OPTIMIZING WHOLE PROGRAMS FOR CODE SIZE

BY

SEAN BARTELL

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana–Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Professor Vikram S. Adve, Chair and Director of Research
Professor Darko Marinov
Assistant Professor Sasa Misailovic
Professor John Regehr, University of Utah

Abstract

Reducing code size has benefits at every scale. It can help fit embedded software into strictly limited storage space, reduce mobile app download time, and improve the cache usage of supercomputer software. There are many optimizations available that reduce code size, but research has often neglected this goal in favor of speed, and some recently developed compiler techniques have not yet been applied for size reduction.

My work shows that *newly practical compiler techniques can be used to develop novel code size optimizations. These optimizations complement each other, and other existing methods, in minimizing code size.* I introduce two new optimizations, Guided Linking and Semantic Outlining, and also present a comparison framework for code size reduction methods that explains how and when my new optimizations work well with other, existing optimizations.

Guided Linking builds on recent work that optimizes multiple programs and shared libraries together. It links an arbitrary set of programs and libraries into a single module. The module can then be optimized with arbitrary existing link-time optimizations, without changes to the optimization code, allowing them to work across program and library boundaries; for example, a library function can be inlined into a plugin module. I also demonstrate that deduplicating functions in the merged module can significantly reduce code size in some cases. Guided Linking ensures that all necessary dynamic linker behavior, such as plugin loading, still works correctly; it relies on developer-provided constraints to indicate which behavior must be preserved. Guided Linking can achieve a 13% to 57% size reduction in some scenarios, and can speed up the Python interpreter by 9%.

Semantic Outlining relies on the use of automated theorem provers to check semantic equivalence of pieces of code, which has only recently become feasible to perform at scale. It extends outlining, an established technique for deduplicating structurally equivalent pieces of code, to work on code pieces that are semantically equivalent even if their structure is completely different.

My comparison framework covers a large number of different code size reduction methods from the literature, in addition to my new methods. It describes several different aspects by which each method can be compared; in particular, there are multiple types of redundancy in program code that can be exploited to reduce code size, and methods that exploit different types of redundancy are likely to work well in combination with each other. This explains why Guided Linking and Semantic Outlining can be effective when used together, along with some kinds of existing optimizations.

Acknowledgments

This dissertation would not have been possible without the guidance of my advisor, Professor Vikram Adve. I relied on his experience and advice, and am grateful for his mentorship.

My work also depended on the contributions Will Dietz, the leader of the ALLVM project from which my research has flowed, and Prof. John Regehr, Nader Bushehri, Om Bhatt, and Yuyou Fan, who helped write papers, write code, and run experiments. I would also like to thank my committee, the other members of Vikram's research group, and everyone else who has given valuable feedback on my work.

My work throughout this program has been funded by the Office of Naval Research under Grant No. N00014-17-1-2996, and by the National Science Foundation under Grant No. 1564274. I am grateful for their support.

Last but not least, I thank my parents and family for their persistent support, and especially for organizing vacations to the Rocky Mountains. Urbana is nice, but there's only so much flatness a man can take.

Table of Contents

Chapter 1 Introduction 1
1.1 Motivation for Reducing Code Size 1
1.2 Novel Optimizations using Newly Practical Techniques 2
1.3 Framework for Comparing Methods 4
1.4 Contributions 5
1.5 Related Work 6
1.6 Organization 7

Chapter 2 Bitcode Database 8
2.1 Introduction 8
2.2 Related Work 8
2.3 MemoDB Design 10
2.4 Storing Bitcode in MemoDB 12
2.5 Obtaining Bitcode 14
2.6 Evaluation 15
2.7 Future Work 15

Chapter 3 Cross-Package Function Deduplication 17
3.1 Introduction 17
3.2 Related Work 17
3.3 Background: Manual and Automatic Multicall 18
3.4 Cross-Package Function Deduplication 19
3.5 Evaluation 24
3.6 Future Work 27
3.7 Conclusion 28

Chapter 4 Guided Linking 29
4.1 Introduction 29
4.2 Related Work 34
4.3 Overview of Guided Linking 37
4.4 Background 39
4.5 Selecting Programs and Libraries to Optimize 44
4.6 Constraining the Dynamic Linker 45
4.7 Optimizing Dynamically Linked Code 52
4.8 Function Deduplication 56
4.9 Evaluation 57
4.10 Future Work 63
4.11 Conclusion 65

Chapter 5 Semantic Outlining 66
5.1 Introduction 66
5.2 Related Work 68
5.3 Candidate Generation 74
5.4 Semantic Clustering 84
5.5 Final Steps 87
5.6 Evaluation 88
5.7 Future Work 91
5.8 Conclusion 92

Chapter 6 Comparison of Code Size Reduction Techniques 94
6.1 Introduction 94
6.2 Related Work 95
6.3 Motivations for Reducing Code Size 96
6.4 Aspects of Different Methods 98
6.5 Specialization and Redesign Techniques 102
6.6 Hardware-Supported Techniques 105
6.7 Compact Storage Formats 108
6.8 Compiler Optimizations 111
6.9 Choosing a Combination of Techniques 114
6.10 Future Work 115
6.11 Conclusion 116

Chapter 7 Conclusion 117

References 118

Chapter 1: Introduction

My thesis is as follows: *Newly practical compiler techniques can be used to develop novel code size optimizations. These optimizations complement each other, and other existing methods, in minimizing code size.* Specifically, I introduce multiple new optimizations, particularly *Guided Linking* and *Semantic Outlining*, and introduce a framework for comparing different kinds of code size reduction methods and choosing effective combinations of them.

1.1 MOTIVATION FOR REDUCING CODE SIZE

Although code size is often neglected as an optimization target in favor of speed, reducing code size has potentially significant benefits at every scale from supercomputers to tiny Internet of Things devices. My work focuses on overall program size and speed, but reducing size can have other benefits as well; see section 6.3 for a longer discussion.

Meeting size limits on embedded devices. Embedded devices have strictly limited flash storage space, ranging from a few megabytes down to as little as 384 bytes [1], which must be sufficient for both code and persistent data. Code that grows too large may not fit on the intended processor, requiring a costly redesign for a larger processor. Many commercial embedded compilers specifically advertise their ability to produce small code [2]–[4], allowing developers to use cheaper microcontrollers with less storage space. Larger embedded systems may run a full, configurable software stack, with a relatively large number of software packages that can be installed after initial deployment [5]–[7], and code size restrictions may prevent desired packages from being installed [8]; code size reductions can lessen the impact of these restrictions.

Increasing speed. Reducing code size can allow more hot code to fit in the instruction cache, and also reduce the amount of the cache that must be replaced when cold code is executed, thereby reducing cache misses and increasing performance [9]–[11]. Smaller code can also be loaded from storage faster and processed faster by just-in-time compilers, useful for performance-critical code that is only executed occasionally, such as mobile app startup code. These advantages motivated Facebook to develop a custom optimization tool to shrink their Android apps [12].

Cheaper software distribution. Frequent software updates are a necessity in order to fix bugs and security holes, but they can come at a significant cost. Firmware updates consume

a significant amount of power on Internet of Things (IoT) devices, about half of which is the cost of wirelessly transmitting the update data [13]; this cost could be directly reduced by shrinking the code. Size reductions are also useful for software such as Docker containers that are built once and then distributed to thousands of nodes [14], [15]. Download size is also a major concern for mobile apps; Facebook developed custom compression algorithms to reduce app download size [16], and Uber expends significant effort to keep their iOS apps under a size limit imposed by the App Store, because exceeding the limit measurably reduces their ability to retain users [17].

1.2 NOVEL OPTIMIZATIONS USING NEWLY PRACTICAL TECHNIQUES

Because code size optimization receives much less attention than performance optimization, several optimization techniques that first became practical in recent years have only been applied in limited ways to code size. In particular, these include the optimization of multiple programs and libraries as a single unit, as well as the use of theorem provers for equivalence checking. In this dissertation, I show how to apply these techniques to the goal of optimizing code size.

Optimizing multiple programs and libraries as a single unit. Support for link-time optimization (LTO) is becoming commonplace, as increased processing power and memory capacities make it feasible to optimize all parts of a single program or library at once. Recent work even combined *multiple programs and libraries* into a single module, which can be processed by the compiler as a single unit [18]. This advancement has opened new frontiers for optimization. I extended this prior work with a new optimization that deduplicates identical functions in the merged module (chapter 3). The prior work has significant limitations on which programs and libraries can be combined, so I developed an improved technique that can support an arbitrary set of software while maintaining full compatibility with dynamically linked code, including plugins. I call this technique *Guided Linking* (chapter 4).

Theorem provers and equivalence checking. The development and increasing practicality of automated theorem provers such as Z3 [19] have made it possible to scalably analyze the semantic properties of nontrivial code sequences [20], [21]. In particular, it is increasingly feasible to check whether two complex code sequences are semantically equivalent [22]–[24]. I show that it is now possible to detect semantically equivalent pieces of code throughout an entire program, even when the pieces are syntactically different. I introduce a new automated technique I call *Semantic Outlining* (chapter 5), in which an entire program or library

is searched for equivalent pieces of code and the equivalent pieces are deduplicated to reduce code size.

1.2.1 The Bitcode Database

As a preliminary infrastructure project, I implemented the Bitcode Database (BCDB), a database that can efficiently store hundreds of real-world programs in the form of the LLVM bitcode, which is the LLVM compiler's intermediate representation (IR). The database is described in chapter 2. The BCDB can detect and deduplicate identical functions, storing many versions of a single program with little overhead. In a BCDB instance containing hundreds of Linux programs, only 34% of functions are unique. The BCDB includes support for distributed processing of analyses and optimizations, and can cache the results of these processes, which is important to make Semantic Outlining practical.

I have built on the BCDB to implement an existing technique called *Software Multiplexing* [18] with the addition of function deduplication features. Software Multiplexing automatically combines several programs and their libraries into one program, which can be smaller and faster than the originals. Using the BCDB, I can deduplicate identical functions in the multiplexed program, even if the duplicates originally came from different programs (chapter 3). By applying these technique to collections of embedded software, I can reduce executable size by 30% to 38%.

1.2.2 Guided Linking

I have also implemented a system I call *Guided Linking* (chapter 4), which extends the benefits of Software Multiplexing and function deduplication to arbitrary dynamically-linked code, including dynamically loaded plugins. The key idea of Guided Linking is that the developer can provide constraints on the behavior of the dynamic linker, such as "definitions in this library will never be dynamically overridden by another library." Given the developer's constraints, and code for a set of programs, libraries, and plugins, the Guided Linker can optimize the set.

Guided Linking can improve size and speed by 5% to 6% on real-world programs such as Python, with speedups greater than 20% on certain Python benchmarks. The Guided Linker can also deduplicate identical functions, even when they occur across different programs and libraries. By combining several versions of the same library (Boost or Protobuf) I can reduce total binary size by 31% to 57% without hurting compatibility.

These techniques could be used to enhance the affect of *any* whole-program optimization,

including my outlining technique.

1.2.3 Semantic Outlining

Although "don't repeat yourself" is a common principle used in software development, compiled code still tends to contain a large amount of redundancy, for several reasons. At the source code level, developers often copy and paste code. Even when source code is not redundant, the compiled IR and machine code may be redundant due to compiler features that produce repetitive code such as macros, template specialization, reference counting, buffer overflow checks, function inlining, and loop optimizations. Outlining is a process that reduces code size by finding duplicate pieces of code and factoring them out into new functions.

This dissertation introduces *Semantic Outlining* (chapter 5), which enhances outlining by using a semantic equivalence checker to identify duplicate code. Equivalence checkers can detect more duplicate code than other tools by using theorem provers to show that two sequences of code have the same effect even when the sequences use completely different operations. The goal of this work is to investigate whether semantic equivalence checks can substantially improve the code size reduction provided by outlining. The Semantic Outliner works by generating a large number of possible outlining candidates, using the equivalence checker to find groups of candidates which are semantically equivalent, choosing groups to outline, and outlining each group by adding a single new function.

Translation validators can only test one pair of code sequences at a time, so a naive approach would apply the validator $O(n^2)$ times. One of my key contributions is a way to significantly reduce the number of pairs that must be compared. The Semantic Outliner does this by eliminating candidates that seem unlikely to be profitable, clustering candidates based on their input and output types, and using a theorem prover to generate test inputs which can be used to further cluster candidates.

1.3 FRAMEWORK FOR COMPARING METHODS

There are a great variety of different code size reduction methods described in the literature, and the boundaries between different kinds are sometimes unclear. Depending on a developer's specific goals, some methods may complement each other to give better results, while others may be redundant or even counterproductive. I introduce a framework (chapter 6) for comparing various aspects of different methods, and explain which combinations of methods are likely to work well for a particular goal. In particular, I indicate which type of

redundancy is exploited by each method, and explain how methods that use different types of redundancy are likely to be effective when combined. Developers can use this framework as a guide when choosing methods to apply to their software.

1.4 CONTRIBUTIONS

Overall, my work provides developers with a suite of new techniques and tools they can use to reduce the size of their code. Even if a developer is already making full use of existing tools, they can get additional improvements by applying Guided Linking and Semantic Outlining. In addition, developers who are trying to determine the best way to shrink their code can refer to my comparison framework for an overview of the trade-offs between different techniques.

Function deduplication and Guided Linking. I and my collaborators published a paper about Guided Linking at OOPSLA 2020 [25], and additionally presented the work at the 2020 LLVM Developers' Meeting [26]. I have published the BCDB implementation, including Guided Linking, as an open source project [27].

1. I identify a set of constraints that can usefully be placed on dynamic linking behavior in order to enable optimizations.

2. I explain how optimizable code can be generated for each possible set of constraints, without breaking compatibility with existing software that relies on the dynamic linker.

3. I implement my proposed system using the LLVM compiler infrastructure.

4. I show experimentally (section 4.9) that my tool can transparently compile large, real-world Linux software packages. As part of this demonstration, I show that I can easily identify sets of constraints that can be applied to real-world software, without breaking anything. I show that my system can improve speed by more than 6% or reduce code size by more than 50% on widely used software packages, depending on the software set and the constraints used; the code size improvement is much more than any previous compiler-based technique I am aware of.

Semantic Outlining. I and my collaborators are in the process of completing a paper on Semantic Outlining.

1. I present a new compiler analysis that determines which subsequences of a function's instructions may be outlined.

2. I present the method of counterexample-guided equivalence clustering, which is necessary for Semantic Outlining to scale to nontrivial programs.

3. I implement my proposed system for several architectures, using the LLVM compiler and Alive2 equivalence checker.

Framework for comparing methods. I and my collaborators are in the process of writing a paper on this framework, including a more thorough evaluation of different methods. We intend this paper to be useful not only to the academic community but also to industry developers who are investigating techniques they can use to reduce the size of their code.

1. I present an overview of different aspects of code size reduction methods and motivations for reducing code size.

2. I discuss a large variety of methods and categorize them in several ways.

3. I give recommendations on choosing methods for a particular application, and determining which methods can be effectively combined with each other.

1.5 RELATED WORK

Memory usage and code size have been important constraints on programmers since the very first programmable computers. At first, machine code and assembly code would have been hand-written with careful attention paid to its size, along with its speed. When general-purpose compilers were introduced, they included features and optimizations to improve both speed and size. For instance, the very earliest versions of Fortran, in the 1950s, included an `EQUIVALENCE` statement programmers could use to overlap variables if necessary to save space [28, pp. 36–37].

Over the decades since then, a large variety of different methods for code size reduction have been introduced. Techniques can be classified in various ways, according to their basic idea, ease of use, effects on program functionality, and so on. Sections 6.5 to 6.8 provide a list of many different types of techniques with references to related work, including discussion of several aspects in which they differ. Chapter 6 as a whole describes different ways of classifying techniques.

The Bitcode Database is a straightforward combination of existing ideas, although to my knowledge the specific combination has not been implemented before. See section 2.2 for discussion of related work. The idea of performing function deduplication across multiple programs and libraries is a straightforward extension of Allmux [18]; see section 3.2 for more discussion.

Guided Linking is essentially an extension of Allmux [18] to include much better support for dynamic linking features, such as plugins and overridden definitions. It can also be seen as an extension of link-time optimization [29] to further increase the scope of optimization, or as an extension of work that performs dead code elimination on shared libraries [30]–[33] to support other types of optimizations. For more information, see section 4.2.

Semantic Outlining is based on the ideas of outlining, which dates back to 1972 [34], and general black-box equivalence checking, which originated around 2000 [35]. Both ideas have an extensive literature, but to my knowledge this dissertation is the first work to combine both. Semantic Outlining is also closely related to clone detection. Section 5.2 discusses the related work in much more detail.

My comparison framework for code size optimizations is not the first such framework to be published. Beszédes, Ferenc, Gyimóthy, *et al.* [36] developed an extensive comparison in 2003, but it is now outdated and it focuses on certain types of technique (mostly those that require special hardware support). Certain other papers have made limited evaluations of multiple techniques; see section 6.2.

1.6 ORGANIZATION

Chapter 2 describes the Bitcode Database (BCDB), including its ability to detect duplicate functions. Chapter 3 describes my implementation of function deduplication across software packages, and chapter 4 describes my more general Guided Linking optimization. Chapter 5 describes Semantic Outlining. Chapter 6 presents my framework for comparing code size reduction methods. Finally, chapter 7 concludes my dissertation.

Chapter 2: Bitcode Database

2.1 INTRODUCTION

As a basis for my research, I have designed and implemented a prototype of an infrastructure project called the Bitcode Database (BCDB). Bitcode is the LLVM Compiler's internal representation and virtual instruction set, and it supports sophisticated compiler analyses and code generation for a wide range of programming languages. The goal of the BCDB project is to provide a unified interface for storing, analyzing, optimizing, and maintaining all software on a system, by replacing standard executable files with a database containing fine-grained pieces of bitcode, along with cached optimization and analysis results. By making an entire system available for processing in a single repository, the BCDB enables novel compiler techniques, including compiler techniques that simultaneously optimize applications and shared libraries, and even separate distinct programs. These techniques are discussed in chapter 3 and chapter 4.

My prototype can store and retrieve arbitrary bitcode modules, and allows arbitrary cached analysis results or other data to be associated with each module. Each module is split into individual functions which are stored separately, using content-addressable storage to combine syntactically identical functions into one database entry even when they are derived from different source modules.

My prototype also supports a REST API which can be used by clients in various programming languages, allowing them to store and retrieve data from the BCDB. This API supports distributed processing: clients can submit requests for processing to be performed on BCDB data, while workers, spread across multiple computers, can receive these requests and send their results to the server. Results are automatically cached in the database, greatly improving performance when the same functions are processed multiple times.

2.2 RELATED WORK

The BCDB is a straightforward combination of a few existing ideas: content-addressable storage (CAS), basic distributed programming concepts, memoization, and persistent storage of a compiler's intermediate representation (IR). To my knowledge, it is the first CAS system designed to store low-level IR in a fine-grained form. However, there is a great deal of closely related work, including content-addressable storage systems, compiler IR repositories, and other projects.

2.2.1 Content-addressable storage

One of the first content-addressable storage systems was designed for hardware in the 1960s [37]. In the 1980s, Merkle trees were developed, in which each leaf node consists of a cryptographic hash of some data, and each internal node contains a hash of its children [38], [39]. This structure was generalized into a Merkle DAG, in which nodes can contain additional data and can be used more than once. Arbitrary data in tree form can be efficiently deduplicated by converting the tree into a Merkle DAG and merging all nodes that have equal hash values. Many storage systems have used Merkle DAGs in this way, such as Git [40] and IPFS [41], as does the BCDB.

In particular, Git repositories are an extremely popular format for storing source code. There is a large body of work on analyzing source code from Git repositories and other sources [42]. However, Git is rarely used to store compiler IR, and unlike the BCDB it has no ability to break IR into smaller parts. Source code repositories can of course be compiled into bitcode form as part of an analysis, but this process adds a significant amount of difficulty to the analysis due to the vagaries of different projects' build systems.

The Bazel build system uses a CAS system to cache the results of build steps, making compilation faster [43]. The output files may contain IR, if link-time optimization is enabled, but Bazel has no special understanding of IR and cannot break it into smaller parts.

The Unison programming language is extremely closely related to BCDB. It stores each source function as a separate entry in a CAS system, caches results of analyses such as type inference, and even includes distributed programming features [44]. The main difference is that Unison works on abstract syntax trees (ASTs) for a purpose-built programming language, whereas BCDB works on low-level IR and supports programs written in existing languages such as C++ and Swift.

2.2.2 IR repositories

Databases have been integrated into compilers in order to speed up interprocedural optimization [45]–[47]. The databases contain some form of IR, and cache the results of optimizing and compiling each part of the code. However, these systems do not incorporate content-addressable storage.

Language runtimes based on virtual machines, such as the Java virtual machine, essentially make IR available for every program and library installed on a system. However, this IR is not integrated into a single database, so operations such as deduplicating functions would be difficult to implement. The initial work on LLVM [48] suggested attaching LLVM bitcode to

executable programs, allowing them to be optimized and re-compiled at install time or even afterwards, but this has the same limitation that programs and libraries are not integrated into a database.

Toman [49] made a server that automatically built and provided Fedora Linux packages in LLVM bitcode form, for purposes of static analysis. Only 60% of packages could be built successfully, and the final product was a simple collection of bitcode files for each package.

One tool converts an LLVM bitcode file to a GraphML database, allowing statistics to be gathered using a graph-based query language [50].

The ALLVM Project [51] was the original basis for the BCDB. It proposes building all software on a system in the form of LLVM bitcode, and it provides a standardized file format, the ALLEXE, for storing each program's bitcode along with the bitcode of its libraries. ALLVM provides a set of tools [52] for working with ALLEXEs, and it extends the Nixpkgs package distribution to build hundreds of Linux packages in ALLEXE format automatically (unpublished work by Will Dietz). The BCDB can be thought of as the next stage in the evolution of ALLVM, with several key improvements:

- Processing IR at the function granularity (ALLVM uses module granularity).
- Deduplication of identical code, allowing hundreds of versions of a program to be stored in the BCDB with limited overhead.
- The ability to cache analysis and compilation results along with the bitcode.
- Distributed processing support.
- A more faithful representation of dynamic linking information.

2.2.3 Other program databases

Another program repository called `llvm-prepo` has been implemented for LLVM [53]–[55]. Unlike the BCDB, which is designed to support general analysis and research tools, `llvm-prepo` is focused on the specific goal of reducing build times, and it stores compiled object code rather than IR. It hooks into the compiler frontend Clang to prevent it from generating IR for pieces of code that are already in the repository.

2.3 MEMODB DESIGN

At the core of the BCDB is a C++ data management library called MemoDB ("memoizing database"). MemoDB itself supports a generic data model and has no specific understanding of bitcode; it provides the content-addressability, distributed processing, and caching functionality that the rest of BCDB is built on.

2.3.1 MemoDB Store

All data in a BCDB instance is stored in a collection of data called a MemoDB Store. Currently, each Store is backed by either a SQLite [56] or RocksDB [57] database. When RocksDB is used, MemoDB enables its compression features to reduce the disk space needed for large Stores. The Store can hold three classes of data items:

Nodes. A Node is a single item of structured data. The contents of each Node are hashed using a cryptographic hash,[1] and the hash value is encoded in the form of an IPLD Content Identifier (CID) [59] that identifies the hash algorithm and type of data. Ignoring the astronomically unlikely possibility of a hash collision, two Nodes will have the same hash value if and only if their contents are exactly identical. Therefore, Nodes are uniquely identified by their CIDs. If two identical Nodes are added to the Store, as determined by comparing their CIDs, they are *deduplicated*: only one copy of the Node is actually stored.

The actual types of value that can be stored in a Node are essentially the same as those that can be stored in JSON [60], with the addition of raw binary data and links to other Nodes. A link is represented using the target Node's CID, effectively making the Store into a Merkle DAG.

Heads. A Head associates a name with an arbitrary Node (identified with its CID). Heads are mainly a convenience, allowing users to work with memorable names (such as `lz4`) rather than long CIDs (such as `uAXGg5AIgIllxYZ4GmUVna9_iFACRJ5LioFpOgvuWWi5_Stzy04M`).

Calls. Calls are the way MemoDB caches the results of arbitrary computations on Nodes. Each Call consists of a name (such as `alive.tv`), a list of zero or more CIDs indicating the arguments, and a CID indicating the result. At most one Call may be present for a given name and argument list.

2.3.2 MemoDB server

MemoDB can be used as a library to directly access the MemoDB Store on disk, but it can also be used in a client/server configuration. The server exposes an HTTP REST API allowing various operations to be performed on the Store. The client connects to this API and performs the desired operations.

Clients can also submit jobs to the server for processing and request their results; a job consists of a Call name and corresponding argument list. If the server finds a matching

[1] Currently 256-bit BLAKE2b [58].

Call already in the Store, it returns its result. Otherwise, it indicates to the client that no result is available yet, and holds the job until another client (called the "worker") indicates that it can process jobs with the given Call name. The server then sends the job to the worker; the worker processes the job, which may involve requesting Nodes from the server and/or submitting other jobs, and submits the result back to the server. The server then adds a corresponding Call to the Store, which can be retrieved by the original client. In this way, MemoDB supports fully distributed processing, up to hundreds of worker processes on different computers.

Aside from the main client implementation in MemoDB itself, separate client implementations have been written in C++ (for the Alive2 worker in section 5.6.1), and in Python (for analysis scripts).

2.4 STORING BITCODE IN MEMODB

The BCDB is built on top of MemoDB and stores LLVM bitcode modules in the MemoDB Store in a form that allows functions to be deduplicated. Each module stored in this way may be the compiled result of a single source file, or a linked version of an entire program or library.

When a module is added to a BCDB instance, each function in the input module is extracted into a separate *single-function module*, along with declarations of all global types, variables and functions referenced by the function. Each of these single-function modules is a fully valid LLVM module that can be analyzed or optimized independently with the standard LLVM tools. The BCDB also generates a *remainder module*, which includes everything from the input module other than the function bodies; in particular, it includes global variables and aliases.

Each of these modules is converted to binary bitcode format, and added to the MemoDB Store as a separate Node. Then, a new Node is created that includes links to the remainder module Node and the single-function module Nodes, each associated with the corresponding function name. This root Node is added to the MemoDB Store, and a corresponding Head is created with a user-specified name for the module.

Thanks to the use of the Merkle DAG, if a pair of identical functions is added to the BCDB in two different modules (or even the same module), only one copy will actually be stored. The root Node(s) will use the same CIDs for both copies, allowing the duplicate to be detected easily. If two completely identical modules are added, their root Nodes will also be duplicates with the same CID. Furthermore, if any Call is calculated for a given function module and stored in the BCDB, it will automatically apply to *all* occurrences of

that particular function, even if they came from different input modules, because of the way Calls are associated with CIDs.

This is a limited method of deduplication that works only for functions that have exactly identical bitcode. It can be easily confused by incidental differences between modules, even if they only contain identical functions. I use several design techniques to reduce these spurious differences:

1. Single-function modules do not include the name of the function being defined. This allows functions with different names to be deduplicated, as long as they are otherwise identical.

2. When functions refer to other functions and global variables, they are deduplicated based on the *names* of the referents, not the values. I chose to use names because when multiple versions of the same code are considered, the values of callees and variables are likely to change even when their names stay the same.

3. I rename all global string constants using a hash of their contents to ensure that identical strings get identical names. LLVM may assign different names to identical global strings in different modules for a variety of reasons; without renaming, functions that use those strings would produce different hashes.

4. I replace pointers to struct types with pointer-to-void when possible, effectively making the struct type opaque. This means that functions that only operate on the pointer and not the target can be detected as equivalent. This replacement is done for the entire module, including within other struct types that contain such pointers. (LLVM is transitioning to the use of opaque pointer types [61], which will make this transformation unnecessary.)

5. I remove all struct type names because type names appear in the IR for readability, but are ignored by compiler passes (LLVM unifies identical struct types regardless of their names).

6. I fill certain string tables used by LLVM when writing bitcode, such as the table of metadata kind names, with a consistent set of strings. Otherwise, because of the way LLVM works, the tables written would vary depending on the strings used in the original module.

Even with these changes, the BCDB's deduplication is limited to closely related functions. In chapter 5, I describe a way to deduplicate code according to a much broader concept

of equivalence, including semantically equivalent functions that are syntactically completely different.

As the name implies, LLVM bitcode is stored as a stream of bit fields that need not be aligned to byte boundaries. Standard compression formats, like Zlib, are less effective on these bit streams because they are designed for byte-aligned data. When saving bitcode, BCDB rewrites the bit streams produced by LLVM to align every field to a byte boundary, while ensuring that the resulting bit stream can still be decoded correctly by LLVM. The resulting bitcode is significantly larger when uncompressed, but significantly smaller when compressed (as when it is stored using RocksDB).

2.5 OBTAINING BITCODE

Given an existing C/C++ software package, one of the simpler ways to create bitcode is to build it with the Clang compiler using the `-fembed-bitcode` option. This option creates normal executable and library files, allowing the package's build system to work as usual, but adds an extra section to each file containing its bitcode. BCDB includes a tool named `bc-imitate` that can extract the bitcode from these files into a separate file, and compile (possibly modified) bitcode files into a replacement for the original executable or library. The `bc-imitate` tool annotates the bitcode file with some additional linking-related information from the original file, such as the list of dynamic libraries it uses, to ensure that it can produce a usable replacement.

A few other means of producing bitcode are worth mentioning. Clang can build bitcode files directly with the `-emit-llvm` or `-flto` options, but a given package's build system may not support the use of bitcode files; these options also require additional work to handle the link command. The WLLVM project offers an alternative way to build executables and libraries with embedded bitcode sections [62], but it is more complicated than `-fembed-bitcode` and significantly slower. LLVM-based compilers for other languages may have an option similar to `-fembed-bitcode`; for example, the `swiftc` compiler for Swift has an `-embed-bitcode` option.

In order to make the best use of the BCDB, it must be possible to build bitcode for large numbers of software packages automatically. Following the same approach as ALLVM, I extended the Nix Packages collection (Nixpkgs) [63] to automatically build packages in bitcode form. Nixpkgs is a collection of software package build instructions written in the Nix functional programming language; it includes more than 60 000 packages, primarily for Linux. I created an overlay for Nixpkgs that switches from GCC to the Clang compiler, enables the `-fembed-bitcode` option, and automatically removes certain other options incompatible with

`-fembed-bitcode`. The overlay includes manual fixes that were needed for certain packages that have compatibility issues, such as errors caused by warning messages printed by Clang but not by GCC, and some packages cannot be built in bitcode form because they are incompatible with Clang.

Bitcode can be built for macOS and iOS packages by using the `ENABLE_BITCODE=YES` and `BITCODE_GENERATION_MODE=bitcode` options for Xcode. An archive containing bitcode files, along with relevant compiler and linker options, will be included in each executable and shared library built. However, `bc-imitate` cannot be extended to support this archive unless several third-party dependencies are added, so instead of using the archive I use the intermediate bitcode files produced by Xcode.

2.6 EVALUATION

I tested the BCDB by using it to store more than a thousand bitcode modules built from hundreds of Nixpkgs packages. The BCDB scales efficiently, forming a database larger than 20 GiB (uncompressed) but only taking a few seconds to add or retrieve a module, which is far faster than actually processing the module with LLVM. Many syntactically identical functions are detected and deduplicated; out of 13 742 007 total functions added to the BCDB, only 4 672 174 (34%) are unique.

As another experiment, I built bitcode for all 100 versions of SQLite [56] from 3.6.10 through 3.26.0 and added each version to a BCDB instance. Figure 2.1 shows the cumulative number of functions in the BCDB. Each new version of SQLite only changes a portion of its functions, so only 16% of the functions are unique overall; the rest are duplicates.

It should be noted that, with the SQLite backend, the total size of the BCDB database file tends to be larger than the total size of the input bitcode modules, even after deduplication. This is due to the large overhead involved in using a separate module for each function. The RocksDB backend does not have this problem because it uses compression; in a similar experiment with many SQLite versions, the BCDB takes up 144 MiB, while the input modules, compressed individually, take up a total of 176 MiB.

2.7 FUTURE WORK

BCDB's distributed processing functionality is currently designed for use by a single development team at a time; there is no authentication or access control functionality. It could be highly useful to add authentication and access control functionality and make it possible to

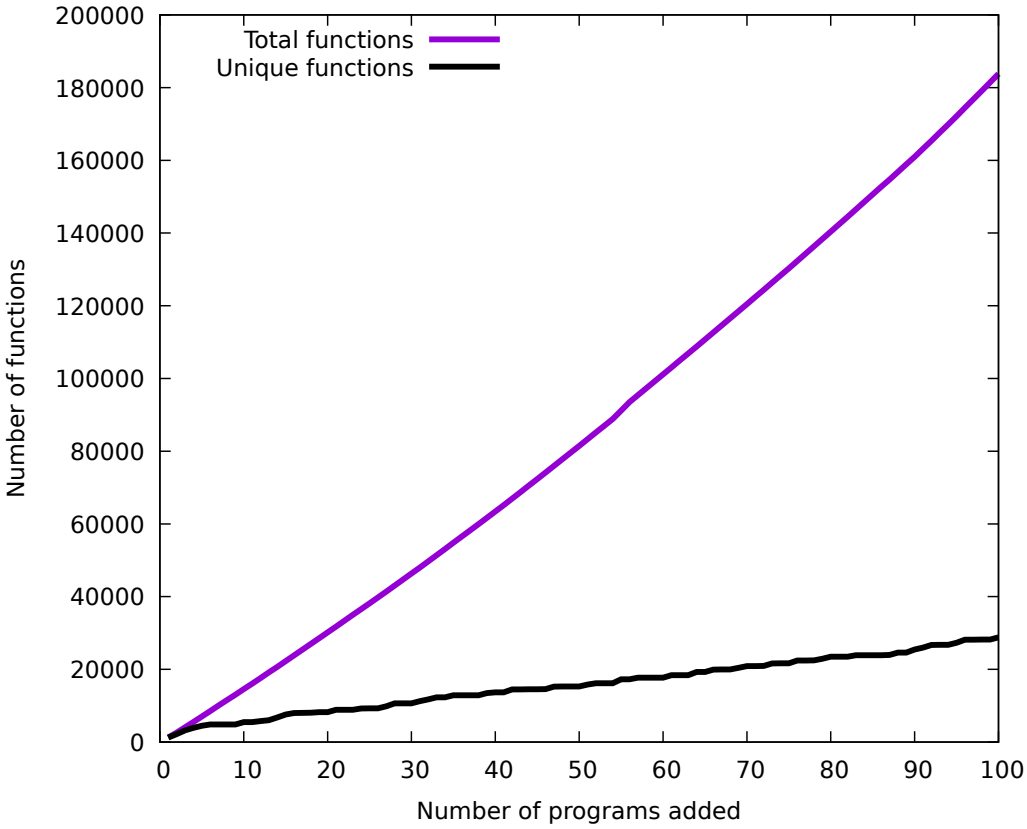


Figure 2.1: Cumulative number of functions in the BCDB as more versions of SQLite are added. After all versions are added, there are 183 785 total functions in the input modules, but only 28 818 unique functions are stored in the BCDB after deduplication.

run MemoDB servers on the open Internet, enabling researchers around the world to share bitcode and optimization results.

A MemoDB Store must currently be centralized on a single computer. If distributed storage were supported, perhaps using protocols based on IPFS, it would be possible for a huge company like Google to store all their compiled code in one BCDB instance.

Many useful applications could be built on top of the functionality provided by BCDB. It would be particularly interesting to add IR search functionality, enabling users to search for particular snippets of LLVM IR across huge numbers of software packages. Developers working on the LLVM project might use such a tool to determine how a given LLVM feature is being used in practice.

Chapter 3: Cross-Package Function Deduplication

3.1 INTRODUCTION

Modern embedded systems often run a full, configurable software stack, with a large number of available software packages that may be selected depending on how the system is being used [5]–[7]. However, such systems often have strict limitations on storage space, constraining the number of packages that can be installed. In this chapter, I demonstrate an automated, application-neutral technique to reduce code size on such systems. Specifically, I introduce BCDBmux, a version of software multiplexing [18] that also deduplicates identical functions, even when the functions appear in separate programs or separately developed packages.

BCDBmux is built on the BCDB described in chapter 2, which it uses to store IR for *all* packages available for an embedded system. The BCDB automatically detects duplicate functions, even when they occur in separate packages. For a desired end-system configuration, consisting of a set of packages that should be installed on the system, BCDBmux is used to extract all the code for those programs from the BCDB and combine them into a single statically linked program, as explained below.

Section 3.2 compares my work with previous work, and section 3.3 gives a more detailed explanation of the particular technique I build upon. Section 3.4 describes BCDBmux and its design. Section 3.5 evaluates the benefits of BCDBmux for the OpenWRT system. Finally, section 3.7 concludes the chapter.

