Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Timo Lindfors

Compression-friendly Encoding for LLVM Intermediate Representation

Master's Thesis
Espoo, May 8, 2014

Supervisor: Professor Heikki Saikkonen
Instructor: Vesa Hirvisalo D.Sc. (Tech.)

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Timo Lindfors		
Title:	Compression-friendly Encoding for LLVM Intermediate Representation		
Date:	May 8, 2014	**Pages:**	68 + 9
Professorship:	Software technology	**Code:**	T-106
Supervisor:	Professor Heikki Saikkonen		
Instructor:	Vesa Hirvisalo D.Sc. (Tech.)		

Distributing software in compiler intermediate representation (IR) instead of native code offers new possibilities for optimization. When IR for the whole program and its dependencies is available at the target system we can use simple but effective install-time optimizations such as constant propagation across library boundaries. Since we know the exact CPU instruction set we can also generate code that optimally matches the target system. Finally, we can execute the software using a just-in-time compiling (JIT) interpreter that can benefit from locally generated profiling information.

Unfortunately the bitcode format that is used to encode the IR of the popular LLVM compiler takes more space than native code. This is mostly because it has been designed to be seekable to allow execution of code without having to read the whole bitcode file. Seekability is a useful property but it is not needed in a software distribution format. Bitcode is also bit oriented which confuses most byte oriented compression tools.

We present a new compression-friendly wire format for the LLVM IR. We compiled 3511 real-world programs from the Debian archive and observed that on average we only need 68% of the space used by gzipped bitcode. Our format achieves this by using two simple ideas. First, it separates similar information to different byte streams to make gzip's job easier. Second, it uses relative dominator-scoped indexing for value references to ensure that locally similar basic blocks stay similar even when they are inside completely different functions.

Keywords: LLVM, compiler, intermediate representation, bitcode, bytecode, data compression

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan tutkinto-ohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Timo Lindfors		
Työn nimi: Pakkausystävällinen koodaus LLVM:n välikielelle			
Päiväys:	8. toukokuuta 2014	**Sivumäärä:**	68 + 9
Professuuri:	Ohjelmistotekniikka	**Koodi:**	T-106
Valvoja:	Professori Heikki Saikkonen		
Ohjaaja:	Tekniikan tohtori Vesa Hirvisalo		

Tietokoneohjelmien levittäminen välikielimuodossa (IR) tarjoaa uusia optimointimahdollisuuksia. Esimerkiksi kun ohjelman ja sen käyttämien ohjelmakirjastojen IR on saatavilla ohjelmaa asennettaessa voidaan vakiolausekkeita sieventää kirjastorajojen yli. Koska ohjelmaa asennettaessa tunnetaan kohdejärjestelmän tarkka käskykanta voidaan IR:stä kääntää natiivikoodia, joka sopii mahdollisimman hyvin kohdejärjestelmälle. Lopulta voimme käyttää IR-muodossa olevien ohjelmien suorittamiseen JIT-kääntäjää, joka voi hyödyntää paikallisessa suorituksessa kerättyä profilointitietoa.

Valitettavasti suositun LLVM-kääntäjän käyttämän IR:n bitcode-koodaus tarvitsee natiivikoodia enemmän tilaa. Tämä johtuu osin siitä, että se on suunniteltu tukemaan koodin osittaista suorittamista ilman, että tulkin tarvitsee lukea bitcode-tiedostoa kokonaan ennen suorituksen alkua. Tämä on hyödyllinen suunnittelupäätös, mutta siitä ei ole hyötyä ohjelmia levitettäessä. Bitcode on myös rakenteeltaan bittivirta, mikä vaikeuttaa tavuvirtoihin suunniteltujen tavallisten pakkausohjelmien työtä.

Esitämme tässä työssä LLVM:n IR:lle uuden koodauksen, joka soveltuu paremmin pakkausohjelmien syötteeksi. Käänsimme 3511 oikeaa ohjelmaa Debian-arkistosta ja havaitsimme, että koodauksemme tarvitsee vain 68% bitcode-koodauksen tarvitsemasta tilasta, kun molemmat koodaukset on pakattu gzip-ohjelmalla. Tuloksen takana on kaksi yksinkertaista ideaa. Ensin samankaltainen tieto jaetaan omiin tavuvirtoihin gzipin työn helpottamiseksi. Sitten arvojen välillä käytetään suhteellisia viittauksia, jotka on rajattu näkyvyystiedon perusteella niin, että eri funktioissa esiintyvät paikallisesti samankaltaiset peruslohkot näyttävät samankaltaisilta myös gzipin näkökulmasta.

Asiasanat:	LLVM, kääntäjä, välikieli, bitcode, tavukoodi, tiedonpakkaus
Kieli:	Englanti

Abbreviations and Acronyms

LLVM	used to stand for Low Level Virtual Machine but is now a proper name
VM	Virtual machine
PC	Program counter
IR	Intermediate representation
AST	Abstract syntax tree
SSA	Static single-assignment
CFG	Control flow graph
ELF	Executable and linkable format
BB	Basic block
JIT	Just-in-time compilation

Contents

Chapter 1

Introduction

LLVM is an emerging compiler infrastructure project that puts emphasis on modularity and life-long program analysis and optimization. In LLVM programs are compiled to an intermediate representation (IR) that can be analyzed and optimized not only at compile-time on developer's computer but also at run-time on end-user's computer. The IR ties all parts of the LLVM infrastructure together so it would be beneficial to keep software in the IR form as long as possible.

Before LLVM IR can be transmitted over network or stored on disk it needs to be encoded so that it can be stored as a sequence of bytes. Unfortunately the current encoding, bitcode, takes more space than native code and thus hinders the widespread adoption of IR unnecessarily especially on mobile devices where bandwidth and storage space are limited.

In this thesis we present a new LLVM IR encoding that takes less space than bitcode. We first describe a simple encoding and then show how two simple compiler-guided optimizations can lead to significant space savings simply by reordering data in the simple encoding so that general purpose compression programs can work more efficiently.

When we implemented our basic encoding and the two optimizations we used a corpus of real-world programs to guide our development, to establish the losslessness of our encoding and to measure how well it can be compressed. Since no such corpus existed we also developed tools to create such a corpus from the software in the Debian archive and ended up compiling 3511 programs to IR form.

Finally we analysed extensively how the corpus behaves when it is compressed using our basic encoding and one or two of our optimizations. As a summary we observed that on average our format takes only 68% of the space taken by compressed bitcode.

This thesis is organized as follows. Chapter 2 introduces the basics of compilers and data compression. It explores related work and shows that most similar work deals with compressing Java programs. Chapter 3 covers the LLVM system and especially the existing bitcode encoding. It also briefly discusses the use of IR as a software distribution format in Google's Chrome web browser. Chapter 4 describes our encoding in detail and the two optimizations that we have on top of it. It explains many of our design decisions. Chapter 5 describes how we built a corpus of LLVM IR to act as a compression benchmark. The chapter continues with an analysis of how our encoding and its various optimizations behave with different kinds of programs. Chapter 6 finally sums everything up and lists potential improvements that could be made.

Chapter 2

Background

2.1 Compilers

Computer programs are typically written in a human-readable programming language that cannot be directly run by a computer. It is possible to write programs directly in the native language of some machine but this is rarely done since higher-level languages (e.g. C, Java and JavaScript) offer abstractions that improve portability across different machines and make code reuse and maintenance easier and less error-prone. There are two main ways to run such programs: interpretation and compilation. [12]

An interpreter reads the source code of a program and executes instructions written in the source language step by step. To simplify the interpretation task it typically first parses the input program and transforms it into some intermediate representation (IR). With some languages (e.g. Java) it is customary to distribute only the IR to end users. Interpreters are relatively easy to develop and maintain but they are often slower than executing native code.

A compiler also translates the input program to IR but instead of executing the IR step by step it generates native code that performs the desired operations. This offers higher performance but ties the compiler to a specific target machine. Early on in the field it was recognized that if we want to support M source languages and N native languages it is unreasonable for us to need to create $M * N$ different compilers [47]. A compiler is typically divided into a front-end that parses the input language and a back-end that generates code for the target machine. These two parts communicate through an IR that tries not to be specific to any source language or target machine. Modern compilers like GCC [9] and LLVM [32] also include a middle-end that does analysis and optimization. If the IR is designed carefully we only

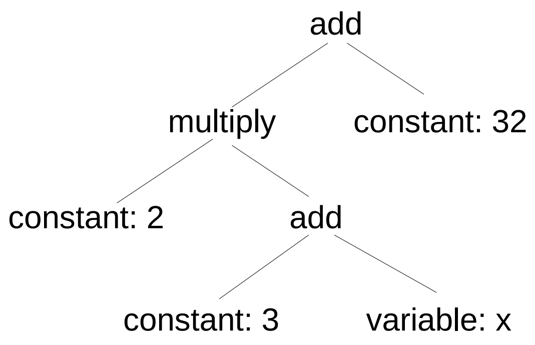

Figure 2.1: Abstract syntax tree for 2 * (3 + x) + 32.

need to create M front-ends and N back-ends and a single optimization pass can work with all of them.

The rest of this section covers different forms of intermediate representation and also introduces some terminology in section 2.1.3.

2.1.1 Abstract syntax trees

When a front-end parses the input program it typically constructs a concrete syntax tree (CST) that contains intricate details like the placement of individual parentheses that are unnecessary for code generation. When we remove these and only preserve the essential semantics of the program we get an abstract syntax tree (AST) where each node corresponds to a construct in the input language. An example of such a tree is presented in figure 2.1. ASTs are often tied to the input language but this does not need to be the case. In the GCC compiler each language front-end internally has its own language specific AST but they all later convert this to a language independent AST called GENERIC. [12]

2.1.2 Three-address code

An AST is useful for verifying the syntax and semantics of a program but it is not the most convenient representation for optimization and code generation since it can contain relatively complex constructs that cannot be broken down into smaller pieces. A three-address code is a sequential and imperative IR where each instruction only performs a single operation and the order of instructions is significant. [12]

The example in listing 2.1 uses only binary operators but unary operators (`a = op b`) and assignments (`a = b`) are naturally also allowed. Also, while

Listing 2.1: Three-address code for 2 * (3 + x) + 32.

```
t1 = 3 + x
t2 = 2 * t1
t3 = t2 + 32
```

an AST can express loops using language specific nodes (`while`, `for`), three-address code lowers these to explicit branch instructions that alter the flow of control.

2.1.3 Basic blocks and flow graphs

A basic block (BB) is a sequence of instructions where the flow of control can only enter the sequence through the first instruction and exit through the last instruction. The last instruction of a basic block is either a branch instruction or an instruction that returns from the current function. No branch in the program ever branches into the middle of the block. Note that function call instructions are permitted inside the basic block. They require special treatment only in languages where they can throw exceptions.

When we assign each instruction of a function to a basic block we can build a control flow graph (CFG): create a node for each basic block and an edge from node A to node B if the last instruction of A can branch to the first instruction of B. Basic block of the first instruction of the function is the entry basic block. Figure 2.2 shows an example with three basic blocks, one branch instruction and one return instruction. Execution of the function begins from the entry basic block BB1.

We say that basic block A dominates basic block B if all flows of control from the entry basic block to B also go through A. It follows that the entry basic block dominates all basic blocks and all basic blocks dominate themselves. In figure 2.2 BB1 dominates BB1, BB2 and BB3. BB2 dominates BB2 and BB3 and BB3 only dominates itself.

The immediate dominator of basic block A is the basic block B that dominates A but does not dominate any basic block that also dominates A. In figure 2.2 BB1 does not have an immediate dominator (the entry basic block never has) but it is the immediate dominator of BB2 and BB3. If we arrange basic blocks into a tree where each node is the child of its immediate dominator we get a dominator tree. This is a concept that we will use later to optimize variable references.

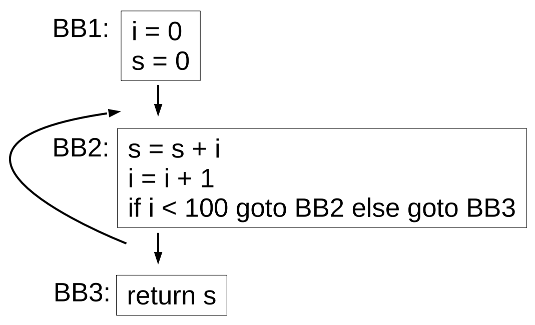

Figure 2.2: Control flow graph of a three-address code function that calculates $\sum_{i=0}^{99} i$.

2.1.4 Static single-assignment

Static single-assignment (SSA) is a refinement of three-address code that makes certain analysis and optimization tasks easier. In SSA variables are assigned only once and then never modified. If a variable in the source language is assigned multiple times the compiler will create a new temporary variable for each assignment. If this sounds confusing it might help to think of SSA as a form of functional programming. [12, 16, 44]

If we try to convert the program in figure 2.2 to SSA form we will encounter some challenges. `s = s + 1` obviously needs a new temporary variable since `s` cannot be assigned twice. We can make that `s2 = s + 1`. Same goes for `i = i + 1`. However, how can the next iteration of the loop refer to the value of `s` from the previous iteration? We cannot use `s` since it is always zero in our example.

To solve this problem SSA includes a special phi instruction that can be used to choose between multiple alternative variables depending on the basic block was used to reach the instruction. We need two of these to convert our example program to SSA form. In figure 2.3 `s2` evaluates to `s` on the first iteration of the loop and to `s3` on all subsequent iterations. `i2` behaves in a similar manner. In SSA a non-phi instruction can only refer to variables that are defined earlier in the same basic block or in some basic block that is a parent of this basic block in the dominator tree. [43]

2.2 Data compression

Data compression is a common solution to cope with limited bandwidth or storage capacity. A compression algorithm tries to exploit the structural

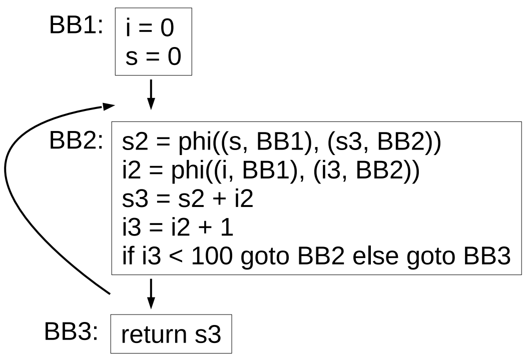
BB1:
i = 0
s = 0
BB2:
s2 = phi((s, BB1), (s3, BB2))
i2 = phi((i, BB1), (i3, BB2))
s3 = s2 + i2
i3 = i2 + 1
if i3 < 100 goto BB2 else goto BB3
BB3:
return s3

Figure 2.3: Program from figure 2.2 after conversion to SSA form.

regularities of an input string of data to find a more compact representation of the same information. In some applications (e.g. audio and video) it is acceptable for the representation to be lossy but here we are only interested in lossless compression where the decompression algorithm can exactly recover the original input.

