and prefixed to the header of the extended basic block to indicate when the dictionary should be reset.

### 4.2.4 Immediate dominator tree

Compression of object code based on the immediate dominator tree employs the algorithm introduced in the previous chapter. Since dictionary maintenance occurs before decoding a node, the depth of the immediate dominator tree is stored as the first pointer in the encoded data. As with the other schemes in this chapter, one LZW pointer is reserved as an escape code for branch instructions and a second for use as an end of basic block indicator for those blocks which do not end on a control flow instruction.

### 4.3 Results

To determine whether the adaptive object code compression techniques described in the previous section give any evidence of being practical, we implemented their compression routines and applied them to a number of benchmarks. The eight programs from the MiBench and MediaBench benchmark suites [GRE+01, LPMS97] which we selected (cjpeg, crc32, djpeg, ispell, rawcaudio, rawdaudio, rijndael and toast) are representative of programs which are found in embedded systems, and have been the target of previous work on code size reduction experiments. While most previous work on object code compression does not need detailed information on the flow of control through the program being compressed, thus allowing the choice of an arbitrary architecture on which to perform their experiments (subject to toolchain availability), we do. This complicates the selection of a target machine architecture due to the difficulty of modifying an existing compiler to output the control flow graph associated with the code it is compiling, or alternatively recovering a program’s control flow graph from the compiled code. Bearing this in mind, the next section describes our choice of target architecture on the basis of extracting a reliable, safe but conservative control flow graph for the code to be compressed.
4.3. RESULTS

4.3.1 Selection of target architecture

Previous work on object code compression has targeted a wide range of computer architectures, with the most popular including RISC machines (Arm, MIPS, PowerPC, sparc), CISC (x86), DSPs (tms320c25, analog device’s sharc) and VLIWs (tms320c6x). The main factor affecting our choice of target architecture for the experiments presented throughout the thesis was the availability of suitable tools for compiling benchmarks and recovering the control flow graph associated with the final statically linked program binary.

Initially we considered targeting Texas Instrument’s TMS320C6x VLIW processor [Ins00], owing to its popularity in recent literature on object code compression. Unfortunately the only good toolchain for the architecture is provided by the vendor and is not made freely available. A one month trial is available for evaluation purposes, and we used this in our evaluation of the architecture as a target for our techniques. Recovering control flow information from the scheduled assembly code generated by TIs compiler is not an entirely straightforward task. Ideally the compiler would be instrumented to output information about the control flow graph for an entire program, however as the source is not available this is clearly not an option. Even if the source were available, frequently the run-time libraries linked to code are provided not as source code, but rather as binary code for which the compiler will not have the control flow graph. To recover control flow information for a statically linked binary, some researchers investigating code compression simply profiled the object code using an instrumented version of a simulator for the architecture, such as SimpleDSP [JRM03]. This approach clearly does not give rise to safe information and without full coverage of the code using the benchmark data, our experience shows that it is unlikely to give results that can be trusted. We prefer to adopt an approach based on static analysis of the code. While CFG construction algorithms are described in the compiler literature, they typically target a compiler’s internal representation where indirect branches (which jump to a target whose address is stored in a register) either do not occur or whose targets are known. For compiled code we need to make conservative assumptions about where the target of such a jump can go — potentially to the start of every basic block in the program. The TI architecture has 5 branch delay slots, which make the problem of finding the first and last operation in each block difficult because branches are allowed to occupy the delay slot of other branches, thus causing control to transfer after it has already begun executing a new path of code. An excellent discussion of the problem, and an algorithm to recover a safe representation of the CFG
4.3. RESULTS

under these circumstances is presented in [KDCW].

Owing to the unavailability of free tools and the likelihood that reconstruction of the CFG would lead to very small basic blocks and an explosion of control flow graph edges (and hence give little opportunity for our scheme to do well), we switched targets when we discovered Diablo, a link-time optimizer for Arm, x86 and MIPS object code [BSP+04, DBKC+03, MDSDB+04]. The tool’s source code is freely available, the architectures on which it operates are all targets of the free open source GCC toolchain, and as a preliminary step to optimization Diablo recovers the CFG for the application being processed. Of the available architectures, we selected MIPS to investigate our technique on. The .text section of ARM binaries does not exclusively contain instruction data, but instead a mix of instruction data and address pools which contain blocks of data (typically addresses) which can be loaded into a register using a pc-relative load instruction, avoiding the overhead of using two instructions to load a 32-bit value into a register or sacrificing one of the machine registers as a global offset table pointer, as in MIPS. With proper handling our techniques can model accesses to the distributed ARM address pools such that they are correctly decompressed when accessed, but we prefer MIPS as a simpler alternative for prototyping. We chose not to target x86 code due to the complexity of decoding x86 instructions; in our prototypes we need to identify control flow instructions, and this is a simple mask and comparison operation on the MIPS ISA due to the fixed-length instruction encoding.

As input, Diablo takes a statically linked program binary and the individual object files from which it was constructed. It links the object files together and ensures the constructed binary is the same as that generated by the native linker, thus ensuring it has interpreted all relocation and symbol information properly [SPC+07]. The relocation information used during the linking step is not necessarily present in the linked program produced by the native linker (as relocations etc. will have been resolved), and by re-linking the program from the original relocatable object files additional information useful in construction of the CFG can be obtained from the objects’ individual symbol tables. Following this step the linked program is disassembled and the control flow graph reconstructed. Optimizations are applied, after which the CFG is relinearized and final addresses are assigned to each node. All symbolic addresses are translated into actual addresses in the binary again. Finally the code is assembled and the final program written to disk.
4.3. RESULTS

Extraction of control flow graphs

Within Diablo, construction of the CFG from the disassembled instructions proceeds as follows. Basic blocks are detected using the leader algorithm [ASU86, Algorithm 9.1] and the relocation information in the following way: the target of any direct conditional or unconditional control flow instruction is the start of a basic block, as is the instruction following any such instruction. In addition to these leaders, all computable addresses (i.e., addresses that can be stored in a register) are conservatively assumed to define entry points to basic blocks and are added to the set. As the compiler generated code is position independent, all control flow targets are either addressed relative to the program counter or are relocatable and information on resolving the relocation is present in the object file. Any unrelocated computations on code addresses can be handled by employing pattern-matching techniques to identify specific compiler idioms used during code generation [SBC+06]. Each basic block then contains all instructions starting at its leader and running up to the next leader in the code. Edges for direct control flow are easily added to the appropriate blocks, and for indirect transfers the instructions at any relocatable addresses are assumed to be the target. By initially assuming that all relocatable code addresses can be the target of indirect control flow a very conservative CFG is built that can contain many unrealizable execution paths. However, information gathered using analyses on this representation can be used to remove unrealizable paths from the graph [SBB05, Section 3.3], [DEMS00, Section 2.2.1].

Diablo uses a supergraph-like whole program interprocedural CFG [Mye81]. Identifying procedures and classifying interprocedural control flow does not necessarily correspond to our simple call-return notion in the source code, due to tail merging and similar techniques which cause non-call-instruction interprocedural transfers. For this reason dummy return nodes are added to each Diablo procedure (which does not necessarily correspond to a procedure from the source), and various edges are added such as link edges connecting a call site to its return site, interprocedural jumps and fallthroughs, and compensating edges which are to non-call interprocedural control flow what return edges are to calls.

We modified the MIPS port of Diablo to traverse the CFG and output information about the boundary between adjacent basic blocks and the control flow graph edges between them. The initial control flow graph constructed by Diablo’s MIPS port cannot be directly related back to the object code from which it was constructed, due to elimination of unreachable and dead code during CFG construction, and an initial mis-
4.3. RESULTS

representation of control flow information (which is subsequently corrected) and the address-agnostic internal representation of control flow instructions. These are non-issues after the binary has been optimized and the final code layout determined. It is not until this point that we invoke our cfg-export routine, producing from an unoptimized binary its optimized counterpart and the associated control flow graph.

Unknown control flow (other than that generated by software interrupt instructions) is modeled by two different functions; call-hell and hell. The entry to call-hell has as predecessors the source of all indirect procedure calls, and the corresponding return sites as successors of the call-hell return node. There is a single call edge connecting the entry of call-hell to the entry block of hell, and a corresponding return edge from hell’s return node to call-hell’s return node. Other unknown control flow caused by indirect jumps is represented by interprocedural jump (ipjump) edges to hell’s entry node and a corresponding compensating edge to the return node of the function from which the ipjump originated (from hell’s return node). The entry of hell is connected to each relocatable procedure by a call edge, and the return node of each such procedure is connected to hell’s return node by a return edge. This two level representation of unknown control flow is useful for modeling architecture specific calling conventions when implementing dataflow analysis. For example, ABI conventions require the caller save certain registers while the callee save others during procedure calls [SCO96]. In the CFG it can be useful to separate calling an unknown procedure (call-hell) from the notion of a procedure being called by an unknown caller (hell) when computing interprocedural live variables, and the different hell-nodes can have different properties attached for modeling worst-case assumptions for the analysis. In our code we have no need for such analyses, and the unknown control flow representation is flattened into a single hell procedure.

The reader must bear in mind that almost all existing work on code compression does not apply link-time optimization before compressing their fully linked binaries. As such these binaries which have not been optimized at link-time may also have been compressing quite a lot of dead and unoptimized code, leaving extra redundancy in the input and potentially allowing their compression schemes achieve good results that probably wouldn’t hold at the same level when applied to link-time optimized code. We will discuss this issue in section 4.3.6.
4.3. RESULTS

4.3.2 Experimental setup

In the following sections we investigate four questions:

- Are the branch blocks used by Lin representative of branch blocks from RISC code? (i.e. could Lin’s technique be applied in a RISC environment)

- What context is available at each data structure granularity? We would expect this to be a predictor of the compression achievable if the context is actually representative of data in the application.