3.2 RELATED WORK

This work is focused on applying deduplication across packages, and it finds duplicate code using the simple function-level syntactic equivalence check implemented in the BCDB (section 2.4). For related work on more sophisticated deduplication techniques, and other general work related to deduplication, see section 5.2 and the techniques in chapter 6 that refer to duplication.

Allmux [18] is the only previous compiler-time technique to reduce code duplication across programs, and BCDBmux builds directly on this work. Allmux uses the Automatic Multicall technique (explained in section 3.3) to merge independent programs into a single multicall program, replacing dynamic linking with static linking at the IR level. Allmux only includes a single copy of each shared library in the multicall program, ensuring that the size benefits of shared libraries are maintained, but it achieves considerable code size reduction

by removing unused functions from libraries and eliminating the overhead of dynamic linking. However, Allmux incorrectly assumes that the static linker will have the same behavior as the dynamic linker, so the programs it produces will behave incorrectly in some cases. For instance, if two unrelated input programs both have weak definitions of the same symbol, Allmux will arbitrarily choose one of the definitions to use and delete the other one, which may lead to unpredictable incorrect behavior at run time. On the other hand, if two input programs have strong definitions of the same symbol, Allmux will refuse to run.

BCDBmux uses the same Automatic Multicall technique as Allmux, but follows it with function-level deduplication. Any duplicate functions in different programs or libraries would be detected by our system but not by Allmux. Such duplication can happen particularly due to statically linked or header-only libraries used by both applications, and can also happen due to manual code copying, software engineering reuse practices, C++ template instantiation, and so on.

3.3 BACKGROUND: MANUAL AND AUTOMATIC MULTICALL

A "*multicall*" program combines multiple programs into one executable, and dispatches them based on either a predefined argument or the name used to invoke the program. Some packages, such as BusyBox [64] and Coreutils [65], are explicitly designed to do this using special support in their source code and build systems. This feature could be added by hand to other software, but doing so would require nontrivial effort from developers. *Automatic Multicall* [18] is a compiler transformation that automates this process without requiring changes in source code: a compiler pass takes several application programs as input and combines them into a single multicall program.

Carrying out these steps after individual executables (e.g., ELF format binaries) have been generated could be quite complex, and it would limit the optimization opportunities available for the multicall program. Instead, Automatic Multicall was implemented using the LLVM compiler's intermediate representation (IR). It merges the LLVM bitcode files for all applications into a single file, renaming internal functions and global variables to minimize name conflicts. It uses a new compiler pass (a simple IR-to-IR transformation) to add a new main function that dispatches to individual program entry points based on the name used to invoke the program. The transformation produces a single merged program in IR form as the output, with the new main function as the entry point. Note that this is purely a compiler transform: no link editing occurs in this step.

```
new_names = {}
for each function f in any input module:
  new_names[f.name] = '__bcdb_body_' + f.name
for each function f in any input module, skipping duplicates:
  f.set_name('__bcdb_body_' + f.name)
  for each name n referred to by f:
    rename n to new_names[n]
  link f into the merged module
```

Listing 3.1: Simplified deduplication algorithm.

3.4 CROSS-PACKAGE FUNCTION DEDUPLICATION

In order to achieve function deduplication across unrelated packages, BCDBmux integrates the BCDB, the automatic multicall transformation described in section 3.3, and the function deduplication optimization, which is described in this section. To use my system, the developer must first build each program and library to be used with BCDBmux in the form of a compiler IR module, which can be added to the BCDB. Different methods of producing the bitcode are described in section 2.5. As each module is added to the BCDB, it is split into separate functions and the functions are deduplicated as described in section 2.4.

Once the BCDB has been filled with IR, BCDBmux can be invoked to automatically generate a multicall program that combines the programs and libraries in the BCDB (either a subset of them, or the entire BCDB). During this process, BCDBmux detects when the same function is used more than once (by a single module or by different modules). When this occurs, it only extracts a single copy of the function's definition, and redirects all uses to that definition, thus deduplicating the function.

The basic algorithm is shown in listing 3.1. We detect duplicate functions by checking whether the functions being loaded from the BCDB have the same CID (see section 2.3.1). If a function was duplicated between any of the original modules (with the same or distinct names), BCDBmux deduplicates it as follows. First, BCDBmux loads a single copy of the function from the BCDB and gives it a unique name. We use the name `__bcdb_body_n`, where n is one of the names originally used for the function; if that name is already in use, we add an arbitrary suffix to make it unique. Then, before the single-function modules are linked together to form the multicall program, BCDBmux updates all references in each module to use the new `__bcdb_body_n` name of the function being referred to. The modules can now be linked together and the multicall process can proceed as before.

The simplified code in listing 3.1 will produce incorrect results in several key situations. The following sections describe the problematic cases, their solutions, and finally the full and correct algorithm.

3.4.1 Function address comparison

Consider the following program:

```
void f0() { do_something(); }
void f1() { do_something(); }
int main() { return &f0 == &f1; }
```

Listing 3.2: A program that uses function address comparison.

The program should return `0` because `f0` and `f1` have different addresses. However, if we applied the code in listing 3.1, `f0` and `f1` would be combined into one function and the program would incorrectly return `1`. In order to prevent this from happening, we can modify `main` so that it doesn't refer to `__bcdb_body_f0` directly. Instead it will refer to *function stubs*, one for each function in the original module, which are functions that simply pass their arguments on to the actual definition in `__bcdb_body_f0`. For example:

```
void __bcdb_body_f0() { do_something(); }
void f0() { return __bcdb_body_f0(); }
void f1() { return __bcdb_body_f0(); }
```

Listing 3.3: The program processed with the simplified algorithm.

This ensures that different functions have different addresses, while enabling us to deduplicate the bodies of different functions. BCDBmux creates stubs for *all* function definitions, and uses the stubs in all cases where a function is referred to or called, including recursive calls. This extra layer of indirection could potentially hurt performance, so I rely on LLVM's optimizer to inline away the stubs when possible.

Limitations of LLVM prevent BCDBmux from creating stubs for varargs functions.[1] BCDBmux currently deduplicates these functions without using stubs, which can potentially lead to incorrect behavior as described above. I expect such behavior to be extremely rare as it can only occur when varargs functions (which are rare) are used with function pointer comparison (which is also rare).

3.4.2 Name conflicts

The simplified algorithm assumes that no two input modules define functions with the same name. When merging two unrelated modules (programs or libraries) into one, this

[1]The only way for a varargs function that does not know the types of its arguments to pass them to another varargs function, in LLVM, is to use a `musttail` call. Unfortunately, some of LLVM's standard optimizations have bugs that break such calls, and we want to apply those optimizations after BCDBmux, so `musttail` calls aren't a viable solution.

assumption can easily be broken; there are often name conflicts if the original modules were not designed to be statically linked together, especially when static functions are considered. We must extend the algorithm to handle name conflicts correctly, so that program behavior is not altered by the merging process.

Specifically, if two functions have the same name but different CIDs, or when one input module defines a function but another input module uses a function with the same name defined in an external library, it is impossible to merge the two functions. Instead, one of the functions must be renamed, and we must update all references to that function to use the new name. The following example shows two input modules with a name conflict, followed by a merged module where one of the conflicting functions has been renamed, allowing the merged module to work correctly.

```
// Input module A:
int func() {
    return 0;
}
void *func_pointer_A = (void*)&func;

// Input module B:
int func() {
    return 1;
}
void *func_pointer_B = (void*)&func;

// Merged output module:
int __bcdb_body_func() {
    return 0;
}
int func() {
    return __bcdb_body_func();
}
void *func_pointer_A = (void*)&func;

int __bcdb_body_func.0() {
    return 1;
}
int func.B() {
    return __bcdb_body_func.0();
}
void *func_pointer_B = (void*)&func.B;
```

Listing 3.4: An example of name conflict.

3.4.3 Indirect conflicts

If two functions have the same name and the same function CID, we would like to merge

them so that we only have one copy of the definition. However, we must take into account any changes that will be made to the body of the function because it refers to other functions that will be renamed. This occurs when two modules being merged have functions with the same name but different CIDs, as in the previous section.

For example, given the following input:

```
// Input module A:
int callee() { return 0; }
int caller() { return callee(); }

// Input module B:
int callee() { return 1; }
int caller() { return callee(); }
```

Listing 3.5: An input program containing an indirect conflict.

The two versions of `callee` have different CIDs, so one of them must be renamed. Suppose module B's `callee` is renamed to `callee.B`. Now, even though both versions of `caller` have the *same* CID, they cannot be merged because they must call different versions of `callee`. Module A's version of `caller` can be used as-is, but module B's version must be rewritten to call `callee.B`. Therefore, we must also rename module B's version of `caller` (both the `__bcdb_body_n` definition and the `caller` stub) so it doesn't conflict with module A's version. The final output would look something like this:

```
// Definitions of callee and callee.B omitted.
int __bcdb_body_caller() { return callee(); }
int caller() { return __bcdb_body_caller(); }
int __bcdb_body_caller.B() { return callee.B(); }
int caller.B() { return __bcdb_body_caller.B(); }
```

Listing 3.6: Output for the indirect conflict program.

When one function is renamed, it may force other functions to be renamed, which may force even more functions to be renamed, and so on. My algorithm uses a graph traversal to determine which functions need to be renamed. After loading all input modules, I build a directed graph called the *global reference graph*. Every node is a global definition (either a function or a global variable). Every edge goes from a function to a global definition it directly references, or from a global variable to a global definition pointed to by the variable's initial value. The global reference graph is similar to a call graph, but it omits all indirect calls because *we only need to consider direct function calls*: indirect function calls through function pointers *can be ignored* because they do not cause conflicts.

Having built the global reference graph, we can traverse it to determine which functions must be renamed. Assuming the graph is acyclic, we visit each node in postorder; this ensures that when we visit a function, we have already visited and renamed everything it refers to. When we visit a global variable, we assign it a unique name. When we visit a function, we generate a tuple consisting of its function CID and the new names of everything it refers to; if we've previously visited a function with the same tuple we can merge it with this one. Otherwise, we give the function a unique name.

```
resolve_all_references()
create_global_reference_graph()
for each SCC in the graph in reverse postorder:
  give unique stub names to every function in SCC
  if there are global variables in the SCC:
    give unique definition names to every function in SCC
    give unique definition names to every variable in SCC
  else:
    tuple = (CIDs of all functions in the SCC,
        new names of all references from the functions)
    if a previous SCC had an equal tuple:
      # merge the functions
      reuse the definition names from the previous SCC
    else:
      give unique definition names to each function in SCC
for each module:
  for each global definition def:
    rename each reference from def to a function,
        using its new stub name
    rename each reference from def to a global variable,
        using its new definition name
    if def is a global variable:
      link def in from the remainder module
      rename def using the new definition name
    else:
      if this definition name is not already loaded:
        link def in from the single-function module
        rename def using the new definition name
      create a stub for def using the new stub name
```

Listing 3.7: Full deduplication algorithm.

3.4.4 Mutual recursion

The previous section assumed there were no cycles in the global reference graph. In practice, cycles can easily arise in the case of mutual recursion, or when a function refers to a global variable that points back to the function. To handle cycles, I build the strongly connected components (SCCs) of the global reference graph, using Tarjan's SCC finding algorithm [66]. I traverse the SCCs themselves in postorder (technically, this is postorder on

the acyclic graph formed by logically collapsing each SCC into a single node and ignoring self-edges). When we process an SCC, we know that every reference from a node in the SCC points to either another node in the SCC, or a node outside the SCC that has already been renamed. If the SCC contains any global variables, it cannot be merged with anything else, so I assign a new name to every node in the SCC. Otherwise, I generate tuples as before for every function in the SCC, and look for a previously visited SCC with the same set of tuples. If a previous matching SCC is available, I merge every function in the current SCC with the corresponding function in the previous SCC; otherwise, I assign a new name to every node in the current SCC.

3.4.5 Full algorithm

The full version of the deduplication algorithm is shown in listing 3.7. It fixes all the problems described above, and also shows how global variables are handled. The input modules are handled in arbitrary order; this only affects the final names of conflicting functions, not the determination of what can be merged.

3.5 EVALUATION

I evaluated BCDBmux for the widely used OpenWRT system, which is a Linux-based software distribution for embedded systems, often used for customizing wireless routers [5]. OpenWRT is highly configurable, offering more than 100 GUI modules and about 10,000 packages that can be installed using the `opkg` package manager. The OpenWRT community lists 748 devices officially supported by the latest release [67], of which 609 have 64 MiB flash or less, and 577 have 16 MiB or less. The community strongly recommends a minimum of 8 MiB for basic software update capabilities, because roughly 4 MiB is used by the kernel and core utilities in a standard install [8]. OpenWRT uses a compressed file system to minimize space usage, and its documentation includes significant discussion on how to avoid space exhaustion when installing packages, such as by manually reclaiming space [8].

3.5.1 Package selection

I evaluated BCDBmux on the OpenWRT distribution fetched from its Git repository on July 17, 2019. In order to find commonly-used sets of packages to test on, I limited my evaluation to software supported by OpenWRT's LUCI system, which is a web GUI that can be used to configure the device running OpenWRT. Although most of the LUCI packages

themselves consist solely of Lua scripts, they depend on many other packages containing binaries and libraries, which BCDBmux can optimize.

I modified the OpenWRT build system to build LLVM bitcode for all packages using Clang and WLLVM [62]. I targeted x86-64, using musl as the C standard library implementation (per OpenWRT default). As OpenWRT is only designed to work with GCC, I needed various small changes to the build scripts and packages in order to build them successfully with Clang; most seriously, I had to disable certain functionality of `libelf` because it used GCC-specific language extensions. OpenWRT by default uses `uClibc++` as the C++ standard library implementation for some packages, but uses the compiler's default implementation for other packages; I disabled `uClibc++` entirely as it was causing link conflicts, and used the compiler's default implementation instead.

I chose three sets of packages to optimize, representing three different use cases for an OpenWRT device:

Home

- `luci`: LUCI server
- `luci-app-e2guardian`: web proxy
- `luci-app-squid`: web proxy
- `luci-app-shairplay`: audio streaming
- `luci-app-aria2`: downloading tool
- `luci-app-samba`: SMB fileserver

Security

- `luci`: LUCI server
- `luci-app-clamav`: virus scanner
- `luci-app-noddos`: automatic firewall
- `luci-app-fwknopd`: pork knocking daemon
- `luci-app-banip`: IP-based banning tool
- `luci-app-cshark`: packet recorder
- `luci-app-firewall`: firewall

Server

- `luci-ssl`: LUCI server with SSL support

- `luci-app-acme`: SSL certificate generator
- `luci-app-ddns`: Dynamic DNS service
- `luci-app-radicale2`: Calendar webapp
- `luci-app-ocserv`: VPN server
- `luci-app-transmission`: BitTorrent service
- `luci-app-mjpg-streamer`: Webcam server

Although I only selected 6 to 7 packages in each set, I also optimized all the packages they depend on, directly or indirectly. The total number of packages ranged from 58 (for Home) to 103 (for Server). I optimized all executable binaries included in the packages, along with all of the shared libraries they link against from any package. However, I excluded plugin files and a few libraries from the OpenWRT base system that are not included in any package.

3.5.2 Basic evaluation

In this section, I apply BCDBmux to each of the three package sets, and compare it against a baseline with no deduplication. First, I use WLLVM to retrieve bitcode modules for every executable and library built by OpenWRT, and add all of the modules to the BCDB. Then, to apply BCDBmux to a package set, I follow the following steps:

1. List the executable binaries included in the selected packages and any packages they depend on. Also list the shared libraries needed by the executables.
2. Using BCDBmux, merge together all of the binaries and libraries into a single multicall module, with a new muxed `main` function that determines which executable is being used by checking `argv[0]` and then calls the appropriate `main` function.
3. Perform LTO on the muxed module using `opt -Oz`, which optimizes for size.
4. Compile the module into an executable ELF file using `llc` and `clang`.
5. Strip unnecessary data from the ELF file using `strip -s -R .comment`.

For the baseline, I use the same list of executables and libraries from step 1; note that this excludes libraries that are not needed by any executable in the package set. I take the bitcode modules produced by WLLVM for each executable and library, and apply steps 3–5 on each module separately. Thus, the baseline includes all the benefits of using LTO on individual executables and libraries, but not the benefits of BCDBmux.

Finally, I compared the size of the single ELF file produced by BCDBmux against the total size of all ELF files in the baseline. The size of ELF files in the Home package set is reduced from 14.5 MiB to 10.2 MiB (30%), Security is reduced from 11.7 MiB to 7.2 MiB (38%), and Server is reduced from 10.7 MiB to 6.9 MiB (36%). On small embedded devices these reductions are very significant. For example, while the baseline Home package set would be unlikely to fit on a device with 16 MiB flash after data files and the Linux kernel are accounted for, the BCDBmux version is much more likely to fit comfortably.

3.5.3 Compilation time

In some applications, it could be important for new packages to be generated very quickly, especially if new package sets are generated often. On the Home package set, which takes the longest to compile, I found that it took only 43 seconds to apply BCDBmux. However, BCDBmux also increases the time needed to perform LTO and compile to machine code, since a much larger module is being processed. On Home, it took about 16 minutes to apply `opt -Oz` and `llc` to the BCDBmux version, compared with only 5 minutes total for all files in the baseline. This amount of time may be acceptable in many scenarios, but it will be too slow for cases where package sets are changed extremely often.

3.6 FUTURE WORK

The current version of BCDBmux does not support plugins or other libraries that are loaded with `dlopen`. It is also not fully compatible with the dynamic linker in some cases, such as when one program refers to a symbol that is defined in multiple libraries in the package set. Chapter 4 describes an improved system that can handle `dlopen` and maintain full compatibility with the dynamic linker.

BCDBmux causes a significant increase in compilation time, because a much larger amount of code is optimized as a single unit using LLVM's LTO functionality. LLVM also supports ThinLTO, an alternative to normal LTO that scales better to large amounts of code because it does not require linking all code into a single module [68]. Future work could extend BCDBmux to work with ThinLTO, improving compilation time.

The current implementation requires several commands to be run manually for each package set that should be optimized, and the resulting multicall program must be installed manually. Future work could integrate BCDBmux into the existing system package manager, so that the user can use their accustomed package management tools without any extra work. A *package server* running on a more powerful computer would receive the list

of desired packages from the user, automatically use BCDBmux to create an optimized package with a single multicall binary, and send it back to the device for installation.

3.7 CONCLUSION

I have presented BCDBmux, which uses package multiplexing and function deduplication to eliminate duplicated code across multiple programs, both within and across separately developed packages. BCDBmux achieves a form of deduplication that has not to my knowledge been achieved before: deduplication of equivalent functions across a full system.

My results show that BCDBmux is able to achieve substantially greater code size reductions than standard compiler approaches for dozens of packages in OpenWRT. In particular, for three distinct OpenWRT software configurations, with 58 to 103 packages each, BCDBmux reduces the total size of executables and libraries by 30% up to 38%. No previous software tool I know of is able to achieve such large reductions in code size fully transparently, and with no impact on the functionality of a system.

Chapter 4: Guided Linking

This work was a collaboration with Will Dietz and Prof. Vikram Adve, and it has been previously published at OOPSLA 2020 [25] and the 2020 LLVM Developers' Conference [26]. This chapter is a slightly extended version of the OOPSLA paper; rights to reprint this material have been retained. The implementation has been published as an open source project [27].

4.1 INTRODUCTION

4.1.1 Motivation

Dynamic linking is ubiquitous on modern Linux, Windows, and MacOS computers. There are three primary motivations for using it. The first is to prevent the duplication of library code that occurs with static linking. The reductions in disk and memory usage can be enormous; compared with static linking, using dynamic linking in a distribution of the Clang and LLVM compiler infrastructure can reduce the total binary size from 1800 MB to 260 MB. Dynamic linking also makes it possible to upgrade a library without waiting to recompile every program that uses it; for basic libraries like OpenSSL, this makes security updates orders of magnitude faster. Finally, dynamic linking allows software to be split into modular components that can be distributed separately and loaded only when necessary. Many popular programs and libraries, such as Apache HTTPD, Nginx, and Qt 5, rely on dynamically loaded modules; even Python, when run interactively, uses `dlopen` to load its `readline` module.

However, dynamic linking is not without its downsides. There is significant size and speed overhead involved in storing symbol tables, resolving symbols, and calling dynamically linked functions; the indirection added to function calls can make the program 4% slower [69]. The extra data needed for the lookup tables and symbol tables takes up extra space both on disk and in RAM. The process of resolving symbols and filling in the lookup tables can be a significant source of overhead, especially for short-lived programs. But the greatest downside of dynamic linking is that it prevents the compiler from optimizing across the boundaries of programs and dynamic libraries. There are two key problems preventing optimization:

1. The compiler only works on a single program or dynamic library at a time, preventing any analysis of external function calls to a dynamic library. Even if the compiler supports link-time optimization, it can optimize function calls across different modules

within a single program or library, but not across different programs and libraries.

2. The compiler can make no assumptions about the runtime behavior of the linker, which depends on many environment settings outside the control or even knowledge of the compiler, such as `LD_PRELOAD`, `LD_LIBRARY_PATH`, and `/etc/ld.so.cache`. With no way to be certain what code will actually be dynamically linked together, the compiler is forced to take the conservative option and assume nothing about the runtime environment can be known.

These problems can be avoided by using static linking instead of dynamic linking. Indeed, this is what Google does for most of their internal software. But system-wide static linking is impractical for most users, for several reasons. In the common case where an existing code base has been designed around dynamic linking, there can be significant effort required to make it compatible with static linking—not only because the build system must be changed, but because the code itself may invoke the dynamic linker (to load plugins) or rely on subtle details of the way the dynamic linker works. Even if the code is compatible with static linking, there can be an enormous increase in disk usage and memory usage because static linking causes library code to be duplicated into every program that uses it. As mentioned above, switching Clang and LLVM from dynamic to static linking could increase the total size of programs and libraries from 260 MB to 1800 MB due to this duplication. Finally, static linking adds even more overhead whenever a library is updated, because everything that uses that library (even indirectly) must be re-linked and redistributed. Imagine if you had to download new versions of most of the software on your computer every time OpenSSL is updated! Given the severe costs of static linking, developers have generally chosen to use dynamic linking instead, despite its costs.

The ultimate goal of our research is to solve both Problem 1 and Problem 2, allowing dynamically linked code to be optimized just as well as statically linked code, while preserving the code size and modularity of dynamic linking. We also, crucially, need to maintain compatibility with existing software that depends on the current behavior of the dynamic linker.

4.1.2 Our approach

In this chapter, we propose a new system called *Guided Linking* that is able to solve both problems above without requiring changes to application code, by making use of some additional input from the developer. Specifically, the developer provides two inputs to the compiler:

Optimized set. The developer provides multiple dynamic objects (programs, libraries, and plugins) to the compiler to be optimized as a single unit, which we call the *optimized set*. The optimized set addresses Problem 1 above by enabling the compiler to work on multiple programs and libraries at once.

Constraints. The developer specifies *constraints* on the runtime dynamic linking behavior of the set. The constraints address Problem 2 above by providing information about the runtime behavior of the linker, allowing the compiler to effectively analyze and optimize the code.

Guided Linking proceeds by moving almost all of the code in the optimized set into a new library, called the *merged library*, which is optimized using existing compiler optimizations. When used on the merged library, these unmodified optimizations are now inherently applied across boundaries no traditional compiler has pierced before: program-to-dynamic-library, dynamic-library-to-dynamic-library, and even program-to-program. By linking the original dynamic objects to the merged library and adding layers of indirection when necessary, Guided Linking can ensure that the new programs and libraries work just like the originals, as long as the constraints are met.

In addition to existing optimizations, Guided Linking also enables new optimization techniques to be developed that are designed specifically for dynamically linked code. In this chapter, we demonstrate a new optimization built on Guided Linking that reduces code size by merging duplicate functions, even when the functions originate from different software packages.

4.1.3 Use cases

Here are several specific examples to illustrate how developers could make use of Guided Linking. Our evaluations in section 4.9 are based on each of these scenarios.

Optimizing a Docker container. Typical Docker containers are excellent targets for Guided Linking because the entire set of software to run in a container is normally known in advance. In addition, containers are often built once and then deployed to hundreds of machines, multiplying the benefit of our optimizations. All programs and libraries in the container can be included in a single optimized set, and the developer can specify strong constraints based on the knowledge that no external programs will ever be used on the system. Section 4.9.1 shows we can increase speed significantly in such a scenario.

Distributing Clang and LLVM. Without compression, the stripped binaries for the Clang compiler, LLVM tools, and associated libraries can take up more than 200 MB of disk space. If a developer wants to redistribute Clang while minimizing the amount of space needed, they can apply Guided Linking to the entire set of LLVM programs and libraries. They can use strong constraints because LLVM is a self-contained project. In this situation, section 4.9.2 shows we can decrease code size by 5% while also increasing the speed of Clang by 5%.

Merging multiple versions of libraries. Some libraries, such as the Boost libraries, often need multiple versions installed at once in order to satisfy the dependencies of different programs. We can reduce code size with Guided Linking by combining all necessary versions of the library and using our function deduplication technique, which is effective even if compatibility concerns mean that only weak constraints can be provided. Section 4.9.3 demonstrates a 57% size reduction by combining 11 versions of Boost into an optimized set and applying Guided Linking.

4.1.4 Benefits

Guided Linking reduces the compile-time uncertainty caused by dynamic linking. Depending on what constraints the developer provides, we can apply arbitrary compiler techniques across the boundaries between programs and dynamic libraries. Unmodified existing optimizations, including all standard link-time optimizations, can easily be applied across these boundaries without requiring any additional effort. Moreover, Guided Linking requires only small changes to build systems, allows the use of a wide range of both imperative and functional languages, and imposes no restrictions on the use of programming strategies such as shared libraries, plugins, or programming language features.

There are several practical benefits of such a system:

Improving speed. By performing inlining and other standard optimizations across dynamic library boundaries, we can improve the speed of any code that uses frequent program–library interaction. Advanced optimizations such as devirtualization can also be used, which may enhance performance further.

Reducing code size within programs. We can reduce code size by applying compiler- and linker-based techniques across program / shared library boundaries, including both basic optimizations like dead function elimination and interprocedural constant propagation, and

advanced techniques like partial specialization. Reducing code size can be highly useful for the reasons discussed in sections 1.1 and 6.3.

Enabling novel optimizations across programs. When components of multiple programs are available to the compiler as a single unit, new types of optimization may become possible. In this chapter, we demonstrate a new technique enabled by Guided Linking that can detect and merge duplicate functions, even when the functions originate from different software packages. This optimization could be integrated into a package server for embedded systems, as proposed in section 3.6, making it possible to install larger numbers of packages on a system with limited space.

4.1.5 Contributions

The technical contributions of this chapter are as follows:

(a) We identify a set of constraints that can usefully be placed on dynamic linking behavior in order to enable optimizations.

(b) We explain how optimizable code can be generated for each possible set of constraints, without breaking compatibility with existing software that relies on the dynamic linker.

(c) We show how our system enables techniques like function deduplication to be applied across dynamic linking boundaries.

(d) We implement our proposed system on top of the LLVM compiler infrastructure.

(e) We show experimentally (section 4.9) that our tool can transparently compile large, real-world Linux software packages. As part of this demonstration, we show that we can easily identify sets of constraints that can safely be applied to real-world software. We show that our system can improve speed by more than 9% or reduce code size by more than 50% on widely used software packages, depending on the optimized set and constraints used; the code size improvement is much more than any previous compiler-based technique we are aware of.

4.1.6 Chapter organization

Section 4.2 discusses related work. Section 4.3 gives a more detailed overview of the Guided Linking process. Section 4.4 explains how dynamic linking works. Sections 4.5

and 4.6 discuss the optimized set and constraints respectively. Section 4.7 describes the Guided Linking process itself, and section 4.8 describes the function deduplication process. Section 4.9 presents our evaluation and results, section 4.10 discusses future work, and section 4.11 concludes the chapter.

4.2 RELATED WORK

Our system is the first we know of that makes use of developer-provided constraints on dynamic linking behavior, but other optimization systems have used other kinds of constraints on run-time behavior. For example, when there is a constraint that certain features of a program will not be used, perhaps because its configuration or input data is known at compile time, the corresponding code can be removed automatically. For discussion of such techniques, see section 6.5. Although conceptually related, these other techniques take advantage of optimization opportunities that are essentially orthogonal to the ones used by Guided Linking; in fact, Guided Linking could make these techniques more effective by allowing them to cross dynamic library boundaries.

There is a great deal of related work on various forms of code deduplication; the novelty of our deduplication technique is that it works across dynamic linking boundaries. See section 5.2.1 for related work, as well as the techniques in chapter 6 that refer to duplication.

4.2.1 Expanding domains of optimization

Over the history of compiler optimizations, the scope of code transformed by a single optimization has been growing larger and larger. The earliest optimizing compilers performed only local optimizations, which consider only a small part of a function. In the 70s and 80s, compilers began to use "global" optimizations, which worked on an entire function at a time [70]. This was soon followed by work on interprocedural optimization, particularly at Rice University [71]. These optimizations are now standard in all common compilers. In the 90s, link-time optimization (then called "cross-module optimization") was becoming more common [29], allowing optimizations to cross the boundary between source code files. Guided Linking is a natural extension to this history, allowing optimizations to expand further and cross the boundary between programs and shared libraries, and even between multiple distinct programs. Our system is the first we know of that can perform arbitrary optimizations across this barrier, without breaking compatibility with existing dynamically linked code.

4.2.2 Optimizing dynamically linked code

Commonly used compilers have a very limited ability to optimize dynamically linked code. For example, the LLVM compiler has a `LibCallSimplifier` pass that optimizes C standard library calls, which is possible because the semantics of these calls are known to the compiler, but it has no equivalent for any other library. This section discusses tools that work in more general cases.

Reducing dynamic linking overhead. Much of the existing work on optimizing dynamically linked code focuses on speeding up the dynamic linking process itself. This is generally done by caching the results of the relocations done by the dynamic linker [72]–[75]. Another option is to have the static linker perform relocations based on the original library versions, so the dynamic linker only needs to update the relocations if the libraries have changed [76]. Several tools go as far as taking a snapshot of the address space after relocations have been processed, then saving the snapshot as a new executable file, which can be run without any further relocation [77]–[80]. None of these techniques eliminate the indirection through the GOT and PLT, and they do not enable any compiler optimizations across dynamic linking boundaries. Guided Linking can sometimes replace dynamic relocations with static references; in other cases, caching can be combined with our system for an additional performance gain.

Other work has proposed modifications to hardware branch predictors to speed up calls through the PLT [69]. Again, in cases where Guided Linking cannot eliminate dynamic relocations, this technique and Guided Linking can be profitably combined.

Dead code elimination. A number of tools can debloat shared libraries by analyzing the programs that use them and removing unneeded code [30]–[33]. The primary goal is to improve security by reducing the attack surface, with code size reduction as a secondary goal. In general, these tools use a custom-built analysis to construct a call graph that encompasses multiple libraries and programs, which is then used to determine which code may be safely eliminated. Some of these tools work directly on binary code, which allows them to debloat libraries that our Guided Linking tool does not support (such as `glibc`, which cannot be built in the form of LLVM IR).

The biggest limitation of these tools is that they cannot reuse existing implementations of compiler analyses and optimizations. Call graph construction and dead code elimination are already implemented in standard compilers, but the debloating tools have to introduce entirely new implementations that are explicitly aware of dynamic linking. In contrast,

Guided Linking allows the compiler's existing analysis and optimization code to be reused without modification. This benefit extends to all possible optimizations that would be worth performing across dynamic linking boundaries; for example, Guided Linking can easily reuse the compiler's support for inlining, whereas extending these other tools to support inlining would require an entirely new implementation with explicit awareness of dynamic linking.

Another major limitation of these tools is that they do not explicitly specify what assumptions they make about the runtime configuration of the dynamic linker. For instance, none of them explain how they interact with `LD_PRELOAD`. Most of them also have limited support for dynamically loading symbols with `dlsym`; however, Davidsson, Pawlowski, and Holz [30] use a developer-provided whitelist to indicate which symbols are needed by `dlsym`, which is very similar to using our NoUse constraint with an exception list. Finally, unlike Guided Linking, these tools cannot optimize libraries unless all of the programs that use them are known in advance.

Arbitrary optimizations. The most aggressive previous technique to optimize across the dynamic linking boundary is the Allmux tool by Dietz and Adve [18]. Allmux takes the compiler IR for a set of programs and all the dynamic libraries they depend on, and statically links the IR together into a single module, including only one copy of each library. The module can be optimized using arbitrary existing optimizations, including inlining and dead code elimination. Then the module is compiled into a single *multicall program*, which can be invoked to run any of the programs that were linked in (see section 3.3). Allmux enables arbitrary compiler optimizations and completely eliminates dynamic linking overhead, but it has severe limitations which make it difficult to use in practice:

1. Allmux requires the entire set of libraries to be optimized together; unlike our system, no libraries can be left out of optimization.

2. Unlike our system, Allmux prohibits external programs from using the libraries that are optimized together.

3. Unlike our system, Allmux prohibits programs and libraries from using `dlopen` and `dlsym`.

4. Allmux uses a static linker to combine programs and libraries that were designed to be dynamically linked, which can lead to incorrect behavior. For instance, when combining two libraries that define the same symbol, Allmux will refuse to run (if both definitions are strong) or arbitrarily delete one definition (if both definitions are weak). Guided Linking handles these cases by falling back to the dynamic linker.

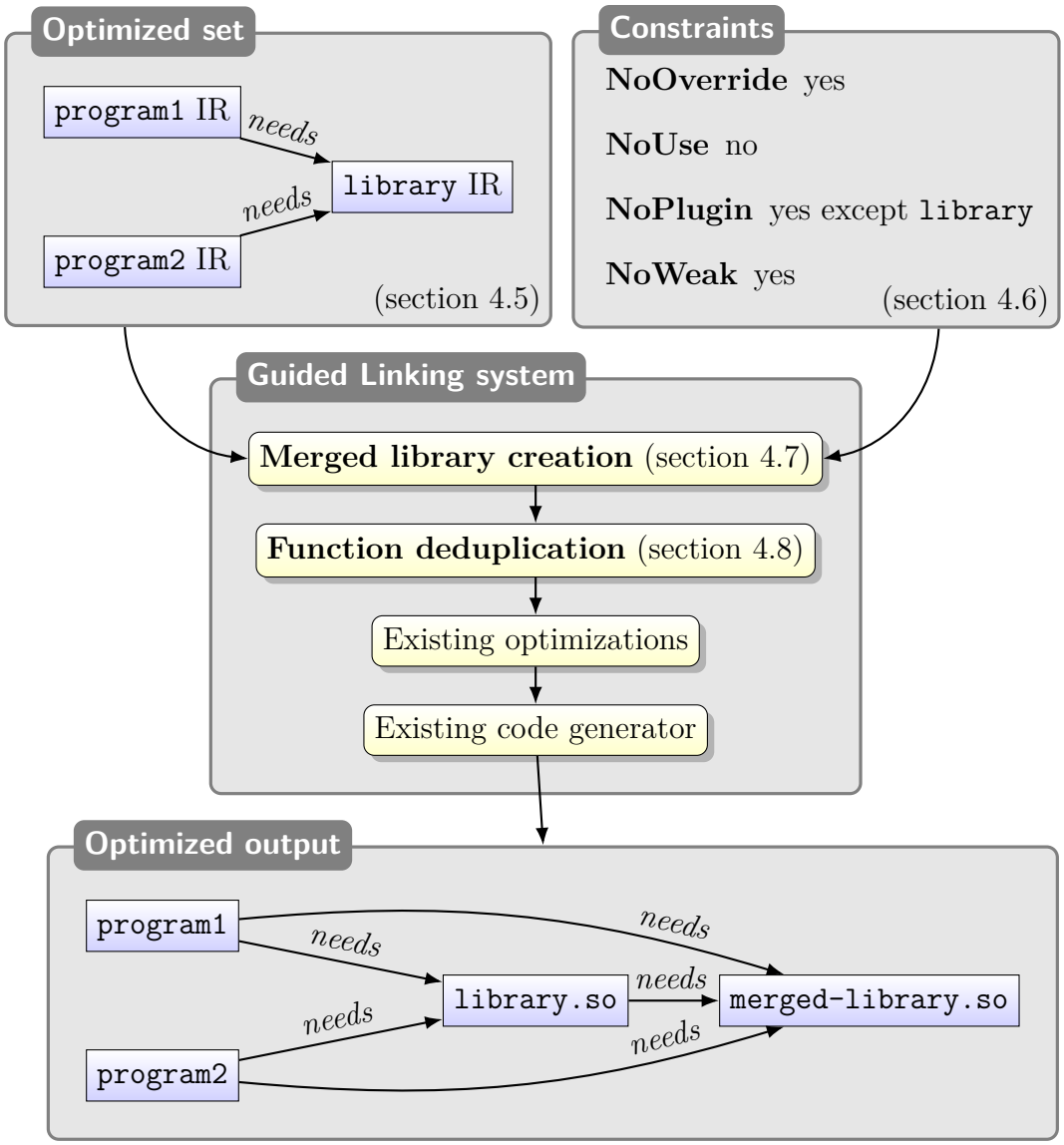


Figure 4.1: The Guided Linking process.