The task of data compression can be divided into two parts. We first create a model of the data and then code the input string with the help of this model. The model can be static or dynamic. A static model is the same for all compressed strings and can be created e.g. by calculating statistical properties of a large corpus of expected data. It has the advantage that the model does not need to be transmitted along with the compressed data since we can assume that the decompression algorithm already has a copy of the static model. [45]

A dynamic model is created separately for each string of input data. It is a good solution for universal compression when we do not know anything about the type of data since it is able to learn the particular regularities of each input string of data. The disadvantages are that we first need to scan over the entire input data to create the model and then we also need to transfer this model to the recipient. We can get rid of the first disadvantage by using an adaptive model that is constantly updated as more data is coded.

The rest of this section briefly introduces the compression algorithms that are behind the gzip program that we use in chapter 5. In the examples we a use a 4-character alphabet $\{a, b, c, d\}$ and an example input string `abaaabcdabaaabcd`.

Character	Probability	Codeword
a	50%	1
b	25%	01
c	12.5%	000
d	12.5%	001

Table 2.1: Huffman code for our example alphabet.

2.2.1 Run-length encoding

Run-length encoding finds groups of repeated characters, or runs, and encodes them using only one character and a number that indicates the length of the run. The implicit static model here is that the input should contain many runs. The concept is very simple but still useful for example in graphics files that have large areas of the same color. Our example string can be compressed to $a_1b_1a_3b_1c_1d_1a_1b_1a_3b_1c_1d_1$ where the numbers in subscript denote run-lengths.

2.2.2 Huffman coding

Huffman coding estimates the occurrence probability of each individual character of the alphabet and tries to assign short codewords for common characters and longer codewords for less common characters. The modeling process begins by sorting the characters by their probability (see table 2.1). The codewords can then be chosen easily if we first build a binary tree (see figure 2.4) as follows: first create a singleton tree for each character, then repeat the following process as long as there are at least two trees left; take the two least probable trees and merge them by creating a new root node and setting the original two trees as its children. Label one edge with `0` and the other one with `1`. Set the the probability of this new tree to the sum of its children's probabilities. [45]

To code a character we need to look at the shortest path from the root of the tree to the character. We use codeword `01` for `b` since we can reach `b` from root node by first taking an edge labeled with `0` and then taking an edge labeled with `1`. Our example string thus compresses to `10111101000001101111010000 01`. This string is enough to reconstruct the original input if the recipient has a copy of our tree. If not, we need to encode the tree too.

A weakness of a Huffman is that its codewords always use an integer number of bits. If we have a very skewed distribution (e.g. 99.99% for `a`) we end up wasting a lot of bits. A solution is to use arithmetic coding that is in

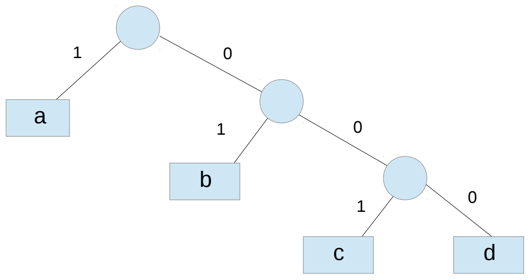

Figure 2.4: Huffman tree for our example alphabet.

many ways similar to Huffman but permits the use of a fractional number of bits per codeword. [45]

2.2.3 LZ77 compression

If the probability of a symbol is independent of the symbols that have appeared before it then Huffman and arithmetic coding are often good solutions. However, data often tends to have structures. In our example `b` is always preceded by `a` and `c` is always preceded by `b`. LZ77 [49] is an adaptive dictionary-based algorithm that uses a fixed-width sliding window to find repeating patterns in the input data. The window is divided into two parts. The search buffer contains data that has already been coded and the look-ahead buffer the data is to be coded next. [45]

At each step of the algorithm we try to find the longest prefix of the look-ahead buffer from the search buffer by scanning from right to left. If a match is found the algorithm outputs a triple that contains the offset of the match, its length and the character that follows the match in the look-ahead buffer. The sliding window is then moved forward by the number of characters in the match plus one. If a match is not found then both offset and length are set to zero. Table 2.2 shows each step of the algorithm applied to our example input when we use eight characters for both buffers. Step 4 exhibits a special case where the match actually extends over to the look-ahead buffer, this is explicitly allowed.

2.2.4 DEFLATE compression

DEFLATE combines LZ77 and Huffman coding. The compressed data consists of a series of blocks. Each block is compressed using LZ77 and the output of LZ77 is then further compressed using Huffman. The exact algorithm is complicated and some of the heuristics are implementation specific

Step	Search buffer	Look-ahead buffer	Output
1		abaaabcd	(0, 0, a)
2	a	baaabcda	(0, 0, b)
3	**a**b	aaabcdab	(2, 1, a)
4	ab**a**	**a**abcdaba	(1, 2, b)
5	abaaab	cdabaaab	(0, 0, c)
6	abaaabc	dabaaabc	(0, 0, d)
7	**abaaabcd**	abaaabcd	(8, 8, _)

Table 2.2: Steps taken by the LZ77 algorithm on our example input. Matched prefixes are in bold.

so we are only going to give an overview here. [20]

The variant of LZ77 that is used in DEFLATE is byte-oriented. It has a 32-kilobyte search buffer and a 258-byte look-ahead buffer. If a match is found the algorithm outputs its length and offset. If a match is not found it simply outputs the next character. The length of the match is limited from 3 to 258 bytes and the offset can address up to 32 kilobytes. It is allowed for the offset to refer to data in the previous block which means that blocks are not compressed independently.

The output of LZ77 is compressed using Huffman coding. For each block we have a pair of Huffman trees. Characters and match lengths are combined into a single alphabet and compressed using one tree while match offsets are compressed using the other tree. The compressor can either use a static tree that is listed in the specification or build a dynamic tree that matches the data of the block more closely. In the later case the trees are encoded in the beginning of the block before the compressed data. The first Huffman tree also contains a special symbol that indicates the end of the block. It is up to the encoder to decide when it is best to start a new block with fresh Huffman trees.

2.2.5 gzip

gzip is a compression program that uses the DEFLATE compression algorithm [20] and stores data in the gzip file format [21]. It is widely used and often referenced in academic literature ([22], [32]) so it serves as a useful metric for evaluating how compression-friendly a file format is. Gzip was created by Jean-Loup Gailly and Mark Adler in 1992 and later adopted by the GNU project [3]. DEFLATE was chosen mostly because use of the popular LZW algorithm [48] was restricted with patents [4].

The gzip file format contains a header, compressed payload and a trailing

CRC-32 checksum of the uncompressed payload. Since there is only one stream for the payload gzip is typically used to compress an archive format like tar that supports multiple files and directories. Some header fields make it difficult to conduct reproducible compression experiments. For example, the MTIME field stores the most recent modification time of the original file and the FNAME field the name of the original file. We noticed that we can make experiments more reproducible by disabling these optional header fields using the `--no-name` option of the gzip command-line interface. [21]

2.3 Related work

Past work on reducing the size of computer programs can be classified in a number of ways. We talk about compaction when the output of the size-reduction process can be executed directly [18, 19, 25, 38] and about compression when a separate decompression step is necessary [22, 23, 37]. If the entire program needs to be fully decompressed before execution can begin we are dealing with a wire format [15, 27, 41] but it is also possible to use an interpreter or a JIT compiler to decompress parts of the program on demand [22, 37, 42].

Binary rewriting tools [19, 25, 38] read native code and generate more compact but equivalent native code as output. They are easy to use since they can be used with existing compilers and run-time environments. However, schemes that operate on the intermediate representation (like AST [46], Java bytecode [15, 17, 27, 41] or SSA [13]) of a compiler have access to much richer semantic information and can perform better.

The exact compression techniques covered in academic literature vary widely from tuning general purpose compression algorithms to employing ad-hoc tricks that are specific to an instruction set or IR. The rest of this section covers existing compaction and compression schemes in more detail and relates them to our own work.

2.3.1 Code compaction

Procedural abstraction identifies common code sequences and abstracts them away to functions that can be reused from multiple locations. It is typically performed before register allocation since it can make otherwise identical code sequences look different. This reduces code size but the extra call and return instructions cause the program to run slower. [12]

Cross-jumping identifies code sequences that end with a branch to the same target and merges them together by keeping only one copy of the sequence and replacing the other instances with a jump instruction that branches to the copy. Since only one extra instruction is added cross-jumping has smaller overhead than procedural abstraction. [12]

Fraser et al. [25] introduce a compressor that combines procedural abstraction and cross-jumping. It uses a suffix tree to identify common subsequences of instructions and takes special care to ignore differences caused by relative branches that look different but branch to the same address. Sequences are classified to be either open or closed. Open sequences end with a branch or return and are suitable for cross-jumping. Closed sequences fall off the end of the sequence and are suitable for procedural abstraction.

Cooper and McIntosh [18] extend the work of Fraser et al. [25] by abstracting away branches and register names. Branches are rewritten to be relative to the program counter and register references are encoded using a reference to the instruction that defined or used the value. This is an important improvement since register allocation can often make otherwise identical code sequences look different. The same paper also notes that traditional compiler optimizations and code compression techniques can be used together to produce best results.

Most modern compilers can optimize code not only for speed but also for size [9, 31, 36]. When we optimize for speed it is useful to align code to a hardware cache line to minimize the number of cache misses or to unroll short loops to reduce the number of executed compare and branch instructions. When the GCC compiler optimizes for size, however, it simply disables such optimizations and tries to apply a simple cross-jumping optimization. [9]

Debray et al. [19] suggest using traditional compiler optimizations at link time since that gives more opportunities for removing redundant and unreachable code after interprocedural constant propagation has been performed. They also note that suffix trees do not cope well with instruction sequences that are semantically equivalent but where some instructions have been reordered and opt to build a CFG instead. With the help of a dominator tree they can then move identical instructions to a point that dominates all occurrences of the instructions.

While the above heuristic approaches try to reduce the size of the program the superoptimizer introduced by Massalin [38] is guaranteed to find the shortest program to compute a given function. It works by exhaustively searching over the space of all possible instruction sequences until it finds a sequence that is equivalent to the input program. This only works for functions that are already short of course but still provides useful information for compiler and algorithm designers. Since testing the equivalence of two instruction sequences is expensive the superoptimizer first runs the candidate sequences with some input values to prune out obviously incorrect sequences.

2.3.2 Code compression

The ARM thumb instruction set can be seen as a form of code compression that is decoded on the fly by the hardware. While normal ARM instructions always take 32 bits the CPU can switch to thumb mode where it executes 16-bit instructions. Not all operations are possible in this mode but since the CPU can switch between modes this is not a major drawback. [34]

Franz and Kistler [23] compress Oberon programs using a variant of the LZW algorithm that operates on the AST. The dictionary is initialized with primitive operations and variables that are in the global scope. When new operations are encoded the dictionary is updated appropriately. When a variable is no longer in the scope it is pruned from the dictionary along with other symbols that refer to that variable. This was originally done to improve the compression ratio but it also has the nice side effect that it is impossible to construct a compressed program that would violate the lexical scoping rules of Oberon.

Ernst et al. [22] introduce a compressed wire code for virtual machine instructions and a compressed interpreteable code called BRISC. The wire code looks at the tree-based lcc IR of the program and splits it into two streams. One stream contains opcodes and the other their operands. Both streams are compressed using move-to-front coding, Huffman coding and finally gzip. The idea of placing semantically similar data into separate streams greatly inspired us in section 4.2 and is also a concept used in the EXI file format [11].

BRISC is a non-wire format that achieves compactness using two simple ideas. Operand specialization creates new opcodes for common opcode-operand patterns. For example if the program often adds the value 5 to a register it makes sense to create an opcode for this specific operation. Opcode combination tries to create opcodes that do the task of two adjacent opcodes. For example if the program often does multiplication and comparison against a specific value it makes sense to promote this to a single opcode. The dictionary that describes both operand specialization and opcode combination is generated using a corpus of typical programs and assumed to be static so that it does not need to be shipped along with the compressed program [37]. [22]

The split-stream dictionary (SSD) scheme uses an input-specific dictionary of instruction sequences and encodes the program using only references to these dictionary entries. Lucco [37] note that programs often reuse short instruction sequences. The sequences can be from one to four instructions long and can not span over more than one basic block. There can be only one branch instruction and it must be the last. The branches are encoded in PC-relative format to make relocated but identical code copies appear identical.

Horspool and Corless [27] take a look at compressing Java class files. They note that typical class files are quite short and contain data in several different formats so that general purpose compression do not have time to adapt. Their CLAZZ format achieves its biggest compression improvement by reordering the constant pool so that entries of the same type are grouped together. This means that CLAZZ does not need to explicitly list the type of each entry anymore. It also allows it to use smaller indices in the code section if it knows that a particular opcode can only reference entries of a specific type. We considered reordering in our own encoding but opted to not do it to ensure that the format stays lossless. CLAZZ separates opcodes and operands like Ernst et al. [22] and compresses them with gzip.

JAZZ [17] improves on CLAZZ by using a unified constant pool for all class files of a Java program and encoding constant pool indices using Huffman coding.

Pugh [41] reorganizes the layout of the Java class files and removes information that is not necessary for running Java programs. He uses separate indices for references to different types of data and applies move-to-front coding to them. The scheme also uses a variable-length encoding for integer literals that is similar to the one used in LLVM bitcode.

Rayside et al. [42] present a non-wire format for Java class files. They replace strings in type references with a tree-like data structure so that common package names do not need to be repeated multiple times. A similar transformation is done for method descriptor strings. Opcodes and opcode pairs are compressed using Huffman coding.

SafeTSA [13, 14, 24, 46] is an innovative way to represent Java programs in an inherently type-safe SSA form. The main design goal is to facilitate better code generation and faster verification. It does this by ensuring that it is impossible to represent illegal programs in the encoding. Just like in the work by Franz and Kistler [23] this has the nice side-effect that the encoding is very compact. The format uses SSA for basic blocks but does not use branch instructions. Instead it encodes the control flow using explicit trees so that the code generator does not need to spend time reconstructing it. They also observed that Java bytecode can represent programs that can not be represented in the Java language. By only supporting control flow structures of the language ("if", "while" and so on) the encoding stays more compact.

SafeTSA achieves type-safety using type separation. Each type uses a separate indexing scheme so that it is impossible for e.g. an integer addition instruction to accidentally refer to a non-integer value. The indexing in

SafeTSA also uses dominator scoping: an instruction can only refer to values that have been defined in the same basic block or in some basic block that dominates the basic block. Our work uses limited type-separation since we have basic blocks, local values and global values in a separate index. We heavily use dominator scoping in section 4.3.

SafeTSA protects against out of bounds accesses to arrays by providing types for both unchecked and checked array indexes. The only way to get a reference to a checked type is to execute an instruction that performs a run-time check. After that the index can be used multiple times without performance penalty. The encoding provides a similar optimization also for run-time null reference checks.