- What compression is achievable? i.e. are these techniques practical for obtaining code size reduction, and does the context accurately predict when compression occurs?

- What is the impact of compilers and link-time optimization on the compressibility of code?

In answering the first three questions, we compiled our eight benchmarks using GCC 3.3.1 with size optimization enabled and then compacted using version 0.3 of the Diablo link-time optimizer. We began by reproducing the results of Madou [MDSDB'04], and then extended Diablo to output information about the basic block boundaries in the optimized binaries along with information about their associated control flow graphs as described in the previous section. For the fourth question, we chose an alternative compression technique which does not require control flow information and applied it over all combinations of compiler/linker optimizations to consider the results.

4.3.3 Basic block / branch block size distribution

This section presents results of comparing the distribution of basic block and branch block sizes over the collection of benchmarks we have used from the mediabench and mibench suites, shown in Figure 4.12. It is clear from the distributions presented for each benchmark that moving from the granularity of a basic block to that of a branch block increases the average block size. The average basic block size is approximately 4.9 instructions (falling within the bounds of 4-7 instructions observed in [PH90]), while the average branch block size is 7.5 instructions. These results clearly demonstrate that the VLIW code which Lin et al. [LXW04] used for their experiments (with an average of 80 branch blocks per program, with average size of 113 instructions) are in no way characteristic of RISC object code optimized for size, and hence that their technique would be inappropriate for application on such architectures.
Figure 4.12: Basic and Branch block size distributions

- **Fast**
  - Rawdata
  - Insdata

- **Slow**
  - Rawdata
  - Insdata

- **Ctc**
  - Rawdata
  - Insdata

- **Ctp**
  - Rawdata
  - Insdata

**Results**
4.3. RESULTS

4.3.4 Available context

In Figure 4.14 we show the available context (average number of instructions which statically precede any given instruction) at each block granularity. As we expect the progression from basic block through to immediate dominator tree improves the amount of context available as each structure includes all context identified by its predecessor. Using context identified by an immediate dominator tree uncovers over 3.5 times the context that was available using Lin’s branch blocks. The significant jump in context shown for the immediate dominator tree can be attributed to the entry code for a program which occurs before the main application begins executing; see Figure 4.13 for an example idom tree from our benchmarks.

4.3.5 Compression results

Here we present results only in terms of code size reduction, leaving an implementation and evaluation of the performance overhead incurred in employing the techniques to future work. We experimented with varying the dictionary size/index length (from 9 to 12 bits) when applying each of our techniques described earlier. Figure 4.15 present the compression ratios (compressed size/uncompressed size) achieved in our experiments for all benchmarks using 9-bit LZW dictionary indices. A compression ratio of 1 indicates no size reduction, greater than 1 expansion, and less than 1 compression. All overheads as described in the relevant sections have been accounted for when computing these results, such as end of block indicators and padding of blocks to word alignment. We do not show our results for larger dictionaries here since none of them showed an improvement over those obtained using the smallest possible dictionary; their results can be found in appendix C. We rationalize this somewhat unexpected behavior as follows:

The control flow graph presented by Diablo is conservative in its representation of unknown control flow, resulting in a shallow dominator tree. The CFG reconstruction algorithm employed by Diablo assumes that indirect branches may pass to any relocation target in the object code. This impacts the compression attainable using our scheme since increasing the dictionary size cannot provide additional context for nodes near the top of the immediate dominator tree, the most common location for a block during our experiments. A tighter integration with the code generation tool chain would provide less conservative information, giving rise to a deeper immediate dominator tree.
Figure 4.13: CRC immediate dominator tree (compiled with -Os and link-time optimized with Diablo)
4.3. RESULTS

Figure 4.14: Context available at each block granularity

Figure 4.15: Compression ratio for 9-bit LZW applied at various variable-length block granularities
Our results show that as we progress from the fine granularity of a single basic block towards that of an immediate dominator tree for the entire program that compression generally improves (that is, the compression ratio decreases). We showed in the previous section that this progression always results in an increase in context, so this improvement in compression results is not to be unexpected. The *disimprovement* when we move from a branch block to an extended basic block may seem unexpected in light of this observation but can be attributed to the additional overhead of the encoding scheme described in section 4.2.3 where we employ context from the extended basic block for compression, but must endure the overhead of padding each basic block in the extended basic block to word alignment, while such overhead is only required once in the branch block scheme where a single branch block (which consists of several basic blocks) is encoded as a single unit.

Despite the improvement in compression as we progress from using fine grained local context to using a coarser grained global context the results we show here do not constitute any code size reduction. The combined overhead of padding all compressed blocks to word alignment rather than byte alignment seems wasteful, and is a direct consequence of our choice of making all jump instructions target addresses in the compressed address space, combined with the MIPS architecture ISA which requires such targets to be aligned on word boundaries. Furthermore as such jump instructions occur relatively frequently in the static object code the use of a full LZW index as an escape code followed by the 4-byte original instruction does little to alleviate the situation.

In the next chapter we adopt an alternative solution to the address mapping problem which shifts the blocks of code we compress from variable-length basic block-oriented ones to fixed-length cache lines.

### 4.3.6 Impact of compile-time and link-time optimization on compression

In this section we discuss the impact of compile-time and link-time optimizations on the compressibility of a computer program. Since compiling for size and compacting at link-time reduces the size of the input data to a code compression routine, we might expect the compression routine to have a harder time at compressing the data. We performed experiments by using the Markov models suggested by Lekatsas and Xie for code compression[Leke00, Xie02], using 1x1, 32x1, 4x4, 32x4 and 32x32 models as characterizations of five different compression schemes, and conditioned each model
4.3. RESULTS

using our benchmarks which were optimized at each optimization level offered by GCC (-o0, -o3, and -oS) and subsequently compacted using the Diablo link-time optimizer [MDSDB+04].

An $n \times m$ Markov model consists of $n$ rows each containing $m$ states. The node at index $j$ on row $i$ has its left and right children on row $(i+1) \% n$ at offsets $\lfloor j/2 \rfloor$ and $m/2 - \lfloor j/2 \rfloor$ respectively. Edges to the left represent the probability of encountering a 0 in the input from this state, while edges to the right represent the probability of encountering a 1 in the current state. An example of a conditioned 16x4 model is shown in Figure 4.16.

We computed the entropy of each model after it was conditioned (shown on the left of Figures 4.17 - 4.24), which gives the number of bits each original bit in the input can be recoded using if we employed an entropy coder (e.g., an arithmetic coder). The entropy for each model is computed as:

$$E = - \sum_{s \in \text{States}} p(s) \cdot \{ p(1|s) \cdot \log_2 p(1|s) + p(0|s) \cdot \log_2 p(0|s) \}$$  \hspace{1cm} (4.1)

where $p(s)$ is the probability of being in state $s$, which is scaled by the binary entropy function for that state.

While the ability to re-encode the input at this reduced bit size does not account for all overheads required to implement a code compression technique (block level access is required which means people typically pad compressed blocks to byte alignment, and LAT tables are typically needed for indexing the compressed code), it can be considered an unsafe lower bound approximation on the gains which might be achievable using the given probability distribution. This predicted size is shown on the right of the figures.

We now make a number of observations about the compression ratio statistic often used to quantify the effectiveness of code compression algorithms, using the entropy as a predictor of compression ratio:

- The entropy is roughly inversely proportional to the size of the input data. Hence the entropy reported for a given scheme can be significantly different based on the preparation of the input binary program (which can be affected by its optimization level and whether or not link-time optimization are applied). For a good example of this see the cjpeg and djpeg benchmarks on the 32x4 models, where the entropy for the one technique varies by as much as 20% from each other. Thus it is important to consider the compression ratio or entropy results
Figure 4.16: Conditioned 16x4 Markov model for crc32 benchmark (optimized with -o0). p(0) is shown as dotted lines, p(1) as solid lines.
4.3. RESULTS

Figure 4.17: Entropy results for cjpeg

Figure 4.18: Entropy results for djpeg

Figure 4.19: Entropy results for crc

Figure 4.20: Entropy results for ispell
4.3. RESULTS

Figure 4.21: Entropy results for rawaudio

Figure 4.22: Entropy results for rawdaudio

Figure 4.23: Entropy results for rijndael

Figure 4.24: Entropy results for toast
4.3. RESULTS

reported in the literature in the context of how the program has been prepared for compression.

- Regardless of whether the entropy significantly changes based on the preparation of the input or not (see the rawcaudio 1x1 model results for an example where the entropy was not much affected by the form of binary preparation), it is important to remember that some of the inputs used to condition the model are smaller than others and the entropy itself does not predict how to best prepare the binary for reduced size – the final compressed size is the real metric of interest here.

- If we use differently prepared inputs on different models, as for the case of different inputs on the same scheme, entropy results do not predict which combination will result in the smallest final binary program. For example, in djpeg's 4x4 model we get an entropy of about .65 for -o0 and, what appears to be a 'worse' entropy of .78 for the 32x1 model when trained with the -o3/link-time optimized binary. In fact, the second scenario results in the smallest binary despite superficially employing the worse compression technique.