4.3 OVERVIEW OF GUIDED LINKING

This section gives a high-level overview of Guided Linking, before the full details of the system are described in sections 4.5 to 4.8. The full Guided Linking process is shown in figure 4.1. After the developer chooses an optimized set (section 4.5) and the constraints they wish to use (section 4.6), the Guided Linking tool creates the merged library (section 4.7) by merging all the code from the optimized set, and creates the wrapper programs and wrapper libraries. The tool also applies function deduplication (section 4.8) across all code added to the merged library. All the generated programs and libraries are then processed with existing compiler optimizations and code generation to produce optimized output code.

Table 4.1: The constraints developers can choose to apply to the runtime behavior of their code.

Constraint	Usually applied to…	Enables…
NoOverride (section 4.6.1)	Definitions that will not be overridden by another definition outside the optimized set.	Resolving references to these definitions statically.
NoUse (section 4.6.2)	Definitions that are never used outside the optimized set, and never loaded with `dlsym`.	Making definitions private and eliminating dead definitions.
NoPlugin (section 4.6.3)	Code that will never be loaded as a plugin or a dependency of one.	Moving function bodies to the merged library without extra indirection.
NoWeak (section 4.6.4)	Symbols that are never used with weak references and are never defined anywhere outside the optimized set.	Moving symbol definitions to the merged library for optimization.

4.3.1 Developer choices

Guided Linking is essentially a new form of link-time optimization (LTO) that optimizes multiple programs and dynamic libraries at once. The developer chooses an *optimized set* of programs, libraries, and plugins to optimize together. Larger sets will provide more optimization opportunities, but they will also reduce some of the flexibility provided by dynamic linking; specifically, when any part of the set is changed, an updated version of the entire set must be built, optimized, and distributed. Depending on their particular requirements, the developer may choose to optimize together a small set of libraries, an entire computer system, or any combination of components in between these extremes.

Like standard LTO, our approach requires all software in the optimized set to be compiled into the compiler's intermediate representation (IR), rather than directly into machine code. If the compiler already supports LTO, this may be as simple as adding a flag to the compiler command line. Most production compilers we know of support LTO today, including GCC, LLVM [48], IBM's XL family [81], and Intel C++ Compiler [82].

Aside from providing the IR for each program and library, the developer also chooses what *constraints* to place on the runtime behavior of the system. The exact set of possible constraints may vary depending on the dynamic linker in use. In this paper, we focus on systems that use the ELF format, such as Linux; other systems are generally simpler [83]. In ELF-based systems, the developer can choose from the possible constraints in table 4.1. In many cases a constraint can be applied across all symbols in the entire optimized set, so the developer need only provide an appropriate flag to our tool. In more advanced situations, the developer can choose to apply these constraints on a symbol-by-symbol basis. For instance,

they might specify that the NoUse constraint applies to all symbols but provide a list of exceptions for plugin entry points.

4.3.2 Our optimizations

Once we have the compiler IR for the optimized set and a list of constraints on dynamic linking behavior for each symbol, we can perform Guided Linking on the set. Our core technique is somewhat similar to automatic multicall (section 3.3): we move all the executable code in the set into a new library, called the *merged library*, which is optimized as a single unit using standard LTO.[1] However, unlike automatic multicall, we maintain full compatibility with the original dynamically linked programs and libraries. The original programs and libraries (including plugins) are left in place as *wrapper programs* and *wrapper libraries*, which export all their original symbols (if necessary), but redirect calls to the actual definitions in the merged library. By adding extra levels of indirection when necessary, we can ensure that the modified programs and libraries have the same runtime behavior as the originals, even when dynamic linking is used, subject to the developer-provided constraints.

Simply by combining everything into a merged library, we enable certain optimizations even without relying on any constraints. Specifically, we extend the function deduplication optimization given in section 3.4 so that it can correctly handle dynamically linked code.

When constraints are provided, we can do more kinds of optimizations on top of the deduplication. We need to handle each symbol differently depending on what constraints the developer applied to it. At one extreme, if no constraints are provided, we can't assume anything. Each reference to the symbol must go through its definition in the wrapper library, which can be overridden at run time just like it could in the original library. At the other extreme, if we can assume all the constraints listed above, we may be able to move the symbol definition into the merged library entirely and even inline it into its callers in other libraries. Table 4.1 gives an overview of the types of optimization enabled by each constraint, and section 4.6 gives the full details.

4.4 BACKGROUND

This section provides an overview of the dynamic linking process, as implemented on ELF-based systems such as Linux. We focus on the aspects of ELF linking behavior most relevant to Guided Linking; readers should consult Kell, Mulligan, and Sewell [83], Drepper [84], and

[1] Although the merged library may be very large, the use of virtual memory ensures that each program will only load the parts of the merged library it actually needs.

the *Linux Programmer's Manual* [85] for a more thorough exploration.

A dynamic library may be loaded at an arbitrary address in memory by the dynamic linker, depending on which parts of the virtual memory space are available. Whenever a dynamic library refers to a function or global variable defined in another library, or even in the same library, the address of the target can't be determined at compile time. Instead, the static linker creates relocation tables that the dynamic linker will fill with the actual addresses at run time.

When a program is run, the dynamic linker essentially performs three steps: (1) it finds the dynamic libraries that the program depends on and loads them into memory; (2) it relocates all the external symbols used by the program and each library; and (3) it calls initializers for the libraries. A "plugin" is simply a dynamic library that is explicitly loaded by the program, e.g., using `dlopen`; the dynamic linker performs these same three steps for the plugin when it is loaded. Each step is described in more detail below.

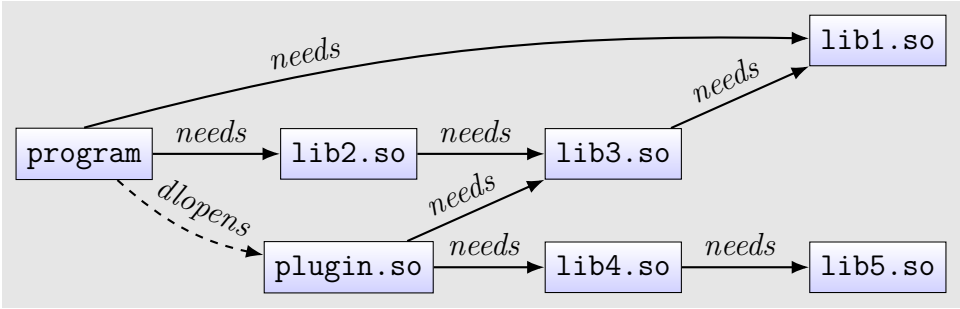
4.4.1 Producing ELF files to be dynamically linked

Much of the work involved in dynamic linking is actually performed in advance by the static linker, `ld`, and the compiler. The static linker must ensure that whenever a reference is made to a symbol in a dynamic library, the reference can be relocated by the dynamic linker so that the correct address is used at run time.

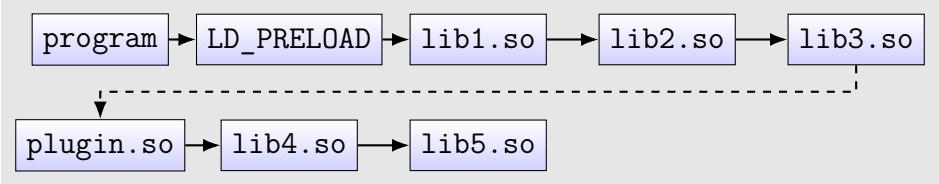
When code in a dynamic library refers to internal symbols in the same library, the compiler generates position-independent code (PIC). Rather than using the absolute address of the symbol, which would need to be relocated at run time, the compiler generates code that uses the *relative* address of the symbol, which can be added to the program counter to get the correct absolute address. PIC will work correctly no matter where in memory the library has been loaded, with no need for relocations.

When code in a dynamic library refers to global variables that may be defined in a different library, dynamic relocations must be used. It would be possible to modify the code itself to use the correct absolute address of the variable, but this would prevent code pages from being shared with other processes that use the same library. Instead, the static linker creates a global offset table (GOT) in the dynamic library with one entry for each external global variable it uses, and it creates a relocation table instructing the dynamic linker to fill in the GOT appropriately. Whenever the library accesses an external global variable, the correct address will be loaded from the GOT.

For dynamic references to functions, the address of the function is placed in the GOT just as for global variables. But when the function is called, an extra layer of indirection is used.



(a) An example program, a plugin it loads at run time, and the dynamic libraries they depend on.



(b) The order in which the dynamic objects will be loaded.

1. `program`
2. `LD_PRELOAD`
3. `lib1.so`
4. `lib2.so`
5. `lib3.so`

(c) The search order for symbols used in global scope: `program`, `lib1.so`, `lib2.so`, or `lib3.so`.

1–5. Same as global scope.
6. `plugin.so`
7. `lib3.so`
8. `lib4.so`
9. `lib1.so`
10. `lib5.so`

(d) The search order for symbols used in local scope: `plugin.so`, `lib4.so`, or `lib5.so`.

Figure 4.2: A dynamic linking example.

The caller actually calls a stub function in the procedure linkage table (PLT), which in turn loads the correct address from the GOT and jumps to it. The PLT allows the caller to use a regular call instruction, which is generally smaller than the code that would be needed to use the GOT directly. The PLT also enables lazy binding (see below).

The relocation table must be used not only for references to symbols that are defined in another library, but **also for references to symbols that are defined in the same library**, when those definitions are publicly exported. This allows a definition of that symbol in another library to *interpose*—that is, to override—a definition in the current library. The dynamic linker's symbol resolution behavior (section 4.4.3) ensures that only one exported definition of a given symbol will actually be in use. This is important for static member variables of C++ templates, for example, which should only have one active definition across the entire program and all its libraries.

4.4.2 Finding and loading dependencies

When a dynamically linked program is first run, the kernel uses its `PT_INTERP` header to determine which dynamic linker to use, loads both the program and the dynamic linker into memory, and starts the dynamic linker. After relocating its own code, the dynamic linker then needs to determine which libraries it should load. It starts by loading each library in the `LD_PRELOAD` environment variable and the `/etc/ld.so.preload` file; this feature can be used to override standard functions such as `malloc` with alternate versions.

Next, the linker must load dependency libraries. It maintains a list of all dynamic objects, starting with the program itself, in the order they were loaded. For each object in the list, the dynamic linker reads the `DT_NEEDED` sections in the object, which contain the names of the libraries that object depends on, and loads each library listed. If a library has already been loaded (as determined by the inode number of the library file), that entry is skipped. Any newly loaded libraries are appended to the end of the list of dynamic objects, and they will in turn have their dependencies loaded; in effect, the dynamic linker performs a breadth-first traversal on the dependency graph of dynamic objects.

When a plugin is loaded with `dlopen`, its dependencies are loaded using the same process. If the plugin depends on an already-loaded library, the library is not loaded twice.

Figure 4.2a shows an example program that depends on several libraries and also dynamically loads a plugin. Figure 4.2b shows the order in which the program and its libraries will be loaded by the dynamic linker. Note that `lib1.so` is loaded before `lib3.so`, even though the latter depends on the former, because the dynamic linker sees `lib1.so` first. Also note that the plugin does *not* cause `lib3.so` to be loaded twice.

`DT_NEEDED` sections often only provide the filename of a library, not the full path. In this case the dynamic linker searches the directories given in the `LD_LIBRARY_PATH` environment variable, the `DT_RUNPATH` section in the same object as the `DT_NEEDED` section, the `/etc/ld.so.cache` file, and finally default directories such as `/usr/lib`. When `dlopen` is called, the dynamic linker uses the `DT_RUNPATH` section from the object containing the call.

4.4.3 Relocating symbol references

After loading all required libraries, all external symbols used in any dynamic object must be resolved to the correct address. The dynamic linker reads the relocation table for each object and stores the resolved address of each symbol into the GOT. If no definition can be found, the linker normally aborts the program with an error; however, if the relocation is marked as a *weak use*, the linker instead uses the null address.

All global variable references are bound when the program first starts. Function calls can instead use *lazy binding*: they are not resolved until the first time the function is called. This is done with extra code in the PLT that calls the dynamic linker to resolve the symbol.

Given that multiple loaded libraries may have definitions of the same symbol, the dynamic linker must search the libraries in the correct order. When resolving references from the program and its dependencies, it searches in the *global scope*, which consists of the program and all its dependencies in the order they were initially loaded. The first definition found is used. For the example program in figure 4.2a, the global scope is shown in figure 4.2c.

When a plugin is loaded with `dlopen`, the behavior depends on flags given to `dlopen`. When the `RTLD_GLOBAL` flag is used, the plugin and its dependencies are added to the global scope, just as if they were dependencies of the main program. However, if the `RTLD_LOCAL` flag is used, which is the default, a new *local scope* is created that consists of the plugin and its dependencies in breadth-first order. References made by the plugin and any newly loaded libraries are first searched in the global scope, and then, if no definition was found, in the new local scope. References made by the original program and libraries are still searched in the global scope, as before, ignoring the plugin and new libraries. Dynamic lookups made by `dlsym` use the same scope as the module calling `dlsym`. For the example program in figure 4.2a, the full search order for a reference in the plugin is shown in figure 4.2d.

4.4.4 Calling initializers and constructors

Before the main program code can start executing, any global variables defined in the

dynamic libraries need to be initialized.[2] This may involve executing code in the library (such as C++ class constructors). The dynamic linker sorts all the dynamic libraries so that each library comes before any library that depends on it,[3] and then calls the initialization code for each library in turn. Similarly, when the program exits, the dynamic linker runs the finalization code in each library, in the reverse order of the initialization code.

4.5 SELECTING PROGRAMS AND LIBRARIES TO OPTIMIZE

The first step in applying Guided Linking is to choose the optimized set: the dynamic objects that should be optimized together. Our technique can optimize an arbitrary set of programs and libraries (including plugins), maintaining any necessary compatibility with external programs that load the optimized libraries. We also allow the optimized set of programs and libraries to link against external libraries which are not optimized. The only real limitation on the size of the optimized set is the fact that the entire set must be compiled and distributed as a single unit.

How should the developer decide which set of programs and libraries to optimize together? It would be possible to use only the programs and libraries within a single software package, in order to preserve compatibility with existing package managers, but this would limit the benefits of Guided Linking. The greatest potential for optimization is achieved when the full system, including *all* programs and libraries, is optimized as a single unit. This is possible in a closed system like a Docker container, where the complete set of software is known before the system is deployed.

On non-containerized systems, however, this is not possible—software can be added and upgraded at any time. These systems can still be fully optimized by using ahead-of-time compilation: instead of relocatable files, software is shipped in the form of IR modules, and Guided Linking is performed again whenever a module is added or removed. For embedded systems where the optimization process is too expensive to perform on the device itself, it can instead be done on a more powerful server before software is deployed to the device, as suggested in section 3.6.

Whenever an upgrade is made to one of the programs and libraries included in the optimized set, the entire Guided Linking process must be performed again. Depending on the size of the set and the compiler optimizations used, this could take several hours, which is undesirable for libraries that must be upgraded quickly (such as cryptography libraries).

[2]The dynamic linker does *not* initialize global variables in the main program; this is instead done by the C runtime library.

[3]If there is a dependency cycle, the order is undefined.

These libraries can be omitted from the optimized set, so they can be upgraded the normal way by simply replacing the library file with a new version.

Finally, it is necessary to omit libraries that are not available in IR form. This can happen because source code for the library is not available or because the library is incompatible with the compiler being used. These libraries must be left out of the optimized set.

4.6 CONSTRAINING THE DYNAMIC LINKER

In order to optimize dynamically linked code, we make use of constraints provided by the developer that bound the possible behavior of the dynamic linker. The list of all constraints is shown in Table 4.1. This section will describe the constraints in detail, and give some intuition about how we can use them for optimization. Details of optimizations enabled by these constraints will be given in section 4.7; in brief, we move as much code as possible into a new library called the merged library, and leave the original programs and libraries in place as wrappers that refer to code in the merged library.

By default, our tool assumes that no constraints are available. The developer can enable a constraint by providing a command-line option, such as `--no-override`, to indicate that the constraint applies to all symbols unless otherwise specified. If there are exceptions for which the constraint does not apply, the developer can list the exceptional symbol names and library names in a separate file, such as the one in listing 4.1, using wildcards to match multiple symbols with one exception. In most cases, developers will need at most a few exceptions per dynamic object.

If the developer accidentally provides constraints that are violated at runtime, there is a risk that the optimized software could exhibit subtle incorrect behavior that would be very difficult to debug. To prevent this from happening, we insert run-time *constraint checks* to be performed when a wrapper library is first loaded by the dynamic linker and whenever the `dlsym` function is called. These events rarely occur on a hot path, so the performance impact of the checks is minimal. If any of the constraints have been violated, the checks will detect the problem and abort the program with a suitable error message, which should allow the developer to identify and fix the incorrect constraint.

We will now explain each of the available constraints.

4.6.1 NoOverride: no external overrides

When a program refers to an external symbol, it's usually obvious what definition of the symbol the developer *intended* to be used. Given the example on the left of figure 4.3,

```
# Each module's PyInit_<modulename> function is loaded with dlsym().
[use]
fun:PyInit_*

# Python modules are loaded as plugins with dlopen().
[plugin]
lib:*/lib-dynload/*
lib:*/site-packages/*

# These libraries may be loaded as dependencies of Python modules.
[plugin]
lib:*libexpat.so*
lib:*libgdbm_compat.so*
lib:*libgdbm.so*
lib:*libpanelw.so*
lib:*libreadline.so*
lib:*libsqlite3.so*
```

Listing 4.1: The constraint exception list we use for Python.

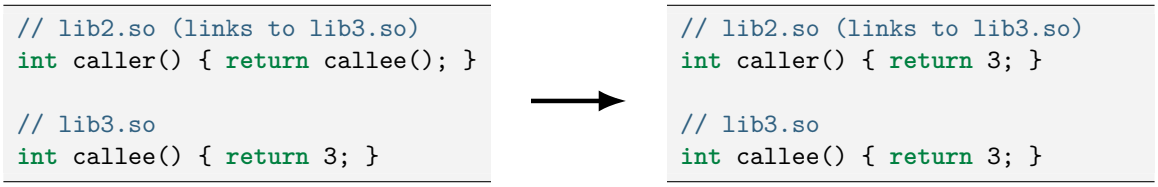


Figure 4.3: An optimization that is only valid when the NoOverride constraint is applied to `callee`.

the developer almost certainly wants `caller` to call the definition of `callee` in `lib3.so`. However, there are various ways the dynamic linker could provide a different definition of `callee`. The user could use `LD_PRELOAD` to load a library containing a different definition, or `LD_LIBRARY_PATH` could lead to a different version of `lib3.so` than expected. The program could also have its own definition of `callee` that interposes the one in `lib3.so` (see section 4.4.1). The compiler can't be sure which definition of `callee` will be used, so it must optimize `lib2.so` without making any assumptions about `callee`; for example, it can't inline `callee` into `caller`.

To allow for better optimization in the common case (when the developer can be confident that the most obvious definition is, in fact, the correct one), we introduce the *no external overrides* (NoOverride) constraint.

When to use it. This constraint, when applied to any exported definition, specifies that the definition will never be overridden at run time by another definition outside the optimized

set.[4] This is trivially true if there are *no* definitions outside the set, but it is also true if the definition inside the set overrides all others, or if the definitions are in different libraries that will never be loaded simultaneously. In the example in figure 4.3, if the developer applied this constraint to the definition of `callee`, it would guarantee that this is the *only* possible definition of `callee` that could be used by `caller`.

Enabled optimizations. When a piece of code refers to a symbol defined in a different program or library, and the symbol has this constraint applied, we can statically determine which definition is used. This allows us to perform arbitrary interprocedural optimizations across multiple programs and libraries, including inlining as shown in figure 4.3.

Constraint check. Whenever an optimized library is loaded, we check the list of NoOverride symbols and raise an error if the active definition of any of them is outside the optimized set. Without the check, if the constraint is violated, a library loaded with `LD_PRELOAD` that attempts to override the symbol would unexpectedly have no effect; or, if the symbol is a global variable, a program and library could end up using two different copies of it.

4.6.2 NoUse: no external uses

In standard practice, developers use tools like the `static` keyword and linker scripts to prevent libraries from exporting internal symbols. The compiler can then safely optimize away the definitions of these symbols. However, these tools only work for symbols that are internal to a single library; they do not work on exported symbols. We introduce the NoUse constraint, which applies to *the optimized set of programs and libraries as a whole*, allowing symbols to be marked as internal even if they are used in multiple places within the optimized set. If we optimize a program along with its libraries, we can often apply the NoUse constraint to *every symbol exported by the libraries*, which would not be possible when optimizing a single library on its own.

When to use it. For symbols that will never be used by external code outside the optimized set and will never be loaded with `dlsym` (even from inside the set). In particular, developers can safely apply this constraint globally whenever they know no external programs will link against the libraries in the optimized set, as long as they provide exceptions for any symbols that may be loaded with `dlsym`.

[4] Interposing definitions are still allowed, as long as they come from within the optimized set, allowing them to be detected at compile time.

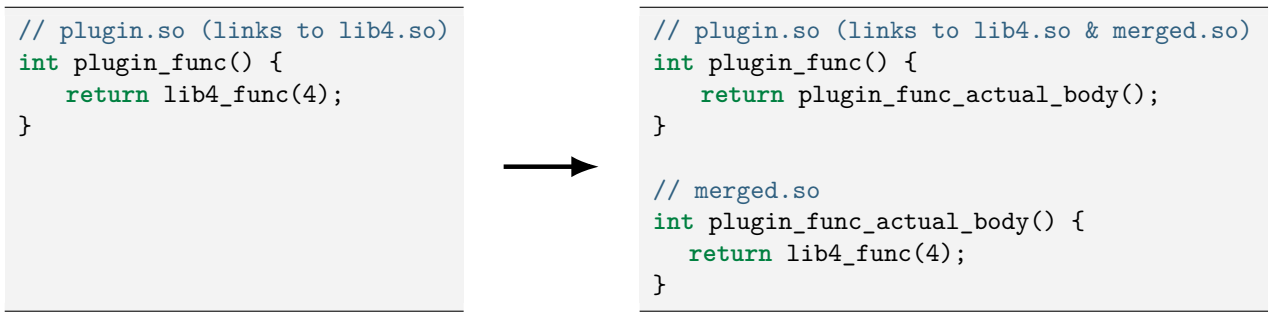


Figure 4.4: A transformation that is only valid when the NoPlugin constraint applies to the use of `lib4_func`.

Enabled optimizations. For symbols with the NoUse constraint, we can *internalize* the symbol: we make it *private to the merged library* and omit it from dynamic linking tables, reducing code size. If the symbol is never used, we can even save significant space by deleting it entirely; or, if it's a function only called in one location, we can inline it into its caller and then delete it.

Constraint check. There are two cases in which we need to perform runtime checks to prevent subtle incorrect behavior from occurring in rare situations. First, if an external program attempts to refer to an internalized symbol, it could unexpectedly resolve to a different definition of the symbol in a library outside the optimized set; we can prevent this by adding code that runs when the merged library is loaded, checks a list of internalized symbols, and raises an error if any of them has a definition in another library that might be used. Second, if any code uses `dlsym` on an internalized symbol, the call may return `NULL` and cause unexpected behavior in the code, which we can detect by hooking `dlsym` to check if the symbol being loaded had been internalized.

4.6.3 NoPlugin: no use in plugin

As described in section 4.4.3, plugins that are loaded with the `RTLD_LOCAL` flag (which is the default) add an extra complication to the symbol lookup process. When resolving a symbol used by the plugin library, or by a library that was loaded as a dependency of the plugin, the dynamic linker searches not only the global scope used by all the libraries but also a local scope specific to the plugin. This plugin-specific lookup scope makes it more difficult to move plugin code into the merged library.

As an example, consider the libraries on the left side of figure 4.4, supposing these are part of the larger set in figure 4.2a. When the original program is run and the dynamic linker

resolves the reference to `lib4_func`, it will search through the libraries shown in figure 4.2d, and eventually find the definition in `lib4.so`. However, if we attempted to move the body of `plugin_func` into `merged.so`, as shown on the right side of figure 4.4, the reference to `lib4_func` might be resolved differently. In particular, if `merged.so` was loaded earlier as a dependency of the program, it will use global scope; `lib4.so` may be missing from global scope, and only present in the local scope of `plugin.so`. In that case, the reference to `lib4_func` from `merged.so` will be undefined.

It's important to note that this problem only occurs when the merged library includes different pieces of code that need to use different scopes. If everything in the merged library should use the global scope—that is, if nothing in the merged library is part of a plugin or a plugin's dependency—the problem cannot occur. Conversely, if everything in the merged library should use the same local scope—that is, everything is part of the same plugin—the problem also cannot occur, presuming the merged library is also loaded as part of that local scope.

Even when these scope conflicts are possible, our system still relies on the ability to move all code into the merged library. We can do this by adding an extra layer of indirection, described in section 4.7.5. The NoPlugin constraint indicates that these conflicts are impossible, allowing us to avoid this extra indirection.

When to use it. Unlike other constraints, which apply to symbol definitions, the NoPlugin constraint applies to external references, such as the reference to `lib4_func` from `plugin.so` in figure 4.4. It is normally applied to every reference in a program or library. The NoPlugin constraint should be applied to all programs, and all libraries that will never be loaded as part of a plugin. In addition, if the entire optimized set belongs to a single plugin, the NoPlugin constraint can be applied to the entire set (despite the name). This constraint can also be applied to the whole set if it can be guaranteed that no two objects in the optimized set will be loaded simultaneously, as when merging multiple versions of the same library.

Enabled optimizations. This constraint allows us to move code to the merged library without adding a potentially expensive layer of indirection, as shown in figure 4.4. Moving the code is not an optimization in and of itself, but it's important to enable our other optimizations.

Constraint check. Whenever a wrapper library is loaded with this constraint applied, we check to ensure it's in the same scope as the merged library and raise an error if it is not. If the constraint were violated without this check, it would generally cause an "undefined

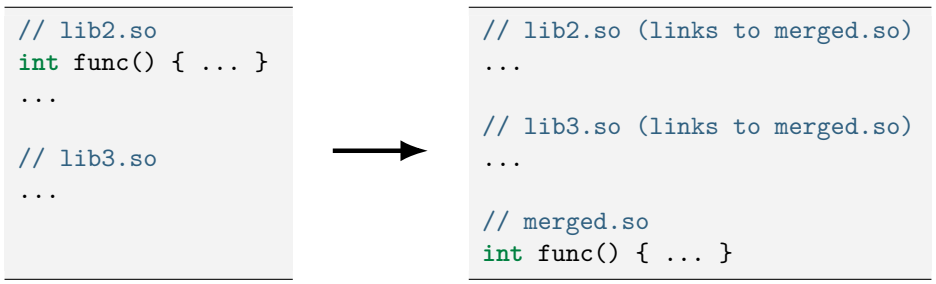


Figure 4.5: A transformation that is only valid when the NoWeak constraint is applied to `func`.

symbol" error as symbols were looked up in the wrong scope, but could potentially cause incorrect behavior if a symbol has multiple definitions and the wrong one is used.

4.6.4 NoWeak: no weak uses or external definitions

Suppose the optimized set includes the libraries on the left side of figure 4.5. Assuming none of the other programs and libraries being optimized exports a symbol named `func`, we might like to move the definition of `func` into `merged.so` for better optimization; because the new `lib2.so` links against `merged.so`, `func` will still be available in any program that uses `lib2.so`. However, this will create a *spurious export* of `func`—any program that links against `merged.so` (perhaps indirectly, via `lib3.so`) will see the definition of `func`, even if it never actually links against `lib2.so`.

In many situations, this is perfectly fine, even when there are external programs that might use the libraries we're optimizing. Suppose an external program `extprog` links against library `lib3.so`, which we have modified to link against `merged.so`. If `extprog` tried to use the symbol `func`, using the original `lib3.so`, it would crash, since no definition of `func` is available. Therefore, it's extremely unlikely that `extprog` actually uses `func`, so the spurious export is probably okay. However, there are two situations where we could see incorrect behavior:

1. Some code behaves differently depending on whether or not `func` is defined. For instance, it uses a weak declaration of `func` and expects the symbol to resolve to `NULL`, or it calls `dlsym("func")` and only behaves correctly if the result is `NULL`. We refer to these uses collectively as *weak uses*. The spurious export could cause `func` to be defined when it was previously undefined, breaking the code.

2. There is a definition of `func` in an external library or plugin, not included in the optimized set. If the external code is loaded by the dynamic linker after the merged

library, its definition of `func` could be interposed by the spurious export, causing the wrong `func` to be used.

The NoWeak constraint guarantees that neither of these situations can occur, and the spurious export is therefore safe. Note that the NoUse constraint implies that these problematic situations are impossible—no external uses implies no weak uses, and interposition of an external definition isn't a problem if the symbol is never externally used—so the NoUse constraint automatically implies the NoWeak constraint.

When to use it. This constraint, applicable to any symbol definition, specifies that (1) there are no weak uses of the symbol (that is, no uses that may only behave correctly if the symbol is undefined), and (2) the symbol is never defined outside the optimized set. Note the difference from the NoOverride constraint: that constraint is only violated by external definitions that would override a definition in the optimized set, but the NoWeak constraint is violated if there are any external definitions at all.

Enabled optimizations. This constraint allows us to move the definition of an exported symbol into another library (specifically, the merged library), as demonstrated in figure 4.5. This allows extra interprocedural optimizations to be applied when other programs and libraries refer to the symbol.

Constraint check. We need to check every external program and library that gets loaded and raise an error if any of them include a weak use or a definition of a NoWeak symbol. We also need a check in `dlsym` to determine if the result of the call would be different without the constraint. If the constraint were violated without these checks, code could use an incorrect definition of the symbol from an unexpected source, when it should have used `NULL` instead.

4.6.5 Special-purpose constraints

There are a few special constraints needed for specific symbols. These constraints are applied automatically by the Guided Linking tool, not manually by developers.

There is an "*unmovable*" constraint, which indicates that a symbol's definition must be left in the wrapper program or wrapper library. This constraint is automatically applied to each program's `main` function and the initialization and finalization code for each library.

There is also an "*always defined elsewhere*" constraint, which should be used for symbols weakly defined in static libraries that cannot be compiled to IR. We automatically apply

this constraint to all symbols defined in the static parts of the C standard library and C++ standard library. In other cases, developers should either use dynamic libraries or compile their static libraries into IR form, rather than linking in unoptimized static libraries.

There are also certain functions, specifically `dlopen` and `dlsym`, that behave differently depending on which dynamic library they are called from (sections 4.4.2 and 4.4.3). We add a special constraint to these functions so they are always called from the correct library.

4.7 OPTIMIZING DYNAMICALLY LINKED CODE

This section explains how our system actually optimizes a set of modules (programs and libraries), given the constraints provided by the developer. As noted in Section 4.3, we move as much code as possible into a new *merged library*. We replace each of the original programs and libraries with a *wrapper program* or *wrapper library* which contains a minimal set of symbol definitions, and links against the merged library to access the rest of the code.

Whenever possible, we replace dynamic references from one module to another, which would have been resolved by the dynamic linker, with *static* references within the merged library. Not only does this reduce the overhead of the dynamic linking itself, but it also allows us to perform optimizations *across dynamic linking boundaries*, simply by applying normal interprocedural optimizations on the merged library.

In order to move as much code as possible into the merged library, we separate each function into a *body function*, which contains the actual code, and a *stub function*, which simply jumps to the body function. All other references to the function use the stub function. An example is shown in figure 4.4. The body function can be moved into the merged library even when the public definition in the stub function cannot. This extra indirection could add overhead to the program, but the stubs can often be optimized away (section 4.7.4). The separation between body functions and stub functions is especially useful because the body functions can be deduplicated, as described in section 4.8.

The rest of this section explains in detail the process of creating the merged library and wrapper programs and libraries. Our implementation is based on the LLVM compiler, and we make reference to certain features of LLVM. We expect the same overall process to work for any compiler that supports LTO and dynamic linking, although the details may differ.

4.7.1 Resolving symbols

The first step is to attempt to resolve all dynamic references made in the input modules. We would like to resolve references statically, so that we can optimize them, but some

references must be left dynamic to ensure correctness. Specifically, we use the following algorithm for each reference:

1. If there is a definition in the same module as the use, and LLVM can determine based on the linkage information in the module that this definition is the exact one that will be used at runtime, we resolve to it. This happens particularly for references to private symbols, such as C functions defined with the `static` keyword.

2. If we can look through the dependencies of the module containing the use within the optimized set and determine which definition is used, and the NoOverride constraint applies to this definition, we resolve to it.[5]

3. Otherwise, the reference will be resolved at run time, by the dynamic linker, just as it would be in the original code.

4.7.2 Choosing where to define symbols

Once we have attempted to resolve each reference, the next step is to decide which stub functions should go in the merged library and which ones should be defined in the wrapper libraries. (All body functions go in the merged library.) There are several cases in which it's impossible to put a definition in the merged library:

1. The symbol is one of the few "unmovable" symbols, such as `main` (section 4.6.5). The stub function must be kept in the wrapper library.

2. The symbol is exported publicly by more than one of the modules being merged. We must keep the stub functions in their respective wrapper libraries and let the dynamic linker decide which definition to use.

3. The symbol is exported publicly, but one of the modules being merged has a reference to a symbol with the same name which could not be resolved statically. We must keep the stub function in the wrapper library and let the dynamic linker decide whether the reference should be resolved to our definition or to an external definition.

4. The symbol is exported publicly, the symbol may be used dynamically (section 4.6.2), and it may have weak uses or external definitions (section 4.6.4). That is, neither the

[5] Our current prototype implements this in simplified form, because it doesn't take the dependency graph into account. It performs static resolution when there is only a single public definition of the symbol among everything being merged. Otherwise, it uses dynamic resolution.

NoUse nor NoWeak constraints apply. We must keep the stub function in the wrapper library, because defining it in the merged library might break other programs that attempt to use it.

The above cases also apply to global variable definitions just as they apply to stub functions. In addition to the above cases where symbols must be kept in the wrapper library, there are several more complex cases that depend on additional attributes of the symbol:

5. When a global variable initializer uses a reference to a symbol that may need to be resolved in local scope (section 4.6.3), and the reference cannot be replaced with a static reference, we need to define the global variable in the wrapper library. For more details, see section 4.7.5.

6. Sometimes there are limitations of the system's compiler or linker that require two symbols to be placed in the same module.[6] In these cases, if one of the symbols is kept in the wrapper library, we must do the same with the other symbol.

After all the above constraints are considered, there will still be many cases where we can freely choose whether a definition goes in the merged library or a wrapper library. Our current implementation puts stub functions in the merged library whenever possible.

4.7.3 Merging and linking

After attempting to resolve references, and deciding which stub functions and global variable definitions should be moved to the merged library, it's time to actually create the merged library and wrapper libraries. Body functions are placed in the merged library. For functions that have their stub functions placed in the merged library, the body function can stay private to the merged library. For functions that should have their stub functions placed in a wrapper library, the body function must be exported from the merged library so that we can use it in the stub function. Similarly, global variables have their definitions placed in the merged library or wrapper library. Private functions in the merged library are renamed as necessary to prevent conflicts.

For references that were statically resolved, we update the reference to make sure it points to the corresponding definition. In certain cases we may move a private symbol to the

[6] On ELF-based systems, this happens when one symbol is an alias of another one; ELF files cannot include an alias to a symbol defined in another module. When using LLVM, this also occurs when a global variable initializer refers to a function using an LLVM `blockaddress` value, which is used for the "computed goto" extension to C/C++. Both the global variable and the function it refers to must be kept in the same module.

merged library but leave its callers/users in a wrapper library, or vice versa. In these cases we export the formerly-private symbol so that the other library can reference it, giving it a new name such as `__merged_func.0` to avoid conflicts.

At this point, the merged library contains code that came from various different programs and libraries, each of which may still have dynamic references to other symbols. In any given execution, only a subset of the programs and libraries may actually have their code used; for the others, there's no guarantee their dynamic references will actually have a definition available. We need to prevent these missing references from causing a dynamic linker error. Therefore, we make all the references *weak*, so if no definition is found, the reference is simply resolved to NULL rather than causing an error.[7] If necessary, we can check at run time whether any of these references would have caused an "undefined symbol" error in the original code, and raise a corresponding error ourselves.

Finally, for each symbol publicly exported by the merged library which has the NoUse constraint (section 4.6.2), and which is not used by any of the wrapper libraries, we make the symbol private.