Chapter 3

LLVM

LLVM is a compiler framework with emphasis on lifelong program analysis and optimization. It is not a compiler as such but rather a collection of modular C++ APIs and utilities that can be used to build traditional compilers, JITs, linkers, interpreters, debuggers and static analysis tools. The project began at the University of Illinois as a research project but it is nowadays backed by large companies like Apple and Google. [32]

Several front-ends (C, C++, Objective-C, ActionScript) and back-ends (X86, PowerPC, ARM, SPARC) have been written for LLVM. They are tied together using a language and hardware independent SSA based intermediate representation, the LLVM IR. The IR is low-level enough to not force semantics of a specific input language but it also supports high-level type information to aid a plethora of different optimization passes. A characteristic property of the IR is that it is designed to be serialized into a file and kept around for the whole lifetime of a program to enable optimizations not only at compile-time but also at link-time, install-time and run-time.

The rest of this chapter is organized as follows. We first introduce the LLVM IR in section 3.1 and then discuss its current binary encoding, the LLVM bitcode, in section 3.2.

3.1 LLVM IR

LLVM IR is the glue that binds most parts of the LLVM framework together. It is a three-address code language in SSA form with a language independent type system. The IR uses an instruction set that is not specific to any source language but still tries to provide primitives for implementing features used by high-level languages and higher-level information to support effective optimizations. [32]

The IR can exist in three different forms that contain the same information: the human readable form is ideal for debugging, the in-memory form can be conveniently manipulated using the LLVM API and the bitcode form (section 3.2) can be stored in a file. In chapter 4 we introduce a fourth encoding which is similar to bitcode but more compact. In this chapter, however, we will be using the notation of the human readable form which is also the notation used by the LLVM Language Reference Manual. [8]

A top-level construct in LLVM IR is the module which can represent an object, a library or a complete stand-alone program. It is composed of types, values and some auxiliary information. Values are grouped to global variables and functions. Functions consist of basic blocks which further contain instructions. The IR does not define a specific entry point for the module.

3.1.1 Types

Each value has a type. Most common primitive types are the integer[1], floating point, label and void types. These can be used to build derived types like array, structure, pointer and function types. Type definitions can be self-referential and form recursive structures. For example the C fragment

```
struct node {
    struct node *next, *prev;
    char name[32];
    double balance;
};
```

produces the following type definition:

```
%0 = type { %0*, %0*, [32 x i8], double }
```

Here `%0` is the type identifier. Such a numeric identifier is used when the type does not have textual name. The braces mark a structure and square brackets mark the array. The asterisk is the same in both IR and C, it marks a pointer. The integer type, `i8`, is an 8-bit integer.

3.1.2 Values

Values can be grouped into four categories: arguments, basic blocks, constants and instructions. Instructions can be further grouped into four categories: terminator instructions, binary instructions, memory instructions

[1]Confusingly, the language reference manual explicitly claims that integers are not primitive types. This is an error and has later been corrected in [10].

and other instructions. By definition, a basic block is a sequence of non-terminator instructions followed by a terminator instruction. Typical terminator instructions include the `ret` return instruction and the `br` branch instruction.

Binary instructions include operations like integer addition and multiplication. They take two values as operands and evaluate to a single value. For example the C fragment

```
3*x + 7
```

produces the following IR when `x` is an `int`:

```
%2 = mul i32 %0, 3
%3 = add nsw i32 %2, 7
```

Here `%0`, `%2` and `%3` are identifiers for the values we have. Just like types, if a value does not have a textual name, a numeric identifier is used in the human readable IR form. The `nsw` flag of the `add` instruction specifies that the compiler can assume that no signed overflow will happen.

All memory accesses are done using dedicated `load` and `store` instructions. Unlike in GCC [39], memory locations are not in SSA form. Accesses are typed but nothing prevents loads and stores to the same memory address using different types. To support typed address arithmetic LLVM provides a special `getelementptr` instruction. It is one of the most commonly misunderstood instructions although it is a simple operation. Given a typed pointer to some derived type it calculates the address of a subelement. For example, if we assume that we have a `struct s1 *ptr` with

```
struct s3 {
    int e[16];
};
struct s2 {
    int  c;
    struct s3 d[10];
};
struct s1 {
    int a;
    struct s2 b[10];
};
```

then the expression

```
&(ptr->b[3].d[4].e[5])
```

Listing 3.1: example1.c

```
#include <math.h>
double f(double s, int x, int *y) {
    for (int i = 0; i < x; i++) {
        s += pow(y[i], 2.2);
    }
    return s;
}
```

translates to

```
%2 = getelementptr inbounds %0* %1, i32 0, i32 1, i32 3, i32 1, i32 4, i32 0, i32 5
```

The indices `3`, `4` and `5` are self-explanatory. The first `0` refers to the fact that we are talking about the object pointed to by the pointer and not e.g. the object that is after it (`ptr[1]`). The two `1` indices come from the fact that `b` is the second element of `struct s1` and `d` is the second element of `struct s2`. The type `i32` refers to the type of these constant indices, it does not have anything to do with the type of the pointed object. The `inbounds` keyword specifies that the value of the instruction is undefined unless `%1` points to an allocated object and value of the `getelementptr` instruction also points to the same object.

3.1.3 Complete example

In this section we will give a walk-through of a complete LLVM IR module. We will start from a simple C compilation unit that you can see in listing 3.1. The example function `f` is simple but demonstrates several important concepts. It uses both primitive types (`double`, `int`) and derived types (`int *`). Since the function has a loop we also get several basic blocks and phi nodes.

To generate LLVM IR we will use the llvm-gcc compiler with the following command line[2]:

```
llvm-gcc -std=c99 -fno-builtin-pow -O2 -emit-llvm -c -o tmp.bc example1.c
opt -load ./LLVMcanonizeBBOrderPass.so -canonizeBBOrderPass -disable-opt \
  -strip -strip-debug -o example1.bc -f tmp.bc
llvm-dis -o example1.ll example1.bc
```

[2]LLVMcanonizeBBOrderPass is explained in chapter 4.1.

Listing 3.2: example1.ll

```
; ModuleID = 'example1.bc'
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:...
target triple = "i386-linux-gnu"

define double @f(double, i32, i32* nocapture) nounwind {
; <label>:3
  %4 = icmp sgt i32 %1, 0
  br i1 %4, label %5, label %15

; <label>:5
  %6 = phi double [ %0, %3 ], [ %12, %5 ]
  %7 = phi i32 [ 0, %3 ], [ %13, %5 ]
  %8 = getelementptr i32* %2, i32 %7
  %9 = load i32* %8, align 4
  %10 = sitofp i32 %9 to double
  %11 = tail call double @pow(double %10, double 2.200000e+00) nounwind
  %12 = fadd double %11, %6
  %13 = add nsw i32 %7, 1
  %14 = icmp eq i32 %13, %1
  br i1 %14, label %15, label %5

; <label>:15
  %16 = phi double [ %0, %3 ], [ %12, %5 ]
  ret double %16
}

declare double @pow(double, double) nounwind
```

You can see the resulting IR in listing 3.2. As you can see we have two functions and three basic blocks. The first function is defined in the module but the second one is only a declaration for an external function. The basic block in the middle contains the body of the for loop of the C function `f`. The `target datalayout` and `target triple` fields describe the target architecture and environment.

Our example uses nine distinct types. It is easy to notice `i1`, `i32`, `i32*` and `double` but there are also other types. The type of the `f` function is `double (double, i32, i32*)` and type of the `pow` function is `double (double, double)`. Since both appear as global values they are represented by a pointer to a memory location and thus we need the corresponding pointer types. Finally, the type of a `br` instruction is `void`.

The example contains 27 values: two functions, three basic blocks, five arguments, three constants and 14 instructions. The execution of the `f` function begins by comparing its second argument against the constant zero. If it is greater then the execution continues in the second basic block, else we exit via the third basic block.

The phi instructions in the beginning of the second basic block bind values of consecutive loop iterations together. If we come from the first basic block the value `%6` is `%0` and if we come from the second basic block it is `%12`. The `getelementptr` computes address of the array element `y[i]` which is then loaded using the `load` instruction. The result is then converted to the double type and given as an argument to the `pow` function.

The return value of the `pow` function is added to the `s` variable using the `fadd` instruction. Then the `i` variable is incremented using the `add` instruction and compared against `x`. If the values are equal we branch to the third basic block else we continue the loop and branch again to the second basic block. The third basic block simply returns the value of the `s` variable.

3.1.4 IR as a software distribution format

One of the motivations for a compression-friendly encoding is to enable widespread software distribution in IR form. Although the IR is a higher level language than native code it currently always still targets some real architecture. The `target datalayout` field of listing 3.2 encodes architecture specific details such as endianness, pointer size and alignment and the `target triple` field explicitly lists the target architecture and OS environment. These fields exist because the IR contains subtle architecture specific assumptions and it would be dangerous to execute the IR on a wrong architecture. [26]

When the C fragment

```
struct foo { int a; int b; };
void bar(struct foo);
int main() {
    struct foo x;
    x.a = 1;
    x.b = 2;
    bar(x);
}
```

is compiled to LLVM IR the number of operands of the resulting call instruction depends on the architecture. Calling conventions vary from architecture to architecture. On X86-64 such a small struct is passed as a single argument [28] but on ARM it is passed as two 4-byte arguments [35]. Since programs need to call external libraries they need to follow the ABI of the target architecture. [30]

The C expression `sizeof(int*)` is lowered to a simple constant integer in the IR. Since all of this is done at compile-time programs can show completely different behavior when they target different architectures. [26, 30]

On X86 LLVM represents the `long double` type using a target specific type `x86_fp80` but on ARM it uses the generic `double` type. [30]

Several strategies can be used to overcome these limitations. The Google portable native client project [29] uses a restricted subset of LLVM IR which always targets the same imaginary architecture:

```
target datalayout = "e-i1:8:8-i8:8:8-i16:16:16-i32:...
target triple = "le32-unknown-nacl"
```

This architecture is little-endian, has always 32-bit pointers and a 32-bit `int` type. Architecture specific types are not supported and programs are also not allowed to make any assumptions on stack direction, alignment or layout.

Wordcode is another interesting solution to the same problem. It uses a two-stage compilation strategy where software is first compiled into a target independent IR which is only later lowered to a target specific IR. The target independent wordcode adds a new type to represent `long double` on all architectures, additional alignment information to all integer types and explicit support for bitfields in structs so that members can be accessed without having to rely on architecture specific memory layout. This wordcode uses the `target triple` of `ovm-none-linux`. When it is lowered to target dependent wordcode most of the new constructs disappear and the result is almost like the existing LLVM IR. [30]

3.2 Bitcode

Bitcode is the binary encoding of LLVM IR used by LLVM 2.6. It was originally introduced in LLVM 2.0 and continues to evolve synchronously with the LLVM IR. Earlier versions used a format called LLVM bytecode which since LLVM 1.4 has been internally compressed using bzip2 [5]. Moving to bitcode was motivated by the need to let the JIT start execution without having to first uncompress the whole file. Bitcode consists of a structured bitstream format and an encoding of LLVM IR on top of it. The format is relatively complicated and not fully documented so we will only try to give an overview here.

3.2.1 Bitstream format

The bitcode specification [33] compares the bitstream format to XML in that it supports nested structures and is not specific to LLVM IR in any way. As the name suggests, the stream consists of bits, not bytes. The stream is built out of four primitive types. Fixed width integers use a fixed number of bits. Variable width integers have some default size but can encode larger numbers by setting their continuation bit to 1. This scheme is somewhat similar to UTF-8. 6-bit characters are used to encode identifiers. Finally, the bitstream is sometimes aligned to a full 32-bit word by padding it with zeroes. This helps byte oriented compression algorithms and makes it possible to memory map parts of the file.

The primitive types are arranged in blocks and data records. Blocks provide a way of representing nested structures. Their header encodes their size so a reader can jump over blocks that it is not interested in. Data records consist of a record code and a number of integers whose meaning depends on the context. If the record code is UNABBREV_RECORD (3) these integers are expected to be 6-bit variable width integers but if it is something else it is expected to denote an abbreviation identifier that has been defined earlier in the file.

Abbreviations are an attempt to compress the encoding by providing denser encoding for commonly used records. Each abbreviation can be defined either for one block or for itself and all of its subblocks. An abbreviation is defined using a DEFINE_ABBREV record. It specifies the number of operands in the abbreviation and their types. Operands can be literal or templates that can be instantiated when the abbreviation is used. It is up to the encoder to choose which abbreviations it wants to use. This means that it is difficult to evaluate the compression-friendliness of bitcode without also mentioning which encoder was used to produce it. Fortunately LLVM 2.6 comes with only one encoder.

3.2.2 IR encoding

LLVM IR encoding on top of the bitstream is not fully documented so we have to rely on the source code of the implementation to understand it [7]. The encoder starts by recursively enumerating global variables, functions, global variable initializers, aliases and aliasees. It is important to note that it does not enumerate instructions, basic blocks or arguments at this stage yet. The encoder then enumerates all types used in the program, even those used by the values that were not enumerated yet.

The encoder emits a MODULE_BLOCK and continues by emitting its subblocks. Type information is in TYPE_BLOCK, constants in CONSTANTS_BLOCK, metadata in METADATA_BLOCK and so on. It then goes through the list of all functions. When it enters a function it enumerates its arguments and instructions. It then emits the number of basic blocks as a FUNC_CODE_DECLAREBLOCKS record and continues emitting instructions of the function. Basic blocks are not encoded explicitly, it is up to the decoder to recognize terminator instructions that by definition separate basic blocks. When all instructions have been emitted the encoder removes arguments and instructions of the function from the enumeration since they can not be referenced from other functions and doing so is useful to keep the indices small.

Chapter 4

The CI4 encoding

This chapter introduces CI4, our encoding for the LLVM IR. We did not originally plan to design a completely new encoding but rather hoped to develop incremental improvements to the existing bitcode format. However, after studying the LLVM IR (chapter 3) and the existing bitcode format we were convinced that a custom format would make it much easier to experiment with new ideas. The bitcode format, as the name suggests, is bit oriented. This confuses many general purpose compression tools that operate on byte streams. Bitcode also supports seeking, it is possible to start executing code without having to parse the whole bitcode file first. This is a useful property but it is not necessary when we are designing a format for compact storage and transmission of programs. Seekability complicates both design and implementation of a file format unnecessarily. Since CI4 is lossless the recipient can always convert it back to bitcode if seekability is desired.

The LLVM IR is relatively complicated. It took significant time to understand how to encode LLVM IR into a custom format and to stay fully lossless. We adopted an incremental development model where we first designed a basic encoding and then started testing optimizations on top of that. This process involved a lot of trial and error and could not have succeeded without a large corpus of test programs (chapter 5). The test programs ensured that we stayed lossless after each incremental improvement but also gave us immediate feedback on how well our optimizations improved the compression ratio.