In summary, entropy/compression ratio is a fair predictor of final size and can be used to compare different compression techniques only if the exact same input is given to each scheme under investigation. For example, in the djpeg results the compression ratio across each of the models decreases for, say, the -o0 without link-time optimization prepared binary. Looking at the (estimated) final compressed code size we get continuously smaller resulting binaries as predicted by the entropy – indicating that each successive compression technique is an improvement over the previous. These observations regarding the compression ratio statistic for evaluating code compression techniques are in agreement with similar observations made in [RS03] where a specific code compression technique was evaluated on VLIW code compiled using a combination of different optimization levels:

It has been found that code compression, and in particular compression ratios, must always be considered in the context of compiler optimization parameters. Compression ratios do differ from one parameter combination to another and unoptimized code seemed to generate higher compression ratios. However, the best compression ratio is not always an indication of best overall size. In general, to obtain the smallest overall size after compression, a compression scheme should be applied to already size-optimized code.
4.3. RESULTS

Bearing these points in mind, the compression ratio results we present here should not directly be compared against those published elsewhere in other research on code size reduction techniques. The difficulty with comparing quantitative results published in the literature is the diversity of the experimental setups employed, as has previously been noted [ABFG+03]. Unfortunately, given the extensive literature on code size reduction techniques [vdWL04], it is impractical to reimplement existing methods to provide a fair comparison each time a new technique is developed. Consider, for example, that the schemes we reference in this thesis have targeted the Compaq Alpha [DE02, DE03, DEMS00], MIPS [MDSDB+04, WC92], A RISC like abstract machine [CM99], ARM [LBCM97, YSO+97], Texas Instruments TMS320C6x [LXW04, Xie02], PowerPC [KMH+98], i386 [LXW04] and PDP 11 [FMW84]. Each of these machines employ a different Instruction Set Architecture, with a distinct encoding, providing different opportunities for compression. The benchmarks evaluated vary from one investigation to another, with some combination of programs from SPEC ’95, SPEC ’92, MiBench, MediaBench and vendor supplied code in the mix. Preparation of the input binaries employed a spread of optimizations ranging from none at all, through ‘normal’ optimization levels, to size optimization. Indeed some papers do not discuss the preparation of the binaries they compress in any detail. Taking into account the fact that different compilers were used (various versions of GCC, the massively scalar compiler, and vendor supplied compilers), it is obvious that the results obtained across papers are simply not amenable to direct comparison. Furthermore some of the techniques described compress statically linked programs, while others do not include any library code in their results. Indeed while some researchers have applied program compaction prior to compression as we do, the vast majority have not.
4.3. RESULTS
Chapter 5

LZW Based Cache Line Compression

In the previous chapter we introduced a family of code compression techniques which operate at the granularity of variable-length blocks of code, where blocks are decompressed between the cache and the processor at run-time. On fully optimized code we saw that increased context helped improve the compression ratios achievable, but in practice the schemes didn’t result in code size reduction. This was attributed to the overheads associated with the choice of encoding scheme for the variable-length basic blocks which needed to be padded to word alignment and to our handling of the mapping from the uncompressed address space to the compressed address space by leaving certain instructions uncompressed. This chapter moves from variable-length blocks to fixed-length cache lines and positions the decompression unit between memory and the instruction cache as in Wolfe’s compressed code RISC processor. In doing so the block padding overhead can be reduced and the mapping from one address space to another can be stored efficiently using Wolfe’s LAT. We first consider using LZW in place of Huffman coding in Wolfe’s CCRP before considering a context-based approach throughout the remainder of the chapter.

5.1 Individual cache lines

We propose replacing the Huffman coding technique employed by Wolfe’s CCRP with an LZW-based scheme. Instead of storing a static or semi-adaptive Huffman coding table in the decompression unit we compute at compression-time an LZW dictionary which will be appropriately initialized and adaptively maintained at run-time during the execution of the program.

As a starting point we investigated the simple idea of compressing each cache line
individual using the LZW algorithm. Before coding each cache line the LZW dictionary contains initial entries for the alphabet \( \Sigma \) of unique bytes 0..255. The cache line is then coded, padded to provide byte alignment, and stored in memory. As before a LAT table is used to locate compressed code in memory. At run-time whenever a cache miss occurs the LZW dictionary is reset, the compressed code is located in memory using the LAT, and the cache line is expanded and placed in the instruction cache.

Unfortunately, except for cache lines of length 64 bytes or greater, the technique results in expansion rather than compression of the input. In the following sections we introduce a new approach where a cache line will be compressed using the context obtainable from those other cache lines guaranteed to have been encountered on all of the possible run-time paths leading to an instruction cache miss for the line. To this end we will employ a new data structure we term the **compulsory miss tree** which identifies precisely those cache lines of interest. This is described in the next section.

### 5.2 Compulsory miss tree

When a processor is unable to locate the data it requires in its cache, a cache miss occurs. These misses can be characterized as either compulsory, capacity or conflict misses. Capacity and conflict misses occur as a result of the physical size of the cache and the associated block placement strategy. A compulsory miss occurs on the first access to a piece of data through the cache, that is, when the data has not previously been loaded before [PH90].

In this section we present an algorithm to construct a compulsory miss tree for a given program and instruction cache. Nodes in this tree represent individual instruction cache lines in the program which may be loaded from memory. When any type of miss occurs for a given node in the tree all ancestors of this node will previously have caused their compulsory miss in the cache to occur.

The construction proceeds as follows: first, the boundaries between basic blocks in the object code are identified. Then we determine the boundaries in the object code between cache line data for loading into the instruction cache. Next the CFG is
5.2. COMPULSORY MISS TREE

```plaintext
construct_compulsory_miss_tree(): CacheLine is

    BasicBlock idomTreeRoot := build_immediate_dominator_tree()
    CacheLine cmTreeRoot := first_cache_line(idomTreeRoot)

    cmTreeRoot.processed := true
    cmTreeRoot.cmTreeDepth := 0
    cmTreeRoot.cmTreeParent := nil

    for each BasicBlock b in pre_order_traverse(idomTreeRoot)
        for each CacheLine cl in touches-cl(b)

            if cl.processed = false then
                BasicBlock a := common_idom_ancestor(touches-bb(cl))
                cl.cmTreeParent := deepest_cmt_cl(a)

                if cl.cmTreeParent = nil then
                    cl.cmTreeParent := deepest_cmt_cl(a.idomTreeParent)
                end -- if

                cl.cmTreeDepth := cl.cmTreeParent.cmTreeDepth + 1
                cl.processed := true
            end -- if

        end -- for each
    end -- for each

    return cmTreeRoot
```

```
def deepest_cmt_cl(x):
    -- Returns the cache line deepest in the compulsory miss tree from
    -- the set touches-cl(x). Returns nil if no cache line in
    -- touches-cl(x) has been added to the tree.

def common_idom_ancestor(x):
    -- Returns the deepest common ancestor of the set of basic blocks x
    -- in the immediate dominator tree.

def first_cache_line(x):
    -- Returns the cache line with lowest address from touches-cl(x).

def build_immediate_dominator_tree():
    -- Constructs an immediate dominator tree for the input program and
    -- returns its root node.
```

Figure 5.2: Compulsory miss tree construction algorithm
5.2. COMPULSORY MISS TREE

<table>
<thead>
<tr>
<th>cl#</th>
<th>touches-bb</th>
<th>obj code</th>
<th>Basic block view</th>
<th>source code</th>
<th>touches-cl</th>
<th>bb</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{A}</td>
<td>0x8C080200</td>
<td></td>
<td>lw t0,512(zero)</td>
<td>{0, 1}</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x8C090204</td>
<td></td>
<td>lw t1,516(zero)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x10080044</td>
<td></td>
<td>beq zero,t0,&lt;C&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x240A0000</td>
<td></td>
<td>li t2,0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>{A}</td>
<td>0x01095020</td>
<td></td>
<td>add t2,t0,t1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x10000002</td>
<td></td>
<td>b 420 &lt;D&gt;</td>
<td>{2, 3}</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x014A5020</td>
<td></td>
<td>add t2,t2,t2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{B, C}</td>
<td>0x01205020</td>
<td></td>
<td>add t2,t1,zero</td>
<td>{3}</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>{B}</td>
<td>0x08C0B208</td>
<td></td>
<td>lw t3,520(zero)</td>
<td>{4, 5}</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x100A0004</td>
<td></td>
<td>beq zero,t2,&lt;F&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x214AFFFF</td>
<td></td>
<td>add t3,t3,t2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>{D}</td>
<td>0x016A5820</td>
<td></td>
<td>addi t2,t2,-1</td>
<td>{5, 6}</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0214AFFF</td>
<td></td>
<td>b &lt;D&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1000FFFF</td>
<td></td>
<td>sw t3,520(zero)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>{D, E}</td>
<td>0x0AC0B208</td>
<td></td>
<td>b &lt;F&gt;</td>
<td>{7}</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1000FFFF</td>
<td></td>
<td>nop</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.3: View of object code from Figure 4.1 showing cache lines (cl), basic blocks (bb) and the touches-cl/touches-bb relations

constructed from the set of basic blocks and the associated immediate dominator tree is derived. Recall that for each basic block in the immediate dominator tree execution of all its ancestors will have preceded execution of the node itself. Although this will have occurred in order from the root of the tree to the node in question, other basic blocks may also have been executed between nodes on that path. When a given basic block is encountered its instructions will execute in sequential order. Thus cache lines containing instructions for a basic block will be found in contiguous memory locations and data from them will be requested in sequential order from the cache.

We define two relations \(\text{touches-cl}\) for basic blocks and \(\text{touches-bb}\) for cache lines (Figure 5.1) which will help us later when we relate the dominance control flow information to the order of compulsory misses in the instruction cache. Given a basic block \(x\), \(\text{touches-cl}\) will return a set containing those cache lines \(y\) which contain part of the data for basic block \(x\). Similarly, given a particular cache line \(q\), \(\text{touches-bb}\) will return the set of basic blocks \(r\) which contribute to the data stored in cache line \(q\).