The merged library, wrapper programs, and wrapper libraries can now be optimized, compiled, and executed using an existing LTO implementation. Because almost all code from the original dynamic modules has been moved to the merged library, the LTO implementation can optimize across the boundaries between the modules just as if they had originally belonged to a single library (albeit with the extra indirection we add depending on the constraints). In addition to standard LTO optimizations, we also apply a code deduplication technique described in section 4.8, which can cross these boundaries thanks to the merging process.

4.7.4 Additional optimizations

We mainly optimize the code after the merged library has been created, by applying a standard set of compiler optimizations. But there are certain optimization opportunities during the creation of the merged library that will not be available after the process is complete. Specifically, we may have a statically resolved call from a body function, which will be placed in the merged library, to a stub function that will be placed in a wrapper library. We might want to inline the stub function into its caller, but we can't do this on the final merged library because the caller and callee are in different dynamic libraries.

In order to apply this optimization, we initially include definitions of all functions in the

[7] As an exception, symbols with the "always defined elsewhere" constraint (section 4.6.5) are not made weak, to ensure that the external definition is always used.

merged library, including stub functions that should actually be defined in wrapper libraries. We run selected optimization passes on this merged library, including constant propagation and inlining of stub functions. Then we remove the extra definitions from the merged library, replacing them with references to the correct definitions in the wrapper libraries.

4.7.5 Handling local scope

We have a problem in cases where a body function without the NoPlugin constraint refers to an external symbol—the symbol may need to be resolved using local scope (sections 4.4.3 and 4.6.3). We put all body functions in the merged library, but the external symbol can only be resolved correctly if it is loaded from the wrapper library. We solve this problem by adding a layer of indirection to these references. We add a constructor to the wrapper library that runs when the library is loaded, taking the address of each such external symbol and storing it in a global variable defined in the merged library. Then, whenever a body function in the merged library needs to use one of these external symbols, we modify it to load the correct address from the new global variable.

This extra layer of indirection will certainly add overhead when libraries are loaded. There will also be overhead at run time because of the indirect symbol lookup. It works similarly to using a normal dynamically linked symbol, which also involves an indirect symbol lookup, but our implementation is less optimized than the dynamic linker's.

There is a similar problem with global variables whose initializers refer to external symbols. We handle these cases much more simply, by leaving the global variable in the wrapper library.

4.8 FUNCTION DEDUPLICATION

The Guided Linking system described above enables existing optimizations to be applied across dynamic-linking boundaries, but it can also enable *new* optimizations that were not previously feasible. As an example of such an optimization, we have implemented the cross-package function deduplication optimization from section 3.4, extended for full compatibility with dynamically linked code. The optimization can detect duplicate functions *anywhere* in the set of programs and libraries being optimized, even when they came from seemingly unrelated software packages, and deduplicate the functions when multiplexing so that only a single copy of the function is included in the final output. This optimization is primarily useful to reduce code size, but it can also increase speed by reducing cache misses.

Our function deduplication optimization is integrated into the optimization system described in section 4.7. Before we link a body function into the merged library, we check whether an equivalent function has already been linked in. The current implementation uses the simple syntactic equivalence check described in section 2.4, but a more sophisticated equivalence check could also be used. If an existing equivalent function is found, we reuse it rather than adding a new copy to the merged library. If many objects are added that consist mostly of duplicate functions, such as many slightly different versions of the same program, the merged library will grow only slowly because of this deduplication. Various additional checks are needed to make sure deduplication is a legal operation, as described in section 3.4. The checks explained in that section are extended to take into account the symbol resolution results from section 4.7.1, so that two otherwise-identical functions will not be merged if their symbols are resolved differently.

4.9 EVALUATION

In order to evaluate our proposed system, we have implemented a prototype based on the LLVM 10.0 compiler [48]. Our prototype implements the full optimization system described in section 4.7, along with the function deduplication technique described in section 4.8, with certain exceptions. First, the special behavior for `dlopen` and `dlsym` described in section 4.6.5 is not yet implemented, which could cause incorrect behavior in certain specific cases. Second, the constraint checks have not yet been implemented, so if an invalid constraint is specified, the optimized program may crash unexpectedly or exhibit incorrect behavior, as explained for each constraint in section 4.6. Note that the constraint checks would only be run when a new library is loaded or when `dlsym` is called, so we expect their impact on program speed to be very limited.

We obtained LLVM IR for the test programs and libraries by compiling them with the Clang compiler and using the `-fembed-bitcode` option. All file size evaluations were performed after applying the `strip --strip-unneeded` command to reduce file size. All performance evaluations were run on a machine with two 24-core Xeon Platinum 8259CL CPUs, NixOS Linux version 20.09pre239228.bd0e645f024, and pyperf version 2.0.0 [86]. In order to reduce variability, we disabled simultaneous multithreading and used the system settings recommended by pyperf.[8]

[8]We booted Linux with kernel options `nosmt isolcpus=4-47 rcu_nocbs=4-47` and ran `pyperf system tune` after boot to apply other recommendations. These settings ensure that each benchmark task runs on its own CPU core.

4.9.1 Optimizing Python in a closed system

Our system can achieve the maximum performance improvement in cases where the entire set of program and libraries on the machine are optimized together. This is possible on closed systems, such as Docker containers, which do not have new software installed after they are originally created. We evaluate the performance of our system in this case by optimizing Python, and assuming that no external programs will use any of the Python libraries.

We perform Guided Linking on the Python interpreter itself, its dynamically loadable plugins (including several external plugins used by our benchmarks), and several libraries the plugins depend on. Because we know there are no external programs left out of the optimized set, we can apply all four of our constraints to everything being optimized, with certain exceptions needed for plugins to work as given in listing 4.1.[9] We compiled the merged library using Clang with the `-O3` option to maximize performance; for our baseline, LTO is applied separately to each library and plugin with Clang `-O3`. We also use profile-guided optimization (PGO) for both our optimized version and the baseline;[10] this is important because Guided Linking introduces many new inlining possibilities, and PGO helps to determine which possibilities are profitable.

We evaluate our optimized version of Python on version 1.0.2 of `pyperformance`, the Python Performance Benchmark Suite [87], which was developed to test the performance of the Python interpreter.[11] Results are shown in figure 4.6. The geometric mean speedup across all benchmarks is 9.2% compared to the LTO+PGO baseline.

Our experiments also showed that the use of PGO makes Guided Linking particularly effective (not shown in the figure). Compared to LTO without PGO or Guided Linking, adding PGO provides a geomean speedup of 3.9%, and adding Guided Linking alone provides a speedup of 5.2%, but *the combination of both provides a speedup of* 13.4%!

Guided Linking achieves these improvements by enabling a variety of LLVM optimizations. Not only can library functions be inlined into a plugin or a program, but inlining can be performed within a single library that was impossible before. There are many different optimizations in play depending on the benchmark; we give details for a few specific examples:

[9] Note that the NoWeak constraint only affects the few symbols given in listing 4.1 under `[use]`. Removing NoWeak affects the speed by less than 0.5%.

[10] We use LLVM's IR-level profiling with the `-fprofile-generate` and `-fprofile-use` options, and generate profiles by running the same benchmarks we use for evaluation. The optimized build and the baseline use two separate profiles.

[11] We run each benchmark with the arguments `--no-locale --affinity <cpu_core> --processes=40 --values=20 --min-time=1`. These arguments increase the number of executions to reduce variability, and we ensure each benchmark uses its own isolated CPU core. We calculate speedups using the mean time per run.

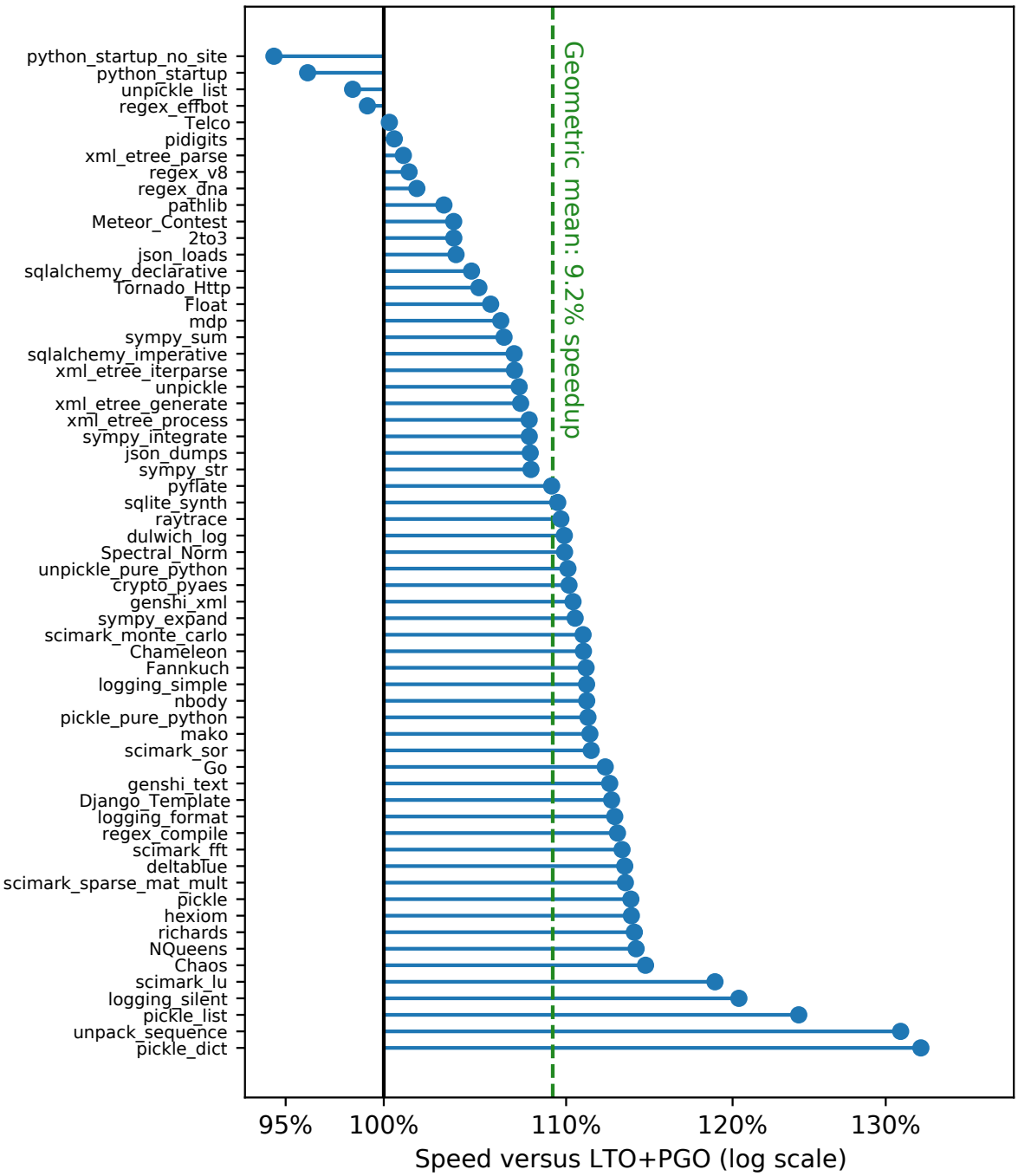


Figure 4.6: Speed of each of the 61 `pyperformance` benchmarks with our optimized build of Python, relative to standard LTO+PGO (higher is better).

- The main Python library defines several public functions that are only used once by the library, such as `_PyObject_GetMethod`. If LLVM inlined this code, it would normally have to leave the original functions in place in case another library calls them, creating duplicate code. Even with LTO and PGO enabled, LLVM's heuristics decide not to inline the functions because duplicating the code is too expensive. Guided Linking

determines that no other code will call these functions and makes them private to the merged library, enabling LLVM to inline them and delete the originals.

- The main Python library has a global variable `PyFrame_Type` containing a pointer to function `frame_dealloc`. With normal LTO, LLVM must conservatively assume that some other library might modify the variable. Guided Linking makes `PyFrame_Type` private to the merged library, so LLVM can determine that its value is never modified, allowing it to inline `frame_dealloc` into code that calls the function pointer.

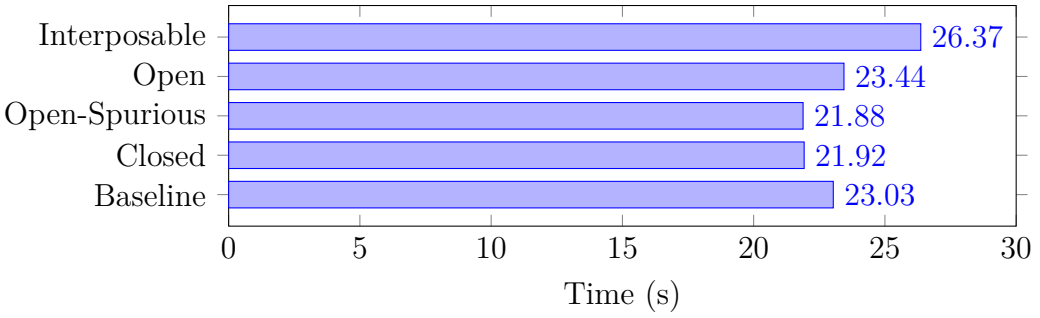
- Functions in one library can be inlined into functions in another, even if the second library is a dynamically loaded plugin. For example, in benchmark `pickle_dict`, most of the execution time is spent in the `_pickle.so` plugin's `save()` function, which makes many calls to the main Python library's `PyDict_Next()` function. Guided Linking moves both of these functions into the merged library and statically resolves the call, allowing LLVM to inline it.

It should be noted that 4 of the 61 benchmarks are slower after Guided Linking. In particular, the two `python_startup` benchmarks are 3.9 to 5.6% slower. Our optimized version of Python actually executes *fewer instructions* on these benchmarks, but incurs 2–3× as many cache misses and 15–18% more page faults. This happens because the Python code is spread throughout the merged library, which also includes code from other libraries. We suspect that performing profile-guided code layout would help reduce these cache misses.

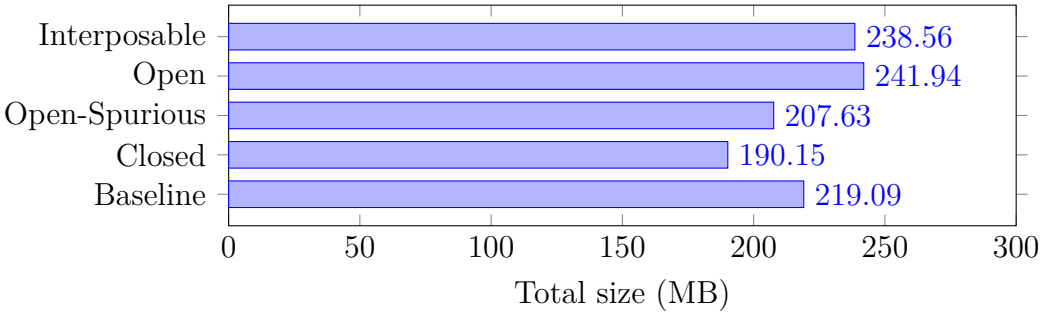
4.9.2 Optimizing Clang and LLVM

In order to evaluate the performance and code-size benefits of our optimizations when used with various combinations of constraints, we apply our system to the Clang 10 and LLVM 10 tools and libraries. We first build Clang and LLVM, together consisting of 94 programs and 227 dynamic libraries, in the form of compiler IR.[12] As a baseline, we compile each program and library to machine code separately, performing LTO with Clang and using the `-O3` option to maximize performance. For additional performance, we use profile-guided optimization just as we did with Python, generating profiles by running the same benchmark we used for evaluation. We then use Guided Linking to optimize all the programs and libraries together, using various sets of constraints, and compile the result to machine code the same way as

[12]LLVM can be built in several different configurations: using static linking for all libraries and tools; using dynamic linking with all library code placed in a single library; or using dynamic linking with code spread out across 149 different dynamic libraries, in addition to the Clang libraries. We use the latter configuration in order to demonstrate that our tool can handle a large set of libraries.



(a) Time to compile and link SQLite (lower is better).



(b) Total ELF file size (lower is better).

Figure 4.7: Evaluation results for PGO-optimized versions of Clang.

the baseline.

We apply the NoPlugin constraint to everything because Clang and LLVM do not normally use plugins. We evaluate several configurations for the other types of constraints:

Interposable NoPlugin. We provide unconditional compatibility with external programs and libraries, but the optimizations we can perform are very limited.

Open NoPlugin, NoOverride. The Clang and LLVM programs will work correctly. External programs will normally work correctly, but certain rare cases will fail, such as using an external library which interposes symbols defined by LLVM.

Open-Spurious NoPlugin, NoOverride, NoWeak. The Clang and LLVM programs will work correctly. External programs will normally work correctly, but certain rare cases will fail, such as using an external library which defines symbols also defined by LLVM.

Closed NoPlugin, NoOverride, NoWeak, NoUse. The Clang and LLVM programs will work correctly, but any external programs which link against the LLVM libraries will fail.

After building each version of the set, we evaluate its performance by running the optimized version of the `clang` compiler to compile and link the SQLite library with the `-O3`

option. The `clang` compiler program will invoke our optimized version of `llvm-ld` in order to link the library.[13]

Results are shown in figure 4.7. Compared with the baseline, our Closed configuration of Clang is 5.1% faster and 13.2% smaller. Our Open-Spurious configuration, which allows external programs to use the LLVM libraries, is 5.3% faster than the baseline and still 5.2% smaller, despite the fact that symbols cannot be made private. (While it may seem surprising that "open-spurious" is faster than "closed", the difference is very small and may be caused by variation in code layout.) PGO and Guided Linking do not combine as well as they did in the Python experiments; the Closed configuration without PGO is 5.5% faster than the baseline without PGO.

The largest portion of the size improvement comes from making symbols private, which allows unused symbols to be deleted. Our system achieves the speed improvement for the Closed and Open-Spurious configurations by performing optimizations across library boundaries, such as by inlining `BasicBlock::getTerminator()` into its callers in other libraries.

The "open" and "interposable" configurations are significantly slower and larger than the baseline. This is because we must keep symbol definitions in the wrapper libraries, but we put body functions in the merged library, and the frequent references back and forth between the wrapper libraries and the merged library add a lot of overhead.

4.9.3 Merging multiple versions of the same library

Our function-level deduplication technique can drastically reduce file size when we optimize together libraries that have large numbers of duplicate functions. This can be useful in practice when merging multiple versions of the same library, which are likely to contain large amounts of unchanged code. Certain libraries, such as `boost` and `protobuf`, often have multiple versions installed at once because different programs depend on different versions of the library; by merging all versions of the library together, we can significantly reduce disk space requirements. Note that merging multiple library versions using our approach is sound, *i.e.*, it does not affect the behavior of programs that link against one of the versions in the optimized set.

We evaluate these space savings on `boost` and `protobuf`, using Guided Linking to optimize together several different versions of each library. We enable the NoPlugin constraint; we

[13] We run each benchmark with the command `pyperf command --processes=80 --values=1 --warmups=1 --no-locale --affinity <cpu_core> clang -fPIC -shared -O3 sqlite-amalgamation.c -ldl -lpthread -o /dev/null`. These arguments execute clang 80 times to reduce variability, and we ensure each benchmark uses its own isolated CPU core. We report the mean time per run.

only expect one version of the library to be used in any given dynamic execution, so plugin scope conflicts are impossible. We do not enable any other constraints, because we want to ensure maximum compatibility with any possible users of the library.

Table 4.2 shows the results. We optimized the merged library with Clang's `-Oz` option to minimize code size. The baseline is the total size of all versions when each library is optimized separately at link time with Clang's `-Oz` option. We get a very significant 31% size reduction when merging 8 versions of `protobuf` together, and an even greater 57% size reduction when merging 11 versions of `boost`. Note that these reductions come from the ability to deduplicate identical functions between multiple versions of the library, as described in section 4.8.

4.10 FUTURE WORK

Ease of use. Our set of constraints is designed to be extremely general, allowing Guided Linking to get as many optimization benefits as possible in different situations. In its current form, Guided Linking can only be used by developers who are willing to spend time understanding each of the constraints and when they can be used for their application. Future work could design a tool that uses static analysis to determine how `dlopen` and `dlsym` are called, automatically identifying exceptions for the NoUse and NoPlugin constraints. Another tool could analyze symbols used by external libraries, including libraries that aren't available in bitcode form, to help generate exceptions for other constraints. And a user study could help determine which situations developers are most likely to encounter when applying Guided Linking, and write predefined lists of constraints with checklists developers can use to ensure the constraints are appropriate. With the checklists and automated tools, developers could use Guided Linking without needing to explicitly reason about the constraints at all.

Table 4.2: File size reduction when combining multiple versions of the same library.

		Total ELF size		
Library	Versions	LTO	Guided Linking	Size reduction
Boost (11 versions)	1.55.0, 1.59.0, 1.60.0, 1.65.1, 1.66.0, 1.67.0, 1.68.0, 1.69.0, 1.70.0, 1.71.0, 1.72.0	384.1 MB	164.0 MB	57%
protobuf (8 versions)	2.5.0, 3.1.0, 3.6.1.3, 3.7.1, 3.8.0, 3.9.2, 3.10.1, 3.11.3	85.5 MB	59.1 MB	31%

Security implications. As is, Guided Linking is likely to have an effect on the security of the optimized software. Incorrect constraints could cause incorrect behavior in the optimized software, including behavior that causes security problems. It will likely make ASLR less effective because all the code is moved to a merged library, which must have its address space randomized as a single unit. On the other hand, the number of return-oriented programming gadgets could either increase or decrease: it could increase because the entire merged library is loaded in memory, including code that would not have been present before, or it could decrease because Guided Linking enables deletion of dead code in libraries. Further research is needed to study these effects and find ways to mitigate them, if necessary.

On the other hand, future work could build on Guided Linking to introduce novel security improvements. When we optimize a program together with its libraries and plugins, we could add a run-time check to the program to ensure only the intended libraries and plugins can be loaded, preventing attacks that cause a malicious library to be loaded [88], [89]. Such attacks have led to high-severity vulnerabilities in a variety of popular software [90]–[96]. Another promising possibility would be the combination of Guided Linking with Control-Flow Integrity (CFI), which prevents large classes of attacks [97]. Guided Linking could allow CFI to work across dynamic-linking boundaries, protecting controw flow between different dynamically linked modules.

Smarter merging. Our current system moves *all* body functions into the merged library, and keeps *all* possible definitions in the merged library as well. Future work should investigate whether performance can be improved by selectively moving some body functions and definitions into the wrapper libraries.

Replacing the dynamic linker. Our current system relies on the standard dynamic linker to perform relocations and symbol resolution at run time. In some cases, such as with scope conflicts as described in section 4.7.5, we need to add extra layers of indirection in order to work with the dynamic linker. We also need to check each symbol ourselves to determine whether any constraints are violated. Future research should explore whether we can reduce this indirection by replacing the dynamic linker, either in whole or in part; special care will be needed to maintain compatibility with external programs and libraries.

Equivalent function detection. Our current system uses a very simple syntactic comparison to detect duplicate functions. It would be straightforward to incorporate more sophisticated function equivalence detection techniques, such as the one by Churchill, Padon, Sharma, *et al.* [24]. If these techniques were added to our tool, the rest of the system would continue to work as-is but with improved deduplication.

Inter-process communication. When we use Guided Linking to optimize together two programs that communicate with each other, it may be possible to optimize the communication code, for example by automatically simplifying the communication protocol. This would be an exciting new direction for compiler optimization research, opening up possibilities that have never to our knowledge been explored before.

4.11 CONCLUSION

We have introduced a new technique, *Guided Linking*, for optimizing dynamically linked software. Our technique maintains all needed functionality of the software, as specified by the developer using constraints on what the software can dynamically link against at run time. We merge most of the dynamically linked code into a single library, statically resolving dynamic references when possible, which enables existing optimizations to be applied across dynamic linking boundaries. We also perform a new optimization that deduplicates identical functions, even if the duplicate functions originally came from separate programs or software packages.

We developed a prototype of Guided Linking on LLVM 10 and evaluated it in several different scenarios on real-world Linux software. Our tool can improve performance by 9% or more with a realistic set of constraints. When optimizing the size of a set of different versions of the same library, our tool can reduce size by more than 50%. No previous software tool we know of is able to achieve such large reductions in code size fully transparently, and with no impact on the functionality of a system.

Chapter 5: Semantic Outlining

This ongoing work is a collaboration with Nader Bushehri, Om Bhatt, Yuyou Fan, Prof. Vikram Adve, and Prof. John Regehr.

5.1 INTRODUCTION

5.1.1 Motivation

In this chapter, we introduce an enhanced version of *outlining*, an automated, compiler-based approach to reduce binary code bloat caused by redundant code. Outlining works by identifying code sequences that have multiple redundant copies in the program. For each such sequence, it replaces all the copies with calls to a single, newly created function containing the sequence. When the sequences are large, or when they have many copies, outlining can significantly reduce code size, without changing the behavior of the program. Unlike code compression, a program optimized with outlining can be executed directly, without needing to be decompressed, so outlining reduces not only the program's size on disk but also its size in RAM and even in the instruction cache. These size reductions are important for the reasons discussed in sections 1.1 and 6.3.

In addition to the size improvement, outlining can sometimes increase speed by reducing cache pressure and I/O overhead, especially in UI-intensive software with few hot loops [17]. In other cases, it reduces speed because of the newly added call instructions. Speed reductions can be minimized by using profile information to avoid outlining hot code.

Outlining a given sequence can be counterproductive if the space overhead of the new function and new call instructions outweighs the benefits of removing redundant copies. To be effective, an outliner must estimate or measure the effect of outlining each sequence, and use only the profitable ones.

The novelty of our technique, *Semantic Outlining*, is that it takes advantage of general semantic equivalence of different code sequences, even when they use very different sequences of operations. This is a much broader concept of equivalence than used in any previous work on outlining; many other outliners are limited to identical contiguous sequences, and the most general previous work searched for subgraphs of the program IR that use identical operations [98], [99], while our work can use equivalence between code that uses completely different operations.

There are two benefits of using general semantic equivalence instead of a narrower kind of equivalence. First, we may find larger sets of redundant sequences, allowing us to replace

more sequences with a single new function. Second, when we create a new function to replace multiple equivalent sequences, we can fill it with the smallest of all the equivalent sequences found, which may be smaller than the alternatives.

5.1.2 Our approach

Our Semantic Outlining algorithm works in several stages, several of which include novel improvements. Figure 5.1 shows an overview of the entire process.

Unlike existing outliners that use identical sequences or subgraphs, we cannot simply search the input program for repeated code, because we want to identify semantically equivalent code even when it has no syntactic similarities. Instead, we must generate a list of candidate instruction sequences that might be worth outlining, so we can perform equivalence checking on them. We first analyze each function to produce an **Outlining Dependence Graph**, indicating which instructions can be outlined together depending on where the call to the new function will be inserted. We then use two different methods to generate candidate sequences. We allow sequences to include noncontiguous instructions and to span multiple basic blocks. The full candidate generation process is described in section 5.3.

After we generate and combine the candidate lists for all functions, the next step is to cluster semantically equivalent candidates together. We use the Alive2 equivalence checking tool [100] to check pairs of candidates for equivalence. In order to reduce the number of pairs checked, we use a new idea we call **counterexample-guided equivalence clustering**, similar to existing ideas such as counterexample-guided abstraction refinement. When Alive2 determines that a pair of candidates is not equivalent, we obtain a counterexample, which is essentially a test input for the candidates that causes them to produce different results. We can use this counterexample as a test input to all other candidates that might be equivalent to the ones just checked; when two candidates produce different results, we know they cannot be equivalent, even without applying Alive2 to them. The full clustering process involves additional phases, and is described in section 5.4.

Our final step is to determine which candidates should actually be outlined and perform the outlining transformation on the input program. When two candidates overlap, we cannot outline both of them; we must choose which one, if any, to outline. This process is described in section 5.5.

5.1.3 Contributions

Our major contributions include:

- We introduce a novel concept, the Outlining Dependence Graph, from which we easily determine which subgraphs may be outlined (section 5.3.2).
- We present a new method for counterexample-guided equivalence clustering (section 5.4.3), which is essential to make Semantic Outlining scale to nontrivial programs.
- We implement our proposed system for several architectures, using the LLVM compiler and Alive2 equivalence checker.

5.2 RELATED WORK

5.2.1 Outlining

Many variations of outlining have been developed. The earliest discussion seems to be from 1972, where it was called "converting expressions to subroutines" and applied to relatively small functions [34]; another early discussion from 1976 called it "procedural abstraction" [101]. Other works have called it "code factoring" or "procedure extraction", often with slight variations in definition. Outlining features have been advertised in commercial compilers for embedded systems [102], [103]. Outlining can be applied to source code [104], compiler ASTs or IR (as in LLVM's IROutliner), machine code being generated by the compiler (as in LLVM's MachineOutliner), or machine code after compilation [105], [106]. When there are large repeated sequences that contain small repeated sequences, outlining can be applied repeatedly to outline both large and small sequences [17].

The simplest candidates to outline are contiguous sequences of identical instructions that do not span multiple basic blocks. Such sequences can be found efficiently using suffix trees [107], with modifications to abstract away trivial differences such as register assignments [108]. Another simple option is to consider candidates that consist of multiple whole basic blocks [106]. When two sequences use different operand values but are otherwise identical, they can be outlined into one function with extra parameters that provide the operand value [105]. Even when some instructions are different, two sequences can be outlined into one function by making the function conditionally execute different instructions depending on the call site [109]–[111].

LLVM includes two outliners. The MachineOutliner is applied to machine code after register assignment, and it uses suffix trees to find identical pieces of code and replace them with call instructions. The MachineOutliner pass is enabled by default for certain architectures including x86-64, ARM, and AArch64. The IROutliner pass is applied to IR code in the middle of the optimization pipeline, and it uses suffix trees to find and outline

code sequences that are identical, except that they can have different constant operand values as mentioned above. The IROutliner does not support sequences that span multiple basic blocks, and it uses simple heuristics to estimate how large the machine code will be after the IR is compiled, which may lead it to make bad decisions. The IROutliner is not enabled by default in any configuration.

Use of dependence graphs. In order to outline noncontiguous sequences of instructions, dependence analysis must be used to ensure the program's behavior is unchanged. Komondoor and Horwitz [112] presented an analysis to determine whether an arbitrary set of instructions can be extracted into a new function. They generally follow the same constraints we use in section 5.3.1, but they do not require the outlining point to dominate the instructions to be outlined. Their approach to control dependencies is very different from ours, although with similar results. Their handling of data dependencies is complicated because they do not use single static assignment form, and their analysis, unlike our outlining dependence graph, does not lend itself to incremental generation of candidates.

Johnson [98] presents an alternative analysis based on a novel graph-based compiler IR. He does not analyze data dependencies, so store and call instructions cannot be outlined. Otherwise, arbitrary matching subgraphs of the IR can be outlined. Dreweke, Wörlein, Fischer, *et al.* [99] generate data flow graphs and search for isomorphic subgraphs using a variation of the gSpan graph mining algorithm; the subgraphs can then be outlined.

Hardware support for outlining. Some researchers have proposed code compression techniques very similar to outlining, but with the aid of extra hardware instructions. The "call-dictionary" instruction [113], [114] acts like a call instruction, but with an extra argument that specifies the number of instructions to execute before returning. This removes the need to explicitly store return instructions, but more importantly it also allows outlined code sequences to partially overlap with each other even if they need to return at different points. Bitmask echo instructions [111] add a mask to specify which subset of instructions should be executed, allowing similar sequences to be merged even if certain instructions are different. The authors reduce code size by 15% with no average change in performance. Echo instructions have been extended with support for nested echo instructions, control flow, calls, and other features [115]–[118].

Clone refactoring. When applied at the source code level, outlining is usually referred to as "clone refactoring" and is generally used for software maintenance purposes, not to reduce compiled code size; see [104] for a recent survey. Most tools focus on Type 1 and Type

2 clones, which are identical aside from whitespace, comments, and (for Type 2) variable and type renaming. Type 3 clones contain syntactically similar code, mixed with additional non-matching statements. Such clones can be detected by finding isomorphic partial slices of program dependence graphs [119], and refactored by taking their dependencies into account [112], which may involve duplicating code [110], [120] or adding parameters to the outlined function [121]. No significant clone refactoring tools have been applied to Type 4 clones, i.e., semantically equivalent clones that are not syntactically similar [104, section 6.1]. One group refactored Type 4 clones they detected by comparing pieces of source code after normalization, but their paper provides insufficient detail to evaluate their work [122].

Other applications of outlining. There is a different form of outlining that doesn't involve deduplicating code, in which the goal is to separate cold code from hot code within the same function, Pettis and Hansen [9] apply this form, which they call "procedure splitting," in the linker as part of a code positioning strategy to improve instruction cache locality. They move hot basic blocks in each function to the start of the function, then move the other basic blocks to a different code segment.

Zhao and Amaral [10], [11] describe an outliner designed to increase code speed that works at the AST level. They apply outlining at a very early stage in the optimization pipeline; in order to prevent outlining from interfering with alias analysis when the address of a local variable would be passed into an outlined function, they add a temporary variable used just for the call. They show that their formulation of the optimal outlining problem, which assumes a fixed upper bound on the number of calls that may be added, is NP-hard. Their outlining technique was intended to enable more hot code to be inlined into its callers, but this actually occurred very rarely; still, they achieve a median speed improvement of 1% and maximum of 6% by improving locality and making other optimizations more effective.

Most work in this area uses profile-guided optimization to determine which code to outline; Mosberger, Peterson, and O'Malley [123] note that this can cause slowdowns if the profiles are misleading, and recommend manually marking the code that should be outlined, especially error handling code.

Outlining has also been used in decompilers, to make code more readable by reversing the effects of inlining [124], [125].

Related transformations. Function merging is an alternative way to factor out repeated code sequences. It works in cases where two functions are largely similar to each other, unlike outlining, which can be used on small similar sequences in functions that are otherwise completely different. In the situations where function merging can be used, it may be more

effective than outlining because no new call instructions need to be added. Highly flexible tools for function merging have been developed [126]–[128].

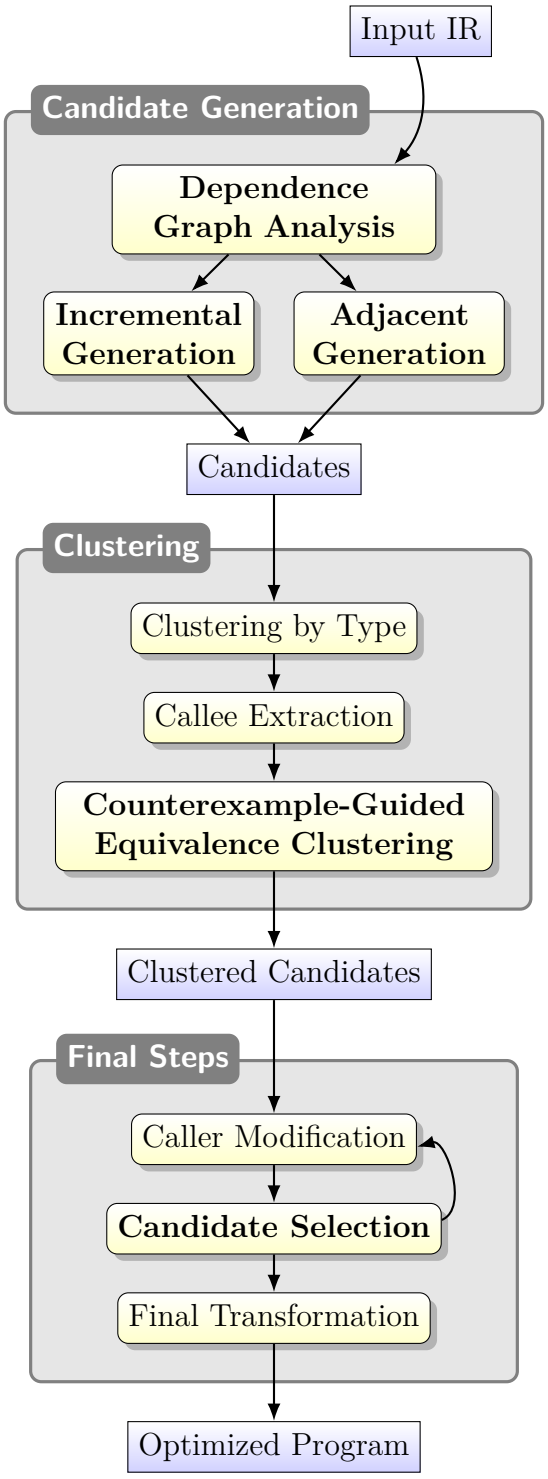


Figure 5.1: The Semantic Outlining process.

Some work has created new virtual machine instructions based on common instruction sequences, which can be seen as a form of outlining [129].