The rest of this chapter is organized as follows. We first introduce our basic encoding for the LLVM IR in section 4.1. Then we cover our two main optimizations: separating similar information to different byte streams in section 4.2 and applying dominator scoping to value references in section 4.3. Finally, in section 4.4 you can see a complete encoding example of a small program.

4.1 Basic encoding

As described in chapter 3 LLVM modules are composed of types, values and some auxiliary information. We encode each type and value in a fairly straightforward way but need to take extra care on how to encode references between types and values efficiently. As we learned in chapter 2.2.5 gzip can recognize repeating patterns only if they are byte-aligned. CI4 uses unsigned 32-bit little-endian integers (`u32`) as its primary data type. We do not attempt to do any bit or byte level packing and leave this to the compression programs to handle. In the extreme case this means that even boolean values are encoded using 32 bits. This might seem foolish but experiments show that at least gzip manages this very well. The only exception to this rule are `i8` arrays which are most commonly used to store strings. We encode them simply using unsigned 8-bit bytes (`u8`) since general purpose compression algorithms are often designed to handle strings in their natural encoding (table A.1) efficiently.

Table 4.1 describes the overall structure of our file format. Each file contains exactly one LLVM module. The module can represent an object file, a library or a complete stand-alone program but we do not need to differentiate between these since they look the same in IR. The magic numbers in the beginning and middle of the file can be used to automatically recognize CI4 and offer a sanity check for the decoder by clearly separating type and value data. The identifier, data layout, target triple and inline assembler fields are global properties of the module. CI4 treats these as opaque strings and does not try to encode them in any special way as they do not usually take much space.

What follows is the description of all types and values in the module. At this point you might be wondering how LLVM knows where the execution of the program is supposed to start. It turns out that the IR does not explicitly specify an entry point at all. It is up to the run-time to decide that it should start execution from the function that is named `main`.

It is important to note that our decoder only does a single pass over the file. This makes handling forward references somewhat difficult and affects many of our design decisions. We encode types and non-instructions before instructions since types cannot refer to values and non-instructions cannot refer to instructions. We also group instructions of the same basic block and function together to make decoding easier.

Type	Count	Description
u32	1	magic number 0x1236115
string	1	identifier
string	1	data layout
string	1	target triple
string	1	inline assembler
u32	1	N_t: number of types
type	N_t	all types
u32	1	magic number 0x1236115
u32	1	N_v - N_{vi}: number of non-instruction values
u32	1	N_{vi}: number of instructions
value	N_v - N_{vi}	all non-instruction values
ref_v	1	F_1: reference to first function
u32	1	N_{F_1}: number of basic blocks in F_1
u32	1	$N_{F_{1,0}}$: number of instructions in basic block 0 of F_1
value	$N_{F_{1,0}}$	instructions of basic block 0 of F_1
u32	1	$N_{F_{1,1}}$: number of instructions in basic block 1 of F_1
value	$N_{F_{1,1}}$	instructions of basic block 1 of F_1
...	...	...
ref_v	1	F_2: reference to second function
u32	1	N_{F_2}: number of basic blocks in F_2
u32	1	$N_{F_{2,0}}$: number of instructions in basic block 0 of F_2
value	$N_{F_{2,0}}$	instructions of basic block 0 of F_2
u32	1	$N_{F_{2,1}}$: number of instructions in basic block 1 of F_2
value	$N_{F_{2,1}}$	instructions of basic block 1 of F_2
...	...	...
...	...	...
u32	1	N_g: number of global variables
ref_v	N_g	references to global variables

Table 4.1: Encoding of an LLVM module.

4.1.1 Type references

The encoding process begins by assigning a numeric identifier to each type used in the module. If we have N_t types we use the integers 0, ..., N_t - 1. The types are sorted by their identifier so that the file format does not need to explicitly encode the identifier. When a type or value needs to refer to a type the reference is encoded as a `u32` which we call ref_t.

Since types can refer to types we can get cyclic structures and thus can not avoid forward references. This is not a problem for the decoder since it can first instantiate N_t opaque types and later use the `refineAbstractTypeTo` method of the LLVM API when it discovers the real details of a type [6].

4.1.2 Value references

Value references are more complicated than type references. If we have N_v values out of which N_{vi} are instructions we assign the integers 0, ..., N_v - N_{vi} - 1 to non-instructions and use N_v - N_{vi}, ..., N_v - 1 for instructions. Note that here "non-instructions" does not include basic blocks even though they too are values in the LLVM IR. The reason for this is that as an optimization we use a separate indexing scheme (ref_bb) for basic blocks that is described later in this section. There is nothing similar to opaque types for values in LLVM IR so forward references become difficult to handle in a decoder that only does a single pass through the file. For example, when the decoder reads an integer multiplication instruction it must have already decoded its operands or it can't instantiate the object that represents the instruction using the LLVM API.

It turns out that there are only two cases where we get cyclic references with values. The first case is obvious to anyone who knows SSA: if the program has a loop we need a phi instruction that can refer to itself either directly or via a chain of other instructions. The second case is not so obvious. It turns out that a global variable in LLVM IR can refer to an initializer constant expression that refers back to the global variable itself. An example of such a construction can be seen in the C fragment

```
char *var1 = (char *)&var1;
```

which translates to the following bit of LLVM IR:

```
@var1 = global i8* bitcast (i8** @var1 to i8*)
```

When the above two causes of cycles are handled specially in the decoder we can find an ordering for the values where no other forward references occur. The basic idea is simple. Before we can encode a value we need to

Listing 4.1: Pseudo code to assign a unique identifier to an LLVM value.

```
assignValueID(V):
    if an ID has been assigned to V:
        return
    if V is a global variable:
        assign a new ID to V
    else:
        assign a dummy temporary ID to V to prevent infinite recursion
    if V is not a phi instruction:
        for all operands OP of V:
            assignValueID(OP)
    if V is not a global variable:
        assign a new ID to V
```

encode its operands. Pseudo code that assigns each value a unique numerical identifier by recursively assigning identifiers to operands of values before themselves is shown in listing 4.1.

There is one further complication associated with value references. If you look at table 4.1 you will see that instructions are grouped by their parent basic block and parent function. Most of the data in programs is in instructions. By grouping instructions and basic blocks together we can avoid having to explicitly specify which instructions are part of which basic blocks. If a function has N_{bb} basic blocks we assign the integers 0, . . . , N_{bb} - 1 to the basic blocks of the function and use these as identifiers. We call such identifiers ref_bb and encode them normally using an `u32`. This is possible since the `br` branch instruction can only refer to basic blocks of the same function. After this optimization most integers in the `bbref` streams were less than 100. This encoding guarantees also that if we have two functions that have identical control flow graphs their `bbref` streams are identical.

If an instruction references (or uses) an instruction we need to encode the operand before its user. It is useful to look at the dominators of a basic block here. If basic block A dominates basic block B we need to encode basic block A before basic block B since basic block B can contain instructions that use instructions in basic block A but not vice versa. To solve this problem we wrote an LLVM pass, `canonizeBBOrderPass`, that topologically sorts basic blocks so that dominators always appear before the blocks that they dominate. This is strictly speaking not a lossless transformation but we argue that this does not affect the semantics of the program in any way. In our benchmarks (chapter 5) we use this ordering for both bitcode and CI4 to make the comparison fair.

Basic type identifier	Basic type
0	primitive
1	opaque
2	integer
3	function
4	array
5	struct
6	vector
7	pointer

Table 4.2: Basic type identifiers.

In some cases the LLVM C++ API returns a NULL pointer when a pointer to a value was expected. This happens for example when we have a return instruction in a function that does not return anything. Such a reference is encoded using the number 0xffffffff.

4.1.3 Type encoding

When each type has been assigned a unique identifier we continue by encoding the types themselves. The types are encoded in ascending order by their identifier so that we do not need to explicitly encode the identifier. This saves space and also makes the format much more compression-friendly. Having a unique but slowly incrementing integer intermixed with type encoding would confuse many general purpose compression programs.

As we saw in chapter 3 the LLVM type system is quite rich. In addition to simple things like integers and pointers it can represent more complex types like structures containing arrays of structures containing pointers to functions. It can also contain recursive or cyclic types, for example a structure can contain a pointer to a structure that contains a pointer to a structure. We can construct all these types using just eight basic types. The types are shown in table 4.2 along with the numerical value, basic type identifier, that we use in our encoding.

Encoding of a specific type begins by encoding its basic type identifier as a `u32` (table 4.3). What follows after that is specific to the type in question. You can see some examples in tables A.2, A.4, A.6 and A.8 or look at a complete example in subsection 4.4.

Type	Count	Description
u32	1	basic type identifier
...	...	type specific encoding

Table 4.3: Encoding of an LLVM type.

4.1.4 Value encoding

Value encoding is similar to type encoding. Like types, we encode values in the order given by their identifier to avoid having to explicitly encode the identifier. This ordering is much more complicated for values but we do not need to worry about it when we encode the values themselves. We divide values into 29 groups and assign each group a numerical value, value type identifier, that you can see in table 4.4.

Just like with types we first encode the value type identifier as an `u32` and then encode the value itself (table 4.5). You can see examples of values in tables A.10 and A.23 or look at the complete example in subsection 4.4.

4.1.5 Limitations

Due to the testing done in chapter 5 we are confident that CI4 is complete enough to encode most real world programs. There are however some known limitations. If we compare our format to the bitcode format the main limitation is that our format is a wire format which is intended to be used to distribute software to end-users. We do not support seeking so it is necessary to decode the complete file before its contents can be run.

We do not support embedded metadata which are mostly used for debugging information. This means that users must first run their programs through

```
opt -strip -strip-debug
```

before converting them to CI4. This should be a reasonable limitation since usually production software is shipped to end-users without debugging information. If the need to add support for debugging information arises the format can be augmented to support it with relatively little work.

We do not support global aliases. They did not seem to be common in our corpus so we opted to not implement them.

We require basic blocks to be ordered so that dominators appear before the blocks that they dominate to make value references work without cycles. Reordering of basic blocks should not change the semantics of the program

Value type identifier	Value type	Description
0	constant_int	constant integer
1	constant_array	constant array
2	global_variable	global variable
3	undef	the undefined value
4	function	function
5	bb	basic block
6	constant_expr	constant expression
7	return_inst	return instruction
8	gep_inst	get element pointer instruction
9	call_inst	call instruction
10	invoke_inst	invoke instruction
11	constant_null	constant NULL value
12	constant_agg_zero	all zero aggregate value
13	store_inst	store instruction
14	load_inst	load instruction
15	binop	binary operator
16	alloca_inst	alloca instruction
17	cmp_inst	cmp instruction
18	phi	phi instruction
19	branch_inst	branch instruction
20	argument	argument of a function
21	select_inst	select instruction
22	cast_inst	cast instruction
23	switch_inst	switch instruction
24	unreachable_inst	unreachable instruction
25	constant_struct	constant struct value
26	inline_asm	inline assembler value
27	constant_fp	constant floating point value
28	extract_value_inst	extract value instruction

Table 4.4: Value type identifiers.

Type	Count	Description
u32	1	value type identifier
...	...	value specific encoding

Table 4.5: Encoding of an LLVM value.

Stream name	Description
main	The default stream
ref	References to values (except basic blocks)
bbref	References to basic blocks
string	i8 arrays
value_op	Value type identifier fields
constant_int	Constant integers
constant_array	Constant arrays
global_variable	Global variables
...	One entry for each value type identifier (see table 4.4)

Table 4.6: Description of streams.

but if this becomes a real problem the format could be modified to save information on basic block ordering so that the decoder can undo this step.

Finally our work is built on top of LLVM 2.6 and can not be expected to work with newer or older LLVM versions without some further development.

4.2 Multiple streams

Our basic encoding is not very compact on its own. Almost all fields are encoded using 32 bits even though less bits could be used. This is not a problem. As explained in section 4.1 our goal is to develop a format that compresses well using general purpose compression tools and thus the uncompressed size is of little interest to us. The important part is that 32-bit values are natural for byte oriented compression tools.

After experimenting with different ideas it turns out that simply reordering the data can improve compression ratio substantially. We decided to split the data into 34 different byte streams and then proceeded to compress these separately. The idea is to group data from similar contexts together. As we learned in chapter 2.2.5 gzip can only recognize repeating patterns if they occur inside a 32 kB window. By putting similar data to the same stream we can improve the probability that this happens. We started by putting everything to a stream named `main` and then gradually identified which parts of the encoding would benefit from a separate stream.

We ended up creating separate streams for value references, value type identifiers, basic block references and strings. We also created one stream for each value type identifier so that similar values would get grouped together. It is surprising that this alone allowed us to do slightly better than the original bitcode format in terms of compressed size. You can see this grouping in table

4.6 and a concrete example in table 4.7.

Our implementation saves each stream to a separate file. This is makes it easy to use external compression programs and analysis tools but is probably not very user-friendly. Since the number of streams is fixed a trivial format for encapsulating all streams to a single file would only need to list the sizes of each stream before their payload. The decoder starts by reading the `main` stream. When it reads N_{vi} it knows how many values it needs to read. It reads value type codes of all values from the `value_op` stream and then variable length data from value specific streams.

4.3 Dominator scoping

After grouping data to separate streams we observed that the `ref` stream takes most of the space in a typical program. This motivated us to rethink the way we encode references. Our original encoding assigned unique indices to all values (except basic blocks) and then encoded them in sequential order so that we do not need to explicitly encode the identifier. The drawback is that if we have two identical functions the local values inside those functions get different indices and gzip can no longer see that the functions are identical. As we learned in chapter 2.2.5 gzip can recognize repeating patterns only if they are truely identical on the byte level. Mixing unique integers to the data stream destroys compresion performance. This can be fixed by using a different indexing scheme for global and local values. We simply chose to use the most significant bit of the identifier to denote whether the reference is global or not. The bitcode encoder does something similar: If the program has N global values and a particular function has N_l local values it uses the identifiers $N, \ldots, N + N_l$ - 1 for local values.

When we look at local value references we can make a few observations. An instruction can only refer to local values that have been defined earlier in the same basic block or in some basic block that is a parent of this basic block in the dominator tree (see section 2.1.4). Instructions often refer to results of nearby instructions. With this in mind we decided to use a relative indexing scheme for local value references. Index 0 refers to the nearest value that can be legally referenced, index 1 to the second to nearest value and so on. This ensures that `ref` stream data for basic blocks with the same value referencing pattern is identical. You can see the effects of this new indexing scheme if you look at the ref_v values of table 4.7 that are in parenthesis.

Encoding data using dominator scoped relative references is easy since the encoder can first build the dominator tree and then traverse it starting from the current basic block towards the entry basic block until it finds the matching value. However, decoding this format in one pass is more complicated since the decoder can not know the dominator tree in advance. To make it feasible to implement a one-pass decoder we decided to explicitly write a reference to the immediate dominator of a basic block before the instructions of that basic block. In table 4.7 you can see two such entries: "immediate dominator of basic block 1 of F_1" and "immediate dominator of basic block 2 of F_1". The first basic block (index 0) does not need this information since it is by definition the entry basic block and thus has no dominators.