Together this knowledge allows us construct the compulsory miss tree by employing the algorithm shown in Figure 5.2. The code on lines 3–8 identifies the root of the compulsory miss tree, that is, the first cache line of instruction data that will miss when the program executes. This corresponds to the cache line containing the first instruction in the program’s entry basic block (the root of the immediate dominator tree). Line 10 performs a pre-order traversal of all basic blocks \(b\) in the immediate
5.2. COMPULSORY MISS TREE

Figure 5.4: Compulsory miss tree

The dominator tree determining for each the point in the compulsory miss tree where its associated cache lines \( cl \) should be attached (line 11). As each cache line \( cl \) may contain data from multiple basic blocks its compulsory miss will occur the first time any of the blocks in touches-bb(\( cl \)) execute. For this reason only the compulsory misses generated on paths to all basic blocks in touches-bb(\( cl \)) may be used as ancestors for \( cl \) in the compulsory miss tree (as determined by lines 15 and 16). In the event that \( b \), the current basic block being traversed, does not contain any cache lines attached to the compulsory miss tree, the misses guaranteed to have previously occurred prior to its execution are those resulting from execution of its immediate dominator. In this situation the immediate dominator’s deepest cache line in the compulsory miss tree is used as ancestor for the first cache line in block \( b \) (lines 18 and 19).

Figure 5.3 shows the simple MIPS assembly language program introduced in Figure 4.1 with its associated machine-level object code. Two views of the program are provided—a basic block view (showing the source code), and a cache line view (showing the corresponding machine-level object code). We use a hypothetical instruction cache that holds eight bytes (two instructions) per line. The \( \text{touches-bb} \) and \( \text{touches-cl} \) relations are also shown. The CFG and immediate dominator tree introduced in Figures 4.2 and 4.4 are those that correspond to this program. Applying the algorithm outlined in Figure 5.2 to the example program yields the compulsory miss tree shown in Figure 5.4.
5.3 Compulsory miss tree based compression

Our improved approach to dynamically building an LZW dictionary for cache line compression begins by building a compulsory miss tree for the program to be compressed. Recall that this tree has the property that whenever a miss occurs for a cache line, all of its ancestors will previously have triggered their compulsory misses.

Compression

We statically allocate space in the LZW dictionary to those cache lines on the path to the largest number of other cache lines, essentially partitioning the dictionary into variable sized divisions. In this way nodes deeper in the compulsory miss tree will have a substantially larger amount of context available in the dictionary for use in their encoding when compared to the naive LZW scheme introduced in section 5.1. The nodes in the compulsory miss tree are weighted by a count of all their descendants. Then, beginning with only the alphabet of bytes $\Sigma = 0..255$ in the dictionary, the cache lines from the compulsory miss tree are compressed in order of their weight. Each cache line uses only those entries placed in the dictionary by its ancestors in the compulsory miss tree for its coding. In this way each cache line inherits the context contributed to the adaptive model by the compression/decompression of its compulsory miss tree parent. Note that the ordering of compression guarantees all of a node’s ancestors will be processed before the node itself and hence the appropriate entries in the dictionary must be present. When the dictionary becomes full no further adaptation can take place and all remaining cache lines are compressed using the entries placed in the (now full) dictionary by their ancestors. Note that each pointer, although fixed in length, cannot be used to address arbitrary entries in the dictionary since not all entries are guaranteed to have been initialized at the point when the decompression of a cache line occurs, hence only those entries statically guaranteed to be initialized may be addressed.

Referring to our example program from Figure 5.3, cache lines will be compressed in the order $\{0,1,4,5,2,3,6,7\}$. The first byte from cache line 0 will be compressed using only the initial alphabet for context with dictionary adaptation starting at entry 256. Subsequent bytes will be processed in a similar fashion using the expanding dictionary for coding purposes. However, at the start of each new cache line the initial context for compression comprises only those entries added to the dictionary by

---

1The order in which nodes of the same weight are processed is arbitrary
5.3. COMPULSORY MISS TREE BASED COMPRESSION

The line’s ancestors in the compulsory miss tree. For example while node 5 will have adapted the dictionary at compression-time prior to the compression of node 2, the encoding of node 2 must not make use of those dictionary entries; only the entries resulting from the compression of nodes 0 and 1, as well as those resulting from processing the node itself, may be referenced when node 2 is being processed. An outline of the process is shown in Figure 5.5.

**Decompression**

LZW generally inserts new entries at the next available location in the dictionary during both compression (as we did above) and decompression. Since we cannot guarantee the total order in which compulsory misses will occur at run-time, we cannot employ such an approach during decompression. If we did, adaptations could occur at different locations in the dictionary to those that occurred during compression thereby invalidating the indices used for the encoding. For example, a sample execution of our program from Figure 5.3 which begins by traversing basic blocks a and b will cause the compulsory misses, and hence decompression, of cache lines in the order 0, 1, 2 and 3. At compression-time, however, the lines were compressed and modified the dictionary in the order 0, 1, 4 and so on. Clearly if we allow decompression at run-time to adapt the dictionary from the next available location, we will not reconstruct the same model used at compression-time, resulting in serious errors in the reconstructed code.

For this reason encoded cache lines require some additional book-keeping information for use by the decompression unit to ensure fidelity of the dictionary model. Each cache line needs to indicate whether or not its decompression should adapt the

<table>
<thead>
<tr>
<th>Index</th>
<th>Phrase</th>
<th>Index</th>
<th>Phrase</th>
<th>Index</th>
<th>Phrase</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x00</td>
<td>0</td>
<td>0x00</td>
<td>0</td>
<td>0x00</td>
</tr>
<tr>
<td>1</td>
<td>0x01</td>
<td>.</td>
<td>.</td>
<td>255</td>
<td>0xFF</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>256</td>
<td>0x8C,0x08</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>255</td>
<td>0xFF</td>
<td>257</td>
<td>0x08,0x02</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>256</td>
<td>init</td>
<td>258</td>
<td>0x02,0x00</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>257</td>
<td>init</td>
<td>259</td>
<td>0x00,0x8C</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>260</td>
<td>.</td>
<td>261</td>
<td>0x8C,0x09</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>258</td>
<td>init</td>
<td>262</td>
<td>0x02,0x04</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>263</td>
<td>init</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>261</td>
<td>.</td>
<td>262</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>260</td>
<td>.</td>
<td>263</td>
<td>.</td>
<td>2^n</td>
<td>init</td>
</tr>
</tbody>
</table>

Figure 5.5: Initial dictionary; dictionary after compressing line 0; and structure of dictionary after five lines lines from the compulsory miss tree have been compressed.
5.4. HARDWARE DECOMPRESSION

dictionary, and if so where entries in the dictionary should be placed as the line is decompressed. A single bit is stored to indicate whether processing a node should lead to a modification of the dictionary. If it does then the following index will identify the location in the dictionary where the first new entry should be placed. This information is readily available at compression-time and is output as part of the compressed cache line encoding. When a miss occurs in the instruction cache at run-time its compressed form is located in memory. If the first bit is a zero it shows that all codes needed for its decoding have already been initialized and this node adds no new entries to the dictionary as it is expanded. However, if the first bit is set to one, then the regular LZW decompression technique is applied adapting from the location identified by the first index of the cache line's encoding. For an example of the encoding see Figure 5.6 where the compressed version of cache line 0 from our example is presented.

Our LAT tables are slightly larger than Wolfe’s since we allow for the expansion of compressed data blocks where they reject it. We need to do this since that is effectively how LZW attains compression: initial data coding causes expansion, but also provides those dictionary entries that later enable us to obtain good compression ratios. The maximum size of a compressed cache line using our scheme includes the overhead when a cache line contributes to the LZW dictionary (1-bit flag plus the index used to address the location to adapt) plus the worst case encoded size when no compression occurs and every byte is replaced by an index to the LZW dictionary (requiring \( x \times n \)-bit indices for a dictionary using indices of length \( n \) bits in a cache line of \( x \) bytes). An appropriate number of bits must be allocated for specifying the size of compressed blocks in the delta coding scheme described for encoding LATs in section 2.6.1, and that is the cause of our increased LAT overhead.

### 5.4 Hardware decompression

We have designed a pipelined LZW decompression unit appropriate for use in a CCRP architecture to implement the technique described above. It was implemented in Verilog HDL and synthesized using Xilinx’s ISE WebPack. The interface to the unit is straightforward: there is a clock input, a reset input, an input address which is the byte
5.4. HARDWARE DECOMPRESSION

address in memory of the compressed cache line to be decompressed, a request line indicating that the input address is valid and decompression should begin and an output buffer which stores the recovered cache line’s data, and a done signal. The high level structure of our design is show in Figure 5.7 and the full Verilog listing is provided in Appendix B.

We assume a byte addressable memory stores the compressed program code, so power reductions via a reduction in required bandwidth for accessing the compressed memory might be realized, and the overhead of padding can be reduced. Fetching of consecutive bytes is handled by stage 0 of the pipeline. Stage 1 buffers a byte since we need to convert the byte stream into a stream of 9-bit LZW pointers. From two bytes (the one just fetched from the compressed memory, byte1, and the one fetched from the compressed memory in the previous cycle, stored in byte0) we form a 16-bit buffer from which 9-bit pointers can be extracted in Stage 2 using a 9-bit 8:1 multiplexor. In Figure 5.8 we show how this operates. After byte0 has been loaded with the first byte in the compressed stream and byte1 has been loaded with the second, the 9-bit mux extracts the first 9 bits of the stream as pointer. On the next cycle, byte0 is loaded with byte1 and byte1 is loaded with the next byte from the stream. The COUNT variable that controls the multiplexor now selects the second 9-bit pointer from the stream, which ignores the first bit of byte0 (which in the previous cycle was the final bit extracted as part of the first pointer). Extraction of 9-bit pointers continues in this fashion however at the end of cycle 7, simply loading byte0 with the value in byte1 is insufficient as all bits of byte1 in the previous cycle were extracted as part of the last pointer output. After every eight cycles we need to stall the remainder of the pipeline for one cycle while a second byte can be read. In this scenario the last pointer passed down the pipeline is reissued from the stall_reissue_pointer register.