Outlining is conceptually related to other transformations which undo normal compiler optimizations, such as strength promotion, software de-pipelining, unprediction and unspeculation, and so on. These techniques are usually used in decompilers [130], [131], but they have also been used to simplify verification [132]. Loop rerolling in particular is closely related to outlining, as it also requires finding and simplifying repeated code; but rerolling is more constrained as the repeated code must occur within a single function and it must be structured in such a way that there is an equivalent loop. In the limited situations where rerolling is possible, outlining will also be possible, but rerolling will give a better result because it simplifies the loop and doesn't create any new functions. Rerolling has been implemented as part of a software pipeliner [133] and a general pattern recognition tool [134]. Stitt and Vahid [130] find rerolling opportunities by using suffix trees to find repeated substrings in a function. Hu, Su, and Wang [131] instead use the function's data dependence graph and identify isomorphic subgraphs, each of which is one iteration of the original loop. In order to reroll subgraphs which are not exactly isomorphic, they perform software de-pipelining, undo peephole optimizations, and merge index registers.

5.2.2 Semantic clone detection

Finding semantically equivalent subsequences in large code bases is not straightforward. The most extensive body of work on this problem can be found in the software engineering literature, where semantically equivalent but syntactically different sequences are referred to as *semantic clones*, *type 4 clones*, or *simions* [135]. For many years, the most general clone detection method available was to generate program dependence graphs (PDGs) and search for isomorphic subgraphs, but this method is difficult to scale and requires the basic structure of the code to be identical [119], [135], [136]. Machine learning has been used for clone detection with features extracted from PDGs [137].

Jiang and Su [138] introduced the first scalable system for finding arbitrary semantically equivalent code sequences in large programs. They extract all possible consecutive sequences of statements, execute each sequence on a series of random input values, and consider sequences to be clones when they produce the same outputs given the same inputs. In general only a small fraction of possible inputs can be tested, so they produce many false positives (28% in one experiment); this would be unacceptable for Semantic Outlining, where any false positive could cause inequivalent sequences to be merged, resulting in incorrect behavior. They can detect clones even when the inputs and outputs have a different ordering or

different types; they ignore side effects of calls and treat return values from calls as additional inputs. They find that only 42% of semantic clones are also syntactic clones, and only 64% of syntactic clones are also semantic clones. A variation reduces the number of code sequences considered by using k-nearest neighbors to partition the control flow graph [139].

Kim, Jung, Kim, *et al.* [140] use a path-sensitive static analysis to compute the abstract memory state after executing each code sequence. Two sequences that produce similar abstract memory states are reported as semantic clones. Loops that execute for more than a few iterations may prevent them from completing the analysis and finding clones.

Krutz, Malachowsky, and Shihab [141], [142] use concolic execution on code sequences, producing a series of path conditions and symbolic values representing the behavior of the code along different control flow paths. They calculate the Levenshtein distance between the concolic execution results for each pair of sequences, reporting a clone if the distance is below a threshold.

5.2.3 Equivalence checking

The simplest method for equivalence checking is to simply check whether the programs are syntactically identical (aside from variable names). Equivalence can be checked modulo inconsequential instruction reorderings by building program dependence graphs, which incorporate both control dependencies and data dependencies, and checking whether the graphs for the two programs are isomorphic; if they are, the programs are equivalent [143]. Programs can be normalized using transformation rules or standard optimizations before equivalence checking, in the hope that they will more likely be equivalent [144, sec. 5.2]. However, these methods only work if the programs are structurally identical after normalization.

For Semantic Outlining, we want to use *general, black-box* equivalence checking, which does not rely on a structural comparison between pieces of code, so that we can take advantage of equivalent sequences even when there is no syntactic similarity. General black-box equivalence checking is in contrast to the original formulation of translation validation [35], in which one of the input programs was derived from the other using a known set of transformations, and semantic differencing [145], in which one of the input programs is a patched version of the other.

Equality saturation [146]–[148] applies a known set of equality-preserving transformations to a program in order to find equivalent programs. Although there may be an exponential number of possible combinations, it is often possible to efficiently calculate and store every possibility using a data structure called the E-PEG; overhead can be reduced further by normalizing the E-PEG [149]. If two programs are equivalent after any sequence

of the known transformations, equality saturation will produce identical E-PEGs for them. This technique can prove equivalence even when certain loop transformations and interprocedural transformations are involved, but is still limited to the known set of implemented transformations.

For loop-free code, the most powerful method for checking equivalence is to convert the code into satisfiability modulo theory (SMT) formulas and ask an SMT solver to prove whether or not the formulas can ever produce different results. Even when the relationship between two code sequences is obscure, they can still be proven equivalent in this way [150], [151]. An SMT solver on its own is insufficient to handle loops and very large pieces of code. Instead, the equivalence checker can identify *cutpoints* each consisting of a control flow point in each program, construct a *simulation relation* that specifies how the states of the two programs relate to each other at each cutpoint, and use an SMT solver to prove that the simulation relation holds and implies equivalence [23], [152], [153]. The simulation relation may be guessed by examining the behavior of the programs when run with automatically generated test inputs [23], [152], or based on counterexamples provided by the SMT solver [154]. By allowing a flexible enough correspondence between control flow in one program and in the other, this idea can be extended to prove equivalence even when optimizations like loop unrolling and loop inversion are used [22]; however, this work scales poorly to unrolled or vectorized code and cannot handle complex control flow relationships [155, sec. 2.4].

Equivalence checking can be reduced to the problem of verifying properties of a *product program* that executes the code from both programs [24], [155], [156]. If the programs are aligned sufficiently well (for instance, if each program has a loop which is merged into a single loop in the product program), standard verification techniques can easily identify invariants and prove equivalence. The most powerful method in existing work [24], [155] works by examining program behavior on test inputs, guessing a weak invariant that indicates which states in one program correspond to the states in another, creating a corresponding alignment and product program, and attempting to prove equivalence. Unlike other work, this method can prove equivalence between vectorized and nonvectorized code involving complex control flow relationships; however, it still cannot handle certain transformations such as loop splitting [24, sec. 5.6].

5.3 CANDIDATE GENERATION

Outlining involves extracting a subset of an existing function's instructions and moving them into a new function, then modifying the original function by deleting that subset of instructions and inserting a call to the new function. We call the new function the *outlined*

callee and the modified original function the *modified caller*. We call the selected subsequence of the original function's instructions the *outlined sequence*; these instructions need not be contiguous, and may come from multiple basic blocks. We use the term *outlining point* to refer to the place in the original function where the new call instruction will be added.

Because we want to use a fully general semantic equivalence checker to find matching candidates, we cannot find candidates using syntax-based algorithms (such as those based on suffix trees). Instead, we need to separately process each candidate we want to check. Section 5.3.1 describes how we determine which candidates are valid for potential outlining, and section 5.3.3 describes how we actually generate the list of candidates.

5.3.1 Constraining the outlining problem

One could imagine an outlining process that allowed arbitrary new instructions to be added when necessary to outline similar sequences of code. However, such a process cannot be readily implemented and could easily lead to an intractable search space. To tame this problem, we first introduce some constraints on which subsequences of instructions are possible to outline, regardless of whether they have duplicates or whether doing so would be profitable. We allow an outlined sequence to contain noncontiguous instruction sequences and arbitrary control flow, including loops. However, we set several restrictions on the outlining process:

- No instructions may be duplicated (though ϕ-nodes may be split; see section 5.3.2).
- No new instructions may be added, except those essential to make the new call and handle its arguments and return values.
- The outlined program may not speculatively execute instructions that would not have been executed in the original program.
- The outlining point must dominate all the instructions to be outlined. This restriction simplifies the analysis that determines which sequences can be outlined.

Under the restrictions above, outlining acts much like a reordering transformation that moves everything in the outlined sequence to the outlining point. Such reordering transformations are valid as long as they preserve all data and control dependencies in the program [157]. These restrictions allow us to construct a data structure, the Outlining Dependence Graph, that we can use to determine which outlining sequences are valid and estimate their potential size savings. Some of these restrictions could be relaxed in future work, as discussed in section 5.7.1.

5.3.2 The Outlining Dependence Graph

As the first step in our outlining process, an Outlining Dependence Graph (ODG) is calculated for each function we will consider optimizing.

Nodes and edges. A node in the ODG is either a real IR instruction or a virtual node. Virtual nodes are used to represent some constraints that involve the outlining point; they correspond to a location in the program and have corresponding dominance relations. For example, a virtual node at the head of a basic block is considered to dominate all other instructions in the block.

Edges in the ODG represent the constraints needed to ensure that all dependencies are preserved. Some edges directly correspond to a data or control dependency, while some are more indirect. There are two types of edges:

- A *normal edge* from B to A means that in order for B to be outlined, either A must also be outlined or A must dominate the outlining point.[1]
- A *forcing edge* from B to A means that in order for B to be outlined, A must also be outlined.

If a subset of the ODG's nodes meets all the constraints given by forcing and normal edges, and it includes a single node that dominates all the others, it corresponds to a valid outlinable subsequence, which we refer to as a *candidate*. The ODG's nodes and edges are determined based on data and control dependencies, described below.

Data dependencies. Suppose instruction B has a data dependency on instruction A. We can ignore read-after-read dependencies, but must consider all other types of dependencies.

If neither or both of the instructions are outlined, the data dependency is preserved in the modified caller or the outlined callee, respectively. If A is outlined but not B, the dependency is again preserved: because the outlining point must dominate A, the new execution of A will still complete before B, preserving the dependency. In the case where A writes a register and B reads it, the outlined callee will need to return the value of A.

The interesting case, necessitating constraints on outlining, is when B is outlined but not A. To preserve the dependency, we must ensure that A is executed before the outlining point is reached. This can be accomplished with a normal edge from B to A in the ODG; note that this edge is in the opposite direction of the edge in the classical data dependence

[1] If A does not dominate B, a normal edge is equivalent to a forcing edge.

graph [157]. In the case where A writes a register and B reads it, the outlined callee will need an argument containing the value of A.

In the example below, there will be a normal edge from `%y` to `%x` due to the read-after-write dependency; this allows a candidate consisting of `%y` and `%loop_done`, but not one consisting of `%y` and the call. As another example, there will be a normal edge from the store instruction to the load due to the write-after-read dependency.

```
loop:
  call void @foo()
  %x = load i32, i32* %ptr
  %y = sub i32 %x, 1
  store i32 %y, i32* %ptr
  %loop_done = icmp eq i32 %y, 0
  br i1 %loop_done, label %exit, label %loop
```

Listing 5.1: An example with data dependencies.

However, in cases where A does not dominate B, such as the load's dependency on the store, representing the dependency with a single normal edge would be too strict. It would only allow B to be outlined without A if A dominated the outlining point, which dominated B, which is impossible because dominance is transitive and A does not dominate B. In fact, outlining B without A is safe as long as all paths from A to B go through the outlining point, ensuring that the dependency is preserved. Let X be the node in the iterated dominance frontier of A that dominates B; there must be a unique such node by the same reasoning that the SSA construction algorithm [158] will have a unique insertion point for a ϕ node defining A that will reach B.[2] If the outlining point is anywhere between X and B, B can be outlined without A.

To represent this dependency in the ODG, we create a virtual node just before the position of X, give B a normal edge to the virtual node, and give the virtual node a forcing edge to A. These edges allow B to be outlined without A, but only when the outlining point is between X and B.

In the example the load instruction has a read-after-write dependency on the store, but the store does not dominate the load. We will create a virtual node X at the head of the `loop` block, give the load a normal edge to X, and give X a forcing edge to the store. This allows the load to be outlined without the store, as long as the outlining point is somewhere in the `loop` block before the load.

[2]In particular, when the dependency is a read-after-write, converting the program to Single Static Assignment (SSA) form would cause a ϕ-node for the definition at A to be inserted at X, which must dominate B. Moreover, the same reasoning also applies to write-after-read and write-after-write dependencies.

Control dependencies. Suppose instruction B has a control dependency on instruction A. We consider not only explicit control dependencies, where A is a conditional branch, but also the implicit control dependencies created when A can throw an exception or exit the program.[3] If neither or both instructions are outlined, the dependency is preserved just as before.

For explicit control dependencies, we cannot outline A without B.[4] Therefore, we create a forcing edge from A to B. In the example below, there are forcing edges from the conditional branch to the call and `%l`, ensuring that the branch can only be outlined if the full `loop` block is also outlined. For implicit dependencies, no such edge is necessary; if A throws or exits it will have the same effect on B even if it is moved into the outlined callee.

```
entry:
  %e = ...
  br label %loop
loop:
  %a = ...
  call void @foo()
  %l = ...
  br i1 %loop_done, label %exit, label %loop
exit:
  %x = ...
```

Listing 5.2: An example with control dependencies.

In order to outline B without A, the outlining point must have the same control dependency on A that B does, whether the dependence is explicit or implicit. This is equivalent to ensuring that A is executed before the outlining point is reached, and all paths from A to B go through the outlining point. This is the same requirement as for data dependencies, so the same solution applies.

For simplicity, we don't perform this analysis for every pair of instructions. Instead, we create a virtual node at the head of each basic block, give each instruction in the block a normal edge to the virtual node, and give the virtual node additional edges to other nodes based on the control dependencies. The effects are the same as if we had handled each instruction's control dependencies separately. Note that we still need separate normal edges for control dependencies within a single block (because of instructions that can throw or exit).

[3] In IR generated from a C++ program with exceptions enabled, *all* call instructions may throw or exit, except when the compiler can specifically prove otherwise.

[4] There are ways to preserve the dependency by adding a new branch instruction in the modified caller that determines whether to execute B. However, the new instruction would violate the no-duplication or no-new-instructions constraints in section 5.3.1.

In the example, we create virtual nodes at the head of each basic block. We give `%l` a normal edge to the virtual node for `loop`, which in turn has a normal edge to the conditional branch. It is then possible to outline a candidate consisting of `%l` and the call, but not one consisting of `%l`, the call, and `%e`. We also give `%l` a normal edge to the call, preventing `%l` and `%a` from being outlined without the call, which would cause `%l` to be executed spuriously if the call throws or exits the program. Together, these edges ensure that all candidates involving `%l` are correct.

There is an additional constraint on the outlining point: if an instruction B is outlined, the outlining point must not have any additional control dependencies that B does not have. For each virtual node created at a block head, we add a normal edge to a new virtual node at the nearest ancestor in the dominator tree that meets this requirement. In the example, we give the virtual node for `exit` a normal edge to a new virtual node inserted before the unconditional branch. Node `%x` has a normal edge to the virtual node for `exit`, and in combination the edges prevent outlining a candidate that consists of `%l` and `%x`, which would incorrectly cause `%x` to be evaluated in every loop iteration.

The rest of this section describes how we deal with several special cases, such as ϕ-nodes and local memory allocations.

ϕ-nodes. If the program is already in SSA form, data dependencies are easier to handle, but the ϕ-nodes need their own special handling because of the way they interact with control flow. We give each ϕ-node normal edges to not only its input values, but also the terminating branch instruction of each basic block referred to by the ϕ-node. This ensures that if we outline the ϕ-node, we also either outline the control flow needed to determine the value of the ϕ-node, or we can handle the ϕ-node by splitting as described in the next paragraph.

In two situations, we may need additional special handling for phi nodes. The first situation is when the outlining point is at the beginning of a basic block and a phi node in the same block has predecessors whose terminators are not outlined. Second, when a phi node *not* in the outlined sequence has predecessor blocks whose terminators *are* outlined. In either case, the phi node cannot be computed correctly in either the caller or the callee. We must split these phi nodes into two, producing one phi node in the caller and one in the callee, in order to evaluate them correctly. In the first case, the value of the phi in the caller must be passed as an argument to the phi in the callee; in the second, the value of the phi in the callee must be returned to the phi in the caller. Note that in some cases, one of the split phi nodes may have only a single operand; these phi nodes are unnecessary, so we omit them and use the operand directly.

Stack allocations. For variables allocated in stack memory (whether implicitly or with the `alloca` function), it's only possible to outline the allocation if we can identify all instructions that can potentially use it. The allocation has a forcing edge to each of these instructions.

Return instructions. We prevent outlining of return instructions; they could only be outlined by adding new code to check whether a return instruction was reached or not, and we don't allow adding new code.

Other cases. Depending on the compiler IR being worked with, there may be extra constraints. For example, an instruction that inspects the stack frame, such as `va_start`, may be impossible to outline.

5.3.3 Generating candidates

The ODG limits which candidates are valid, but in a function with n instructions and few dependencies, there may still be $O(2^n)$ valid sequences for outlining. It is clearly impractical to generate every candidate for large functions. Instead, we choose candidates using a combination of two methods.

Incremental growth. Suppose we have a valid candidate, and we want to add new instructions to its outlined sequence in order to make a new valid candidate. We want to add as few instructions as possible in a single step, and we want to limit ourselves to instructions we know to have a control flow or data flow interaction with the instructions already in the candidate; specifically, we consider instructions to which at least one instruction already in the outlined sequence has a normal edge in the ODG.

Given these considerations, there is a unique minimal set of instructions we can add to generate a new valid candidate. Let S be the set of nodes our current candidate has normal edges to. If S has at least two elements, let A and B be two arbitrary nodes from S. Then we can iteratively remove nodes from S until we are able to generate a new candidate:

- If S is empty, we stop using this candidate as it cannot be extended further.
- If S has a single element, we add it to the candidate, and this process is done.
- If A dominates B, we cannot add A to the candidate without adding B, because that would violate the normal edge to B, so we can remove A from S and only consider adding B.

- Conversely, if B dominates A, we remove B from S.
- If neither A nor B dominates the other, we cannot add either alone because then there would be no valid place to put the outlining point; instead we examine the dominance tree to find the nearest common ancestor C of both A and B, and replace A and B with C in S.

The new candidate may violate some ODG edges, either because there are forcing edges or the new node was a node C as generated above and nodes A and B still need to be added. We iteratively add further nodes needed to satisfy edges until there are no more violations. Then the procedure is finished.

So the "Incremental Growth" method starts by generating a candidate for each instruction in the function, consisting of that instruction alone. Then each candidate is extended by adding as few nodes as possible to make a new candidate, as described above. If only virtual nodes were added, we extend the candidate repeatedly using the same process until at least one real instruction is added. This process continues to generate new candidates from existing ones until there are no more instructions that can be added to them.

For a function with n instructions, this process generates n initial candidates, and for each initial candidate can generate up to $n - 1$ additional candidates (because each additional candidate adds at least one more instruction, and there are $n - 1$ additional instructions that may be added). Therefore, this process generates $O(n^2)$ candidates.

Note that when there are forcing edges, they may cause this process to generate duplicate candidates. We detect the duplicates and eliminate them.

Adjacent instructions. Because the incremental generation technique focuses on dependencies, it may miss groups of related instructions that are still useful for outlining. For example, consider code that initializes all the fields of a large structure with different values. There may be no dependencies between the instructions that initialize different fields, and yet all these instructions are closely related and may be worth outlining together if the code is duplicated elsewhere in the program.

Therefore, in addition to the candidates already generated, we also generate candidates consisting of 2 or more adjacent instructions within a single basic block. We skip candidates with a forcing edge that would require a non-adjacent instruction to be added.

Implementation limits. To prevent performance pathologies, we set the following default limits on the candidates that can be generated; these values can be tuned separately for different target applications, if necessary.

- Maximum number of instructions in a candidate generated using incremental growth: 200.
- Maximum number of instructions in a candidate generated using adjacent instructions: 10.
- Maximum combined number of arguments and return values needed for the outlined callee: 10.

These limits reduce the total number of candidates to a maximum of $209n$: $200n$ for incremental growth and $9n$ for adjacent instructions.

5.3.4 Filtering with code size estimates

When generating a candidate, we can estimate the code size savings that outlining that candidate would achieve. If the size of the new call instruction is larger or equal to the estimated size of the outlined callee, it is likely impossible to profitably outline that candidate regardless of how many duplicates it has. We filter out such candidates when generating them and do not process them further.

To determine how profitable it would be to outline a given candidate, we need to estimate its effect on final program size. For the purposes of this approximation, we ignore the potential effects of outlining on instruction selection and register pressure. Outlined callees will generally pass their arguments and return values in registers, not on the stack, so we ignore the possibility that new loads and stores will be needed. There are three components of this effect:

- OutlinedSetSize: the total compiled size, in bytes, of the instructions in the outlined set.
- CallSize: the compiled size, in bytes, of the newly inserted call instruction. We use a per-architecture constant estimate.
- FunctionOverhead: the additional size, in bytes, needed to create the outlined callee function. We use per-architecture constant estimates that depend on whether or not we are outlining a call instruction. On the architectures we studied, functions with call instructions generally need an additional entry in the exception handling frame table (the `.eh_frame` section in ELF files), and on some architectures they also need additional instructions to manage the frame pointer.

Table 5.1: Size estimate values, in bytes, for various architectures.

	ARM	THUMB	AArch64	RISC-V	x86-64	WebAssembly
Call instruction (per caller)	4	4	4	8	5	6
Return instruction (per callee)	4	2	4	4	1	1
Frame pointer management instructions (per callee that includes call instructions)	8	4	8	16	0	0
Exception handling frame (per callee that includes call instructions)	8	8	16	16	16	0
Function alignment (per callee)	–	–	–	–	16	–
Additional overhead (per callee)	0	0	0	0	0	3

Table 5.1 shows the estimates we use for CallSize and FunctionOverhead. We can't get a good estimate of OutlinedSetSize directly from the IR instructions; there are too many interactions between different instructions during compilation. Instead of using a complicated cost model, we simply compile the original function into machine code, measure the size of each machine instruction, and assign the size of those bytes to the corresponding IR instruction. We do this once per original function, in parallel with calculating the ODG. In order to determine which IR instructions correspond to which machine instructions, we assign a different debugging line number to each IR instruction before compilation, and read the debugging information associated with each machine instruction afterwards.

This method of estimating instruction size can sometimes produce strange results. For example, if two IR instructions are compiled into a single machine instruction, we will only assign a size to one of the IR instructions. Later steps in our outlining process will calculate the actual size of the outlined callee and perform more accurate filtering.

Clearly, the cost size estimation would be easier and more accurate if we were to outline code at the machine IR level instead of the mid-level LLVM IR. However, doing so would make equivalence checks less effective. Structural equivalence would be affected by incidental back-end transformations such as register allocation or instruction reordering. Even more importantly, by working at the IR level, we can support a large number of target architectures using a single semantic equivalence checker designed for an architecture-independent IR; if we worked on machine instructions, we would require a lot more effort to construct formal semantics for large and complex machine ISAs, which would need to be repeated for every single target architecture (there are almost two dozen supported by LLVM alone, and far more in GCC).

Given these three estimates, we can compute two new values:

- F = FunctionOverhead + OutlinedSetSize. This is the size that will be added if any duplicate of the candidate is outlined.
- S = OutlinedSetSize − CallSize. This is the size that will be saved for each duplicate of the candidate outlined, including the first.

Some candidates will have a negative or zero value of S. This means that they will never be profitable according to our estimate, no matter how many duplicates they have, because replacing the original instructions with a call instruction will not make the modified caller any smaller. We eliminate these candidates during the candidate generation step.

The F and S values depend only on the target architecture, the total size of the instructions to be outlined, and whether or not the candidate includes a call instruction. They can easily be calculated incrementally as candidates are being generated, making it very efficient to filter out unprofitable candidates.

5.4 SEMANTIC CLUSTERING

At this stage, for each original function, we have generated candidates in the form of subsets of the outlining dependence graph. We also have estimated values of F and S for each candidate. We pool together the candidates from every function into a single set; we want to find and outline equivalent code that occurs anywhere in the program, regardless of which functions it originated in. The next step is to cluster the candidates so that each one is grouped with other candidates that have a semantically equivalent callee; if we outline multiple candidates from the same group, we only need to add a single callee to the optimized program, reducing code size.

We currently require candidates to be fully semantically equivalent, without modifications, for them to be outlined together; see section 5.7 for discussion of other possibilities. We will start with large clusters and progressively refine them until only candidates with equivalent callees are grouped together.

5.4.1 Clustering by type

Two callees cannot be equivalent unless their types and global references are identical,[5] so our initial clusters are based on the type and globals.

[5]Technically, two callees with different global references can be equivalent if some of the references occur in dead code. We assume that dead code has already been removed by the compiler, so this cannot occur.

We can determine a callee's type and global references without actually generating its code: argument types and return types are based on data dependencies between the outlined set and the rest of the original function, while the global references consist of all globals used by the outlined instructions. We reorder the argument list and return value list based on type, to ensure that otherwise-equivalent callees are not clustered separately simply because of argument ordering.[6] We also replace all pointer types with pointer-to-void.

At this point, we use our size estimates to calculate the best-case size savings for each cluster, assuming every candidate in the cluster has the same callee. If the savings are zero or negative, we filter out the cluster. This filter only affects a small fraction of candidates, because many candidates share the same type.

5.4.2 Callee extraction

In order to analyze the candidates further, we need to generate the actual callee functions. (Callers are not modified until later on.) This is mostly a straightforward process, based on copying the outlined sequence into a new function. When the outlined sequence uses values that come from outside the sequence, we replace them with new arguments; when values calculated in the outlined sequence will be needed by code outside the sequence, we add them to a return value.[7] When possible, we mark the callee to ensure it is optimized for size and choose a calling convention that uses registers for as many arguments and return values as possible, in order to reduce the time and space overhead of the call.

After generating the callee functions, we compile them individually into machine code, and measure their actual code size. We use the actual size to replace our size estimate F from this point onward.

5.4.3 Counterexample-guided equivalence clustering

We now have multiple clusters of candidates, with an extracted callee function for each candidate. We need to refine the clusters into smaller ones, so that each cluster consists of candidates that have semantically equivalent callees. We can easily check for callees that are syntactically identical with each other, and merge the corresponding candidates. Checking pairs of syntactically different callees is more difficult. We can use an equivalence checker, such as the Alive2 translation validator,[8] on each pair. The simplest option is to check each

[6] A limitation of our method is that this only works on arguments of different types.

[7] The compiler must allow functions to return multiple values (perhaps using a structure type).

[8] Alive2 actually checks *refinement*, not equivalence. A function A refines to a function B if the permissible behaviors of B are a subset of those of A; in particular, if A has undefined behavior on some inputs, B's

pair of callees in the same cluster, but clusters can contain millions of candidates and we would need $O(n^2)$ checks, making this option prohibitively expensive.

We use a new technique we call **counterexample-guided equivalence clustering** to drastically reduce the number of checks needed. This technique is similar to other counterexample-guided techniques, such as counterexample-guided abstraction refinement. When processing a cluster, we first apply the equivalence checker to an arbitrary pair of callees in the cluster. If the equivalence checker proves they are equivalent, we merge them and continue with another pair. If the equivalence checker proves they are inequivalent, and provides a counterexample, we use the counterexample as a test input that will cause the two functions to produce different results. We then use an interpreter to evaluate every callee in the cluster on the new test input, and subdivide the cluster based on the results. We can then restart the process using the new, smaller, clusters.

```
candidate_A:
  %result = add i32 %arg0, %arg0
candidate_B:
  %result = add i32 %arg0, 1
candidate_C:
  %result = shl i32 %arg0, 1
```

Listing 5.3: An example cluster.

For example, suppose the cluster initially contains the three candidates above. If we apply the equivalence checker to candidates A and B first, it will report that they are inequivalent, and provide a counterexample such as `arg0=2` on which they have different behavior. On this counterexample, candidate A returns 4 while candidate B returns 3. When we interpret candidate C on the same input, we get a result of 4. So we create a new cluster containing candidates A and C, which still need to be checked for equivalence, and another new cluster containing only candidate B.

Ideally, each test input will help improve the clusters effectively. Even if we only divide the original cluster into 2 equally sized clusters at each step, we reduce the number of equivalence checks to $O(n)$ and the number of interpreter invocations to $O(n \log n)$. However, there's no guarantee that we will get equally sized clusters; we could also get one cluster of size 1 and another of size $n - 1$.[9] In order to prevent such cases from taking an absurdly long time, we set limits after which we will give up on the cluster. We may also get inconclusive

behavior on those inputs is irrelevant. For simplicity, we only use equivalent pairs for outlining, which are those pairs that refine in both directions.

[9]We can even potentially get a single cluster of size n, due to limitations of our interpreter. In particular, undefined values in LLVM have an exponential number of possible values at run time, and our interpreter does not consider all possible values.

results from Alive2 (such as when it times out on a large function); if we repeatedly get inconclusive results when checking the same callee, we give up on that callee and remove it from the cluster.

Note that callees are never considered equivalent based on interpreter results alone; in order to end up in the same cluster after the process is finished, they must either be syntactically identical or proven equivalent with the equivalence checker.

5.5 FINAL STEPS

5.5.1 Selecting candidates to outline

At this point, we have a list of outlining candidates for the entire program, estimate code size S for each modified caller function, measured code size F for each callee function, and a clustering of candidates so that all candidates in the same cluster have semantically equivalent callees. We want to maximize the profitability of the set of candidates we choose to outline. However, we have one constraint: candidates that overlap (outlining the same instructions from the same function) cannot both be outlined.[10]

For each cluster of identical callees, we choose a single callee from the cluster with the smallest compiled size. We can choose any set of candidates to outline, letting $X_m = 1$ if candidate m should be outlined, as long as none of them overlap. We get a code size benefit of S_m for each candidate m outlined; however, for every cluster containing one or more candidates chosen for outlining, we must pay a code size penalty of S_n for the callee n that must be included in the optimized program. Let $Y_n = 1$ if callee n must be included.

This problem can be expressed as an integer linear programming (ILP) problem as follows. Let C be the set of all candidates (m, n), where m is the candidate number and n is the callee chosen for the cluster. (Each m will appear only once in C, but each n may appear multiple times when there are equivalent callees.) Let M_i be the set of candidates that outline instruction i. Then the problem is:

$$\begin{aligned} \text{maximize} \quad & \sum S_m X_m - F_n Y_n \\ \text{subject to} \quad & X_m \leq Y_n \text{ for each } (m,n) \in C \\ & \sum_{m \in M_i} X_m \leq 1 \text{ for each } i \\ \text{and} \quad & X_m, Y_n \in \{0,1\} \end{aligned} \tag{5.1}$$

[10] Future work could relax this restriction and allow candidates to overlap when one is a subset of another. Or, we could apply the whole outlining process multiple times, as in [17].

This problem is NP-complete: if all $F_n = 0$ and $S_m = 1$, it simplifies into the Maximum Set Packing problem. In practice, we get good (albeit suboptimal) results with a greedy algorithm that starts with an all-zero solution and repeatedly chooses a candidate m and sets $X_m = 1$ following the process described in the next paragraph, greedily choosing the m that will maximize the savings for the next solution.

When adding a candidate m with corresponding callee n, we must check for overlapping candidates m' and set $X_{m'} = 0$ for each one; this process can cause some values in Y to be reset to 0. Then, if $Y_n = 0$, we set it to 1 and loop over all candidates m' that use callee n, setting $X_{m'} = 1$ for each one that does not overlap with an already-selected candidate. This can cause multiple values in X to be set to 1 in a single step.

In this greedy algorithm, instead of using the estimated value of S_m for each candidate, we modify the caller for each candidate that could be added, compile it, and use its actual measured size. We update the measured size as new candidates are added, ensuring that it includes both the candidates already chosen and the new candidate we are considering. This ensures that our outlining decisions take into account unpredictable, nonlinear effects of outlining multiple candidates from the same function. In this way, our candidate selector combines both the flexibility of working with compiler IR and the code size accuracy of working with machine code.

5.5.2 Final transformation

We have already generated the extracted callees and modified callers, so all we need to do now is add the extracted callees to the program and replace the original functions with the appropriate modified callers.

5.6 EVALUATION

5.6.1 Implementation

We implemented our technique using LLVM and the Alive2 equivalence checker. We use an Alive2-based interpreter we built that supports Alive2's counterexamples, including values specific to the LLVM semantics such as poison.[11]

[11]There are still some aspects of LLVM semantics that the interpreter doesn't fully implement. In particular, undefined values (which represent many possible values) may be handled inaccurately. If the interpreter produces incorrect results in these cases, we may fail to recognize equivalent candidates (if the interpreter incorrectly returns different results) or perform unnecessary equivalence checks (if the interpreter incorrectly returns equal results), but our outliner will not produce incorrect output.

We find data dependencies using LLVM's MemorySSA analysis, which calculates SSA form for memory loads and stores as if the contents of memory were a single register. We use MemorySSA because it is reasonably efficient to compute, and we can use the phi nodes it creates to easily determine where to place virtual nodes. MemorySSA has the limitation that is does not fully distinguish between different memory objects, so our data dependencies are imprecise (but still sound); this is not an inherent limitation of our method, which could be implemented using a more precise analysis. MemorySSA is only designed for read-after-write and write-after-write dependencies, so to determine write-after-read dependencies we additionally use a modified version of MemorySSA that treats all memory accesses as if they were writes.

For the normal edges mentioned at the end of section 5.3.2, we don't actually create virtual nodes; we create normal edges going to the branch instruction. This is sound, but will prevent certain candidates from being generated (namely, those that would include the virtual node before a conditional branch, but not the conditional branch itself).

Our implementation is based on MemoDB and BCDB, and makes full use of MemoDB's distributed processing and caching features section 2.3. In fact, the Alive2 analysis worker (which needs a special version of LLVM due to Alive2's design) runs in separate processes from the rest of the Semantic Outlining analysis (which uses normal LLVM), and all the process communicate by connecting to a MemoDB server. Semantic Outlining is very resource-intensive, and these features help make it more practical to use.

5.6.2 Evaluation

We have performed an initial evaluation of our outliner compared with LLVM's IROutliner and MachineOutliner. The results are shown in table 5.2. We used three programs from the SPEC CPU 2017 benchmark suite, as well as two libraries written in the Swift programming language. We compiled each subject into LLVM bitcode and optimized it with `opt -Oz`. Then we generated four new versions of the bitcode using different IR-level outlining settings: no outlining, our outlining tool without semantic equivalence checks (syntactically identical candidates only), our full outlining tool with semantic equivalence checks, and LLVM's IROutliner. For each of these versions, we used LLVM to generate three object files with different machine code outlining settings: no outlining, normal MachineOutliner, and MachineOutliner iterated 5 times (as in [17]). Finally, we measured the size of the `.text` section of the object files.

In several cases, combining our outliner with MachineOutliner gives better results than either one alone. This makes sense, as each one can outline code that the other cannot; our

Table 5.2: Code size reductions after various outlining combinations; larger is better. "Syntactic" is our outliner without any semantic equivalence checking. "Semantic" is our outliner with semantic equivalence checking. "MO" is the MachineOutliner. "MO^5" is the MachineOutliner iterated 5 times.

		No IRO	Syntactic	Semantic	IROutliner
SPEC CPU 2017 deepsjeng_r (x86-64) 66 490 B	No MO		0.9%	1.4%	0.0%
	MO	1.8%	2.3%	**2.6%**	1.8%
	MO^5	1.9%	2.3%	**2.6%**	1.9%
SPEC CPU 2017 deepsjeng_r (AArch64) 66 392 B	No MO		0.3%	0.3%	−0.1%
	MO	3.6%	3.7%	3.7%	3.4%
	MO^5	3.7%	**3.8%**	**3.8%**	3.5%
SPEC CPU 2017 leela_r (AArch64) 98 946 B	No MO		**1.0%**	**1.0%**	−0.1%
	MO	−1.1%	−0.3%	−0.2%	−1.2%
	MO^5	−1.2%	−0.5%	−0.4%	−1.4%
SPEC CPU 2017 xz_r (AArch64) 149 457 B	No MO		−0.9%	−0.9%	0.1%
	MO	2.6%	4.3%	4.3%	2.4%
	MO^5	2.7%	**4.6%**	**4.6%**	2.5%
Swift library RIBs (AArch64) 71 783 B	No MO		1.9%	**2.8%**	−0.1%
	MO	−1.5%	−0.3%	0.5%	−1.8%
	MO^5	−2.7%	−1.6%	−1.0%	−2.8%
Swift library SnapKit (AArch64) 87 755 B	No MO		4.6%	4.7%	0.2%
	MO	5.3%	7.5%	**7.6%**	4.7%
	MO^5	5.0%	6.9%	6.8%	4.3%

outliner can outline identical code that would become different after register allocation and other processes, and can handle noncontiguous instructions; MachineOutliner can outline machine instructions that are added by the backend.

The best results for each program are provided by our outliner, usually in combination with MachineOutliner. Even without semantic equivalence, our outliner is usually more effective than IROutliner. This may be because our outliner determines what to outline by actually compiling the modified code and measuring its size, which is much more accurate than IROutliner's crude cost model. Our outliner also supports noncontiguous instruction sequences, which IROutliner cannot outline.