4.4 Encoding example

In this section we will encode the example LLVM IR in listing 3.2 into CI4. You can see the result in table 4.7. Since the table is quite long it might be helpful to first revisit table 4.1 to first get an idea of the overall structure. As described in section 4.1 all items are either 32-bit (`u32`) or 8-bit (`u8`) unsigned little-endian integers but some have more abstract names: ref_t is a reference to a type, ref_v is a reference to a value and ref_bb is a reference to a basic block. Types and values are listed in ascending order by their identifier and identifiers are chosen so that instructions get grouped by their basic block and basic blocks by their parent function.

The encoding is relatively straightforward. If you look at for example value 18 you can see that it is encoded using four numbers. 15 is the value type identifier that tells us that we are dealing with a binary operator and 8 is the opcode that narrows the instruction down to `fadd`. The remaining two numbers are references to other values. Value 17 is the return value of the `pow` function and 12 is one of the phi instructions. Both of these refer to values that were defined earlier in the file. After dominator scoping these references become 0 and 5 respectively. This is because value 17 is just before value 18 and value 12 is the fifth possible value that value 18 can refer to.

The only forward references with values in our example occur with values 12 and 13 that refer to values 18 and 19 respectively. All other values refer only to values that were defined earlier in the file. There is some room for optimization. For example we explicitly encode the `void` type since that is the type of the `br` instruction but nothing in fact refers to this type in CI4.

Table 4.7: Complete encoding example. Values in parenthesis in the data column show the result after dominator scoping (section 4.3).

Stream	Type	Data	Description
main	u32	1…	magic number (19095829)
main	u32	11	identifier (“example.bc”)
main	u8	’e’	
main	u8	’x’	
main	u8	’a’	
…	…	…	
main	u32	117	data layout (“e-p:32:32:32-i1:8:8-…”)
main	u8	’e’	
main	u8	’-’	
main	u8	’p’	
…	…	…	
main	u32	14	target triple (“i386-linux-gnu”)
main	u8	’i’	
main	u8	’3’	
main	u8	’8’	
…	…	…	
main	u32	0	inline assembler (“”)
main	u32	9	N_t: number of types
			type 0: double (double, i32, i32*)*
main	u32	7	- pointer type
main	u32	0	- address space
main	ref_t	1	- pointee type
			type 1: double (double, i32, i32*)
main	u32	3	- function
main	u32	0	- not vararg
main	ref_t	2	- return type
main	u32	3	- number of arguments
main	ref_t	2	- type of first argument
main	ref_t	3	- type of second argument
main	ref_t	4	- type of third argument
			type 2: double
main	u32	0	- primitive type
main	u32	2	- LLVM type id (2 = double)
			type 3: i32
main	u32	2	- integer type
main	u32	32	- bit width
			type 4: i32*
main	u32	7	- pointer type
main	u32	0	- address space
main	ref_t	3	- pointee type
			type 5: double (double, double)*
main	u32	7	- pointer type
main	u32	0	- address space

Continued on next page.

Table 4.7: Encoding example continued from previous page.

Stream	**Type**	**Data**	**Description**
main	ref_t	6	- pointee type
			type 6: double (double, double)
main	u32	3	- function
main	u32	0	- not vararg
main	ref_t	2	- return type
main	u32	2	- number of arguments
main	ref_t	2	- type of first argument
main	ref_t	2	- type of second argument
			type 7: i1
main	u32	2	- integer type
main	u32	1	- bit width
			type 8: void
main	u32	0	- primitive type
main	u32	0	- LLVM type id (0 = void)
main	u32	1...	magic number (19095829)
main	u32	10	N_v - N_{vi}: number of non-instruction values
main	u32	14	N_{vi}: number of instructions
			value 0: the function “f”
value_op	u32	4	- function
function	ref_t	1	- type
function	u32	0	- linkage (0 = external linkage)
function	u32	1	- name (“f”)
function	u8	’f’	
function	u32	0	- return attributes
function	u32	0	- attributes of first argument
function	u32	0	- attributes of second argument
function	u32	2097152	- attributes of third argument (nocapture)
function	u32	32	- function attributes (nounwind)
function	u32	0	- calling convention (C)
function	u32	0	- alignment
			value 1: the function “pow”
value_op	u32	4	- function
function	ref_t	6	- type
function	u32	0	- linkage (0 = external linkage)
function	u32	3	- name (“pow”)
function	u8	’p’	
function	u8	’o’	
function	u8	’w’	
function	u32	0	- return attributes
function	u32	0	- attributes of first argument
function	u32	0	- attributes of second argument
function	u32	32	- function attributes (nounwind)
function	u32	0	- calling convention (C)
function	u32	0	- alignment
			value 2: argument double %0
value_op	u32	20	- argument

Continued on next page.

Table 4.7: Encoding example continued from previous page.

Stream	**Type**	**Data**	**Description**
argument	ref_t	2	- type
ref	ref_v	0	- parent function
			value 3: argument i32 %1
value_op	u32	20	- argument
argument	ref_t	3	- type
ref	ref_v	0	- parent function
			value 4: argument i32* %2
value_op	u32	20	- argument
argument	ref_t	4	- type
ref	ref_v	0	parent function
			value 5: argument %0
value_op	u32	20	- argument
argument	ref_t	2	- type
ref	ref_v	1	- parent function
			value 6: argument double %1
value_op	u32	20	- argument
argument	ref_t	2	- type
ref	ref_v	1	- parent function
			value 7: i32 0
value_op	u32	0	- constant integer
constant_int	u32	32	- bit width
constant_int	u32	1	- value as a string
constant_int	u8	'0'	
			value 8: double 2.200000e+00
value_op	u32	27	- floating point constant
constant_fp	u32	1	- double precision
constant_fp	u32	1...	most significant word (1073846681)
constant_fp	u32	2...	least significant word (2576980378)
			value 9: i32 1
value_op	u32	0	- constant integer
constant_int	u32	32	- bit width
constant_int	u32	1	- value as a string
constant_int	u8	'1'	
main	ref_v	0	reference to F_1
main	u32	3	N_{F_1}: number of basic blocks in F_1
main	u32	2	$N_{F_{1,0}}$: number of instructions in basic block 0 of F_1
			value 10: %4 = icmp sgt i32 %1, 0
value_op	u32	17	- compare instruction
cmp_inst	u32	43	- opcode
cmp_inst	u32	38	- predicate (sgt)
ref	ref_v	3	- first operand
ref	ref_v	7	- second operand
			value 11: br i1 %4, label %5, label %15
value_op	u32	19	- branch instruction
branch_inst	u32	0	- conditional
bbref	ref_bb	1	- first target

Continued on next page.

Table 4.7: Encoding example continued from previous page.

Stream	Type	Data	Description
bbref	ref_bb	2	- second target
ref	ref_v	10/(0)	- condition value
main	u32	10	$N_{F_{1,1}}$: number of instructions in basic block 1 of F_1
bbref	ref_bb	(0)	immediate dominator of basic block 1 of F_1
			value 12: %6 = phi double [%0, %3], [%12, %5]
value_op	u32	18	- phi instruction
phi	ref_t	2	- type
phi	u32	2	- number of incoming basic blocks
ref	ref_v	2	- value if coming from first incoming basic block
bbref	ref_bb	0	- first incoming basic block
ref	ref_v	18/(3)	- value if coming from second incoming basic block
bbref	ref_bb	1	- second incoming basic block
			value 13: %7 = phi i32 [0, %3], [%13, %5]
value_op	u32	18	- phi instruction
phi	ref_t	3	- type
phi	u32	2	- number of incoming basic blocks
ref	ref_v	7	value if coming from first incoming basic block
bbref	ref_bb	0	first incoming basic block
ref	ref_v	19/(3)	- value if coming from second incoming basic block
bbref	ref_bb	1	- second incoming basic block
			value 14: %8 = getelementptr i32* %2, i32 %7
value_op	u32	8	- get element pointer instruction
gep_inst	u32	2	- number of operands
ref	ref_v	4	- first operand
ref	ref_v	13/(0)	- second operand
gep_inst	u32	0	- not inbounds GEP
			value 15: %9 = load i32* %8, align 4
value_op	u32	14	- load instruction
load_inst	u32	0	- not volatile
load_inst	u32	4	- alignment
ref	ref_v	14/(0)	- operand
			value 16: %10 = sitofp i32 %9 to double
value_op	u32	22	- cast instruction
cast_inst	u32	37	- opcode
ref	ref_v	15/(0)	- operand
cast_inst	ref_t	2	- destination type
			value 17: %11 = tail call double @pow(double %10 ...
value_op	u32	9	- call instruction
call_inst	u32	0	- calling convention (C)
call_inst	u32	1	- tail call
call_inst	u32	3	- number of operands
ref	ref_v	1	- first operand (function)
ref	ref_v	16/(0)	- second operand
ref	ref_v	8	- third operand
call_inst	u32	0	- return attributes
call_inst	u32	0	- attributes of second operand

Continued on next page.

Table 4.7: Encoding example continued from previous page.

Stream	Type	Data	Description
call_inst	u32	0	- attributes of third operand
call_inst	u32	32	- function attributes (nounwind)
			value 18: %12 = fadd double %11, %6
value_op	u32	15	- binary operator
binop	u32	8	- opcode
ref	ref_v	17/(0)	- first operand
ref	ref_v	12/(5)	- second operand
			value 19: %13 = add nsw i32 %7, 1
value_op	u32	15	- binary operator
binop	u32	7	- opcode
ref	ref_v	13/(5)	- first operand
ref	ref_v	9	- second operand
binop	u32	0	- unsigned wrap
binop	u32	1	- no signed wrap (nsw)
			value 20: %14 = icmp eq i32 %13, %1
value_op	u32	17	- compare instruction
cmp_inst	u32	43	- opcode
cmp_inst	u32	32	- predicate (eq)
ref	ref_v	19/(0)	- first operand
ref	ref_v	3	- second operand
			value 21: br i1 %14, label %15, label %5
value_op	u32	19	- branch instruction
branch_inst	u32	0	- conditional branch
bbref	ref_bb	2	- first target
bbref	ref_bb	1	- second target
ref	ref_v	20/(0)	- condition value
main	u32	2	$N_{F_{1,2}}$: number of instructions in basic block 2 of F_1
bbref	ref_bb	(0)	immediate dominator of basic block 2 of F_1
			value 22: %16 = phi double [%0, %3], [%12, %5]
value_op	u32	18	- phi instruction
phi	ref_t	2	- type
phi	u32	2	- number of incoming basic blocks
ref	ref_v	2	- value if coming from first incoming basic block
bbref	ref_bb	0	- first incoming basic block
ref	ref_v	18/(3)	- value if coming from second incoming basic block
bbref	ref_bb	1	- second incoming basic block
			value 23: ret double %16
value_op	u32	7	- return instruction
ref	ref_v	22/(0)	- return value
main	u32	0	number of global variables

Chapter 5

Experiment

This chapter evaluates how well CI4 can be compressed using gzip. A typical benchmarking approach here would be to compile and encode a few test programs and then study how well they compress. We took a different approach and eventually built a few thousand real world programs. This was a lot of work but we feel that we had good reasons. First, we knew that at least some of the inefficiencies of bitcode were related to value references in larger programs [32]. We feared that too small test programs would not expose these issues. Secondly, we needed a versatile corpus of programs to ensure that our encoding was really complete and could handle all of LLVM IR losslessly. During the development phase we found cases that we had not initially thought of at all. Thirdly, we used this corpus to guide our development. Having a comprehensive corpus meant that we could avoid accidentally tuning our encoding to only compress well with a small set of test programs.

The rest of this chapter is organized as follows. We first describe our corpus and how it was created in section 5.1. Then we show how CI4 compares against bitcode in a benchmark where both encodings are compressed using gzip in section 5.2.

5.1 Corpus of LLVM IR

Finding a set of programs compiled to LLVM IR for our compression experiment turned out to be a challenge. We could not find any existing sufficiently large corpus of LLVM IR so we needed to build our own. The next challenge was to choose what software to build. An automated and reproducible build process was desirable: We would not have time to manually compile any significant number of programs and we wanted to make sure that somebody else can repeat our experiments.

We chose to compile software from the open source Debian [1] 5.0 operating system to LLVM IR. Debian has a large pool of software and can be fully rebuilt from source in an automated fashion. We proceeded by carefully considering what subset of Debian we should try to build and then augmented the build process to produce LLVM IR instead of native code. Finally, we picked only those packages that built successfully. Since Debian is open source we can freely distribute this resulting corpus along with its source code to any interested parties.

5.1.1 Software selection

Debian is composed of thousands of source packages. A source package consists of source code, metadata and scripts that control the build process. When a source package is built it produces one or more binary packages. Debian provides these binary packages for twelve different processor architectures. It is natural to think of LLVM IR as a yet another new processor architecture. To some degree LLVM IR is platform independent but the module data layout fixes things like endianness, stack alignment and pointer sizes. In languages like C, programs can even construct build-time constants that depend on these values. We thus need to build LLVM IR that targets some real architecture.

We chose to build binary package for the `i386` architecture which is for commodity 32-bit Intel and AMD hardware. Support for `i386` is very mature in LLVM and it has the same pointer size as the popular ARM architecture. There are several variants of the ARM instruction set and ARM is common on mobile devices so it might make sense to distribute software as LLVM IR. By using same pointer size we hope to ensure that our encoding works efficiently also for LLVM IR that targets ARM.

We limited our focus to packages that were written in the C or C++ languages so that we could use llvm-gcc and llvm-g++. Debian has a tagging system called debtags that can be used to classify which programming languages are used in a given package. However, out of the 22309 binary packages only 1902 were marked using the `implemented-in::c` tag. Such a low number is a sign that the tagging system has not been fully adopted yet and can not be used for our package selection. We developed a better selection heuristic. Firstly, start with the list of `i386` binary packages that include ELF binaries in a `bin` directory. This excludes libraries, plugins and software written in interpreted languages. Only 4713 binary packages, from 3878 distinct source packages, satisfy this condition. Secondly, try to build the packages with an augmented build process that produces LLVM IR and

Listing 5.1: GCC spec file for producing LLVM IR instead of native code.

```
%rename cc1 old_cc1
*cc1:
%{old_cc1} -emit-llvm

*invoke_as:
%{!S:-o %|.s | llvm-as-wrapper %(asm_options) %|.s %A }

*linker:
llvm-ld-wrapper

*link:
%{shared:-shared}

*lib:
%{pthread:-lpthread}

*libgcc:
*startfile:
*endfile:
```

ignore the packages that fail to build.