Stage 3 takes the pointer generated from stage 2 and uses it to lookup the appropriate data from the LZW dictionary. As the first 256 pointers simply represent themselves in the dictionary we use mux A to determine whether the dictionary entry should come from memory where run-time generated entries are stored, or whether it should be directly generated from the LZW pointer. The most significant bit of the pointer can be used to determine which is required. The dictionary memory is addressed by the remaining 8 bits from the LZW pointer and is a dual port 67-bit memory. Entries in the memory consist of a three bit length field followed by eight bytes of data, of which the length field specifies the number of fields to be considered valid (Since entries in the dictionary contain at least two valid bytes a three bit length field is sufficient). The sec-
5.4. HARDWARE DECOMPRESSION

ond port is used for writing run-time adaptations to the dictionary and will be discussed later. Mux B is used to forward a dictionary entry which was just written to memory in the same cycle as the read should have occurred. Mux C handles the special LZW case when a dictionary entry is read which will not be written to the memory until the end of the next cycle. At the end of stage 3 the dictionary entry is saved to a register for generating a new dictionary entry in the next stage (recall the LZW update inserts the string just located concatenated with the first byte of the next string decompressed into the dictionary). It also incorporates this recovered string into the output buffer via appropriate shifting, masking and oring.

Stage 4 generates a new dictionary entry for the dictionary memory, and stage 5 stores data for forwarding and control of the dictionary and the multiplexors in stage 3.

Controlling where adaptations take place in the dictionary involves inspecting the first bit of the buffer after a new request is initiated and either loading the next_dict_index using the following nine bits (which imposes a delay of 1 cycle before actual decompression begins) or directly beginning to issue LZW pointers to the rest of the pipeline in the same cycle.

Synthesis for a Virtex2 FPGA (XCS2V500) using XST, without using any user constraints requires 318 flip flops, 1528 4 input LUTs and a total of 806 Slices. For speed grade -4 the clock period given is 12.053ns giving a clocking frequency of 82.97Mhz. Lin does not report which Xilinx FPGA they target, but reports that their decompression engine occupies 1129 4 input LUTs and operates at 47.3MHz.

The time required to decode a compressed cache line in pipeline cycles is as follows: There is a latency of 4 cycles when the cache line doesn’t add entries to the LZW dictionary and 5 if it does, until the first data is written to the output buffer and the pipeline is filled. On each subsequent cycle an LZW pointer is processed and data is written to the buffer, except every time 8 pointers have been processed and we need to stall the pipeline for one cycle to fill both byte0 and byte1 with valid data. There is an average requirement of 5 cycles to perform a lookup in the LAT, which is added to the overhead of decompressing the cache line.

A comparable model which doesn’t use compression can fetch data from memory at a rate of 1 word (32 bits) per cycle, thus requiring lineSize/4 cycles to fetch a cache line of lineSize on a miss. The relative slowdown when using our compressed code implementation is:
5.5 Results

As in the previous chapter we employ the benchmarks discussed in section 4.3. We first present results in terms of the compression achievable by the scheme before considering whether it might be a practical technique with respect to its implementation in hardware.

5.5.1 Compression results

We experimented with varying both the dictionary size/index length (from 9 to 12 bits) and cache line size (ranging from 16 to 128 bytes). Using its assigned virtual address
Figure 5.7: Block level diagram of LZW decoder pipeline
5.5. RESULTS

Figure 5.8: Extracting a 9-bit stream from a byte stream

the first instruction of the object code did not always align to the start of a cache line boundary, nor did the code necessarily fill the final cache line in which it was contained; in these situations the cache lines were padded with zeros until they became correctly aligned. We padded all compressed cache line encodings to byte alignment. Our results account for all overhead involved in the scheme including the LAT tables (requiring a constant overhead of 6.25%, 3.515%, 1.953125% and 1.072419% of the original program code size for cache lines of length 16, 32, 64 and 128 bytes respectively, using dictionary indices of size ranging from 9 to 15 bits per index and eight cache lines per LAT entry).

Figures 5.9 and 5.10 present the compression ratios (compressed size/uncompressed size) achieved in our experiments for all benchmarks using various length cache line sizes with 9-bit LZW dictionary indices. On average 7%, 14%, 19% and 19% of the original code size is eliminated using our compulsory miss tree based technique for cache lines of length 16, 32, 64 and 128 bytes respectively, compared with reductions of -12%, -6%, 3% and 10% for regular LZW. We do not present results for larger dictionaries since none of them showed an improvement over those just described (as previously discussed in section 4.3).

These results show that simply moving to a fixed-length block compression tech-
5.5. RESULTS

Figure 5.9: Compression ratio for 9-bit LZW applied at various cache line lengths

nique alone does not justify the use of an adaptive compression technique. While it
cuts down on the overhead associated with the mapping from uncompressed addresses
to their compressed counterparts, the use of additional context beyond the boundary
of a single cache line is required for the technique to be considered in any reasonable
way effective. Furthermore, it seems that arbitrarily increasing the size of the cache
lines improves the compression of cache lines individually, but does not do so for the
compulsory miss tree technique where disimprovements in the results were observed
at a line length of 128 bytes. While such a long line may be uncharacteristic of an
embedded system, the best results we observed occur for a 64-byte line which is not
entirely unrealistic.

5.5.2 Performance evaluation

Having now demonstrated an adaptive compression technique which gives promising
code size reductions we consider whether the technique is practical. Since compres­
sion is performed offline, the decompression routine is the target of our investigation.
We will consider the performance of our decompression routine using the models de­
scribed by equations 5.1, 5.2 and 5.3. Our investigation assumes that the time spent
executing code throughout the application is distributed evenly over all code and, on
5.5. RESULTS

Figure 5.10: Compression ratio for 9-bit LZW applied at various cache line lengths over a compulsory miss tree

that basis, we use the average time to decompress any cache line in our evaluation of these equations. More accurate results would be obtained by profiling execution of the application and storing the trace of instruction addresses actually requested however, due to the lack of a simulator for MIPS capable of executing statically linked applications, this was not an available option.

We assume the time to access the LAT ($C_{yc_{LAT}}$) is 5 cycles, and the number of cycles required for the pipeline to decode the data ($C_{yc_{Pipeline}}$) is the average over all cache lines in the application of the latency required to start writing data to the output buffer, the number of cycles during which the pipeline is filled plus the number of stalls required to buffer a byte for byte 0. We range the miss rate $mr$ from 0 to 1.0. We select $C_{yc_{Arb}}$ as 20 cycles for bus arbitration. For the model in equation 5.3 $C_{yc_{MemAccess}}$ and $C_{yc_{Wait}}$ are determined by the speed of computer memory and the rate at which the pipeline/CPU are clocked. If we assume the clock operates at 82Mhz (the speed at which our pipeline can operate, according to synthesis results by Xilinx tools), and the memory being accessed is Intel’s StrataFlash P33 with a random access time of 85ns and a burst read rate of 52Mhz, then 9 cycles are needed for a random access in

2James Larus' excellent SPIM simulator [PH04, Appendix A— Assemblers, Linkers, and the SPIM Simulator] does not execute binary programs. It reads and executes assembly language programs.
5.5. RESULTS

Figure 5.11: cnpeg

memory \((C_{\text{MemAccess}})\) and 2 cycles for sequential reads thereafter \((B_{\text{stRead}})\). Were the pipeline/CPU implementation engineered carefully to allow operation at a faster rate of 200MHz, \(C_{\text{MemAccess}}\) would become 18 cycles and \(B_{\text{stRead}} 4\). Our results for the simple model in equation 5.1, the arbitration model in 5.2 and the memory model in equation 5.3 at 82MHz and 200MHz are shown for each benchmark in Figures 5.11 - 5.18.

In the results for the simple model, at a 100% miss rate we see that the relative time for accessing the smallest cache line is greatest, because in the short cache line the initial latency of filling the pipeline cannot be masked and the compression for these short lines is the worst. As the cache line sizes increase we get to mask the latency of filling the pipeline and as we have better compression (requiring that we fetch less data and use fewer cycles to decompress it), the relative performance to just filling the line improves.

In the arbitration model we see that at a miss rate of 100% the biggest cache line has the worst performance relative to the base model, and the shortest cache line has the best. This is because in the smallest cache line the arbitration overhead adds significantly to the fetching time in the base system, but doesn’t add much (relatively)
5.5. RESULTS

Impact of Compression on Cache access time

Simple model

Memory model 82Mhz

Arbitration model

Memory model 200Mhz

Figure 5.12: crc

Figure 5.13: jpeg
5.5. RESULTS

![Graphs showing the impact of compression on cache access time for different memory models and compression sizes. The graphs compare simple and arbitration models, with memory models 82Mhz and 200Mhz.](image)

**Figure 5.14**: ispell

**Figure 5.15**: rawcaudio
5.5. RESULTS

![Impact of Compression on Cache access time](image1)

**Simple model**

![Impact of Compression on Cache access time](image2)

**Arbitration model**

Memory model 82Mhz

Figure 5.16: rawaudio

Memory model 200Mhz

![Impact of Compression on Cache access time](image3)

Simple model

![Impact of Compression on Cache access time](image4)

Arbitration model

Memory model 82Mhz

Memory model 200Mhz

Figure 5.17: rijndael
5.5. RESULTS

to the decoding time in the compressed system. When the cache lines are largest, the addition to the decoding time is still negligible compared with the original decoding time, but as the fetching time increases the arbitration overhead no longer dominates the denominator in our model and so the relative performance decreases.