In most cases, the results of our outliner are almost the same with and without semantic equivalence checking. There are several possible reasons for this. The programs we used in

this initial evaluation may be too small to have significant amounts of redundant code; we intend to evaluate our outliner on much larger programs, which may have different results. Alternatively, there may be redundant code that our outliner is unable to find because some of the code isn't yet supported by Alive2 or the interpreter, or because of the constraints on which candidates our outliner can generate. A final possibility is that semantically equivalent but syntactically different code is simply too rare to be very useful in outlining.

5.7 FUTURE WORK

5.7.1 Outlining more code

There are a number of ways Semantic Outlining could be extended to outline more code, achieving greater size reductions on the same programs:

- We could consider code sequences to be equivalent even if they have some constant operands that differ, and outline them by replacing these constant operands with arguments to the outlined callee. Some other outliners are capable of doing this, but there are extra questions be resolved about how this would work for semantically equivalent code sequences.
- We could outline code sequences that are only equivalent if certain instructions are removed, by keeping those instructions in the outlined callee but making them conditional on an argument to the callee. It isn't clear how this should work when using semantic equivalence.
- For instructions that can safely be speculatively executed, we could ignore their control dependencies.
- We could outline only part of a loop, making the outlined callee return the number of loop iterations so the modified caller can execute the remaining loop instructions the correct number of times. Similarly, we could outline parts of an if-then-else, or outline return instructions, if the callee returns enough information about control flow.
- We could take advantage of cases where one candidate refines to another, but the candidates are not fully equivalent. This would complicate the process of selecting candidates (section 5.5.1).
- We could explore better search methods than the greedy search described in section 5.5.1.

5.7.2 Speeding up outlining

By analyzing the most profitable candidates our tool selects for outlining, we may be able to find heuristics that distinguish profitable candidates from unprofitable ones. If we can use those heuristics to quickly filter out candidates that seem unprofitable, we can make the Semantic Outlining process much faster.

If Semantic Outlining is still too slow for some situations, we can still use it to improve less powerful, syntax-based outliners. For example, if a certain pattern of semantically equivalent code is highly useful for Semantic Outlining, we can add an analysis to specifically recognize that pattern and use it in normal, non-semantic outlining.

5.7.3 Extending beyond single programs

When combined with software multiplexing (chapter 3) or Guided Linking (chapter 4), our Semantic Outlining tool will be able to outline code that appears in multiple programs and libraries into a single function. The results could potentially be significantly better than outlining a single program, and should be investigated.

5.7.4 Improving speed and power usage

Our Semantic Outliner is currently designed to minimize size, without regard for the speed or power usage of the optimized code. We could improve these factors by explicitly considering the tradeoffs involved; for example, we could use profiling information to avoid outlining hot code. When selecting a callee from a set of ones, we could consider the callee's speed in addition to its size.

5.7.5 Other applications

Our technique for finding equivalent code sequences may be interesting to the software engineering community as a new form of clone detection. It can also be used as an automatic way to generate test cases for individual functions and parts of functions.

5.8 CONCLUSION

We have introduced *Semantic Outlining*, a new form of outlining that offers increased benefits for code size. Semantic Outlining uses two new ideas: the Outlining Dependence

Graph, to efficiently find code sequences that can be outlined, and counterexample-guided equivalence clustering, which efficiently determines which code sequences are semantically equivalent. With these ideas, we can effectively find and outline semantically equivalent pieces of code that are missed by other outlining implementations.

Chapter 6: Comparison of Code Size Reduction Techniques

6.1 INTRODUCTION

There are a large variety of code size reduction methods described in the literature and available for use. Depending on what motivations a developer has for reducing the size of their code, some techniques may be especially effective, while others may be unhelpful. In many cases, the best solution will involve a combination of multiple techniques that complement each other, but only some combinations will be effective; some combinations are redundant or even counterproductive.

Unfortunately, there is limited guidance for developers deciding which techniques to use, especially if they are considering experimental ideas from research papers. In this chapter, I discuss many different techniques, compare them along several different dimensions, and explain how an effective combination can be chosen.

Developers may have several different motivations for reducing code size, such as meeting size limits, reducing software distribution overhead, improving speed and power usage, and increasing security. Section 6.3 discusses these motivations in more detail. Depending on motivation, developers may be interested in different aspects and tradeoffs of size reduction methods; I describe several in section 6.4.

In this chapter, I divide size reduction methods into four main classes: Specialization and Redesign (described in section 6.5), Hardware-supported Techniques (section 6.6), Compact Storage Formats (section 6.7), and Compiler Optimizations (section 6.8). Section 6.4.1 explains this classification.

The best size reductions can be achieved by combining many different techniques. As an example, consider the Scheme implementation presented by Yvon and Feeley [159]. In order to fit their implementation in 4 kB of JavaScript code, they use a combination of many different techniques that work well together:

- Manual design (section 6.5.1): the implementation omits many features that would be included in a full-size implementation, such as parser error messages and some kinds of error checking.

- Compact virtual ISA (section 6.7.5): most code is stored in an internal representation, which is compact but must be interpreted by native code.

- Software decompression (section 6.7.4): the initial program is stored in a compact string which must be decoded before the interpreter starts.

- Standard optimizations such as dead code elimination (section 6.8.1) are applied to the Scheme library included in the interpreter.

- The interpreter can be specialized to run a prespecified Scheme program (similar to section 6.5.2).

Section 6.9 explains which techniques are likely to be effective when combined in this way.

6.2 RELATED WORK

Papers that describe specific code size reduction techniques often have a brief discussion of other techniques in a "related work" section, but few publications have performed deeper comparisons. The most notable such publication is by Beszédes, Ferenc, Gyimóthy, *et al.* [36], who provide a very detailed comparison of twelve code size reduction techniques that existed up to 2001. They classify techniques along six dimensions: whether they need a decompressor, interpreter, or similar; what hardware type they are designed for; which hardware unit the decompressor is implemented in; whether the decompressed code is identical to the original code or just functionally equivalent; what level of code (such as source code, IR, or machine code) the technique is applied to; and the granularity of blocks that are compressed independently. My classification scheme is inspired in part by theirs, although I have made changes to cover a wider range of techniques. They also classify the results of each technique based on code size reduction, execution speed, energy consumption, and other factors, relying on the evaluations given in the original papers. Only two of the techniques covered work purely in software; the rest require hardware decompression support. This contrasts with my comparison, which includes many software-only methods.

Several tools have combined multiple code size reduction techniques. Lau, Schoenmackers, Sherwood, *et al.* [111] combine software-only outlining, outlining with special hardware instructions, and IBM CodePack, which compresses individual instructions; they find that hardware-supported outlining gives somewhat better results than software-only outlining with little benefit from combining both, but on the other hand, combining CodePack with outlining gives additional benefits compared to using either one alone. A high-level overview of three other tools is given in [160]. One of the three was extended to support multiple types of deduplication, outlining, specialization, and compression, and [161, Table I] gives a detailed breakdown of the effects of each technique as implemented. In this case, each technique offers a moderate benefit on its own, and the best results are obtained by combining all techniques.

Chabbi, Lin, and Barik [17, Section III] offer a recent comparison of several experimental techniques implemented for LLVM, but they use only one application and do not explore modifications to the tools. They find poor results (at most 2% reduction) for everything except the MachineOutliner.

Latendresse and Wiel [162] have a list of code compression papers, although it has not been updated since 2004. I used their list to find some of the papers I compare.

6.3 MOTIVATIONS FOR REDUCING CODE SIZE

Although code size is often neglected as an optimization target in favor of speed, reducing code size has potentially significant benefits at every scale from supercomputers to tiny Internet of Things devices. Depending on the motivation, different code size reduction techniques may be suitable for a given application. This section describes several possible motivations.

6.3.1 Meeting size limits

Embedded systems tend to have strictly limited storage space, which must be sufficient for both the application and its data. The total space available is often only a few megabytes or less. If the application code is too large, it may be impossible to run on the planned hardware, necessitating a costly redesign. Therefore, software for embedded systems is generally designed with close attention to its size. Techniques that reduce code size can allow extra functionality to be added in the same amount of space, and/or reduce the amount of developer effort needed to keep size low.

Many embedded systems today run a full, configurable software stack, with a relatively large number of available software packages [5]–[7]. Because storage space is at a premium, space limitations may prevent desired packages from being installed, or may require tedious manual steps to free up space by prioritizing and deleting the less important packages. These limitations severely limit the freedom to add useful functionality.

Outside the realm of embedded software, size limits may be set by software distribution platforms. For example, the Apple app store has a default size limit of 200 MB, above which apps can only be downloaded over Wi-Fi. This limit has been a major concern for app developers like Uber, which gets 20% fewer first-time bookings when their UberRider app crosses the limit [17]. This is a clear example where excessive app size has lead to measurable reductions in revenue. End users also have limited space to install apps, especially if they are using older devices or cheap devices common in developing countries [12].

6.3.2 Reducing download and install time

Frequent software updates are a necessity in order to fix bugs and security holes, but they can come at a significant cost. Firmware updates consume a significant amount of power on IoT devices, about half of which is the cost of wirelessly transmitting the update data [13]; this cost could be directly reduced by shrinking the code. Size reductions are also useful for Docker containers that are built once and then distributed to thousands of nodes [14], [15], and for mobile apps to improve the experience of end users downloading the app; major developers like Uber and Facebook have spent substantial effort on app size for this reason [16], [17]. Size may also be important when transmitting updates to deep space applications, which may have much less bandwidth available than terrestrial networks. For instance, NASA's Deep Space Network currently supports bandwidths less than $100\,\mathrm{kbit\,s^{-1}}$ [163].

6.3.3 Improving security

Some techniques can remove unneeded features from software, improving security by eliminating any security holes in those features [164]–[166]. This is especially effective in application-specific systems like Docker containers; large amounts of code not needed for the particular application can be removed [14], [15]. The Office of Naval Research recently initiated a program to reduce bloat and complexity in their software, motivated by the need to reduce its attack surface [167], and my work has been partly funded by this program.

6.3.4 Improving speed

For programs with nontrivial amounts of code, the processor can spend a significant amount of time waiting for instructions to be loaded from memory into the instruction cache. Optimizations that improve cache locality can be used to speed up applications ranging from automotive software to huge datacenter applications [168]–[170]. Reducing code size can allow more of the hot code to fit in the instruction cache, and also reduce the amount of the cache that must be replaced when cold code is executed, reducing cache misses and increasing performance [9]–[11].

Smaller code also means less code to load from permanent storage during cold start, which can improve speed when a user is switching between different mobile apps [12]. Another case where code loading is important is for web apps; if Javascript and Webassembly code can be made smaller, it will be downloaded faster, reducing the time it takes to load the page

and improving the user experience [171], [172].

6.3.5 Improving power usage

Power consumption is not only a critical concern for battery-powered devices but also for supercomputers [173]. Memory caches are often responsible for a large portion of the power consumption of a system [174], and code size reduction is one potential way to reduce cache pressure. Accessing an L2 cache may take twice as much power per byte as an L1 cache, and accessing main RAM may take twenty times as much [175], [176].

Instruction caches are one of the main sources of power consumption on multi-core IoT devices [177]. On a typical battery-powered consumer device, common tasks such as web browsing spend more energy on main memory access than on actual computation [174]. Even when data access is the bottleneck, reducing code size may help by freeing up extra cache space for data.

6.4 ASPECTS OF DIFFERENT METHODS

This section describes several important dimensions along which code size reduction methods can be compared. Developers will want to determine which dimensions are important for their application, and narrow down methods based on where they fall along those dimensions. For example, a developer seeking to reduce cache usage will be particularly interested in the "where effective" aspect, which classifies techniques based on whether they affect code size in cache (among other places).

6.4.1 Overall classification and special requirements

I divide code size reduction methods into four classes, shown in table 6.1. The first two classes have special requirements that make them unsuitable for some applications: Specialization and Redesign techniques require changes to the semantics or source code of the software, and Hardware-supported Techniques require special hardware modifications that are not commonly available. The other two classes can be implemented without modifying the source code or semantics or using special hardware, so they are more broadly usable. Compiler Optimizations refer to any techniques that change a program's instructions into other instructions; some such techniques are actually performed by the linker, or by a binary rewriter after the program has been linked. Compact Storage Formats are other software-based techniques that change the way the program's instructions are represented, for example

by compressing them.

6.4.2 Type of redundancy exploited

Software includes various types of redundancy, and different techniques reduce size by eliminating different types. In many cases, techniques that use different types of redundancy can be combined to get better results than either one alone, while techniques that use the same type of redundancy may work poorly in combination. This is just a rule of thumb, and there are many complications to consider; see section 6.9 for more discussion. Most techniques exploit one of the following types of redundancy.

- **Functional equivalence:** two programs may be semantically different but perform equally well for a given use case; for instance, one version of the software may have unnecessary features. Test-based debloating, which removes unneeded features, exploits functional equivalence.

- **Semantic equivalence:** for a given program, there are many other programs that will produce identical results; it may be possible to find a smaller one. For example, peephole optimizations exploit semantic equivalence.

- **Predictability:** executable code is far from random—it has a highly predictable structure. Some instructions and immediate values are much more common than others, and adjacent instructions are correlated with each other. Some studies have found that the average entropy of executables is around 5 bits per byte with simple, local prediction models, which can be compared to plain text at 4.3 bits per byte [178], [179]. By using shorter encodings for more common code sequences, and longer encodings for

Table 6.1: Basic classification of techniques.

Class	Special Requirements
Specialization and Redesign (Section 6.5)	Minor or major changes to functionality
Hardware-supported Techniques (Section 6.6)	Small or large changes to hardware
Compact Storage Formats (Section 6.7)	None
Compiler Optimizations (Section 6.8)	None

less common ones, the same program can be represented in fewer bytes. For example, Huffman coding can be used on individual instructions.

- **Duplication:** certain pieces of code may appear repeatedly throughout the whole program. For example, outlining exploits duplication.
- **Cross-Program Duplication:** pieces of code, such as libraries, may be used by multiple programs on a system. For example, shared libraries exploit cross-program duplication.

6.4.3 Where effective

Depending on what techniques are used, software may have different sizes when stored as source code files, stored as object code, transferred over the network, loaded in main memory, and loaded in the instruction cache. Most of the techniques I discuss do not affect source code size, but can be used to shrink code in all the other cases. Some exceptions are worth noting:

- Source code size is only affected by manual redesign (section 6.5.1) or clone refactoring (section 6.5.4).
- Compression (sections 6.6.2 and 6.7.4) may only affect code size at certain stages. For example, a given implementation may compress code during network transfer but decompress it before storing it to disk.
- When software is distributed in the form of a compact IR (section 6.7.5) and then compiled at run time, the compact IR will not save size in memory or cache.
- Dynamic deduplication (section 6.7.6) only affects memory and cache size, not network or disk size.

6.4.4 When applied

Size reduction tools may be applied at different times during the compilation process. Manual redesign and clone refactoring (sections 6.5.1 and 6.5.4) are applied to source code before compilation. Compact ISAs and hardware outlining (sections 6.6.1 and 6.6.3) generally affect the compiler backend. Compression and compact storage formats (sections 6.6.2 and 6.7) are applied after final machine code has been generated. And dynamic deduplication (section 6.7.6) isn't applied until run time. Other methods can generally be applied

to a compiler's intermediate representation (IR), and they can be interleaved with other IR transformations.

All methods that can be implemented in a compiler could also potentially be applied by the linker, or by using binary rewriting to modify programs that have already been compiled, although reconstructing information such as the call graph may be difficult. This may be the only way to reduce code size of legacy programs that are only available in binary form.

6.4.5 Effect on size

All techniques discussed here are intended to reduce code size, but some are more effective than others. In some cases, the more effective techniques have disadvantages that make them less appealing, such as severely slowing down the program or requiring special hardware support. In other cases, it makes sense to implement a more effective technique first, and then add some less effective techniques for additional improvements.

The tables in this chapter include a subjective rating based on the reported size savings of each technique. Note that these ratings are based on the more effective cases reported in the literature, and other applications may get worse results.

6.4.6 Effect on speed and power usage

All else equal, smaller code will execute faster and use less power because there will be fewer cache misses. However, many types of code reduction add additional overhead to the program, which may negate these benefits. For instance, outlining adds new call instructions that must be executed, and the overall effect of outlining may increase or decrease speed, depending on the exact experimental setup. When a tool decreases speed, the impact can be reduced by applying it only to cold code, assuming high-quality profiling information is available; alternatively, the tool can be applied to known-cold code such as error handling routines [123]. The tables in this chapter include a rating based on the reported change in speed caused by each technique; power usage is generally correlated with execution time.

6.4.7 Effect on security

Manual redesign (section 6.5.1) and testing-based debloating (section 6.5.3) can specialize software by removing unneeded features, which can directly improve security by eliminating insecure code [164]–[166].

Most other code size reduction methods do not affect the semantics of the program, so they have no effect on security during normal operation. In programs that have buffer overflow vulnerabilities, these methods can still affect security by increasing or reducing the number of return-oriented programming (ROP) gadgets. This effect must be evaluated with care because the total number of gadgets is not a good indication of security; as few as 11 gadgets of specific types may be sufficient for an attack [180], [181].

6.4.8 Effect on debugging and maintenance

Manual redesign and clone refactoring can improve the quality of source code, easing maintenance of the software. The other methods do not affect source code, but they can still interfere with debugging of the software. For example, outlining will introduce new function calls that don't correspond to any function in the original program, making stack traces more difficult to understand, while code compression will cause problems for a debugger that can't decompress the code. Problems like these can be avoided by adding sufficient support to debugging tools, so they can report what the program state would be without the size reduction technique, but such support is not always available.

6.5 SPECIALIZATION AND REDESIGN TECHNIQUES

Table 6.2: Summary of specialization and redesign techniques.

Technique	Size	Speed	Type of Redundancy	Pros/Cons
Manual redesign (Section 6.5.1)	++++	Varies	Functional equivalence	+ Maximum potential savings + Can improve speed, security, maintainability − Requires major effort
Partial evaluation (Section 6.5.2)	+++	+++	Functional equivalence	+ Sometimes improves speed − Requires known configuration
Testing-based debloating (Section 6.5.3)	+++	?	Functional equivalence	+ Can improve security − Requires tests − Can break programs if tests aren't exhaustive
Clone refactoring (Section 6.5.4)	+	?	Duplication	+ Can improve source code quality − May require developer interaction

6.5.1 Manual redesign

The most dramatic reductions in code size can only be achieved by redesigning and rewriting software with code size in mind. Especially large reductions are possible when code size is so crucial that other requirements can be relaxed and functionality can be removed. A redesign can reduce code size at every level, from source code to the processor's code cache, and will also affect data size, speed, security, maintainability, and other aspects of the software. Because the developers have so much flexibility to make application-specific tradeoffs, few general statements can be made about the effects of manual redesign except that it requires more developer effort than any other technique, especially considering the effort needed to test and debug the new code.

Hobbyists can create minimal programs in less than 256 bytes [182] and full graphical demos with music in only 64 kilobytes [183] by designing an entire program with careful consideration to code size, writing every line of code from scratch, excluding features that are not strictly necessary (such as the ability to configure the program), and using code compression. The costs of applying this level of manual optimization to a large software project would be enormous. Not only would the entire software development process take far longer, but any library code being used would need just as much effort for optimization. There are extremely few projects for which this level of effort would be cost-effective.

Although redesigning software is mainly a manual process, automated tools can still make the process easier. Some programming language features, such as C++ templates and Rust's monomorphisation, can lead a large amount of machine code to be generated from a single line of source code [184]; Uber uses linting tools to warn developers who overuse such features so they can choose a better design [17]. Developers can make their software customizable for a particular application, so that unneeded code can be deleted or replaced with stubs [185], potentially with a maintenance cost [186], [187]; research has found ways to simplify this process and automatically generate working stubs [166].

6.5.2 Partial evaluation

Programs are often deployed repeatedly to different machines with the same configuration inputs, such as command line options and configuration files. *Partial evaluation* can be used to optimize a program for one particular configuration; the part of the program that reads and processes the configuration is evaluated in advance, and code that is no longer reachable afterwards can be removed, including code for features disabled by the configuration. One tool achieved some improvements by using existing compiler optimizations, and also creating

specialized copies of functions when some of their call sites used constant arguments [188]. Better results are achieved by using more aggressive versions of certain optimizations, such as loop unrolling and interprocedural constant propagation [189], [190]. Results vary significantly depending on the program and chosen configuration; size reductions of 20% to 60% can be attained in many cases [190].

Unlike testing-based debloating, partial evaluation is a sound transformation; it will not change program behavior as long as the program's configuration inputs do not change. Aside from code size, partial evaluation has also been proposed to improve speed [191], providing a speedup greater than 50% in one case [190]. Partial evaluation has also been suggested to increase security [188]–[190], but because it is a sound transformation, it may have limited effects; see section 6.4.7.

6.5.3 Testing-based debloating

Software often includes features that are unnecessary for a particular use case, but cannot simply be disabled with a configuration option. If the user is able to perform tests that exercise all needed features of the program, dynamic analysis can be used to automatically detect and remove code that was not needed during the tests. These debloating methods can potentially remove very large amounts of code, and can directly improve security by removing insecure features that could potentially be used at run time but are not actually needed. However, there is a significant risk that some of the removed code will turn out to be necessary at run time and the program will break. For example, important error handling code may be removed for errors that do not occur during testing. This risk has been identified as an essential limitation of testing-based debloating [192].

The simplest implementation is to measure the code coverage of the tests and remove all uncovered code; variations of this approach have been used to remove unnecessary files from containers [14] and unnecessary functions from web applications [193]. If additional tests are available that exercise unwanted features, it is safer to remove only code covered by those tests, and retain code which is not covered at all (which could be important error-handling or exception-handling code) [194]. Another way to make debloating safer is to include additional uncovered code that may be necessary for the desired features, based on control-flow heuristics [195]. If the software already uses configuration options, as used by the Linux kernel, the debloating tool can find configuration options to disable rather than directly deleting code, which may be less likely to lead to errors [192].

A more aggressive approach is to use delta debugging [196] or genetic search [194], which iteratively try to delete sequences of source code in order to find the smallest program that

passes the tests. These methods can get better results by deleting covered code that has no effect on the test cases, but the process is extremely slow; even when reinforcement learning is used to predict which code can be removed, delta debugging takes several hours to debloat programs with about 10 000 lines of code [164], and genetic search has not been demonstrated on programs this large [194].

6.5.4 Clone refactoring

Clone refactoring involves finding duplicate sequences in source code and refactoring them into new functions, eliminating the duplication. It is essentially a form of outlining (section 6.8.4) applied at the source code level, but it is used primarily for software maintenance purposes, and reductions in compiled code size are merely a side effect. Recent surveys of clone detection and refactoring papers can be found in [104], [197]. Although most parts of the refactoring process can be automated, clone refactoring tools still tend to rely on developer interaction, if only to choose which clones should be refactored.

6.6 HARDWARE-SUPPORTED TECHNIQUES

Table 6.3: Summary of hardware-supported techniques.

Technique	Size	Speed	Type of Redundancy	Pros/Cons
Compact ISAs (Section 6.6.1)	+++	++	Predictability	+ Can improve speed + Can improve power − Major changes to hardware − Major changes to compilers etc.
Hardware Decompression (Section 6.6.2)	+++	−/+	Predictability	+ Some versions also apply to data − Moderate changes to hardware
Hardware Outlining (Section 6.6.3)	++	−/+	Duplication	− Requires new hardware instructions

6.6.1 Compact ISAs

The design of a computer's instruction set architecture (ISA) has a major effect on the size of each instruction and the number of instructions required to implement a given piece of code; total code size in one ISA may be more than 50% larger than another [198], [199], [200, Sec. 5.5]. Weaver and McKee [199], [201] wrote the same program in assembly language on

31 different architectures and reported on which architectural features correlate with code size; CISC and embedded architectures generally had smaller code, non-embedded RISC architectures generally had larger code, and the sole VLIW architecture they studied had some of the largest code. Compact ISAs can not only reduce code size but also improve performance (by reducing cache misses) and power efficiency [200], [202]; however, ISA changes require extensive effort to modify not only the hardware processor but also the compiler, debugger, etc.

Commercial ISAs have had extensions written specifically in order to reduce code size, such as MIPS16 [203], THUMB [204], and the RISC-V C Extension [200]; see [200, Sec. 5.1] for a history of such ISAs. Additional compact ISAs have been proposed in the literature, using structured control flow [205], new 8-bit opcodes for a 16-bit ISA [202], or mask bits indicating which VLIW instruction fields are present [206].

6.6.2 Hardware decompression

Instead of modifying the core processor, another option is to store code in a compressed format in main memory and modify the memory hardware to decompress code as it is loaded. There is a large body of work on general hardware decompression in the memory hierarchy that applies to both code and data, sometimes including newly-written data; for recent reviews, see [207]–[209]. I will focus here on papers that specifically consider code compression.

Aside from reducing size, hardware decompression has been used to significantly reduce cache misses [210] and power usage [211], [212] and improve performance [212]–[214]. Debugging may be affected; some implementations are transparent to the debugger [215], [216], but others would need a modified debugger that understands compressed code. Effectiveness varies significantly based on how compact the target ISA is; one technique reduced code size to about 50% on SHARC and ARM, but only to 85% of its original size on the compact Thumb instruction set [217].

A variety of different compression methods have been implemented. Individual instructions can be encoded so that common instructions use fewer bits and uncommon ones use more bits, using Huffman coding [178], [218], [219], arithmetic coding [217], or dictionary-based coding [220], [221]. Compression can be improved by splitting each instruction into separate fields and compressing each field using a different code [178], [219], [222], [223]. Fields can be predicted from previous fields using Markov models [217]. Models and coding tables can be system-wide or specific to a particular program.

Programs frequently jump from one piece of code to another, and decompressors must

be able to quickly load the target code. If compressed instructions are independent from each other and aligned to a byte or word boundary, the normal jump and call instructions may suffice; the decompressor can start from an arbitrary instruction given its address. If instructions are compressed as independent bit sequences, jump and call instructions can be modified to address individual bits [219], and it has been argued that this is the best solution [224]; however, this solution may require changes to the ISA and processor core. Another option is to independently compress each cache line (or similarly sized block), using a lookup table to map between compressed addresses and uncompressed ones [218].

Commercial hardware decompressors have included IBM CodePack [215], [216], a commercially available system for PowerPC, which worked by dividing each instruction into multiple parts and using Huffman coding on each part. The mAgicV DSP architecture, based on a VLIW instruction set, also included a hardware decompressor [225]. More recently, some MediaTek systems-on-a-chip such as the MTK6260 had an undocumented hardware decompressor that replaced common instructions with shorter sequences of bits, similar to Huffman coding [226].

When hardware decompression is used with a particularly simple encoding scheme, it becomes similar to a Compact ISA (section 6.6.1). On the other hand, hardware decompression is also closely connected to software decompression (section 6.7.4). Hardware implementations may be much faster, but may limit which compression techniques can be used. One paper is based on software decompression, but modifies the hardware slightly so decompressed code can be written directly into the code cache [227].

6.6.3 Hardware outlining

Outlining is normally performed purely in software (see section 6.8.4), but some researchers have proposed new hardware instructions to make outlining more effective. The "call-dictionary" instruction [113], [114] acts like a call instruction, but with an extra argument that specifies the number of instructions to execute before returning. This removes the need to explicitly store return instructions, but more importantly it also allows outlined code sequences to partially overlap with each other even if they need to return at different points. Bitmask echo instructions [111] add a mask to specify which subset of instructions should be executed, allowing similar sequences to be merged even if certain instructions are different. The authors reduce code size by 15% with no average change in performance. Echo instructions have been extended with support for nested echo instructions, control flow, calls, and other features [115]–[118].

Table 6.4: Summary of compact storage format techniques.

Technique	Size	Speed	Type of Redundancy	Pros/Cons
Software Multiplexing (Section 6.7.1)	+ + +	++	Cross-Program Duplication	+ Enhances other optimizations − Incompatible with dynamic linking features − Multiplexed programs must be distributed as a single unit
Shared Libraries (Section 6.7.2)	+ + +	−−	Cross-Program Duplication	+ Programs and libraries can be distributed separately + Unrelated programs can share RAM for libraries − Can lead to dependency hell − Prevents optimization of program–library interactions − Can introduce security holes
Guided Linking (Section 6.7.3)	++	++	Cross-Program Duplication	+ Compatible with dynamically linked software + Enhances other optimizations − Requires developer to understand the constraint system
Software Decompression (Section 6.7.4)	+ + +	−−	Predictability, possibly Duplication	+ Highly effective − Significant slowdown if applied in RAM
Compact Virtual ISA (Section 6.7.5)	++	− − −	Predictability	− Very slow unless a JIT is used
Dynamic Deduplication (Section 6.7.6)	++	−	Cross-Program Duplication	+ Can be applied across multiple VMs − Can only deduplicate page-aligned data

6.7.1 Software multiplexing

When multiple programs share the same code, they can be combined into a single program using software multiplexing, and the shared code can be deduplicated. See chapter 3 for further discussion. Multiplexing can enhance the effect of other duplication-based size reducers by applying them across multiple programs and libraries. The main limitations of multiplexing are that the multiplexed programs must be distributed and updated as a single unit, and programs that rely on dynamic linking behavior (such as plugins) will not work correctly. Shared libraries avoid those limitations, and can be combined with Guided Linking to get some of the same benefits as multiplexing.

6.7.2 Shared libraries

Shared libraries are practical way for developers to reduce code size [228]. Whenever a piece of code is used by more than one program on a system, moving it into a library will prevent duplication of the code. Shared libraries can ensure only one copy of the code is transferred over the network, stored on disk, and loaded in RAM. Thanks to the benefits of shared libraries, and the fact that they can help developers organize their code, they have become ubiquitous on smartphones, desktop computers, and larger systems, but they do have some key drawbacks [69], [72], [228], [229]: compiler optimizations are unable to apply across the application / shared-library boundary, and shared libraries increase both startup costs (because the libraries must be loaded and linked at run time) and later execution costs (because of indirect function dispatch). Shared libraries can also harm security if they are used without sufficient care, such as by allowing a malicious library to be loaded [88], [89], which has led to high-severity vulnerabilities in a variety of popular software [90]–[96].

Some optimizations have been extended to take program information into account when they are applied to shared libraries, particularly dead code elimination [30]–[32]. Guided Linking is a much more general way to apply optimizations across programs and shared libraries, and is described in the next section.

6.7.3 Guided Linking

For software that uses dynamic linking, Guided Linking provides a way to optimize code across dynamic linking boundaries. See chapter 4 for a detailed discussion. Guided Linking causes dramatic size reductions in certain specific cases, such as when combining multiple versions of the same library; in other cases, it gets moderate benefits. The main limitation of Guided Linking is that the developer must be able to provide correct constraints on the software's dynamic linking behavior.

6.7.4 Software decompression

Executable files can be compressed effectively with standard lossless compression algorithms, such as DEFLATE, Zstandard, LZMA, etc. Compression ratios can be improved by taking advantage of the unique structure of executable code, such as by transforming relative offsets into absolute offsets [230, Sec. "E8E9"], splitting instruction into multiple fields and compressing different field types separately [16], [231], or changing instruction ordering or register assignment [231], [232].

The biggest limitation of standard compression algorithms is that they are not designed to support random access; the whole file must be decompressed at once. This isn't a problem if the compression is only used during network transfer, but it may be problematic if compression is used on disk, as the entire program must be decompressed and loaded in RAM when it is run. Using whole-program compression on code stored in RAM would be impractical because the entire program would need to be decompressed each time a cache miss occurred.

Instead of compressing the whole program at once, researchers have proposed several designs that compress each function separately, decompressing it when it is called [161], [233]–[235]. It then becomes possible to compress code in main RAM, with only a small fraction of functions uncompressed at any given time; however, managing the memory allocated for the decompressed functions can be somewhat complicated [233].

In general, software decompression will significantly slow down execution whenever new code must be decompressed. This overhead can be reduced by compressing only cold code, so decompression will rarely be necessary, but only if accurate profiling information is available. Alternatively, the operating system can detect which parts of memory are not being accessed at runt ime, whether code or data, and compress them automatically; the `zram` module for Linux works this way.

Hardware decompression (section 6.6.2) may be faster than software decompression, but it requires specialized hardware and is less flexible than software tools.

6.7.5 Compact virtual ISA

Several popular platforms, including Java, .NET, and Android, distribute software in the form of virtual machine code instead of physical machine code. A number of papers have investigated virtual machine designs that make the code smaller than equivalent physical machine code. The virtual machine can be tuned for the particular application being compiled, with frequent code sequences replaced with a single instruction [129], [236]. Hot code can be compiled to physical machine code while other code is compiled to virtual machine code, obtaining the benefits of both [237], [238]. Interpreters are several times slower than native execution; if interpretation is too slow, the program can be distributed as compact virtual machine code and just-in-time compiled to physical machine code [236], [239], with a possible increase in memory usage. Some methods go beyond compressing the instructions and redesign the entire executable format to be more compact [239]–[241].

Compact virtual ISAs can be compared with compact hardware ISAs (section 6.6.1); virtual ISAs are much more flexible and can be customized to a particular application, but they

are also much slower to execute. When common sequences of instructions are replaced with a single new instruction, the effect is similar to outlining (section 6.8.4).

6.7.6 Dynamic deduplication

Code can be deduplicated at run time by detecting identical memory pages in virtual memory and merging them so they occupy the same space in physical memory [242], [243]; similar methods can be used for disk space or cache lines [244]. This could be useful on any system with virtual memory, but achieves especially dramatic savings on virtualization servers that host many virtual machines running the same OS. However, these techniques may require significant time overhead to search for duplicate pages, and they can only deduplicate code aligned in such a way that the pages are identical.

Slinky [229] minimizes overhead by precomputing a cryptographic hash of each code page, used to find duplicate pages. It greatly reduces the storage and bandwidth costs of static linking; however, it still incurs a 20% storage space increase and a 34% network bandwidth increase over shared libraries.

6.8 COMPILER OPTIMIZATIONS

Table 6.5: Summary of compiler optimization techniques.

Technique	Size	Speed	Type of Redundancy	Pros/Cons
Standard Optimizations (Section 6.8.1)	+++	Varies	Semantic Equivalence	+ Easy to apply + Can improve speed and power
Optimization Tuning (Section 6.8.2)	++	−/+	Semantic Equivalence	+ Easy to apply + Can improve speed and power
Superoptimization (Section 6.8.3)	++	−/+	Semantic Equivalence	+ Can improve speed and power − Extremely slow to apply
Outlining (Section 6.8.4)	++	−/+	Duplication	+ Sometimes improves speed
Semantic Outlining (Section 6.8.5)	?	?	Duplication	+ More powerful than normal outlining − Extremely slow to apply
Function Merging (Section 6.8.6)	++	−/+	Duplication	+ Less overhead than outlining − Less powerful than outlining

6.8.1 Standard optimizations

Many of the standard optimizations implemented in popular compilers can reduce code size. Common subexpression elimination, unreachable code elimination, dead code elimination, global value numbering, and constant propagation will generally reduce code size. Many peephole optimizations have the same effect. Jain, Bora, Kumar, *et al.* [245] tested various combinations of the LLVM compiler's standard size optimizations; they found that CFG simplification and scalar replacement of aggregates alone can reduce code size by about 30%, by simplifying inefficient code generated by the compiler frontend. A group of 11 further transformations is responsible for most of the rest of the code size improvement.

Inlining often increases code size, but can still be beneficial if used judiciously; inlining very small functions and functions that are only called in one location is often an improvement. Inlining implementations are often tuned with a focus on speed, and improved results are possible by tuning them for size [246], [247].

The effect of standard optimizations can be enhanced by using more accurate analyses. For instance, using a more precise abstract domain for abstract interpretation can enable more sections of dead code to be detected and eliminated [248]. Standard optimizations can also be enhanced by analyzing larger units of code. Popular compilers support interprocedural dead code and dead global elimination, and can also use the linker to detect and delete functions that are never used by a given program.[1] Compilers that support Link-Time Optimization (LTO) can perform optimizations like inlining and interprocedural constant propagation across source file boundaries.

6.8.2 Optimization tuning

Compilers' built-in optimization sequences are not necessarily the most effective ones for a given program. More effective sequences of already implemented optimizations can be found manually, or by genetic search or other automatic algorithms [249]. For a survey of machine learning methods to find optimization sequences, see [250]. For detailed investigations of optimization sequences in recent LLVM versions, see [251].

6.8.3 Superoptimization

Superoptimization [150], [151] is an extremely slow, but more powerful alternative to standard local compiler optimizations. It works by searching the space of all possible programs

[1] For example, with the `-ffunction-sections` and `--gc-sections` options for Clang and GCC, and the `/Gy` and `/OPT:REF` options for Microsoft Visual Studio.

to find one that is better (faster, smaller, etc.) than the input program but produces the same results. An equivalence checker is used to discard programs that cannot be proven equivalent to the input. Superoptimizers can find code size optimizations that are missed by a traditional compiler; the main disadvantage is that the search process is extremely slow.