5.1.2 Build process

To compile a Debian source package to LLVM IR we used the regular Debian `dpkg-buildpackage` tool in a special environment where all compiler and linker invocations go to our wrapper script that passes a custom GCC spec file to the llvm-gcc driver program using the `-specs=` option [9]. The spec file that you can see in listing 5.1 causes the compiler to generate LLVM IR instead of native code. It also uses the LLVM linker instead of the standard system linker that only works for native code and omits the use of several native run-time objects like `libgcc`, `crti.o` and `crtn.o`.

We noticed that some packages run their test suites during the build. Some packages also build tools that generate code and expect to be able to run these during the build. When LLVM IR is stored as bitcode it is not directly executable. These problems are typical in cross-compilation.

Inspired by scratchbox [2] we investigated the possibility of using the `binfmt_misc` module of the Linux kernel to define a new executable format but then realized that LLVM IR also lacks information about dependent

Listing 5.2: LLVM IR encapsulated in a shell script for transparent execution during build process.

```
#!/bin/sh
file0=$(mktemp /tmp/XXXXXXXXXXX)
tail −c +00000180 < "$0" | head −c 00581432 > "$file0" || exit 1
lli −load=libncursesw.so "$file0" "$@"
ret=$?
rm −f "$file0"
exit $ret
BC\xc0\xde!\x0c\x00\x00\xcb7\x02...
```

libraries. This forced us to write a linker wrapper, `llvm-ld-wrapper`, that generates a shell script that encapsulates the LLVM IR. The shell script runs the program using the LLVM interpreter with the right set of libraries. You can see an example of such a script in listing 5.2. When the build was done we extracted the bitcode from those shell scripts and added them to our corpus.

5.1.3 Build results

We ended up with 3511 bitcode files with total size of 384 megabytes. These were from 1589 distinct source packages which means that only 41% of the source packages built successfully. Since we do not actually plan to introduce a new architecture but instead just need a corpus of LLVM IR this is acceptable. As described in chapter 4.1 we then ran all of the files through `canonizeBBOrderPass` and assigned each bitcode file a unique numeric identifier from the set 0, . . . , 3510.

You can see the size distribution of our corpus in figure 5.1. This looks like log-normal distribution centered around 25 kB. Since not all packages built successfully this is obviously biased towards smaller and simpler programs that have a higher probability of building successfully. To reduce the effects of this bias we will be looking at compression ratios as a function of program size.

5.2 Compression benchmark

The design goal of CI4 was to be compression-friendly. For this reason we will mostly be comparing gzipped versions of CI4 against gzipped bitcode. Uncompressed sizes are shown too but they are mostly only to give a better

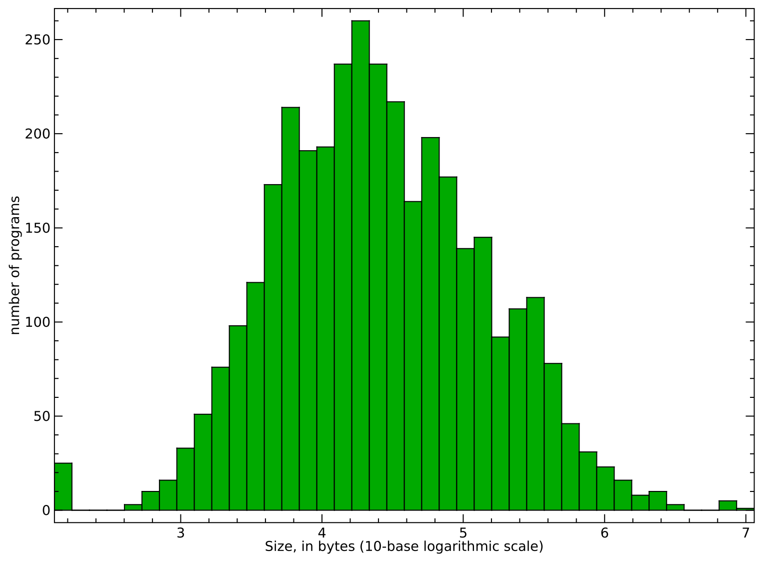

Figure 5.1: Our corpus consists of 384 megabytes of uncompressed LLVM bitcode. The above figure shows the size distribution in logarithmic scale.

picture of how our encoding behaves. We chose gzip because it is an established general purpose compression algorithm. We are reasonably confident that CI4 should behave well also with other compression tools like bzip2 or lzma since we did not design our encoding specifically for gzip. We used gzip version 1.3.12 with the default compression level (`-6`).

5.2.1 Bitcode and native code

As explained in chapter 4 we initially hoped to incrementally improve bitcode. For this reason it was useful to look at how well bitcode compresses currently using gzip. Figure 5.2 shows that typical compression ratios are between 70% and 90%. It also shows that the ratio improves slightly when programs get bigger and that the deviation is large.

Just to get some perspective to the size efficiency of bitcode we used the LLVM `llc` tool to generate i386 native code for all of our bitcode files. The comparison is not entirely fair since native code is not lossless as type and symbol information is lost. Figure 5.3 shows that gzipped native code takes around 50% to 70% of the space of gzipped bitcode. The ratio improves as programs get larger. If we want to distribute software as LLVM IR it would be helpful to get closer to the sizes of native code.

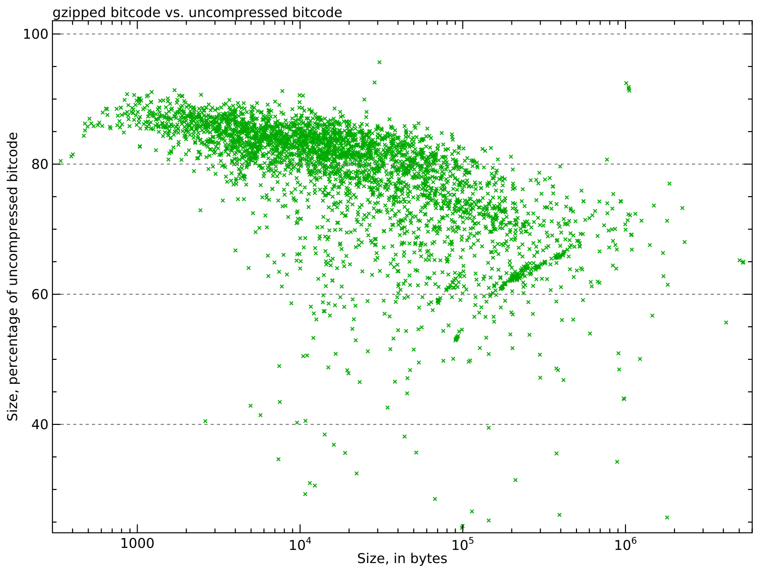

Figure 5.2: Size of gzipped bitcode as a function of uncompressed bitcode size.

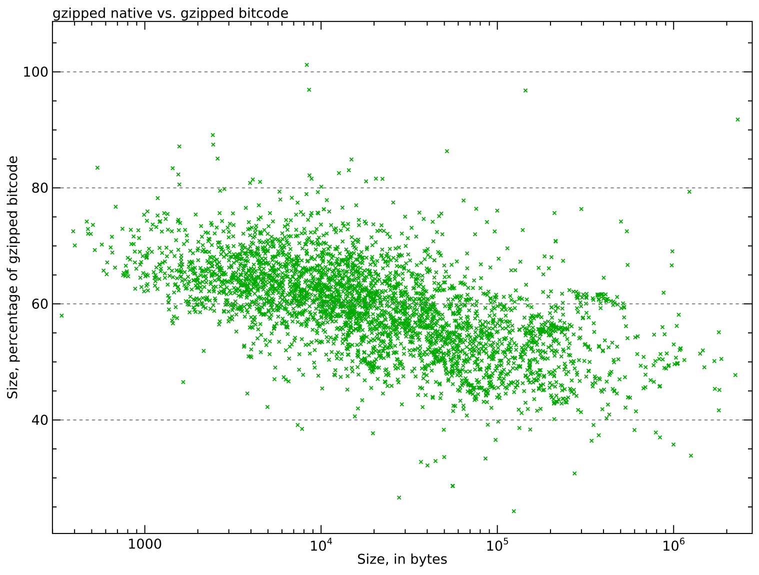

Figure 5.3: Size of gzipped native code as a function of gzipped bitcode size. The comparison is not entirely fair since conversion to native code is lossy.

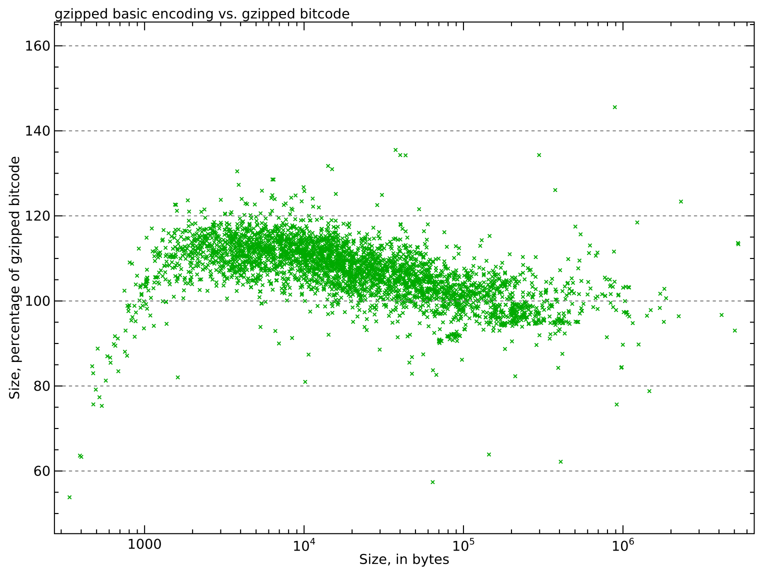

Figure 5.4: Gzipped size of our basic encoding as a function of gzipped bitcode size. Even this simple encoding manages to break even with bitcode at around 100 kB.

5.2.2 Basic encoding

After some time we gave up on trying to improve bitcode and instead designed our own basic encoding (chapter 4.1). We then then implemented an encoder and decoder with the help of the corpus: We started from the smallest program and implemented the necessary parts of the encoding to make lossless encoding and decoding work. We then moved to the second largest program and so on until we were able to successfully handle the whole corpus.

Figure 5.4 shows how our basic encoding (chapter 4.1) compares against bitcode when both are gzipped. There are some interesting effects. Programs that are less than a kilobyte seem to compress better. At 10 kB we are at 110% and at 100 kB we are approaching the same size as bitcode. Our explanation for this is two-fold. CI4 is not very compact. The uncompressed form takes significantly more space than bitcode. Our encoding compresses well with gzip but gzip gets more efficient only with larger programs where it can learn about the data.

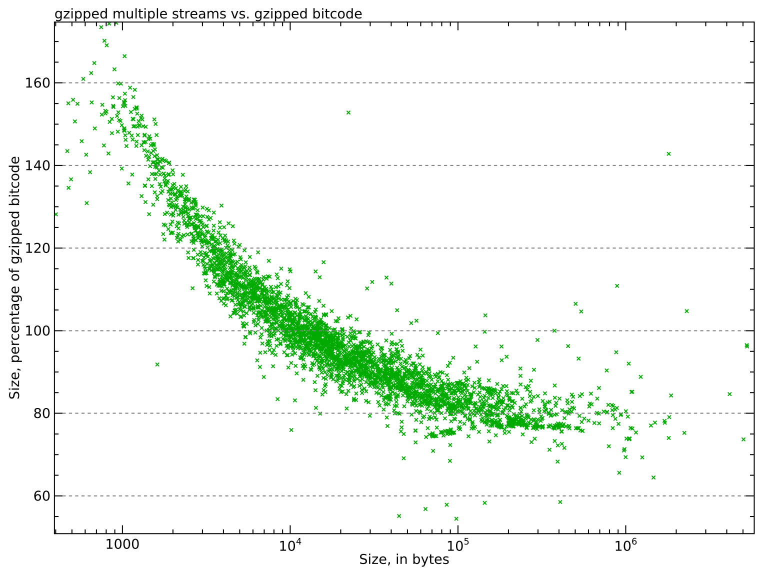

Figure 5.5: Gzipped size after separating data to multiple streams (chapter 4.2) as a function of gzipped bitcode size. Our format performs poorly with small programs since data is split to many streams and each stream contains very little data.

5.2.3 Separate streams

We were happy that our basic encoding was able to reach compressed sizes that were only slightly larger than bitcode. We proceeded by implementing the separation of data to multiple streams as described in chapter 4.2. This optimization does not change the uncompressed size of our encoding in any way but if you compare figures 5.4 and 5.5 you can see a very significant improvement in the compressed size. With basic encoding small programs perform better than bitcode but when data is separated to streams we see an opposite effect. The ratio improves when programs get larger and the break-even point is again at 10 kB. At 100 kB we are at 90% and at 1 MB we are already at 80% of the gzipped bitcode size. The outlier that you can see in the upper right corner is test program 1178, fflite_time, a speech synthesis tool that includes audio samples inside its binary.

Figure 5.6 shows the relative sizes of each stream in our corpus (this is with dominator scoping already applied). Most space is taken by value references, value code identifiers, call instructions and basic block references. When we compress each stream and each program separately using gzip the relative sizes change. Figure 5.7 shows that value references now take 45%

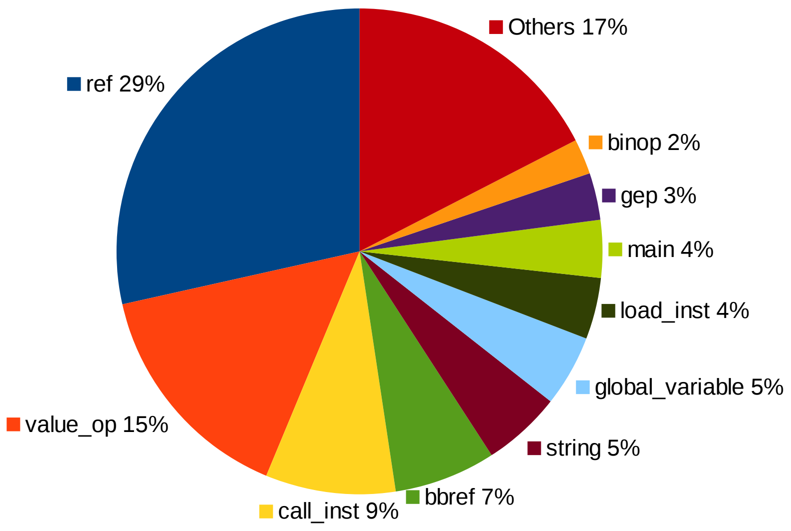

Figure 5.6: Relative sizes of uncompressed streams in our corpus. This is with dominator scoping already applied. We are mostly interested in compressed size but this chart is still a useful reference.

of the space. The second smallest stream, string, is probably something that can not be compressed any further. Basic block references could be compressed by applying dominator scoping also to basic block references.