When we introduce a real memory to the system we reduce the overheads observed with our scheme with respect to the base technique further. We can mask up to 4 cycles of burst read time per access in our pipeline by fetching a word from memory and dispatching consecutive bytes on each clock cycle, and in the cases where good compression arises we need to fetch fewer words from memory than the base system. In benchmarks where good compression was observed (e.g. cjpeg) the slowdown at 82MHz is less than 50% over the base system for the longest cache line, while rijndael and toast, which did not compress as well have overheads in excess of 50%. By increasing the clock to 200Mhz the cost of accessing memory becomes a dominating factor in the base schemes and our code compression approach can, under certain circumstances, lead to a relative increase in performance due to the reduction in time spent waiting for data to transfer from memory. While rijndael sees no performance improvement for a 64-byte line at 200Mhz, cjpeg which compressed better (and hence requires less
5.5. RESULTS

data from memory during a line refill) does. Such performance improvements over the non-compressed architecture are in addition to the memory size reduction benefits which were the source of our original interest in code compression schemes. This observation that performance increases over native code might be realized in the presence of memories with long latencies is in keeping with other’s results.

Tomiyama and Yasuura [TY97] investigated code placement techniques for cache miss rate reduction in embedded systems. While they target the SPARC architecture and we’ve been investigating MIPS, before application of their placement techniques they observed cache miss rates of 6.8%, 7.11%, 12.44% and 0.18% for a direct mapped 1k cache with lines of 32 bytes in length for four different benchmarks (grep, sed, indent and compress). In all our results we show full graphs for miss rates ranging from none (0%) to a miss on every access to the cache (100%), while in reality we should probably constrain our interest in these results to the 0-20% range.

A naive consideration of our technique independent of a full architecture might assume memory could be accessed at the same rate as our pipeline is clocked, and in the absence of a cache this results in a slowdown by a factor in excess of 4 over a system without compression (our simple model results). When a more realistic model is introduced, where the overhead of bus arbitration is taken into account, the relative degradation in our results is reduced to a factor of about 2.5. Considering the performance of a commercial memory and the speed at which a direct synthesis of the pipeline can be clocked at further reduces the overhead to a factor of about 1.4, and a hypothetical system where a careful implementation of our pipeline can be clocked at 200Mhz is shown to offer potential speedups over the base system. Finally, considering a low miss rate can mask the overhead of our scheme when it causes a slowdown, there is a strong indication that the technique we present should be considered practical.
5.5. RESULTS
Chapter 6

Conclusions

6.1 Future work

There are a number of avenues for further research based on the material presented in this thesis; we briefly discuss them in the remainder of this section before summarizing our contributions in the final paragraphs.

6.1.1 Immediate dominator DAG / compulsory miss DAG

The dominator tree for a CFG is the transitive reduction of the solution to computing the dominance relation between all nodes in the graph. Using a path-sensitive dominance computation rather than the traditional path-insensitive technique, the idom relation may result in a directed acyclic graph (DAG) rather than a tree when we work with an interprocedural control flow graph (Figure 6.1). This arises due to the path-sensitive

![Diagram](image)

Figure 6.1: Interprocedural control flow graph (i) dominator tree (ii) and dominator DAG (iii)
nature of control flow where a return node may only be visited if its corresponding call node was also visited. Consider call site 2 and its associated return site 4 in the example where dashed lines are not paths which are taken at runtime but indicate the point at which the call to a procedure will continue execution after the procedure returns. We know that \(2 \text{ dom } 4\) since a return node will only be reached if its corresponding call site was encountered. Furthermore we know that the exit node \(c\) of the called procedure also dominates the return site 4. However by the definition of \(idom\) in section 3.2 both 2 and \(c\) strictly dominate 4 but do not strictly dominate any other dominator of 4 (since 2 doesn't dominate \(c\) nor vice versa), which leads to the construction of a DAG rather than a tree for the transitive reduction. We call the transitive reduction of the path-sensitive dominators computation in an interprocedural control flow graph the dominator DAG. A fast algorithm for computing interprocedural dominance has recently been proposed [SPB07].

Based on the dominator DAG for a program, it should be clear that an associated compulsory miss DAG can be constructed to provide additional context during compression improving on the schemes introduced in Chapter 5. Implementing the compulsory miss DAG and adapting our compulsory miss tree based technique to work with DAGs rather than trees constitute research which can be investigated immediately.

### 6.1.2 Scratchpad memories

Small on-chip static RAMs termed scratchpad memories have been proposed as a design alternative for on-chip cache memory in embedded systems due to their reduced area requirements and low power consumption [BSL+02]. As the data stored in these memories is typically selected by the programmer at design time, or automatically chosen by a scratchpad aware compiler, they lend themselves to determining worst case execution time of a task better than a system employing a cache memory. While the majority of interest in scratchpad memories has focused on storage of data in the on-chip memory, recent work has considered the execution of instruction data directly from this faster memory. There are two approaches to storing code in the scratchpad: it can be statically selected and remain in the memory throughout execution of the program; or alternatively it can be moved in and out of the scratchpad dynamically as described by Steinke [SGW+02]. In such a scheme code which is executed only from the scratchpad could be stored in compressed form in memory and decoded as it is moved into the scratchpad. As the blocks of code being moved are still quite small
6.1. FUTURE WORK

the adaptive techniques we presented in this thesis could help improve both static code size and power consumption.

Other work on scratchpad memories has considered their use as part of a demand-paged memory scheme [ELS06] where code is partitioned into blocks which can reside on the scratchpad and blocks which should not. The code appropriate for execution from the scratchpad is moved into the on-chip memory in response to a page fault trigged by the system’s memory management unit. As the pages being moved into the scratchpad are of fixed size, a compulsory fault tree analogous to the compulsory miss tree can be constructed. Using such a tree the compression algorithm described for caches can readily be adopted for use at this coarser granularity.

6.1.3 Low-entropy code generation

As discussed in section 2.5, the backend of a compiler performs three important tasks: instruction selection; register allocation; and instruction scheduling. These aspects of code generation are typically targeted at producing code which executes quickly—usually achieved by aligning it with a model of how the target architecture will execute each instruction at runtime. Speed is not the only metric that needs to be targeted during the code generation phase. Instead of attempting to minimize code execution speed, the three backend algorithms can be targeted at minimizing memory traffic, static code size, power consumption etc. Our initial interest in object code compression stemmed from the idea that, given a code compression scheme, the compiler might be modified to generate object code which was more susceptible to compression by incorporating a knowledge of the compression algorithm into the code generation process. This idea is not entirely new and has been noted by several other researchers investigating code compression systems [Xie02, Lef00]; nevertheless there has been relatively little work in the field considering the possible impact compiler optimizations might have when targeted at generating code with low entropy for compressed code systems.

Reordering operations within a VLIW instruction was investigated by Ros in [RS03] targeting a simple semi-adaptive dictionary algorithm and scheduling code for reduced entropy targeting a more complicated dictionary technique was considered by Lekatsas in [Lek00]. Scheduling code for a prediction-by-partial-match (PPM) code compression algorithm was considered in [DKV03]. Cooper and McIntosh [CM99] used a register renaming technique to improve on Fraser’s code compaction algorithm [FMW84]. Register renaming was also investigated by Ros in the context of a code compression algorithm based on Hamming distances [RS05, RS04]. Hines considered adapting all
three backend algorithms to target his Instruction Register File which is essentially a semi-adaptive dictionary compression algorithm exposed to the underlying architecture [HWT06]. We are aware of no work on instruction selection or register allocation targeting statistical code compression techniques, nor any that targets the adaptive dictionary algorithms proposed in our work [GA06] or in Lin’s [LXW07, LXW04].

6.2 Summary

In this thesis we have investigated the use of adaptive data compression algorithms for compressing object code in a manner that is transparent to the underlying processor on which the code executes. In Chapter 2 we described the types of memory used for code storage in embedded systems and discussed the advantages associated with reductions in memory size requirements: specifically improvements in battery life due to decreased energy consumption; increases in performance due to reduced memory access time; reductions in production costs due to increased yield; and reductions in software development and maintenance cost due to effective memory size increase. We reviewed the design of instruction sets from the perspective of code size before presenting introductory material on data compression to provide appropriate background for the remainder of the thesis. In section 2.6 we classified previous work on code size reduction in terms of a number of facets (whether the technique compacts or compresses code, what representation of the code is compressed, and at what granularity the technique operates), giving an overall picture of the approaches taken by other researchers in the past. This classification was followed by a summary of the most well-known code size reduction schemes and those code compression techniques most relevant to our work.

Chapter 3 presented our Graph LZW algorithm, an adaptive technique based on LZW for compressing small quantities of text which are organized as a directed rooted graph. The formal definition of immediate dominance and the computation of this relation on a graph were introduced before disclosing our novel approach to LZW dictionary maintenance as a stack like structure in section 3.3. To investigate the effectiveness of our algorithm it was applied to a collection of PDA websites, where it was found to reduce the compressed text size by approximately 20% more than regular LZW, and required only minor modifications to the standard LZW decompression routine. Notwithstanding the simplicity of our decoding routine, the results presented are comparable with those achieved by gzip which requires a considerably more complex
decoder combining both static Huffman and LZ77 decoding routines. Based on gzip's combination of an adaptive dictionary-based technique with a semi-adaptive code allocation phase we considered the impact of using a canonical Huffman code to represent the dictionary indices used by our Graph LZW algorithm and concluded that the overhead required in the decoder would not represent a justifiable penalty for the marginal improvements likely to arise.