6.8.4 Outlining

Outlining works by identifying code sequences that have multiple redundant copies in the program. For each such sequence, it replaces all the copies with calls to a single, newly created function containing the sequence. See chapter 5 for more discussion.

6.8.5 Semantic Outlining

Semantic Outlining is an extension to outlining that searches for groups of semantically equivalent code sequences throughout the program, replacing them with calls to a new function containing the shortest sequence. See chapter 5. Semantic Outlining can make improvements that normal outlining cannot; the downside is that it is extremely slow, comparable to superoptimization.

6.8.6 Function merging

As an alternative to outlining, duplicate code sequences in different functions can be merged by combining the functions into one. Function merging avoids adding new call instructions, but it is more limited than outlining because it requires the merged functions to have significant similarities, while outlining can deduplicate small pieces of code in functions that are otherwise completely different from one another.

Koch, Franke, Bhandarkar, *et al.* [126], [127] attempt to merge functions when they have the same type signature, isomorphic CFGs, corresponding basic blocks having equal numbers of instructions, and corresponding instructions having identical result types (but possibly different operations and operands). They use hashing to quickly identify functions that can be merged. When there are non-identical instructions, the merged functions use conditional branches to execute the appropriate instructions depending on the caller. They achieve a total code size reduction of 3% to 4% depending on architecture, with a modest slowdown of 0.3% because the added call overhead is partially offset by improved cache performance.

Rocha, Petoumenos, Wang, *et al.* [128] also merge functions using control flow polymorphism, choosing functions that use similar opcodes and similar types to merge, but they

use a more flexible design based on sequence alignment that can merge functions even when they have different signatures and CFGs. They reduce code size by up to 30%, with minimal effect on speed because they only modify cold code.

6.9 CHOOSING A COMBINATION OF TECHNIQUES

This section gives a brief guide to choosing code size reduction techniques, considering how techniques interact with each other. In general, the best results can be obtained by choosing one method from each paragraph below; however, some are only useful in specific cases or require large amounts of effort, so typically only a few methods will be used.

Standard optimizations and/or superoptimization. The compiler's built-in optimizations are always useful, easy to enable, and should always be used. Tuning can be explored as an easy way to increase effectiveness. If compile time is not important, superoptimization can be used instead of or in addition to some standard optimizations. Standard optimizations and superoptimization generally combine well with all other methods, although if the optimized code is less predictable than unoptimized code, compression may be less effective.

Manual redesign. Redesign can almost always be used to reduce code size. However, given that it can involve a very large amount of developer effort, it may only be worthwhile as a last resort if other techniques are insufficient. Manual redesign generally combines well with all other methods (although the redesigned code may have less redundancy for other methods to exploit).

Partial evaluation or testing-based debloating. Partial evaluation is worth trying if the program will be deployed with a fixed configuration, as it can be very effective in some cases. Testing-based debloating can also be very effective, but carries a risk that it will break the program, so should only be used with great caution. Combining these two methods may not be effective, as they may both delete the same code; however, either method will combine well with other types of methods.

Software multiplexing, or shared libraries with Guided Linking. Whenever there are multiple programs that use the same code, software multiplexing or shared libraries can be used effectively to deduplicate the code. Multiplexing will be more effective because it doesn't introduce any dynamic linking overhead, and it allows other optimizations to be

applied to the whole set of programs at once. When multiplexing is not an option because the programs must be distributed independently, or because the programs rely on dynamic linker behavior, shared libraries can be used instead. In this case, Guided Linking can be applied to improve optimization. Multiplexing and shared libraries generally combine well with other types of methods.

Duplication-based methods. Whole-program compression is highly effective to reduce program size transferred over the network and possibly stored on disk, but it can't be used to reduce program size in memory. Outlining is a good way to get a moderate size reduction, optionally enhanced by hardware instructions (if available) or Semantic Outlining (if compile time is not important). Function merging is less general than outlining, and the best methods are very slow, so it is less likely to be worth pursuing. All these methods will reduce each other's effectiveness if used in combination, because they all take advantage of the same type of redundancy; still, it may be worth combining whole-program compression (for network transfer) and outlining (for run time). All these methods will generally combine well with other types of methods.

Predictability-based methods. When changing hardware is feasible, the use of compact ISAs or hardware decompression can be highly effective. All other predictability-based methods will add overhead at run time, so in cases where speed is important they should only be applied to cold code, if at all. Software decompression, compact executable formats, and compact virtual ISAs can all be effective depending on the needs of the system. All these methods will reduce each other's effectiveness if used in combination, because they all take advantage of the same type of redundancy. In particular, if instructions are encoded in the form of a bit stream, further compression will not be very effective. All these methods will generally combine well with other types of methods.

6.10 FUTURE WORK

In this chapter, I have given rough ratings of the code size reduction potential and speed improvement potential of different techniques. I have based these ratings on the size and speed changes reported by papers on each technique, but the quality of the ratings can be questioned because each paper tends to use different subject programs, architectures, and benchmark methods. Future work should perform an apples-to-apples comparison of all the most important methods using the same subject programs, architectures, and benchmarks. In order to cover the wide range of use cases in which code size reduction is desirable, the

comparison should evaluate several different scenarios, such as embedded software, mobile apps, and desktop programs.

In section 6.9, I give recommendations about which techniques are useful in different situations, and which techniques are likely to combine well together, based on reasoning about the way each techniques work. Future work should test these recommendations and determine whether the actual effects of combining techniques confirm my reasoning.

This chapter provides guidance to help a developer choose which methods are worth evaluating, but their actual evaluation will still be difficult, because there is no centralized way to find implementations of the different methods. Future work could build a software distribution that includes implementations of many different methods, using an up-to-date compiler, so that developers can evaluate different methods more easily.

6.11 CONCLUSION

In this chapter, I have provided a review of a large variety of code size reduction methods. I have given several considerations that would be relevant for someone interested in reducing the size of their code, along with an overview of the methods and combinations of methods that seem most promising. I believe this work will be useful to developers who are investigating different techniques to reduce code size.

Chapter 7: Conclusion

In this dissertation I have argued that code size reduction is an important goal for systems ranging from Internet of Things devices to large-scale computers, and that newly feasible compiler techniques have the potential to significantly increase the level of code size optimization that can be accomplished.

I have presented the Bitcode Database (BCDB), an infrastructure project that serves as the basis for all my research projects, and described how it can be used to detect duplicate functions in LLVM IR. Building on the BCDB, I introduced an improved version of Software Multiplexing that can deduplicate identical functions even when they appear in different software packages.

I have introduced two novel code size optimization methods based on newly practical compiler techniques. The first is Guided Linking, a new technique for applying compiler optimizations, including function deduplication, across dynamic linking boundaries to reduce the overhead of dynamic linking. I described how this can be done without breaking compatibility with existing software as long as the developer providers certain constraints on the behavior of the dynamic linker.

My second novel method is Semantic Outlining, which entails finding and deduplicating equivalent pieces of code in large programs, even if the pieces are syntactically very different. Semantic Outlining is based on two new ideas, the Outlining Dependence Graph and counterexample-guided equivalence clustering, which together make it more efficient to generate and test large numbers of outlining candidates.

I have also presented a comparison of a wide variety of code size reduction techniques, both my own and other important techniques from the literature. Along with this comparison, I have described which combinations of techniques are likely to be especially effective, indicating that both Guided Linking and Semantic Outlining can complement each other and other ideas.

Taken together, Guided Linking, Semantic Outlining, and my comparison of techniques are highly useful for the goal of optimizing code size in a variety of situations.

References

[1] "PIC10F200/202/204/206," Microchip Technology, Data Sheet DS40001239F, 2014. [Online]. Available: https://www.microchip.com/en-us/product/pic10f200 (visited on 10/07/2021) (cit. on p. 1).

[2] "ARM Keil development tools," Emprog Inc. (n.d.), [Online]. Available: https://www.emprog.com/emprog/arm-keil/ (visited on 10/07/2021) (cit. on p. 1).

[3] "Green Hills optimizing compilers," Green Hills Software. (n.d.), [Online]. Available: https://www.ghs.com/products/compiler.html (visited on 10/07/2021) (cit. on p. 1).

[4] "STM8, ST7 compiler," Raisonance. (2017), [Online]. Available: https://raisonance.com/stm8-compiler.html (visited on 10/07/2021) (cit. on p. 1).

[5] [SW] The OpenWrt Project, *OpenWrt*, URL: https://openwrt.org, VCS: https://git.openwrt.org/ (cit. on pp. 1, 17, 24, 96).

[6] M. Vögler, J. M. Schleicher, C. Inzinger, and S. Dustdar, "A scalable framework for provisioning large-scale IoT deployments," *ACM Transactions on Internet Technology*, vol. 16, no. 2, Mar. 2016. DOI: 10.1145/2850416 (cit. on pp. 1, 17, 96).

[7] E. Baccelli, C. Gündoğan, O. Hahm, *et al.*, "RIOT: An open source operating system for low-end embedded devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, Dec. 2018. DOI: 10.1109/JIOT.2018.2815038. [Online]. Available: https://www.riot-os.org/assets/pdfs/riot-ieeeiotjournal-2018.pdf (cit. on pp. 1, 17, 96).

[8] The OpenWrt Project, *OpenWrt documentation*, Sep. 5, 2021, ch. 4/32 warning. [Online]. Available: https://openwrt.org/supported_devices/432_warning (visited on 11/05/2021) (cit. on pp. 1, 24).

[9] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proc. ACM SIGPLAN 1990 Conf. Programming Language Design and Implementation (PLDI '90)*, June 20–22, 1990. DOI: 10.1145/93548.93550 (cit. on pp. 1, 70, 97).

[10] P. Zhao and J. N. Amaral, "Function outlining and partial inlining," in *Proc. 17th Int. Symp. Computer Architecture and High Performance Computing (SPAC-PAD'05)*, Oct. 24–27, 2005. DOI: 10.1109/CAHPC.2005.26. [Online]. Available: https://webdocs.cs.ualberta.ca/~amaral/papers/ZhaoAmaralSBAC05.pdf (cit. on pp. 1, 70, 97).

[11] P. Zhao and J. N. Amaral, "Ablego: A function outlining and partial inlining framework," *Software: Practice and Experience*, vol. 37, no. 5, 2007. DOI: 10.1002/spe.774. [Online]. Available: https://webdocs.cs.ualberta.ca/~amaral/papers/ZhaoAmaralSPE07.pdf (cit. on pp. 1, 70, 97).

[12] M. Greenia, B. Maher, and S. Nay. "Optimizing Android bytecode with ReDex," Facebook. (Oct. 1, 2015), [Online]. Available: https://engineering.fb.com/2015/10/01/android/optimizing-android-bytecode-with-redex/ (cit. on pp. 1, 96, 97).

[13] J. Bauwens, P. Ruckebusch, S. Giannoulis, I. Moerman, and E. De Poorter, "Over-the-air software updates in the Internet of Things: An overview of key principles," *IEEE Communications Magazine*, vol. 58, no. 2, Feb. 2020. DOI: 10.1109/MCOM.001.1900125 (cit. on pp. 2, 97).

[14] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "CIMPLIFIER: Automatically debloating containers," in *Proc. 2017 11th Joint Meeting Foundations of Software Engineering (ESEC/FSE 2017)*, Aug. 2017. DOI: 10.1145/3106237.3106271 (cit. on pp. 2, 97, 104).

[15] [SW] DockerSlim contributors, *DockerSlim*, URL: https://dockersl.im/(visited on 08/22/2020), VCS: https://github.com/docker-slim/docker-slim (cit. on pp. 2, 97).

[16] S. Bhatia. "Superpack: Pushing the limits of compression in Facebook's mobile apps," Facebook. (Sep. 13, 2021), [Online]. Available: https://engineering.fb.com/2021/09/13/core-data/superpack/ (cit. on pp. 2, 97, 109).

[17] M. Chabbi, J. Lin, and R. Barik, "An experience with code-size optimization for production iOS mobile applications," in *2021 IEEE/ACM Int. Symp. Code Generation and Optimization (CGO)*, Mar. 2021. DOI: 10.1109/CGO51591.2021.9370306 (cit. on pp. 2, 66, 68, 87, 89, 96, 97, 103).

[18] W. Dietz and V. S. Adve, "Software multiplexing: Share your libraries and statically link them too," in *Proc. ACM Programming Languages*, vol. 2, OOPSLA, Nov. 2018. DOI: 10.1145/3276524 (cit. on pp. 2, 3, 7, 17, 18, 36).

[19] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, ser. Lecture Notes in Computer Science, vol. 4963, 2008. DOI: 10.1007/978-3-540-78800-3_24 (cit. on p. 2).

[20] T. A. L. Sewell, M. O. Myreen, and G. Klein, "Translation validation for a verified OS kernel," in *Proc. 34th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '13)*, 2013. DOI: 10.1145/2499370.2462183 (cit. on p. 2).

[21] N. Bjørner and L. de Moura, "$Z3^{10}$: Applications, enablers, challenges and directions," in *6th Int. Workshop Constraints in Formal Verification (CFV)*, 2009. [Online]. Available: https://leodemoura.github.io/files/cfv09.pdf (visited on 11/05/2021) (cit. on p. 2).

[22] M. Dahiya and S. Bansal, "Black-box equivalence checking across compiler optimizations," in *Proc. 15th Asian Symp. Programming Languages and Systems (ASPLAS 2017)*, ser. Lecture Notes in Computer Science, vol. 10695, 2017. DOI: 10.1007/978-3-319-71237-6_7 (cit. on pp. 2, 74).

[23] J. P. Lim and S. Nagarakatte, "Automatic equivalence checking for assembly implementations of cryptography libraries," in *Proc. 2019 IEEE/ACM Int. Symp. Code Generation and Optimization (CGO 2019)*, Feb. 2019. DOI: 10.1109/CGO.2019.8661180. [Online]. Available: https://people.cs.rutgers.edu/~sn349/papers/casm-verify-cgo-2019.pdf (cit. on pp. 2, 74).

[24] B. R. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic program alignment for equivalence checking," in *Proc. 40th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '19)*, 2019. DOI: 10.1145/3314221.3314596. [Online]. Available: https://theory.stanford.edu/~aiken/publications/papers/pldi19.pdf (cit. on pp. 2, 64, 74).

[25] S. Bartell, W. Dietz, and V. S. Adve, "Guided linking: Dynamic linking without the costs," in *Proc. ACM Programming Languages*, vol. 4, OOPSLA, Nov. 2020. DOI: 10.1145/3428213 (cit. on pp. 5, 29).

[26] S. Bartell and V. S. Adve, "Guided linking: Shrinking and speeding up dynamically linked code," presented at the 2020 LLVM Virtual Developers' Meeting (Oct. 6–8, 2020), LLVM Foundation. [Online]. Available: https://youtu.be/QyNJKllZP4I (cit. on pp. 5, 29).

[27] [SW] S. Bartell, *The Bitcode Database with Guided Linking* Nov. 20, 2020, VCS: https://github.com/yotann/bcdb, SWHID: ⟨swh:1:snp:5ff30152e382f7a05b7e56fab58514f642d849d7⟩ (cit. on pp. 5, 29).

[28] *The Fortran automatic coding system for the IBM 704, Programmer's reference manual*, International Business Machines Corporation, Oct. 15, 1956. [Online]. Available: https://archive.computerhistory.org/resources/text/Fortran/102649787.05.01.acc.pdf (visited on 11/05/2021) (cit. on p. 6).

[29] A. Srivastava and D. W. Wall, "A practical system for intermodule code optimization at link-time," Digital Western Research Laboratory, WRL Research Report 92/6, Dec. 1992. [Online]. Available: https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-92-6.pdf (cit. on pp. 7, 34).

[30] N. Davidsson, A. Pawlowski, and T. Holz, "Towards automated application-specific software stacks," in *Computer Security (ESORICS 2019)*, ser. Lecture Notes in Computer Science, vol. 11736, 2019. DOI: 10.1007/978-3-030-29962-0_5 (cit. on pp. 7, 35, 36, 109).

[31] A. Ziegler, J. Geus, B. Heinloth, T. Hönig, and D. Lohmann, "Honey, I shrunk the ELFs: Lightweight binary tailoring of shared libraries," *ACM Trans. Embedded Computing Systems*, vol. 18, no. 5s, Oct. 2019. DOI: 10.1145/3358222 (cit. on pp. 7, 35, 109).

[32] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating binary shared libraries," in *ACSAC '19: Proc. 35th Annual Computer Security Applications Conf.*, 2019. DOI: 10.1145/3359789.3359823 (cit. on pp. 7, 35, 109).

[33] C. R. Mulliner and M. Neugschwandtner, "CodeFreeze: Breaking payloads with runtime code stripping and image freezing," presented at the Black Hat Briefings USA (Aug. 6, 2015). [Online]. Available: https://www.mulliner.org/security/codefreeze/ (cit. on pp. 7, 35).

[34] C. M. Geschke, "Global program optimizations," Ph.D. dissertation, Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA, USA, Oct. 1972. [Online]. Available: https://apps.dtic.mil/sti/citations/AD0762621 (cit. on pp. 7, 68).

[35] G. C. Necula, "Translation validation for an optimizing compiler," in *Proc. ACM SIGPLAN 2000 Conf. Programming Language Design and Implementation (PLDI '00)*, May 2000. DOI: 10.1145/349299.349314 (cit. on pp. 7, 73).

[36] Á. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto, "Survey of code-size reduction methods," *ACM Computing Surveys*, vol. 35, no. 3, Sep. 2003. DOI: 10.1145/937503.937504 (cit. on pp. 7, 95).

[37] G. F. Coulouris, J. M. Evans, and R. W. Mitchell, "Towards content-adressing in data bases," *The Computer Journal*, vol. 15, no. 2, May 1, 1972. DOI: 10.1093/comjnl/15.2.95 (cit. on p. 9).

[38] R. C. Merkle, "Method of providing digital signatures," U.S. Patent 4309569A, Jan. 5, 1982 (cit. on p. 9).

[39] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology (CRYPTO '87)*, ser. Lecture Notes in Computer Science, vol. 293, 1988. DOI: 10.1007/3-540-48184-2_32 (cit. on p. 9).

[40] [SW] L. Torvalds, *Git*, URL: https://git-scm.com/ (cit. on p. 9).

[41] J. Benet. "IPFS - content addressed, versioned, P2P file system." (2014), [Online]. Available: https://arxiv.org/abs/1407.3561 (cit. on p. 9).

[42] D. D. R. Barros, F. Horita, K. C. Silva, and I. S. Wiese, "A mining software repository extended cookbook: Lessons learned from a literature review," in *Brazilian Symp. Software Engineering (SBES '21)*, 2021. DOI: 10.1145/3474624.3474627 (cit. on p. 9).

[43] Google, *Remote caching - Bazel main*, June 15, 2021. [Online]. Available: https://docs.bazel.build/versions/main/remote-caching.html (visited on 11/05/2021) (cit. on p. 9).

[44] P. Chiusano, "Unison: A new distributed programming language," presented at the Strange Loop Conf. (Sep. 13–14, 2019). [Online]. Available: https://youtu.be/gCWtkvDQ2ZI (cit. on p. 9).

[45] A. Carle, K. D. Cooper, R. T. Hood, K. Kennedy, L. Torczon, and S. K. Warren, "A practical environment for scientific programming," *Computer*, vol. 20, no. 11, Nov. 1987. DOI: 10.1109/MC.1987.1663418 (cit. on p. 9).

[46] M. Burke and L. Torczon, "Interprocedural optimization: Eliminating unnecessary recompilation," *ACM Trans. Programming Languages and Systems*, vol. 15, no. 3, July 1993. DOI: 10.1145/169683.169678 (cit. on p. 9).

[47] P. W. Sathyanathan, W. He, and T. H. Tzen, "Incremental whole program optimization and compilation," in *2017 IEEE/ACM Int. Symp. Code Generation and Optimization (CGO)*, 2017. DOI: 10.1109/CGO.2017.7863742 (cit. on p. 9).

[48] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation and Optimization (CGO 2004)*, Mar. 2004. DOI: 10.1109/CGO.2004.1281665. [Online]. Available: https://llvm.org/pubs/2004-01-30-CGO-LLVM.pdf (cit. on pp. 9, 38, 57).

[49] M. Toman, "LLVM IR service for Fedora," M.S. thesis, Masaryk University Faculty of Informatics, Brno, Czech Republic, Aug. 2, 2013. [Online]. Available: https://is.muni.cz/th/n9bfn/?lang=en (cit. on p. 10).

[50] A. Denisov. "Exploring LLVM bitcode interactively." (Feb. 28, 2020), [Online]. Available: https://lowlevelbits.org/exploring-llvm-bitcode-interactively/ (visited on 10/08/2021) (cit. on p. 10).

[51] V. S. Adve, W. Dietz, J. Regehr, and J. Criswell. "ALLVM project." (2016), [Online]. Available: https://publish.illinois.edu/allvm-project/ (visited on 08/23/2020) (cit. on p. 10).

[52] [SW] W. Dietz, *allvm-tools*, VCS: https://github.com/allvm/allvm-tools, SWHID: ⟨swh:1:snp:dd2eefd26056cb7dbe5d8a756b386f3897271225⟩ (cit. on p. 10).

[53] P. Bowen-Huggett, "Toy programming demo of a repository for statically compiled programs," presented at the 2016 US LLVM Developers' Meeting (San Jose, CA, USA, Nov. 3–4, 2016). [Online]. Available: https://llvm.org/devmtg/2016-11/#talk22 (cit. on p. 10).

[54] P. Camp and R. Gallop, "Targeting a statically compiled program repository with LLVM," presented at the 2019 European LLVM Developers' Meeting (Brussels, Belgium, Apr. 8–9, 2019). [Online]. Available: https://youtu.be/mlQyEBDnDJE (cit. on p. 10).

[55] [SW] P. Camp and R. Gallop, *llvm-project-prepo*, SN Systems (Sony Interactive Entertainment), VCS: https://github.com/SNSystems/llvm-project-prepo, SWHID: ⟨swh:1:snp:1dca3f7c53437c19838ff75a9c3ee17b0c0f4db1⟩ (cit. on p. 10).

[56] [SW] SQLite Consortium, *SQLite*, URL: https://sqlite.org/ (cit. on pp. 11, 15).

[57] [SW] Facebook Database Engineering Team, *RocksDB: A Persistent Key-Value Store for Flash and RAM Storage*, VCS: https://github.com/facebook/rocksdb (cit. on p. 11).

[58] M.-J. O. Saarinen and J.-P. Aumasson, "The BLAKE2 cryptographic hash and message authentication code (MAC)," RFC 7693, Nov. 2015. DOI: 10.17487/RFC7693. [Online]. Available: https://www.rfc-editor.org/info/rfc7693 (visited on 08/24/2020) (cit. on p. 11).

[59] Protocol Labs. "CID (Content IDentifier) specification." (Sep. 8, 2020), [Online]. Available: https://github.com/multiformats/cid (cit. on p. 11).

[60] T. Bray, "The JavaScript object notation (JSON) data interchange format," RFC 8259, Dec. 2017. DOI: 10.17487/RFC8259. [Online]. Available: https://www.rfc-editor.org/info/rfc8259 (visited on 10/08/2021) (cit. on p. 11).

[61] A. Eubanks. "Opaque pointers." (July 14, 2021), [Online]. Available: https://llvm.org/docs/OpaquePointers.html (visited on 11/20/2021) (cit. on p. 13).

[62] [SW] WLLVM contributors, *WLLVM*, VCS: https://github.com/travitch/whole-program-llvm, SWHID: ⟨swh:1:snp:557567461d68164a0312240689371f18e15dd475⟩ (cit. on pp. 14, 25).

[63] [SW] Nixpkgs contributors, *Nixpkgs*, URL: https://nixos.org/nixpkgs/manual/, VCS: https://github.com/NixOS/nixpkgs, SWHID: ⟨swh:1:snp:da0e3e4a3eff6fb6370259fd2bdfcf932fa6ac69⟩ (cit. on p. 14).

[64] D. Vlasenko. "BusyBox: The Swiss Army knife of embedded Linux." (Apr. 11, 2008), [Online]. Available: https://busybox.net/about.html (visited on 07/28/2020) (cit. on p. 18).

[65] W. Mertens. "Coreutils: Single binary build." (June 21, 2016), [Online]. Available: https://github.com/NixOS/nixpkgs/pull/16406 (visited on 07/28/2020) (cit. on p. 18).

[66] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Computing*, vol. 1, no. 2, June 1972. DOI: 10.1137/0201010 (cit. on p. 23).

[67] *OpenWrt documentation*, The OpenWrt Project, ch. Table of Hardware: Full details. [Online]. Available: https://openwrt.org/toh/views/toh_extended_all (visited on 07/28/2020) (cit. on p. 24).

[68] T. Johnson, M. Amini, and X. D. Li, "ThinLTO: Scalable and incremental LTO," in *Proc. 2017 Int. Symp. Code Generation and Optimization (CGO)*, Feb. 4, 2017. DOI: 10.1109/CGO.2017.7863733 (cit. on p. 27).

[69] V. Agrawal, A. Dabral, T. Palit, Y. Shen, and M. Ferdman, "Architectural support for dynamic linking," in *Proc. 20th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, Mar. 14, 2015. DOI: 10.1145/2694344.2694392. [Online]. Available: http://compas.cs.stonybrook.edu/~mferdman/downloads.php/ASPLOS15_Architectural_Support_for_Dynamic_Linking.pdf (cit. on pp. 29, 35, 109).

[70] M. Auslander and M. Hopkins, "An overview of the PL.8 compiler," in *Proc. 1982 SIGPLAN Symp. Compiler Construction*, June 1, 1982. DOI: 10.1145/872726.806977 (cit. on p. 34).

[71] K. D. Cooper, K. Kennedy, and L. Torczon, "The impact of interprocedural analysis and optimization in the $\mathbf{R}^n$ programming environment," *ACM Trans. Programming Languages and Systems*, vol. 8, no. 4, Aug. 1, 1986. DOI: 10.1145/6465.6489 (cit. on p. 34).

[72] D. B. Orr, J. Bonn, J. Lepreau, and R. Mecklenburg, "Fast and flexible shared libraries," in *Proc. USENIX Summer 1993 Technical Conf.*, Apr. 1993. [Online]. Available: https://www.cs.utah.edu/flux/papers/shlibs.html (cit. on pp. 35, 109).

[73] M. N. Nelson and G. Hamilton, "High performance dynamic linking through caching," in *Proc. USENIX Summer 1993 Technical Conf.*, 1993. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/cinci93/nelson.html (cit. on p. 35).

[74] J. Jelínek, "Prelink," Red Hat, Inc., Tech. Rep., Mar. 4, 2004. [Online]. Available: https://people.redhat.com/jakub/prelink.pdf (visited on 08/24/2020) (cit. on p. 35).

[75] C. Jung, D.-K. Woo, K. Kim, and S.-S. Lim, "Performance characterization of prelinking and preloading for embedded systems," in *Proc. 7th ACM & IEEE Int. Conf. Embedded Software (EMSOFT '07)*, Sep. 2007. DOI: 10.1145/1289927.1289961 (cit. on p. 35).

[76] W. W. Ho, W.-C. Chang, and L. H. Leung, "Optimizing the performance of dynamically-linked programs," in *Proc. USENIX 1995 Technical Conference*, 1995. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/neworl/ho.html (cit. on p. 35).

[77] [SW] scut, *reducebind.c* Jan. 7, 2003, URL: https://packetstormsecurity.com/files/30760/reducebind.c.html(visited on 08/24/2020) (cit. on p. 35).

[78] [SW] B. Blackham, *CryoPID* 2005, VCS: https://github.com/maaziz/cryopid (cit. on p. 35).

[79] [SW] V. Reznic, *Statifier* 2016, URL: http://statifier.sourceforge.net/(visited on 08/24/2020) (cit. on p. 35).

[80] [SW] Ermine authors, *Ermine - Linux Portable Application Creator* 2018, URL: http://www.magicermine.com/(visited on 08/24/2020) (cit. on p. 35).

[81] "Code optimization with the IBM XL compilers on Power architectures," IBM Corporation, document number 317699, Dec. 13, 2018. [Online]. Available: https://www.ibm.com/support/pages/node/317699 (visited on 07/09/2020) (cit. on p. 38).

[82] *Intel® C++ Compiler 19.1 developer guide and reference*, Intel Corporation, ch. Interprocedural Optimization (LTO). [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/interprocedural-optimization-ipo.html (visited on 07/09/2020) (cit. on p. 38).

[83] S. Kell, D. P. Mulligan, and P. Sewell, "The missing link: Explaining ELF static linking, semantically," in *Proc. 2016 ACM SIGPLAN Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '16)*, 2016. DOI: 10.1145/2983990.2983996. [Online]. Available: https://www.cl.cam.ac.uk/~pes20/rems/papers/oopsla-elf-linking-2016.pdf (cit. on pp. 38, 39).

[84] U. Drepper. "How to write shared libraries." (Dec. 10, 2011), [Online]. Available: https://akkadia.org/drepper/dsohowto.pdf (visited on 07/09/2020) (cit. on p. 39).

[85] *Linux programmer's manual*, version 5.07, Linux man-pages project, Aug. 2, 2019, ch. ld.so(8). [Online]. Available: https://man7.org/linux/man-pages/man8/ld.so.8.html (cit. on p. 40).

[86] [SW] Python Software Foundation, *Python pyperf module* 2020, URL: https://pyperf.readthedocs.io/en/latest/, VCS: https://github.com/psf/pyperf (cit. on p. 57).

[87] [SW] Python Software Foundation, *The Python Performance Benchmark Suite* 2020, URL: https://pyperformance.readthedocs.io/, VCS: https://github.com/python/pyperformance (cit. on p. 58).

[88] CWE Content Team, *CWE-426: Untrusted search path*, in *Common Weakness Enumeration*, The MITRE Corporation, June 25, 2020. [Online]. Available: http://cwe.mitre.org/data/definitions/426.html (visited on 07/26/2020) (cit. on pp. 64, 109).

[89] CWE Content Team, *CWE-427: Uncontrolled search path element*, in *Common Weakness Enumeration*, The MITRE Corporation, June 25, 2020. [Online]. Available: http://cwe.mitre.org/data/definitions/427.html (visited on 07/26/2020) (cit. on pp. 64, 109).

[90] *CVE-2017-4921*, in *National Vulnerability Database*, affects VMware vCenter Server, National Institute of Standards and Technology, Oct. 2, 2019. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-4921 (visited on 07/26/2020) (cit. on pp. 64, 109).

[91] *CVE-2017-7494*, in *National Vulnerability Database*, affects Samba, National Institute of Standards and Technology, Oct. 21, 2018. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-7494 (visited on 07/26/2020) (cit. on pp. 64, 109).

[92] *CVE-2019-4094*, in *National Vulnerability Database*, affects IBM DB2, National Institute of Standards and Technology, Oct. 9, 2019. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-4094 (visited on 07/26/2020) (cit. on pp. 64, 109).

[93] *CVE-2019-8801*, in *National Vulnerability Database*, affects Apple iTunes installer, National Institute of Standards and Technology, Dec. 30, 2019. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-8801 (visited on 07/26/2020) (cit. on pp. 64, 109).

[94] *CVE-2019-14271*, in *National Vulnerability Database*, affects Docker, National Institute of Standards and Technology, Aug. 28, 2019. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-14271 (visited on 07/26/2020) (cit. on pp. 64, 109).

[95] *CVE-2019-19726*, in *National Vulnerability Database*, affects OpenBSD, National Institute of Standards and Technology, Dec. 27, 2019. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-19726 (visited on 07/26/2020) (cit. on pp. 64, 109).

[96] *CVE-2020-1458*, in *National Vulnerability Database*, affects Microsoft Office, National Institute of Standards and Technology, July 20, 2020. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2020-1458 (visited on 07/26/2020) (cit. on pp. 64, 109).

[97] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity, Principles, implementations, and applications," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Nov. 7, 2005. DOI: 10.1145/1102120.1102165. [Online]. Available: https://www.microsoft.com/en-us/research/publication/control-flow-integrity/ (cit. on p. 64).

[98] N. E. Johnson, "Code size optimization for embedded processors," University of Cambridge Computer Laboratory, Cambridge, UK, Technical Report UCAM-CL-TR-607, Nov. 2004. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-607.pdf (cit. on pp. 66, 69).

[99] A. Dreweke, M. Wörlein, I. Fischer, D. Schell, T. Meinl, and M. Philippsen, "Graph-based procedural abstraction," in *Int. Symp. Code Generation and Optimization (CGO'07)*, Mar. 2007. DOI: 10.1109/CGO.2007.14 (cit. on pp. 66, 69).

[100] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, "Alive2: Bounded translation validation for LLVM," in *Proc. 42nd ACM SIGPLAN Int. Conf. Programming Language Design and Implementation (PLDI '21)*, 2021. DOI: 10.1145/3453483.345430. [Online]. Available: https://www.cs.utah.edu/~regehr/alive2-pldi21.pdf (cit. on p. 67).

[101] T. A. Standish, D. F. Kibler, and J. M. Neighbors, "Improving and refining programs by program manipulation," in *ACM '76: Proc. 1976 Annual Conf.*, 1976. DOI: 10.1145/800191.805652 (cit. on p. 68).

[102] W. G. Wong. "Optimization breeds application efficiency," Electronic Design. (Sep. 2003), [Online]. Available: https://www.electronicdesign.com/technologies/embedded-revolution/article/21768560/optimization-breeds-application-efficiency (visited on 07/14/2020) (cit. on p. 68).

[103] "Raisonance CodeCompressor for 8051, ST5 and CoolRISC," MicroController Pros Corporation. (2003), [Online]. Available: https://microcontrollershop.com/raisonance_codecompressor.php (visited on 07/14/2020) (cit. on p. 68).

[104] M. Mondal, C. K. Roy, and K. A. Schneider, "A survey on clone refactoring and tracking," *J. Systems and Software*, vol. 159, Jan. 2020, ISSN: 0164-1212. DOI: 10.1016/j.jss.2019.110429 (cit. on pp. 68–70, 105).

[105] M. J. Zastre, "Compacting object code via parameterized procedural abstraction," Ph.D. dissertation, Department of Computer Science, University of Victoria, Victoria, BC, Canada, 1995. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.4235 (cit. on p. 68).

[106] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Trans. Programming Languages and Systems*, vol. 22, no. 2, Mar. 2000. DOI: 10.1145/349214.349233 (cit. on p. 68).

[107] C. W. Fraser, E. W. Myers, and A. L. Wendt, "Analyzing and compressing assembly code," in *Proc. 1984 SIGPLAN Symp. Compiler Construction*, June 1984. DOI: 10.1145/502874.502886 (cit. on p. 68).

[108] K. D. Cooper and N. McIntosh, "Enhanced code compression for embedded RISC processors," in *Proc. ACM SIGPLAN 1999 Conf. Programming Language Design and Implementation (PLDI '99)*, May 1999. DOI: 10.1145/301631.301655 (cit. on p. 68).

[109] C. Castelluccia, W. Dabbous, and S. O'Malley, "Generating efficient protocol code from an abstract specification," in *IEEE/ACM Trans. Networking*, vol. 5, 4, Aug. 1997. DOI: 10.1109/90.649465 (cit. on p. 68).

[110] R. Komondoor and S. Horwitz, "Effective, automatic procedure extraction," in *11th IEEE Int. Workshop Program Comprehension*, 2003. DOI: 10.1109/WPC.2003.1199187 (cit. on pp. 68, 70).

[111] J. Lau, S. Schoenmackers, T. Sherwood, and B. Calder, "Reducing code size with echo instructions," in *CASES '03: Proc. 2003 Int. Conf. Compilers, Architecture and Synthesis for Embedded Systems*, Oct. 2003. DOI: 10.1145/951710.951724. [Online]. Available: https://cseweb.ucsd.edu/~calder/abstracts/CASES-03-ECHO.html (cit. on pp. 68, 69, 95, 107).

[112] R. Komondoor and S. Horwitz, "Semantics-preserving procedure extraction," in *Proc. 27th ACM SIGPLAN–SIGACT Symp. Principles of Programming Languages (POPL '00)*, Jan. 2000. DOI: 10.1145/325694.325713 (cit. on pp. 69, 70).

[113] S. Y.-H. Liao, S. Devadas, and K. Keutzer, "Code density optimization for embedded DSP processors using data compression techniques," in *Proc. 16th Conf. Advanced Research in VLSI (ARVLSI)*, 1995. DOI: 10.1109/ARVLSI.1995.515626 (cit. on pp. 69, 107).

[114] S. Y.-H. Liao, S. Devadas, and K. Keutzer, "A text-compression-based method for code size minimization in embedded systems," *ACM Trans. Design Automation of Electronic Systems*, vol. 4, no. 1, Jan. 1999. DOI: 10.1145/298865.298867 (cit. on pp. 69, 107).