5.2.4 Dominator scoping

Our second optimization, dominator scoping from chapter 4.3, was much more complicated to implement. It affects mostly the `ref` stream but also changes the uncompressed size when it adds immediate dominator information to the `bbref` stream. Figure 5.8 shows the results when both separate stream and dominator scoping optimizations are applied. The break-even point stays at 10 kB but the trend is much steeper. At 100 kB we are at 70% of gzipped bitcode size and at 1 MB we are already at 60%. We are beginning to approach the compression ratios of native code.

5.2.5 Discussion

Since we use a corpus to measure the performance of our compression program we need to ask how well the corpus represents real-world usage. Our corpus is based on real-world programs and not synthetic benchmark programs but it can still be biased towards particular types of programs.

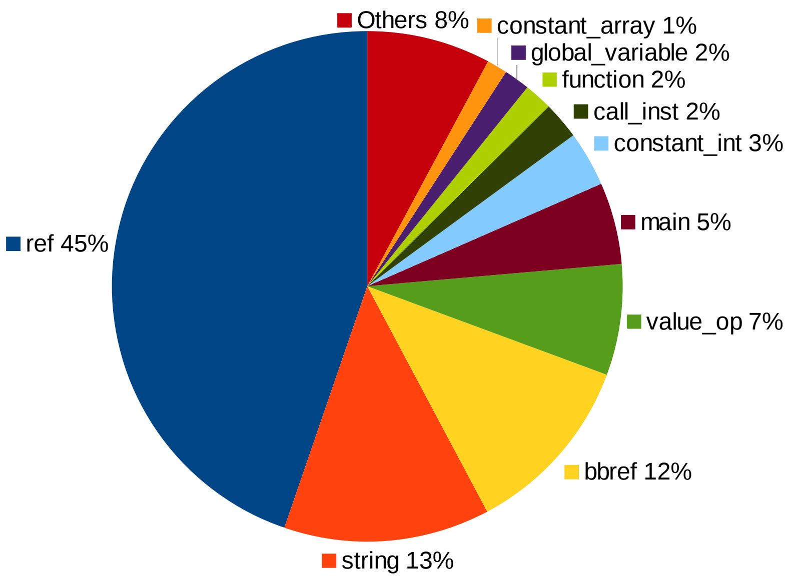

Figure 5.7: Relative sizes of gzipped streams in our corpus. This is with dominator scoping already applied. Most information is in value references and there is not much we can do to further compress strings.

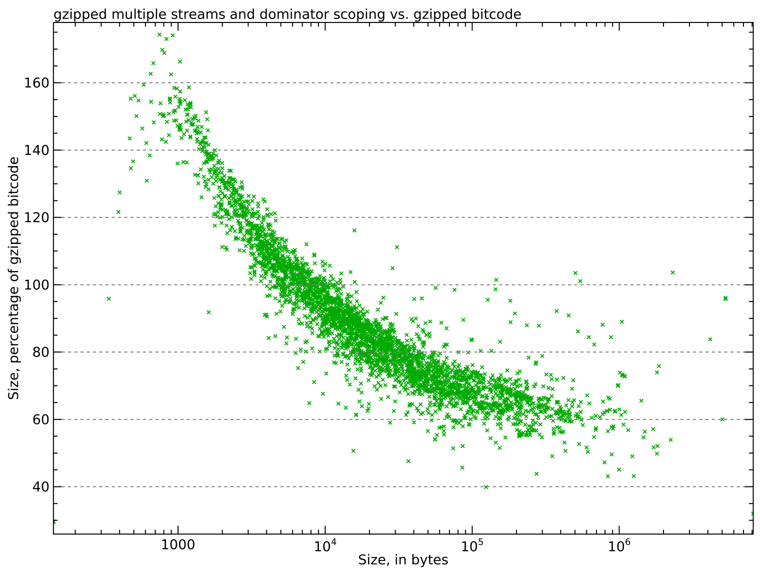

Figure 5.8: Gzipped size after separating data to multiple streams and applying dominator scoping as a function of gzipped bitcode size.

Programs consist of data and code. In our encoding contents of the string, CONSTANT_INT, CONSTANT_FP, CONSTANT_ARRAY and CONSTANT_STRUCT streams are regarded as data. The ratio of data and code varies from program to program. Figure 5.9 shows a histogram of percentage of data in a program in our corpus. We see that most programs have between 5% and 25% of data and the median of data inclusion percentage is around 15%.

We could not find an easy way to verify if the histogram of data inclusion in our corpus matches real-world programs. However, we can try to see if data inclusion percentage affects compression ratio in the first place. Figure 5.10 is an enhanced version of figure 5.8 and shows programs that contain more than 15% of data with red squares. We can see that most of the outliers that compress better than average have less than 15% of data in them and most of the outliers that compress worse have more than 15% of data in them. This makes sense since our encoding concentrates mostly on optimizing code size.

Finally it is interesting to look if different types of programs compress better than others. In the Debian archive binary packages are arranged to sections based their role in the system [40]. The classification is not very accurate and the classification is sometimes done rather arbitrarily but in any case it does give us a way to group packages.

Table 5.1 shows the total size of each section in CI4 encoding when compared to the gzipped bitcode size. Program size affects compression performance of our encoding and at the same time the program size distribution in each Debian package section is different. To see how the section affects compression performance we chose to include only programs of similar size in this table and included only programs that are larger than $10^{4.5}$ bytes (32 kB) and smaller than 10^5 bytes (1 MB). From the table we can see that programs related to desktop environments (gnome, graphics, x11, web) seem to compress worse than development tools (devel, math, editors, perl). Unfortunately we don't have enough data to understand why this is the case.

5.2.6 Summary

Figure 5.11 shows the relative sizes when the whole corpus is encoded in six different formats. We can see that gzipped native code takes almost half of the gzipped bitcode. If we are to distribute software in LLVM IR format it would be helpful to get closer to native code.

Our basic encoding is not very compact but it compresses to almost the size of gzipped bitcode. When we separate the data to streams the uncompressed size does not change but the gzipped size drops by 19% to 83% of

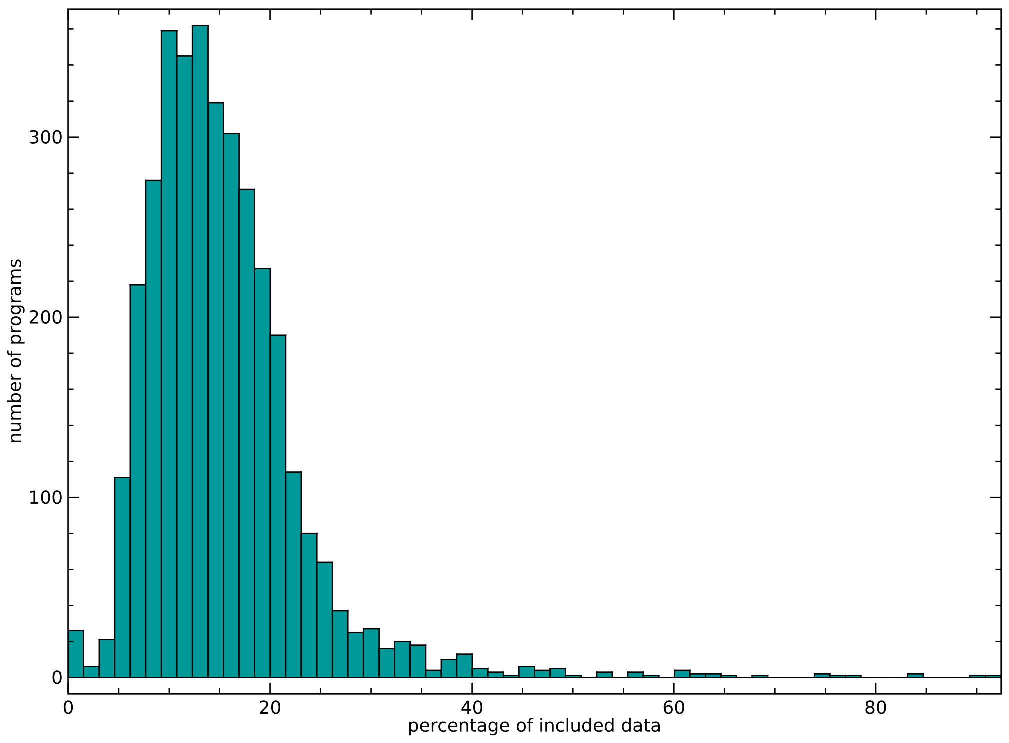

Figure 5.9: Histogram of percentage of data in a program. Contents of the string, CONSTANT_INT, CONSTANT_FP, CONSTANT_ARRAY and CONSTANT_STRUCT streams are regarded as data.

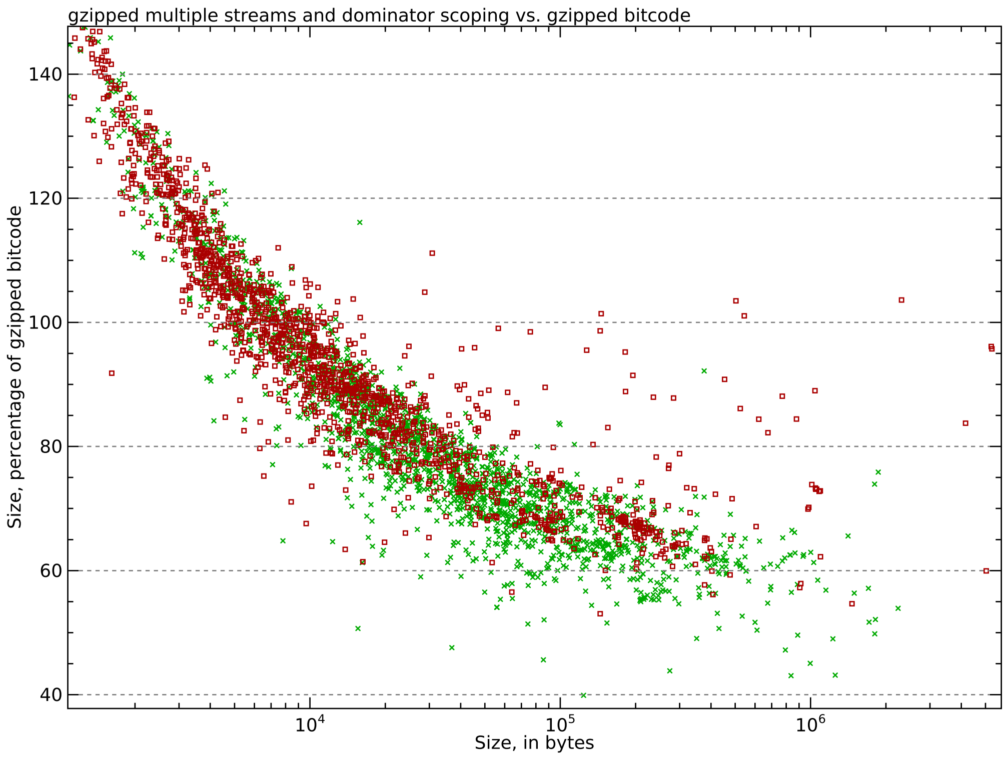

Figure 5.10: Gzipped size after separating data to multiple streams and applying dominator scoping as a function of gzipped bitcode size. Programs that contain more than 15% of data are marked with a red square.

Section	Number of programs	Total size
math	59	67%
devel	42	69%
otherosfs	11	69%
editors	11	70%
graphics	81	70%
science	64	70%
admin	28	71%
misc	16	71%
net	83	71%
hamradio	14	72%
mail	36	72%
sound	40	73%
text	31	73%
utils	87	73%
comm	11	74%
gnome	17	74%
web	16	74%
x11	42	75%
electronics	14	76%

Table 5.1: Total size of Debian package sections in CI4 encoding, percentage of gzipped bitcode size. Includes only programs that are larger than $10^{4.5}$ bytes (32 kB) and smaller than 10^5 bytes (100 kB) and only sections that have at least 10 programs.

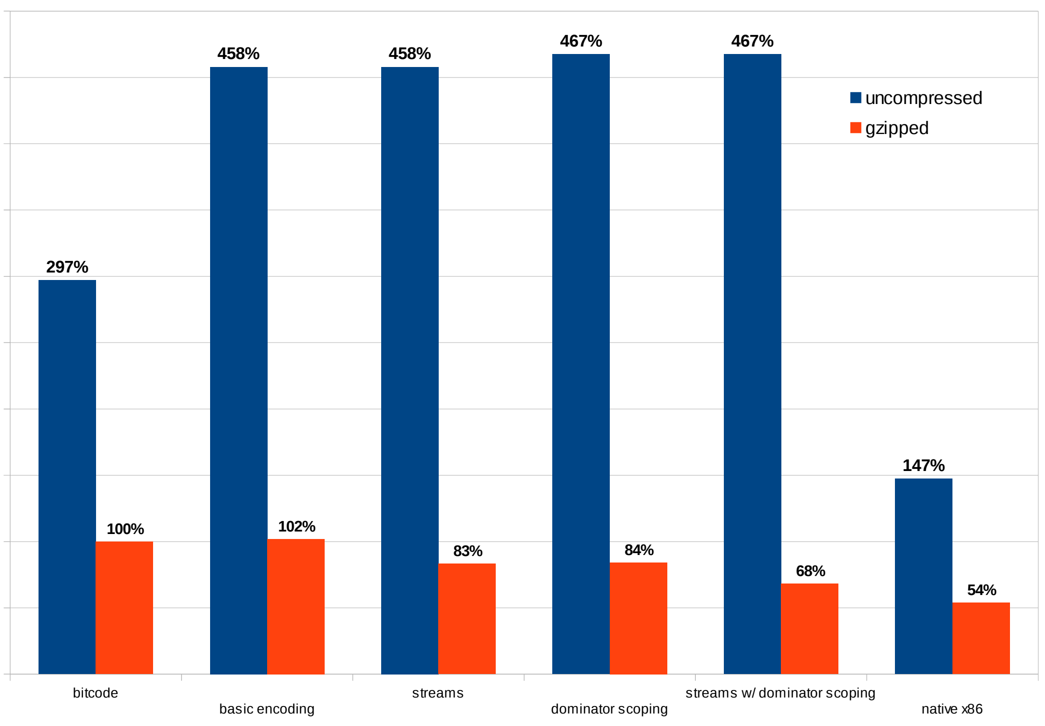

Figure 5.11: Total size of our corpus in various formats, percentage of gzipped bitcode size. Native code is shown here only for reference, it is not equivalent to bitcode unlike the others.

gzipped bitcode.

Almost the same happens when we apply dominator scoping to value references but here the uncompressed size also grows a bit since the encoding adds information about immediate dominators.

When we combine both optimizations we get another 18% improvement. This gives a hint that the optimizations are independent of each other. We are finally only 26% larger than gzipped native code and 32% smaller than gzipped bitcode.