In Chapter 4 we considered how Graph LZW might be applied to the control flow graph representation of a program's object code. We showed that basic blocks, branch blocks, extended basic blocks and immediate dominator trees--all popular structures targeted by optimization within compilers--essentially form a family of related data structures capturing different levels of context which can be exploited by an adaptive approach to code compression. The immediate dominator tree captures all context offered by the others, so our Graph LZW applied to object code can be considered a generalization of Lin's branch block compression, which is the only previous work on adaptive code compression at a granularity smaller than an entire procedure. We discussed the difficulties of implementing the technique in a manner transparent to the underlying processor on which the code executes and describe our solutions to the problems which arise. In section 4.3.1 we discussed our choice of architecture on which our experiments were performed and the difficulties of recovering a control flow graph representation of a fully linked binary which is an essential input to our compression routine. We showed that Lin's work on VLIW code is unlikely to be applicable to the RISC code on which we work due to the substantially smaller branch blocks extracted from our code than those he worked with. The compression ratios achieved for our techniques do not, unfortunately, reduce the size of the input programs. We attributed this to the overheads associated with the encoding scheme employed for variable-length blocks and our method of locating code in the compressed memory. The raw encoded data does decrease in size for each compression scheme and this suggests that a variation of the technique may indeed be practical if these overheads can be reduced. In section 4.3.6 we considered the impact of compile and link time optimization on compression and concluded that great care is needed when using the compression ratio metric as a basis on which to quantitatively compare code compression techniques.

Building on the work from Chapter 4 we investigated the use of an LZW technique in Chapter 5 where fixed-length cache lines are the target of compression and the decompression unit is positioned between memory and the instruction cache. By
moving to fixed-length blocks Wolfe's LAT can be used to efficiently store the mapping required for looking up compressed code which needs only to be padded to byte alignment rather than word alignment, hence addressing the main overheads associated with our techniques from Chapter 4. Initially we considered the compression of each cache line individually but as expected this does not result in code size reduction for cache lines shorter than 64 bytes in length. In section 5.2 we presented a novel data structure termed the **compulsory miss tree** (and an algorithm for its construction) which encodes for each cache line in the program those other cache lines that are guaranteed to have caused their compulsory miss prior to the first miss for the line in question. Based on this data structure we presented an alternative code compression algorithm in section 5.3 which statically allocates space in the LZW dictionary to those cache lines on the path to the largest number of other cache lines, essentially partitioning the dictionary into variable sized divisions. Using the compulsory miss tree based technique, elimination of 7%, 14%, 19% and 19% of the original code is achievable for cache lines of length 16, 32, 64 and 128 bytes respectively. Motivated by these promising code size reduction results we designed a pipelined decompression unit in hardware using Verilog (described in section 5.4) which can operate at a speed of 82.97Mhz. Section 5.5.2 considered the performance of our system when integrated with a number of simple memory models and showed that in the most realistic model (employing a shared bus with an arbitration overhead, and a flash memory with a burst read mode) using a direct synthesis of our pipeline incurred a speed overhead in the region of about 1.4 over a system not using code compression. If a careful implementation of our pipeline allowed a faster clock speed of 200Mhz, potential speedups over the base system can be demonstrated.

Ultimately the techniques presented in this thesis allow us to conclude that the pervasively claimed random access decompression requirement for object code compression [LHJC05, LHW00, MS05, EEF+97, XWL01b, LHC+03, ABFG+03, LSSC03, NMC03, OW02] is unnecessarily restrictive. Furthermore we refute the claims made by other researchers that adaptive compression models are inappropriate, impractical or impossible to use for object code compression since they do not allow for random access or work well for small blocks of data [XWL03, KMH+98, WC92, KW94, LW99b, CAP99, Luc00, LW98, LW99a, LDK95, LDK99].
Appendix A

Example Graph Structured Text Application

In this appendix we give the source code for a prototype web 2.0 application which implements the Graph LZW algorithm described in Chapter 3. The majority of the code in client.html is written in JavaScript and the modifications required to convert the client from a regular LZW decoder to a Graph LZW decoder are shown on lines 10, 11, 90, 91 and 105. The encoded Graph LZW data is listed in compressed.txt while the simple server which handles requests sent from the client is shown in server.php and written in PHP.

A.1 client.html

```html
<html>
<head>
<title>Adaptive Compression of Graph Structured Data</title>
<script language='javascript'>
// parameterize code for both regular and context-based lzw
var isIdomTreeBased = true;
// data structure to enable context-based lzw
var end.of.adaptations = new Array(128); // can only handle idom trees of depth 128
end.of.adaptations[0] = 256;
// global lzw dictionary
var pointerLength = 9;
var dictionarySize = Math.pow(2, pointerLength);
var nextIndex = Math.pow(2, 8);
var parentIndex = new Array(dictionarySize);
for(var i = 0; i < dictionarySize; i++) parentIndex[i] = -1;
```
```javascript
var characters = new Array(dictionarySize);
for(var i = 0; i < 256; i++) characters[i] = String.fromCharCode(i);

// lz77 decode functions
function unwind(ptr)
{
    var decoded = "";
    var stack = new Array();
    var si = 0;
    var curr = ptr;
    while(curr != -1)
    {
        stack[si] = characters[curr];
        si++;
        curr = parentIndex[curr];
    }
    for(var i = si - 1; i >= 0; i--)
    {
        decoded += stack[i];
    }
    return decoded;
}

function decode(ptr, nextPtr)
{
    // decode the pointer
    var decoded = unwind(ptr);
    // update the dictionary
    if(nextPtr != -1 && nextIndex < dictionarySize)
    {
        if(nextPtr == nextIndex)
        {
            // special case - nextPtr isn't in the dictionary yet!
            characters[nextIndex] = decoded.charAt(0);
            parentIndex[nextIndex] = ptr;
            nextIndex++;
        }
        else
        {
            // regular case
            characters[nextIndex] = unwind(nextPtr).charAt(0);
            parentIndex[nextIndex] = ptr;
            nextIndex++;
        }
    }
    return decoded;
}
```
function decodeStream(stream)
{
    var pointers = stream.split(",");

    // fix type. pointers is an array of ints, not strings
    for(var i = 0; i < pointers.length; i++)
    {
        pointers[i] = parseInt(pointers[i]);
    }

    var decodedStream = "";
    var startIndex = 0;

    // reset the dictionary appropriately
    if(isIdomTreeBased)
    {
        startIndex++;
        nextIndex = end.of.adaptations[pointers[0]];
    } else {
        nextIndex = Math.pow(2, 8);
    }

    for(var i = startIndex; i < pointers.length; i++)
    {
        var nextPointer = i < pointers.length - 1 ? pointers[i + 1] : -1;
        decodedStream += decode(pointers[i], nextPointer);
    }

    // maintain end_of_adapt[]
    if(isIdomTreeBased)
    {
        end.of.adaptations[pointers[0] + 1] = nextIndex;
    }

    return decodedStream;
}

// utility and debug functions
function render(content)
{
    var contentDiv = document.getElementById("content");
    contentDiv.innerHTML = content;
}

// ajax
function requestData(id)
{
    var req = new XMLHttpRequest();
    req.onreadystatechange = renderResponse;
    req.open("GET", "server.php?id=" + id, true);
    req.send(null);
}

function renderResponse()
A.2. server.php

```php
<?php
$id = $_REQUEST['id'];
$data = @file('compressed.txt');
if(!@$data)
    echo "Couldnt_open_file";
    exit(-1);

    foreach($data as $entry) {
    list($key, $value) = explode('=', $entry);
    if($key == $id) {
        echo trim($value);
    }
}?

A.3. compressed.txt

2:1,84,104,270,66,97,110,107,32,111,102,32,73,282,108,350,100,323,280,282,284,286,
1:1,84,104,270,66,111,111,107,32,111,102,32,75,101,266,115,323,280,282,284,286,288,
32,60,97,32,104,114,101,102,61,34,106,97,118,97,115,99,257,112,116,58,282,113,117,
101,115,116,68,97,116,97,40,49,41,34,62,110,111,100,270,49,60,47,97,62,32,124,277,
279,281,283,285,287,289,291,293,295,297,299,301,303,305,40,50,309,311,313,270,50,
317,319
Appendix B

Pipeline Implementation

In this appendix we provide the full Verilog source code for the pipelined decompression unit described in Chapter 5, section 5.4.

B.1 Verilog listing

```
module lw_expander(clk, reset, address, req, output_buffer, done);

// --
// interface

input clk, reset;
input [31:0] address;
input req;

output [16*8-1:0] output_buffer;
output done;

reg [16*8-1:0] output_buffer;
reg done;