[115] C. W. Fraser, "An instruction for direct interpretation of LZ77-compressed programs," Microsoft Research, Tech. Rep. MSR-TR-2002-90, Sep. 2002. [Online]. Available: https://www.microsoft.com/en-us/research/publication/an-instruction-for-direct-interpretation-of-lz77-compressed-programs/ (cit. on pp. 69, 107).

[116] P. Brisk, J. Macbeth, A. Nahapetian, and M. Sarrafzadeh, "A dictionary construction technique for code compression systems with echo instructions," in *Proc. 2005 ACM SIGPLAN/SIGBED Conf. Languages, Compilers, and Tools for Embedded Systems (LCTES '05)*, 2005. DOI: 10.1145/1070891.1065926 (cit. on pp. 69, 107).

[117] C. W. Fraser, "An instruction for direct interpretation of LZ77-compressed programs," *J. Software: Practice and Experience*, vol. 36, no. 4, Apr. 10, 2006. DOI: 10.1002/spe.702 (cit. on pp. 69, 107).

[118] I. Stubdal, A. Karaduman, and H. Amano, "Code compression with split echo instructions," *IEICE Trans. Information and Systems*, vol. E92.D, no. 9, 2009. DOI: 10.1587/transinf.E92.D.1650 (cit. on pp. 69, 107).

[119] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *SAS 2001: Static Analysis*, ser. Lecture Notes in Computer Science, vol. 2126, July 2001. DOI: 10.1007/3-540-47764-0_3. [Online]. Available: https://research.cs.wisc.edu/wpis/papers/sas01.pdf (cit. on pp. 70, 72).

[120] R. Ettinger, S. Tyszberowicz, and S. Menaia, "Efficient method extraction for automatic elimination of type-3 clones," in *2017 IEEE 24th Int. Conf. Software Analysis, Evolution and Reengineering*, 2017. DOI: 10.1109/SANER.2017.7884633 (cit. on p. 70).

[121] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Trans. Software Engineering*, vol. 41, no. 11, 2015. DOI: 10.1109/TSE.2015.2448531 (cit. on p. 70).

[122] X. Li, X. Su, P. Ma, and T. Wang, "Refactoring structure semantics similar clones combining standardization with metrics," in *Proc. Int. Conf. Soft Computing Techniques and Engineering Application*, ser. Advances in Intelligent Systems and Computing, vol. 250, 2014. DOI: 10.1007/978-81-322-1695-7_41 (cit. on p. 70).

[123] D. Mosberger, L. L. Peterson, and S. O'Malley, "Protocol latency: MIPS and reality," Department of Computer Science, University of Arizona, Tucson, AZ, USA, Tech. Rep. 95-02, 1995. [Online]. Available: https://www.cs.arizona.edu/sites/default/files/TR95-02.pdf (cit. on pp. 70, 101).

[124] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, "Helping Johnny to analyze malware: A usability-optimized decompiler and malware analysis user study," in *2016 IEEE Symp. Security and Privacy (SP)*, 2016. DOI: 10.1109/SP.2016.18 (cit. on p. 70).

[125] Y. Liu, Y. Zhao, L. Zhang, and K. Liu, "Subgraph isomorphism based intrinsic function reduction in decompilation," *J. Software Engineering and Applications*, vol. 9, 2016. DOI: 10.4236/jsea.2016.93007 (cit. on p. 70).

[126] T. J. K. E. von Koch, B. Franke, P. Bhandarkar, and A. Dasgupta, "Exploiting function similarity for code size reduction," in *Proc. 2014 SIGPLAN/SIGBED Conf. Languages, Compilers and Tools for Embedded Systems (LCTES '14)*, June 2014. DOI: 10.1145/2597809.2597811 (cit. on pp. 71, 113).

[127] T. J. K. E. von Koch and P. Bhandarkar, "Code size reduction using similar function merging," presented at the 2013 LLVM Developers' Meeting (San Francisco, CA, USA, Nov. 6–7, 2013). [Online]. Available: https://llvm.org/devmtg/2013-11/#talk3 (cit. on pp. 71, 113).

[128] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Function merging by sequence alignment," in *2019 IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO)*, 2019. DOI: 10.1109/CGO.2019.8661174. [Online]. Available: http://homepages.inf.ed.ac.uk/hleather/publications/2019_functionmergesequencealign_cgo2019.pdf (cit. on pp. 71, 113).

[129] B. Marks, "Compilation to compact code," *IBM J. Research and Development*, vol. 24, no. 6, Nov. 1980. DOI: 10.1147/rd.246.0684 (cit. on pp. 72, 110).

[130] G. Stitt and F. Vahid, "New decompilation techniques for binary-level co-processor generation," in *ICCAD-2005, Proc. 2005 Int. Conf. Computer-Aided Design*, Nov. 2005. DOI: 10.1109/ICCAD.2005.1560127. [Online]. Available: https://www.cs.ucr.edu/~vahid/pubs/iccad05_bp.pdf (cit. on p. 72).

[131] E.-W. Hu, B. Su, and J. Wang, "Instruction level loop de-optimization, Loop rerolling and software de-pipelining," in *Computer and Information Science 2015*, ser. Studies in Computational Intelligence, vol. 614, 2016. DOI: 10.1007/978-3-319-23467-0_15 (cit. on p. 72).

[132] X. Yin, J. Knight, and W. Weimer, "Exploiting refactoring in formal verification," in *Proc. 2009 IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN)*, 2009. DOI: 10.1109/DSN.2009.5270355 (cit. on p. 72).

[133] B. Su, S. Ding, and J. Xia, "URPR - an extension of URCR for software pipelining," in *Proc. 19th Annu. Workshop Microprogramming (MICRO)*, Dec. 1986. DOI: 10.1145/19551.19541 (cit. on p. 72).

[134] C. W. Keßler, "Symbolic array data flow analysis and pattern recognition in numerical codes," in *Programming Environments for Massively Parallel Distributed Systems*, ser. Monte Verità, 1994. DOI: 10.1007/978-3-0348-8534-8_6 (cit. on p. 72).

[135] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," School of Computing, Queen's University, Kingston, Ontario, CA, Tech. Rep. 2007-541, Sep. 26, 2007. [Online]. Available: https://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf (cit. on p. 72).

[136] T. A. D. Henderson and A. Podgurski, "Rethinking dependence clones," in *11th IEEE Int. Workshop Software Clones (IWSC)*, 2017. DOI: 10.1109/IWSC.2017.7880512 (cit. on p. 72).

[137] A. Sheneamer and J. Kalita, "Semantic code detection using machine learning," in *2016 15th IEEE Int. Conf. Machine Learning and Applications (ICMLA)*, 2016. DOI: 10.1109/ICMLA.2016.0185 (cit. on p. 72).

[138] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *ISSTA '09: Proc. 18th Int. Symp. Software Testing and Analysis*, July 2009. DOI: 10.1145/1572272.1572283 (cit. on p. 72).

[139] D. Kong, X. Su, S. Wu, T. Wang, and P. Ma, "Detect functionally equivalent code fragments via k-nearest neighbor algorithm," in *2012 IEEE 5th Int. Conf. Advanced Computational Intelligence (ICACI)*, 2012. DOI: 10.1109/ICACI.2012.6463128 (cit. on p. 73).

[140] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: Memory comparison-based clone detector," in *ICSE '11: Proc. 33rd Int. Conf. Software Engineering*, 2011. DOI: 10.1145/1985793.1985835 (cit. on p. 73).

[141] D. E. Krutz, S. A. Malachowsky, and E. Shihab, "Examining the effectiveness of using concolic analysis to detect code clones," in *SAC '15: Proc. 30th Annual ACM Symp. Applied Computing*, Apr. 2015. DOI: 10.1145/2695664.2695929 (cit. on p. 73).

[142] D. E. Krutz and E. Shihab, "CCCD: Concolic code clone detection," in *2013 20th Working Conf. Reverse Engineering (WCRE)*, 2013. DOI: 10.1109/WCRE.2013.6671332 (cit. on p. 73).

[143] S. Horwitz, J. Prins, and T. Reps, "On the adequacy of program dependence graphs for representing programs," in *POPL '88: Proc. 15th ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages*, Jan. 1988. DOI: 10.1145/73560.73573 (cit. on p. 73).

[144] S. Dasgupta, S. Dinesh, D. Venkatesh, V. S. Adve, and C. W. Fletcher, "Scalable validation of binary lifters," in *PLDI 2020: Proc. 41st ACM SIGPLAN Int. Conf. Programming Language Design and Implementation*, June 2020. DOI: 10.1145/3385412.3385964. [Online]. Available: https://sdasgup3.github.io/files/pldi_2020.pdf (cit. on p. 73).

[145] D. Jackson and D. A. Ladd, "Semantic diff: A tool for summarizing the effects of modifications," in *Proc. 1994 Int. Conf. Software Maintenance (ICSM)*, 1994. DOI: 10.1109/ICSM.1994.336770 (cit. on p. 73).

[146] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *POPL '09: Proc. 36th Annual ACM SIGPLAN–SIGACT Symp. Principles of Programming Languages*, 2009. DOI: 10.1145/1480881.1480915. [Online]. Available: http://www.cs.cornell.edu/~ross/publications/eqsat/ (cit. on p. 73).

[147] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," *Logical Methods in Computer Science*, vol. 7, no. 1, Mar. 28, 2011. DOI: 10.2168/LMCS-7(1:10)2011. [Online]. Available: http://www.cs.cornell.edu/~ross/publications/eqsat/ (cit. on p. 73).

[148] M. Stepp, R. Tate, and S. Lerner, "Equality-based translation validator for LLVM," in *Computer Aided Verification, CAV 2011*, ser. Lecture Notes in Computer Science, vol. 6806, 2011. DOI: 10.1007/978-3-642-22110-1_59 (cit. on p. 73).

[149] J.-B. Tristan, P. Govereau, and G. Morrisett, "Evaluating value-graph translation validation for LLVM," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Design and Implementation (PLDI)*, June 2011. DOI: 10.1145/1993498.1993533. [Online]. Available: http://nrs.harvard.edu/urn-3:HUL.InstRepos:4762396 (cit. on p. 73).

[150] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Proc. 18th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2013. DOI: 10.1145/2499368.2451150 (cit. on pp. 74, 112).

[151] R. Sasnauskas, Y. Chen, P. Collingbourne, *et al.* "Souper: A synthesizing superoptimizer." (Nov. 13, 2017), [Online]. Available: https://arxiv.org/abs/1711.04422 (cit. on pp. 74, 112).

[152] R. Sharma, E. Schkufza, B. R. Churchill, and A. Aiken, "Data-driven equivalence checking," in *Proc. 2013 ACM SIGPLAN Int. Conf. Object Oriented Program. Systems Lang. Applications (OOPSLA)*, Oct. 2013. DOI: 10.1145/2544173.2509509 (cit. on p. 74).

[153] T. Kasampalis, D. Park, Z. Lin, V. S. Adve, and G. Roşu, "Language-parametric compiler validation with application to LLVM," in *Proc. 26th ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, Apr. 2021. DOI: 10.1145/3445814.3446751. [Online]. Available: https://daejunpark.github.io/asplos21.pdf (cit. on p. 74).

[154] S. Gupta, A. Saxena, A. Mahajan, and S. Bansal, "Effective use of SMT solvers for program equivalence checking through invariant-sketching and query-decomposition," in *Theory and Applications of Satisfiability Testing, SAT 2018*, ser. Lecture Notes in Computer Science, vol. 10929, 2018. DOI: 10.1007/978-3-319-94144-8_22 (cit. on p. 74).

[155] B. R. Churchill, "Blackbox equivalence checking of program optimizations," Ph.D. dissertation, Computer Science Department, Stanford University, Stanford, CA, USA, June 2019. [Online]. Available: https://theory.stanford.edu/~aiken/publications/theses/churchill.pdf (cit. on p. 74).

[156] A. Zaks and A. Pnueli, "CoVaC: Compiler validation by program analysis of the cross-product," in *Formal Methods, FM 2008*, ser. Lecture Notes in Computer Science, vol. 5014, 2008. DOI: 10.1007/978-3-540-68237-0_5. [Online]. Available: https://llvm.org/pubs/2008-05-CoVaC.html (cit. on p. 74).

[157] R. Allen and T. Kennedy, *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, Sep. 26, 2001, ISBN: 978-0-08-051324-9 (cit. on pp. 75, 77).

[158] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, Oct. 1991. DOI: 10.1145/115372.115320 (cit. on p. 77).

[159] S. Yvon and M. Feeley, "A small Scheme VM, compiler, and REPL in 4K," in *Proc. 13th ACM SIGPLAN Int. Workshop Virtual Machines and Intermediate Languages*, Oct. 2021. DOI: 10.1145/3486606.3486783. [Online]. Available: http://www-labs.iro.umontreal.ca/~feeley/papers/YvonFeeleyVMIL21.pdf (cit. on p. 94).

[160] B. De Bus, D. Kästner, D. Chanet, L. Van Put, and B. De Sutter, "Post-pass compaction techniques," *Comm. ACM*, vol. 46, no. 8, Aug. 2003. DOI: 10.1145/859670.859696 (cit. on p. 95).

[161] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere, "Automated reduction of the memory footprint of the Linux kernel," *ACM Trans. Embedded Computing*, vol. 6, no. 4, Sep. 2007. DOI: 10.1145/1274858.1274861 (cit. on pp. 95, 110).

[162] M. Latendresse and R. van de Wiel. "The code compression bibliography." (Nov. 15, 2004), [Online]. Available: http://www.iro.umontreal.ca/~latendre/codeCompression/ (cit. on p. 96).

[163] P. Falke, "The Deep Space Network - a technology case study and what improvements to the Deep Space Network are needed to support crewed missions to Mars?" M.S. thesis, Department of Telecommunications, State University of New York, Polytechnic Institute, Utica, NY, USA, May 2017. [Online]. Available: https://soar.suny.edu/bitstream/handle/20.500.12648/1087/falke-2017-thesis-final.pdf?sequence=1 (cit. on p. 97).

[164] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proc. 2018 ACM SIGSAC Conf. Computer and Communication Security (CCS)*, Jan. 2018. DOI: 10.1145/3243734.3243838 (cit. on pp. 97, 101, 105).

[165] Y. Chen, S. Sun, T. Lan, and G. Venkataramani, "TOSS: Tailoring online server systems through binary feature customization," in *Proc. 2018 Workshop Forming Ecosystem Around Softw. Transformation (FEAST)*, Jan. 2018. DOI: 10.1145/3273045.3273048. [Online]. Available: https://www2.seas.gwu.edu/~tlan/papers/TOSS_FEAST_2018.pdf (cit. on pp. 97, 101).

[166] M. D. Brown and S. Pande, "CARVE: Practical security-focused software debloating using simple feature set mappings," in *Proc. 3rd ACM Workshop Forming an Ecosystem Around Software Transformation (FEAST)*, Nov. 2019. DOI: 10.1145/3338502.3359764 (cit. on pp. 97, 101, 103).

[167] Office of Naval Research, "Late-stage software customization and complexity reduction S&T for legacy naval systems," Broad Agency Announcement N00014-17-S-B010, Feb. 28, 2017. [Online]. Available: https://www.onr.navy.mil/work-with-us/funding-opportunities/announcements (visited on 11/22/2021) (cit. on p. 97).

[168] H. Falk and H. Kotthaus, "WCET-driven cache-aware code positioning," in *Proc. Int. Conf. Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2011. DOI: 10.1145/2038698.2038722 (cit. on p. 97).

[169] G. Ottoni and B. Maher, "Optimizing function placement for large-scale data-center applications," in *Proc. 2017 Int. Symp. Code Generation and Optimization (CGO)*, 2017. DOI: 10.1109/CGO.2017.7863743 (cit. on p. 97).

[170] P. Stratis and A. Rajan, "Speeding up test execution with increased cache locality," *Software Testing, Verification and Reliability*, vol. 28, no. 5, e1671, 2018. DOI: 10.1002/stvr.1671 (cit. on p. 97).

[171] M. Trivellato. "WebAssembly load times and performance." (Sep. 17, 2018), [Online]. Available: https://blog.unity.com/technology/webassembly-load-times-and-performance (visited on 11/22/2021) (cit. on p. 98).

[172] N. Fitzgerald and A. Crichton. "Shrinking .wasm code size." (2018), [Online]. Available: https://rustwasm.github.io/docs/book/reference/code-size.html (visited on 11/22/2021) (cit. on p. 98).

[173] P. Kogge, K. Bergman, S. Borkar, *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA, TR 2008-13, Sep. 28, 2008. [Online]. Available: http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf (cit. on p. 98).

[174] A. Boroumand, S. Ghose, Y. Kim, *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proc. 23rd Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018. DOI: 10.1145/3173162.3173177 (cit. on p. 98).

[175] D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller, "Characterizing the energy consumption of data transfers and arithmetic operations on x86-64 processors," in *Proc. 1st Int. Green Computing Conf. (IGCC)*, 2010. DOI: 10.1109/GREENCOMP.2010.5598316. [Online]. Available: https://tu-dresden.de/zih/forschung/ressourcen/dateien/abgeschlossene-projekte/benchit/2010_IGCC_authors_version.pdf?lang=en (cit. on p. 98).

[176] E. Vasilakis, "An instruction level energy characterization of ARM processors," Department of Computer Science, University of Crete, Tech. Rep. FORTH-ICS/TR-450, Mar. 2015. [Online]. Available: https://projects.ics.forth.gr/carv/greenvm/files/tr450.pdf (cit. on p. 98).

[177] C. Jie, I. Loi, L. Benini, and D. Rossi, "Energy-efficient two-level instruction cache design for an ultra-low-power multi-core cluster," in *Proc. 2020 Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2020. DOI: 10.23919/DATE48585.2020.9116212 (cit. on p. 98).

[178] M. A. Kozuch and A. Wolfe, "Compression of embedded system programs," in *Proc. 1994 IEEE Int. Conf. Computer Design (ICCD)*, Oct. 1994. DOI: 10.1109/ICCD.1994.331903 (cit. on pp. 99, 106).

[179] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security & Privacy*, vol. 2007, no. 02, 2007. DOI: 10.1109/MSP.2007.48 (cit. on p. 99).

[180] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: Size does matter in Turing-complete return-oriented programming," in *6th USENIX Workshop on Offensive Technologies (WOOT 12)*, Aug. 2012. [Online]. Available: https://www.usenix.org/conference/woot12/workshop-program/presentation/homescu (cit. on p. 102).

[181] M. D. Brown and S. Pande, "Is less *really* more? Towards better metrics for measuring security improvements realized through software debloating," in *12th USENIX Workshop Cyber Security Experimentation and Test (CSET)*, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/cset19/presentation/brown (cit. on p. 102).

[182] SizeCoding contributors. "Case studies." (2020), [Online]. Available: http://www.sizecoding.org/wiki/Case_Studies (visited on 07/18/2020) (cit. on p. 103).

[183] pouët.net contributors. "Prodlist," pouët.net. (2020), [Online]. Available: http://www.pouet.net/prodlist.php?order=views&type%5B0%5D=64k&page=1 (visited on 07/18/2020) (cit. on p. 103).

[184] D. Wood, "Polymorphisation: Improving Rust compilation times through intelligent monomorphisation," M.S. thesis, School of Computing Science, University of Glasgow, Glasgow, Scotland, Apr. 15, 2020. [Online]. Available: https://davidtw.co/media/masters_dissertation.pdf (cit. on p. 103).

[185] R. Queiroz, L. Passos, M. T. Valente, C. Hunsen, S. Apel, and K. Czernecki, "The shape of feature code: An analysis of twenty C-preprocessor-based systems," *Software & Systems Modeling*, vol. 16, 2017. DOI: 10.1007/s10270-015-0483-z (cit. on p. 103).

[186] D. Santos and C. N. Sant'Anna, "How does feature dependency affect configurable system comprehensibility?" In *2019 IEEE/ACM 27th Int. Conf. Program Comprehension (ICPC)*, 2019. DOI: 10.1109/ICPC.2019.00016 (cit. on p. 103).

[187] W. Fenske, J. Krüger, M. Kanyshkova, and S. Schulze, "`#Ifdef` directives and program comprehension: The dilemma between correctness and preference," in *2020 IEEE Int. Conf. Software Maintenance and Evolution (ICSME)*, 2020. DOI: 10.1109/ICSME46990.2020.00033 (cit. on p. 103).

[188] G. Malecha, A. Gehani, and N. Shankar, "Automated software winnowing," in *Proc. 30th Annual ACM Symp. Applied Computing (SAC)*, Apr. 2015. DOI: 10.1145/2695664.2695751. [Online]. Available: http://www.csl.sri.com/users/gehani/papers/SAC-2015.Winnow.pdf (cit. on p. 104).

[189] H. Sharif, M. Abubakar, A. Gehani, and F. M. Zaffar, "TRIMMER: Application specialization for code debloating," in *Proc. 2018 33rd ACM/IEEE Int. Conf. Automated Software Engineering (ASE)*, 2018. DOI: 10.1145/3238147.3238160. [Online]. Available: http://www.csl.sri.com/users/gehani/papers/ASE-2018.Trimmer.pdf (cit. on p. 104).

[190] A. A. Ahmad, A. R. Noor, H. Sharif, *et al.*, "TRIMMER: An automated system for configuration-based software debloating," *IEEE Trans. Softw. Eng.*, July 8, 2021. DOI: 10.1109/TSE.2021.3095716 (cit. on p. 104).

[191] C. Smowton and S. Hand, "`make world`," in *Proc. 13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, 2011. [Online]. Available: https://www.usenix.org/legacy/events/hotos11/tech/final_files/Smowton.pdf (cit. on p. 104).

[192] H.-C. Kuo, J. Chen, S. Mohan, and T. Xu, "Set the configuration for the heart of the OS: On the practicality of operating system kernel debloating," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 1, Mar. 2020. DOI: 10.1145/3379469. [Online]. Available: https://sibin.github.io/papers/2020_SIGMETRICS_Cozart_HsuanChiKuo.pdf (cit. on p. 104).

[193] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: Quantifying the security benefits of debloating web applications," in *Proc. 28th USENIX Security Symp.*, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/azad (cit. on p. 104).

[194] J. Landsborough, S. Harding, and S. Fugate, "Removing the kitchen sink from software," in *GECCO'15: Companion Publication 2015 Genetic and Evolutionary Computation Conf.*, July 2015. DOI: 10.1145/2739482.2768424 (cit. on pp. 104, 105).

[195] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "RAZOR: A framework for post-deployment software debloating," in *28th USENIX Security Symposium (USENIX Security 19)*, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/qian (cit. on p. 104).

[196] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, Feb. 2002. DOI: 10.1109/32.988498. [Online]. Available: https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/delta-debugging.pdf (cit. on p. 104).

[197] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection," *IEEE Access*, vol. 7, May 22, 2019, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2918202 (cit. on p. 105).

[198] J. Bunda, D. Fussell, W. C. Athas, and R. Jenevein, "16-bit vs. 32-bit instructions for pipelined microprocessors," in *Proc. 20th Annual Int. Symp. on Computer Architecture (ISCA '93)*, May 1993. DOI: 10.1145/173682.165159 (cit. on p. 105).

[199] V. M. Weaver and S. A. McKee, "Code density concerns for new architectures," in *2009 IEEE Int. Conf. Computer Design (ICCD)*, 2009. DOI: 10.1109/ICCD.2009.5413117. [Online]. Available: http://web.eece.maine.edu/~vweaver/papers/iccd09/ (cit. on p. 105).

[200] A. S. Waterman, "Design of the RISC-V instruction set architecture," Ph.D. dissertation, EECS Department, University of California, Berkeley, CA, USA, Jan. 3, 2016. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html (cit. on pp. 105, 106).

[201] V. M. Weaver. "ll: Exploring the limits of code density." (May 26, 2017), [Online]. Available: http://web.eece.maine.edu/~vweaver/papers/iccd09/ll_document.pdf (cit. on p. 105).

[202] H. Lozano and M. Ito, "Increasing the code density of embedded RISC applications," in *2016 IEEE 19th Int. Symp. Real-Time Distributed Computing (ISORC)*, 2016. DOI: 10.1109/ISORC.2016.33 (cit. on p. 106).

[203] K. D. Kissell, "MIPS16: High-density MIPS for the embedded market," Silicon Graphics MIPS Group, 1997 (cit. on p. 106).

[204] *ARM7TDMI technical reference manual*, version 3, ARM Limited (cit. on p. 106).

[205] A. S. Tanenbaum, "Implications of structured programming for machine architecture," *Communications of the ACM*, vol. 21, no. 3, Mar. 1978. [Online]. Available: https://cs.vu.nl/~ast/Publications/Papers/cacm-1978.pdf (cit. on p. 106).

[206] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," *IEEE Transactions on Computers*, vol. 37, no. 8, Aug. 1988. DOI: 10.1109/12.2247 (cit. on p. 106).

[207] S. Sardashti, A. Arelakis, P. Stenström, and D. A. Wood, "A primer on compression in the memory hierarchy," *Synthesis Lectures on Computer Architecture*, vol. 10, no. 5, 2015. DOI: 10.2200/S00683ED1V01Y201511CAC036 (cit. on p. 106).

[208] S. Mittal and J. S. Vetter, "A survey of architectural approaches for data compression in cache and main memory systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 27, no. 5, May 1, 2016. DOI: 10.1109/TPDS.2015.2435788 (cit. on p. 106).

[209] D. R. Carvalho and A. Seznec, "Understanding cache compression," *ACM Trans. Architecture and Code Optimization*, vol. 18, no. 3, June 2021. DOI: 10.1145/3457207 (cit. on p. 106).

[210] I.-C. K. Chen, P. L. Bird, and T. N. Mudge, "The impact of instruction compression on i-cache performance," University of Michigan EECS Department, Technical Report CSE-TR-330-97, 1997. [Online]. Available: http://eecs.umich.edu/techreports/cse/97/CSE-TR-330-97.pdf (cit. on p. 106).

[211] L. Benini, A. Macii, E. Macii, and M. Poncino, "Selective instruction compression for memory energy reduction in embedded systems," in *ISLPED '99: Proc. 1999 Int. Symp. Low Power Electronics and Design*, Aug. 1999. DOI: 10.1145/313817.313927 (cit. on p. 106).

[212] E. W. Netto, R. Azevedo, P. Centoducatte, and G. Araujo, "Multi-profile based code compression," in *Proc. 41st Design Automation Conf.*, July 2004. DOI: 10.1145/996566.996635 (cit. on p. 106).

[213] K. Shrivastava and P. Mishra, "Dual code compression for embedded systems," in *2011 24th Int. Conf. VLSI Design*, Jan. 2011. DOI: 10.1109/VLSID.2011.13 (cit. on p. 106).

[214] H. Hajimiri, K. Rahmani, and P. Mishra, "Synergistic integration of dynamic cache reconfiguration and code compression in embedded systems," in *2011 Int. Green Computing Conf. and Workshops*, July 2011. DOI: 10.1109/IGCC.2011.6008580 (cit. on p. 106).

[215] T. M. Kemp, R. K. Montoye, J. D. Harper, J. D. Palmer, and D. J. Auerbach, "A decompression core for PowerPC," *IBM J. Res. Development*, vol. 42, no. 6, Nov. 1998. DOI: 10.1147/rd.426.0807 (cit. on pp. 106, 107).

[216] A. Orpaz and S. Weiss, "A study of CodePack: Optimizing embedded code space," in *Proc. 10th Int. Symp. Hardware/Softw. Codesign (CODES)*, 2002. DOI: 10.1145/774789.774811 (cit. on pp. 106, 107).

[217] H. Lekatsas and W. Wolf, "SAMC: A code compression algorithm for embedded processors," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 12, Dec. 1999. DOI: 10.1109/43.811316 (cit. on p. 106).

[218] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture," in *Proc. 25th Annual Symp. Microarchitecture (MICRO 25)*, Oct. 1994. DOI: 10.1109/ICCD.1994.331903 (cit. on pp. 106, 107).

[219] M. Breternitz Jr. and R. Smith, "Enhanced compression techniques to simplify program decompression and execution," in *Proc. Int. Conf. Computer Design VLSI in Computers and Processors*, Oct. 1997. DOI: 10.1109/ICCD.1997.628865 (cit. on pp. 106, 107).

[220] M. Ros and P. Sutton, "A Hamming distance based VLIW/EPIC code compression technique," in *CASES '04: Proc. 2004 Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, Sep. 2004. DOI: 10.1145/1023833.1023853 (cit. on p. 106).

[221] M. Thuresson and P. Stenstrom, "Evaluation of extended dictionary-based static code compression schemes," in *CF '05: Proc. 2nd Conf. Computing Frontiers*, May 2005. DOI: 10.1145/1062261.1062278 (cit. on p. 106).

[222] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain, "Code compression based on operand factorization," in *Proc. 31st Annu. ACM/IEEE Int. Symp. Microarchitecture*, Dec. 1998. DOI: 10.1109/MICRO.1998.742781. [Online]. Available: https://computersystemslaboratory.files.wordpress.com/2015/02/araujo1998micro.pdf (cit. on p. 106).

[223] Y.-L. Jeang, J.-W. Hsieh, and Y.-Z. Lin, "An efficient instruction compression/decompression system based on field partitioning," in *48th Midwest Symp. Circuits and Systems*, Aug. 2005. DOI: 10.1109/MWSCAS.2005.1594495 (cit. on p. 106).

[224] E. C. R. Hehner, "Special feature: Computer design to minimize memory requirements," *Computer*, vol. 9, no. 08, Aug. 1976. DOI: 10.1109/C-M.1976.218677 (cit. on p. 107).

[225] *mAgicV DSP architecture document*, 7011A-DSP-12/08, Atmel, Dec. 2008. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/doc7011.pdf (visited on 10/18/2021) (cit. on p. 107).

[226] [SW] D. Morrison, *mtk_fw_tools* 2018, URL: https://github.com/donnm/mtk_fw_tools(visited on 07/23/2020) (cit. on p. 107).

[227] C. Lefurgy and T. N. Mudge, "Fast software-managed code decompression," in *Proc. 6th Int. Symp. High-Performance Computer Architecture (HPCA-6)*, Jan. 2000. DOI: 10.1109/HPCA.2000.824352 (cit. on p. 107).

[228] J. R. Levine, *Linkers and Loaders*. San Francisco, CA, USA: Morgan Kaufmann, Oct. 25, 1999, ISBN: 978-1-55-860496-4. [Online]. Available: https://linker.iecc.com/ (cit. on p. 109).

[229] C. S. Collberg, J. H. Hartman, S. Babu, and S. K. Udupa, "SLINKY: Static linking reloaded," in *Proc. 2005 USENIX Annual Technical Conf. (ATC)*, 2005. [Online]. Available: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/slinky-static-linking-reloaded (cit. on pp. 109, 111).

[230] M. Mahoney. "Data compression explained." (Apr. 15, 2013), [Online]. Available: http://mattmahoney.net/dc/dce.html (visited on 07/22/2020) (cit. on p. 109).

[231] M. Drinić, D. Kirovski, and H. Vo, "Code optimization for code compression," in *Int. Symp. Code Generation and Optimization, CGO 2003*, Mar. 2003. DOI: 10.1109/CGO.2003.1191555 (cit. on p. 109).

[232] M. Ros and P. Sutton, "A post-compilation register reassignment technique for improving Hamming distance code compression," in *CASES '05: Proc. 2005 Int. Conf. Compilers, Architectures and Synthesis for Embedded Systems*, Sep. 2005. DOI: 10.1145/1086297.1086311 (cit. on p. 109).

[233] D. Kirvoski, J. Kin, and W. H. Mangione-Smith, "Procedure based program compression," *Int. J. Parallel Programming*, vol. 27, no. 6, Dec. 1999. DOI: 10.1109/C-M.1976.218677 (cit. on p. 110).

[234] S. Debray and W. Evans, "Profile-guided code compression," in *Proc. ACM SIGPLAN 2002 Conf. Program. Language Design Implementation (PLDI)*, May 2002. DOI: 10.1145/512529.512542 (cit. on p. 110).

[235] D. Citron, G. Haber, and R. Levin, "Reducing program image size by extracting frozen code and data," in *EMSOFT '04: Proc. 4th ACM Int. Conf. Embedded Software*, Sep. 2004. DOI: 10.1145/1017753.1017800 (cit. on p. 110).

[236] J. Ernst, W. Evans, C. W. Fraser, T. A. Proebsting, and S. Lucco, "Code compression," in *Proc. ACM SIGPLAN 1997 Conf. Programming Language Design and Implementation*, May 1997. DOI: 10.1145/258916.258947 (cit. on p. 110).

[237] T. Pittman, "Two-level hybrid interpreter/native code execution for combined space-time program efficiency," in *Papers Symp. Interpreters and Interpretive Techniques*, July 1987. DOI: 10.1145/29650.29666 (cit. on p. 110).

[238] K. Hong, J. Park, S. Kim, *et al.*, "TinyVM: An energy-efficient execution infrastructure for sensor networks," *Software: Practice and Experience*, vol. 42, no. 10, Oct. 2012. DOI: 10.1002/spe.1123 (cit. on p. 110).

[239] M. S. O. Franz, "Code-generation on-the-fly: A key to portable software," Ph.D. dissertation, Swiss Federal Institute of Technology Zurich, 1994 (cit. on p. 110).

[240] D. Rayside, E. Mamas, and E. Hons, "Compact Java binaries for embedded systems," in *CASCON '99: Proc. 1999 Conf. Centre for Advanced Studies on Collaborative Research*, Nov. 1999. [Online]. Available: https://dl.acm.org/doi/10.5555/781995.782004 (cit. on p. 110).

[241] W. Pugh, "Compressing Java class files," in *Proc. ACM SIGPLAN 1999 Conf. Programming Language Design and Implementation*, 1999. DOI: 10.1145/301631.301676. [Online]. Available: http://www.cs.umd.edu/~pugh/pack.pdf (cit. on p. 110).

[242] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *Proc. 5th Symp. Operating Systems Design and Implementation (OSDI)*, Dec. 2002. DOI: 10.1145/844128.844146. [Online]. Available: http://www.waldspurger.org/carl/papers/esx-mem-osdi02.pdf (cit. on p. 111).

[243] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using KSM," in *Proc. Linux Symp.*, 2009. [Online]. Available: https://www.kernel.org/doc/ols/2009/ols2009-pages-19-28.pdf (cit. on p. 111).

[244] M. Kleanthous and Y. Sazeides, "CATCH: A mechanism for dynamically detecting cache-content-duplication in instruction caches," *ACM Trans. Architecture and Code Optimization*, vol. 8, no. 3, Oct. 2011. DOI: 10.1145/2019608.2019610 (cit. on p. 111).

[245] S. Jain, U. Bora, P. Kumar, V. B. Sinha, S. Purini, and R. Upadrasta, "An analysis of executable size reduction by LLVM passes," *CSI Transactions on ICT*, vol. 7, June 2019. DOI: 10.1007/s40012-019-00248-5 (cit. on p. 112).

[246] M. Trofin, Y. Qian, and E. Brevdo. "Machine learned policy for inlining -Oz." (Apr. 2020), [Online]. Available: https://reviews.llvm.org/D77752 (visited on 11/22/2021) (cit. on p. 112).

[247] V. S. Pacheco, T. R. Damásio, L. F. W. Góes, F. M. Q. Pereira, and R. C. O. Rocha, "Inlining for code size reduction," in *SBLP'21: 25th Brazilian Symp. Programming Languages*, Sep. 2021. DOI: 10.1145/3475061.3475081. [Online]. Available: https://homepages.dcc.ufmg.br/~fernando/publications/papers/SBLP21Pacheco.pdf (cit. on p. 112).

[248] N. Redini, R. Wang, A. Machiry, Y. Shoshitaishvili, G. Vigna, and C. Kruegel, "BinTrimmer: Towards static binary debloating through abstract interpretation," in *Proc. 16th Int. Conf. Detection Intrusions and Malware, and Vulnerability Assessment, DIMVA 2019*, ser. Lecture Notes in Computer Science, vol. 11543, 2019. DOI: 10.1007/978-3-030-22038-9_23 (cit. on p. 112).

[249] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *LCTES '99: Proc. ACM SIGPLAN 1999 Workshop Languages, Compilers, and Tools for Embedded Systems*, May 1999. DOI: 10.1145/314403.314414 (cit. on p. 112).

[250] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys*, vol. 51, no. 5, Jan. 2019. DOI: 10.1145/3197978 (cit. on p. 112).

[251] A. F. da Silva, B. N. B. de Lima, and F. M. Q. Pereira, "Exploring the space of optimization sequences for code-size reduction: Insights and tools," in *CC 2021: Proc. 30th ACM SIGPLAN Int. Conf. Compiler Construction*, Mar. 2021. DOI: 10.1145/3446804.3446849. [Online]. Available: https://homepages.dcc.ufmg.br/~fernando/publications/papers/FaustinoCC21.pdf (cit. on p. 112).