Chapter 6

Conclusion

In conclusion, we have developed a lossless and compression-friendly encoding for LLVM IR that on average takes only 68% of space of the existing bitcode encoding, when both formats are compressed using gzip. We began this work by defining a simple but complete encoding for the IR. This was not a completely straight-forward task but demonstrates well how modular LLLVM is. We continued the work by defining two simple optimizations on top of the basic encoding and then benchmarked these against the original bitcode encoding and the basic encoding. For the benchmark we built several thousand real-world programs from the Debian archive into IR and then used them to guide our optimization efforts.

6.1 Future work

There are several details in the CI4 encoding that could be improved. The separate streams optimization currently uses gzip to compress all streams. It might be beneficial to dynamically determine which compression algorithm suits best for each stream, either globally or on a per-program basis. Also, the compressors typically start with a generic dictionary suitable for all types of data. It should be possible to preheat the compressors by using an initial dictionary derived from typical data from our corpus.

The dominator scoping optimization could restrict indexing based on value type. For example the condition field of a branch can legally refer only to values of type `i1` and it generally does not make sense for them to refer to constant values.

The CI4 compression tools were developed to benchmark the compression optimizations. If we want to use CI4 in production we need to port the compression tools to newer LLVM versions and also come up with a container format that offers at least some form of compatibility to older versions.

Bibliography

[1] The Debian operating system. WWW page of the Debian project: `http://debian.org`. Accessed 11 Dec 2012.

[2] Scratchbox cross-compilation toolkit. WWW page of the scratchbox project: `http://www.scratchbox.org`. Accessed 11 Dec 2012.

[3] The GNU operating system. WWW page of the GNU project: `http://www.gnu.org`. Accessed 11 Dec 2012.

[4] The gzip project. WWW page of the gzip project: `http://www.gzip.org`. Accessed 11 Dec 2012.

[5] LLVM 1.4 release notes, 2005. WWW page of the LLVM community: `http://llvm.org/releases/1.4/docs/ReleaseNotes.html`. Accessed 11 Dec 2012.

[6] LLVM API documentation, LLVM 2.6, 2009. WWW page of the LLVM community: `http://llvm.org/doxygen/`. Accessed 5 Jan 2010.

[7] Source code of LLVM 2.6, directory lib/bitcode/, 2009. LLVM community.

[8] LLVM language reference manual, LLVM 2.6, 2009. WWW page of the LLVM community: `http://llvm.org/releases/2.6/docs/LangRef.html`. Accessed 11 Dec 2012.

[9] *Using the GNU Compiler Collection*. Free Software Foundation Inc., 2010.

[10] LLVM language reference manual, SVN revision 122606, 2010. LLVM community.

[11] Efficient XML interchange (EXI) format 1.0. Technical report, W3C, 2011.

[12] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools.* Pearson Education, Inc., 2007.

[13] Wolfram Amme. SafeTSA: a type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 137–147. ACM Press, 2001.

[14] Wolfram Amme, Jeffery Von Ronne, and Michael Franz. SSA-based mobile code: Implementation and empirical evaluation. *ACM Transactions on Architecture and Code Optimization*, 4(2):2007, 2007.

[15] Denis N. Antonioli. Car: The class archive format. Technical report, Department of Information Technology, University of Zurich, 2001.

[16] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.

[17] Quetzalcoatl Bradley, R. Nigel Horspool, and Jan Vitek. JAZZ: An efficient compressed format for Java archive files. In *1998 conference of the Centre for Advanced Studies on Collaborative research.* IBM Press, 1998.

[18] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. In *ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 139–149. ACM Press, 1999.

[19] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

[20] P. Deutsch. RFC 1951: DEFLATE compressed data format specification version 1.3, 1996. Internet Engineering Task Force.

[21] P. Deutsch. RFC 1952: GZIP file format specification version 4.3, 1996. Internet Engineering Task Force.

[22] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. In *ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 358–365. ACM Press, 1997.

[23] Michael Franz and Thomas Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[24] Michael Franz, Wolfram Amme, Matthew Beers, Niall Dalton, Peter H. Fröhlich, Vivek Haldar, Andreas Hartmann, Peter S. Housel, Fermín Reig, Jeffery von Ronne, Christian H. Stork, and Sergiy Zhenochin. *Foundations of Intrusion Tolerant Systems*, chapter Making Mobile Code Both Safe and Efficient, pages 337–356. IEEE Computer Society Press, 2003.

[25] Christopher W. Fraser, Eugene W. Myers, and Alan L. Wendt. Analyzing and compressing assembly code. *ACM SIGPLAN Notices*, 19(6): 117–121, June 1984.

[26] Dan Gohman. LLVM IR is a compiler IR, 2011. WWW page of the LLVM mailing list archive: `http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-October/043719.html`. Accessed 28 Dec 2012.

[27] R. Nigel Horspool and Jason Corless. Tailored compression of Java class files. *Software – Practice and Experience*, 28(12):1253–1268, 1998.

[28] Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V application binary interface, AMD64 architecture processor supplement, draft version 0.90, 2003. WWW page of the AMD Operating System Research Center: `http://www.x86-64.org/documentation/abi-0.99.pdf`. Accessed 11 Dec 2012.

[29] Google Inc. Introduction to portable native client, 2012. WWW page of the Google Portable Native Client project: `http://www.chromium.org/nativeclient/pnacl/building-and-testing-portable-native-client`. Accessed 28 Dec 2012.

[30] Jin-Gu Kang. More target independent LLVM bitcode, 2011. Presentation at the LLVM European User Group Meeting. Available at `http://llvm.org/devmtg/2011-09-16/`. Accessed 11 Dec 2012.

[31] Mark Lacey. Optimizing your code with Visual C++, 2003. WWW page of Microsoft corporation: `http://msdn.microsoft.com/en-us/library/aa290055(v=vs.71).aspx`. Accessed 11 Dec 2012.

[32] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[33] Chris Lattner and Joshua Haberman. LLVM bitcode format, LLVM 2.6, 2009. WWW page of the LLVM community: `http://llvm.org/releases/2.6/docs/BitCodeFormat.html`. Accessed 11 Dec 2012.

[34] ARM Ltd. ARM architecture reference manual, 2005.

[35] ARM Ltd. Procedure call standard for the ARM architecture, release 2.09, 2012.

[36] ARM Ltd. ARM compiler toolchain using the compiler, version 4.1, 2013. WWW page of ARM Ltd.: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0472c/index.html`. Accessed 11 Dec 2012.

[37] Steven Lucco. Split-stream dictionary program compression. *ACM SIGPLAN Notices*, 35(5):27–34, May 2000.

[38] Henry Massalin. Superoptimizer: a look at the smallest program. *SIGARCH Computer Architecture News*, 15(5):122–126, October 1987.

[39] Diegno Novillo. Memory SSA - a unified approach for sparsely representing memory operations. In *Proceedings of the GCC Developers' Summit*, 2007.

[40] The Debian project. The Debian policy manual, version 3.9.5.0, 2013.

[41] William Pugh. Compressing Java class files. In *ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 247–258. ACM Press, 1999.

[42] Derek Rayside, Evan Mamas, and Erik Hons. Compact Java binaries for embedded systems. In *1999 conference of the Centre for Advanced Studies on Collaborative research*, pages 1–14. IBM Press, 1999.

[43] Jeffery Von Ronne, Wolfram Amme, and Michael Franz. SafeTSA: An inherently type-safe SSA-based code format. Technical report, University of Texas at San Antonio, Department of Computer Science, 2006.

[44] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM Press, 1988.

[45] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 2006.

[46] Christian H. Stork. *WELL: A language-agnostic foundation for compact and inherently safe mobile code*. PhD thesis, University of California, 2006.

[47] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel. The problem of programming communication with changing machines. *Communications of the ACM*, 1(8):12–18, 1958.

[48] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 1984.

[49] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3), 1977.

Appendix A

IR encoding

As explained in chapter 3 LLVM modules are composed of types, values and some auxiliary information. While chapter 4 and table 4.1 discuss the high-level structure of our encoding this appendix contains a series of compact tables that document the low-level encoding details of each type and value.

Our format is based on 8-bit (`u8`) and 32-bit (`u32`) unsigned little-endian integers. Some metadata such as function names is encoded by first encoding the length of the string as `u32` and then encoding each byte as a `u8`. Table A.1 shows this in tabular form.

Type	Count	Description
u32	1	N: length of the string in bytes
u8	N	bytes of the string, can include zero bytes

Table A.1: Encoding of string. This is not a type or value, it is only for storing metadata.

A.1 Types

Type	Count	Description
u32	1	LLVM type identifier, from C++ enum llvm::Type::TypeID

Table A.2: Encoding of primitive type.

Type	Count	Description
		(nothing to encode)

Table A.3: Encoding of opaque type.

Type	Count	Description
u32	1	bitwidth of the integer

Table A.4: Encoding of integer type.

Type	Count	Description
u32	1	1 if the function takes a variable number of arguments, 0 otherwise
ref_t	1	return type
u32	1	N_{args}: number of parameters
ref_t	N_{args}	type of parameter

Table A.5: Encoding of function type.

Type	Count	Description
u32	1	number of elements
ref_t	1	element type

Table A.6: Encoding of array type.

Type	Count	Description
u32	1	1 if struct is packed, 0 otherwise
u32	1	N_{args}: number of elements
ref_t	N_{args}	element type

Table A.7: Encoding of struct type.

Type	Count	Description
u32	1	address space
ref_t	1	pointee type

Table A.8: Encoding of pointer type.

A.2 Non-instruction values

Type	Count	Description
ref_t	1	type of argument
ref_v	1	parent function

Table A.9: Encoding of argument value.

Type	Count	Description
u32	1	bitwidth of the constant
string	1	value of the constant in base 10

Table A.10: Encoding of constant_int value.

Type	Count	Description
u32	1	1 if the value has double precision, 0 otherwise
u32	1	most significant word
u32	1	least significant word

Table A.11: Encoding of constant_fp value.

Type	Count	Description
ref_t	1	type of the array
u32	1	N_{args}: number of values
ref_v	N_{args}	value

Table A.12: Encoding of constant_array value.

Type	Count	Description
u32	1	opcode
		if opcode is getelementptr:
u32	1	N_o: number of operands
ref_v	N_o	operand
		if opcode is bitcast, inttoptr, ptrtoint or zext:
ref_v	1	operand
ref_t	1	target type
		if opcode is shl, or, and or sub:
ref_v	1	first operand
ref_v	1	second operand
		if opcode is icmp:
u32	1	predicate (e.g. less-than, equal or greater-than)
ref_v	1	first operand
ref_v	1	second operand

Table A.13: Encoding of constant_expr value.

Type	Count	Description
ref_t	1	type

Table A.14: Encoding of constant_null value.

Type	Count	Description
ref_t	1	type

Table A.15: Encoding of constant_agg_zero value.

Type	Count	Description
ref_t	1	type
u32	1	N_{args}: number of elements
ref_v	N_{args}	element

Table A.16: Encoding of constant_struct value.

Type	Count	Description
ref_t	1	type of the variable
u32	1	1 if the variable is constant, 0 otherwise
u32	1	linkage type
u32	1	1 if the variable has initializer, 0 otherwise
ref_v	1	initializer
string	1	name of the variable
u32	1	1 if the variable is thread local, 0 otherwise
u32	1	required alignment in bytes

Table A.17: Encoding of global_variable value.

Type	Count	Description
ref_t	1	type of the value

Table A.18: Encoding of undef value

Type	Count	Description
ref_t	1	type of the function
u32	1	linkage type
string	1	name of the function
u32	1	return attributes
u32	N_{args}	parameter attributes
u32	1	function attributes
u32	1	calling convention

Table A.19: Encoding of function value.

Type	Count	Description
u32	1	1 if the assembler has side effects, 0 otherwise
ref_t	1	type
string	1	string describing the assembler
string	1	string describing constraints

Table A.20: Encoding of inline_asm value.

A.3 Instructions

Type	Count	Description
ref_v	1	return value

Table A.21: Encoding of return_inst value.

Type	Count	Description
u32	1	N_{args}: number of operands
ref_v	N_{args}	operand

Table A.22: Encoding of gep_inst value.

Type	Count	Description
u32	1	calling convention
u32	1	1 if tail call, 0 otherwise
u32	1	N_{args}: number of parameters
ref_v	1	target function
u32	N_{args}	parameters
u32	1	return attributes
u32	N_{args}	parameter attributes
u32	1	function attributes

Table A.23: Encoding of call_inst value.

Type	Count	Description
u32	1	N_{args}: number of operands
ref_v	1	first operand
ref_bb	1	second operand, normal target
ref_bb	1	third operand, unwind target
ref_v	$N_{args} - 3$	operand
u32	1	calling convention
u32	1	return attributes
u32	N_{args}	parameter attributes
u32	1	function attributes

Table A.24: Encoding of invoke_inst value.

Type	Count	Description
u32	1	1 if the store is volatile, 0 otherwise
u32	1	alignment in bytes
ref_v	1	destination
ref_v	1	value to store

Table A.25: Encoding of store_inst value.

Type	Count	Description
u32	1	1 if the store is volatile, 0 otherwise
u32	1	alignment in bytes
ref_v	1	source

Table A.26: Encoding of load_inst value.

Type	Count	Description
u32	1	opcode
ref_v	1	first operand
ref_v	1	second operand

Table A.27: Encoding of binop_inst value.

Type	Count	Description
ref_t	1	type
ref_v	1	size of allocation
u32	1	alignment in bytes

Table A.28: Encoding of alloca_inst value.

Type	Count	Description
u32	1	opcode
u32	1	predicate
ref_v	1	first operand
ref_v	1	second operand

Table A.29: Encoding of cmp_inst value.

Type	Count	Description
ref_t	1	type
u32	1	N_{args}: number of incoming values
		repeat N_{args} times:
ref_v	1	incoming value
ref_bb	1	basic block

Table A.30: Encoding of phi value.

Type	Count	Description
u32	1	1 if the branch is unconditional, 0 otherwise
		for unconditional branches:
ref_bb	1	target basic block
		for conditional branches:
ref_bb	1	target if the condition is true
ref_bb	1	target if the condition is false
ref_v	1	condition

Table A.31: Encoding of branch_inst value

Type	Count	Description
ref_v	1	condition
ref_v	1	true value
ref_v	1	false value

Table A.32: Encoding of select_inst value.

Type	Count	Description
u32	1	opcode
ref_v	1	operand
ref_t	1	target type

Table A.33: Encoding of cast_inst value.

Type	Count	Description
ref_v	1	condition
ref_bb	1	default destination
u32	1	N_{args}: number of cases
		repeat N_{args} times:
ref_v	1	case value
ref_bb	1	destination if the condition matches the case value

Table A.34: Encoding of switch_inst value.

Type	Count	Description
		(nothing to encode)

Table A.35: Encoding of unreachable_inst value.

Type	Count	Description
ref_v	1	aggregate operand
u32	1	N_{args}: number of indices
u32	N_{args}	index into the aggregate operand

Table A.36: Encoding of extract_value_inst value.