// -------------------
// internal state

reg [2:0] state;
```
B.1. VERILOG LISTING

```verilog
reg [7:0] byte0;
wire [7:0] byte1; // this is latched in the memory output
wire [15:0] buffer; // 16bit buffer of two bytes
reg [8:0] pointer;
reg [8:0] stall_reissue_pointer;
wire [8:0] issued_pointer;
reg [31:0] next; // address of next sequential byte in mem to request
wire [31:0] req_address; // address which were requesting from the byte memory
reg [2:0] count; // which offset in buffer are we requesting a pointer from?
reg [8:0] last_pointer; // pipeline latch incase we opt not to take the data from the dict mem
reg [7:0] next_dict_index; // next location to be written into in the dict mem
wire [66:0] dict_out;
wire [66:0] dict_in;
reg dict_write_enable;
wire [66:0] dictionary_entry_mem;
wire [66:0] dictionary_entry_lzw_special;
wire [66:0] dictionary_entry_lzw_regular;
wire [66:0] dictionary_entry;
reg [66:0] last_dictionary_entry;
reg [66:0] last_dict_in;
wire [4:0] output_buffer_count;
wire [4:0] next_output_buffer_count;
wire [127:0] next_output_buffer;
wire ad;
reg adapt;

// memory units
memory YourInstanceName_memory {
    .addr(req_address), // Bus [7 : 0]
    .clk(clk),
    .dout(byte1)); // Bus [7 : 0]
}
dictionary YourInstanceName_dictionary {
    .addra(issued_pointer[7:0]), // Bus [7 : 0]
    .addrb(next_dict_index), // Bus [7 : 0]
    .clka(clk),
    .clkb(clk),
    .dinb(dict_in), // Bus [66 : 0]
    .douta(dict_out), // Bus [66 : 0]
    .web(dict_write_enable));
```

B.1. VERILOG LISTING

// combinatorial logic

assign req_address =
  ((state == 'FETCH) | (state == 'WAIT)) ?
  address : next;

assign buffer = {byte0, byte1};

assign dictionary_entry_mem = last_pointer[8] ?
  dict_out : [ 3'b000, last_pointer[7:0], 56'b0];

assign dictionary_entry_lzw_regular =
  (next_dict_index - 1'b1 == last_pointer[7:0]) & last_pointer[8] ?
  last_dict_in : dictionary_entry_mem; // handle forwarding of data entries

assign dictionary_entry_lzw_special =
  (last_dictionary_entry[66:64] + 1'b1,
   last_dictionary_entry[63:0]) |
  (last_dictionary_entry[63:56] << ((6 - last_dictionary_entry[66:64])*8));

assign dictionary_entry =
  (next_dict_index == last_pointer[7:0]) & last_pointer[8] ? // handle the lzw special case
  dictionary_entry_lzw_special : dictionary_entry_lzw_regular;

assign next_output_buffer_count =
  output_buffer_count == 5'bl000 ?
  output_buffer_count : output_buffer_count + dictionary_entry[66:64] + 1'b1;

always @(buffer, count)
  begin
    case(count)
      3'b000: pointer <= buffer[15:7];
      3'b001: pointer <= buffer[14:6];
      3'b010: pointer <= buffer[13:5];
      3'b011: pointer <= buffer[12:4];
      3'b100: pointer <= buffer[11:3];
      3'b101: pointer <= buffer[10:2];
      3'b110: pointer <= buffer[9:1];
      3'b111: pointer <= buffer[8:0];
    endcase
  end

assign dict_in =
  [last_dictionary_entry[66:64] + 1'b1,
   last_dictionary_entry[63:0]] |
  dictionary_entry[63:56] << ((6 - last_dictionary_entry[66:64])*8});

// distance to shift is linebytes - (size + 1 + 1)
// => 8 - (lde[66:64]) + 1 + 1
// => 6 - lde[66:64]
wire [16*8+8*8+1:0] shifted_entry;
assign shifted_entry = dictionary_entry[63:0] << ((16 - output_buffer_count)*8);
B.1. VERILOG LISTING

```verilog
counter [127:0] or_component;
assign or_component = shifted_entry[16*8+8-1:8*8];
assign next_output_buffer = output_buffer | or_component;
assign ad = buffer[15];
assign issued_pointer =
(state == 'BUFFER_BYTE) ? stall_reissue_pointer : pointer;

// Synchronous logic
always @ (posedge clk or negedge reset) begin
  if ('reset)
    stall_reissue_pointer <= 9'b0;
  else
    stall_reissue_pointer <= pointer; // we assume we stall for only one cycle max
end

// Main asm
always @ (posedge clk or negedge reset) begin
  if ('reset)
    begin
      next <= 32'b0;
      byte0 <= 8'b0;
      count <= 3'b000;
      last_pointer <= 9'b0;
      last_dictionary_entry <= 67'b0;
      output_buffer_count <= 4'b0;
      output_buffer <= 127'b0;
      next_dict_index <= 8'b0;
      last_dict_in <= 67'b0;
      done <= 1'b0;
      adapt <= 1'b0;
    end
  else
    case(state)
      'FETCHB1:
        begin
          output_buffer <= 127'b0;
          count <= 3'b000; // bit 0 is the adapt line
          next <= address + 1'b1;
        end
      'FETCHB0:
        begin
          byte0 <= byte1;
          next <= next + 1'b1;
        end
      'LUPPO:
        begin
```
192 begin
193 adapt <= ad;
194 count <= count + 1'bl;
195 byte0 <= byte1;
196 next <= next + 1'bl;
197
198 last_pointer <= issued_pointer;
199
200 if(ad)
201 begin
202 // we'll stall in the next cycle to actually lookup the first pointer.
203 next_dict_index <= pointer[7:0]; // where do our adaptations begin?
204
205 end
206
207 'LUPP1_EXPANDP0:
208 begin
209 dict_write_enable <= 1'bl; // we now have the first ptr in the expand stage,
210 // so begin writing to the dict in the next cycle.
211
212 last_dictionary_entry <= dictionary_entry;
213 output_buffer_count <= next_output_buffer_count;
214 output_buffer <= next_output_buffer;
215
216 count <= count + 1'bl;
217 last_pointer <= issued_pointer;
218
219 byte0 <= byte1;
220 next <= next + 1'bl;
221
222 end
223
224 'STEADY:
225 begin
226 last_dict_in <= dict_in;
227 next_dict_index <= next_dict_index + 1'bl;
228
229 last_dictionary_entry <= dictionary_entry;
230 output_buffer_count <= next_output_buffer_count;
231 output_buffer <= next_output_buffer;
232
233 count <= count + 1'bl;
234 last_pointer <= issued_pointer;
235
236 byte0 <= byte1;
237 next <= next + 1'bl;
238
239 if(next_output_buffer_count == 5'd10000) // were about to wait()
240 begin
241 done <= 1'b1;
242 dict_write_enable <= 1'b0;
243
244 end
245 // else
246 // if(count == 3'b111)
247 // dict_write_enable <= 1'b0; // disable it for the stall
248 // (strictly this isn't necessary... as the stall has the last

B.1. VERILOG LISTING
B.1. VERILOG LISTING

```verilog
// pointer reissued, so it just repeats the dict update as before)
end

BUFFER_BYTE:
// stall everything and buffer a byte. note that the last ptr
// will be reissued to the dictionary.
begin
next <= next + 1'b1;
byte0 <= byte1; // rest of pipeline is stalled
count <= 3'b0;

if(adapt)
dict_write_enable <= 1'b1;
end

MOD_STALL:
// this is the actual first ptr lookup, as the last state was doing a
// dict adapt location. unlike buffer byte we do not reissue the last
// pointer during this state!
begin
count <= count + 1'b1;
next <= next + 1'b1;
byte0 <= byte1;
last_pointer <= issued_pointer;
end

WAIT:
begin
count <= 3'b001;
nex t <= address + 1'b1;
if(req)
begin
done <= 1'b0;
output_buffer_count <= 5'b0;
output_buffer <= 127'b0;
end
end
default:
begin
endcase

end

// next state computation logic
always @ (posedge clk or negedge reset)
begin
if("reset")
state <= 'FETCHBl;
else
begin
endcase

end
end
```

136
300     state <= 'MOD_STALL; // need another cycle to get the actual pointer
301     else
302         state <= 'LUPP1_EXPANDP0;
303     'LUPP1_EXPANDP0:
304         state <= 'STEADY;
305     'STEADY:
306         if(count == 3'b111)
307             state <= 'BUFFER_BYTE;
308     else if(next_output_buffer_count == 5'b10000)
309         state <= 'WAIT;
310     'BUFFER_BYTE:
311         state <= 'STEADY;
312     'WAIT:
313         if(req) state <= 'FETCHBO;
314     'MOD_STALL:
315         state <= 'LUPP1_EXPANDP0;
316     default:
317         begin
318             end
319     endcase
320     end
321
322 endmodule
Appendix C

Detailed Compression Ratio Results

In this appendix we provide full compression ratio results for each benchmark on which we tested our LZW-based code compression algorithms described in chapters 4 (Basic Block, Branch Block, Extended Basic Block and IDOM Tree) and 5 (cacheline from section 5.1 and cacheline CMT from section 5.3, each applied on cache lines of length 16, 32, 64 and 128 bytes). We applied each scheme to our benchmarks using LZW dictionaries requiring between 9 and 12-bit indices.
Compression ratios for various techniques

Figure C.1: cjpeg

Figure C.2: crc
Figure C.3: djpeg

Figure C.4: ispell
Compression ratios for various techniques

Figure C.7: rijndael

Figure C.8: toast
Bibliography


[BYTJ05] Coon; Brett, Miyayama; Yoshiyuki, Nguyen; Le Trong, and Wang; Johannes. System and method for translating non-native instructions to native instructions for processing on a host processor, 2005.


Timothy J. Harvey Keith D. Cooper and Todd Waterman. Building a control-flow graph from scheduled assembly code.


Haris Lekatsas, Jörg Henkel, Srimat Chakradhar, Venkata Jakkula, and Murugan Sankaradass. CoCo: a hardware/software platform for rapid


BIBLIOGRAPHY


