ALIGNING INTENT AND BEHAVIOR IN SOFTWARE SYSTEMS: HOW PROGRAMS COMMUNICATE & THEIR DISTRIBUTION AND ORGANIZATION

BY

WILLIAM B. DIETZ

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 at Urbana-Champaign, 2020

Urbana, Illinois

ABSTRACT

Managing the overwhelming complexity of software is a fundamental challenge because complexity is the root cause of problems regarding software performance, size, and security. Complexity is what makes software hard to understand, and our ability to understand software in whole or in part is essential to being able to address these problems effectively. Attacking this overwhelming complexity is the fundamental challenge I seek to address by simplifying how we write, organize and think about programs. Within this dissertation I present a system of tools and a set of solutions for improving the nature of software by focusing on programmer's desired outcome, i.e. their intent.

At the program level, the conventional focus, it is impossible to identify complexity that, at the system level, is unnecessary. This "accidental complexity" includes everything from unused features to independent implementations of common algorithmic tasks. Software techniques driving innovation simultaneously increase the distance between what is intended by humans – developers, designers, and especially the users – and what the executing code does in practice. By preserving the declarative intent of the programmer, which is lost in the traditional process of compiling and linking and building software, it is easier to abstract away unnecessary details. The Slipstream, ALLVM, and software multiplexing methods presented here automatically reduce complexity of programs while retaining intended function of the program. This results in improved performance at both startup and run-time, as well as reduced disk and memory usage.

To Clare, Brian, Noelle, and Chex.

ACKNOWLEDGMENTS

This has been a long and difficult journey, and there are many people I would like to thank and words I wish I could find to thank them with. With humility and gratitude I apologize for anyone forgotten and for inadequately expressing my thanks to both you and even those mentioned; thank you to each of you wonderful people! Thank you for the many ways you helped me to learn and grow into the scholar and person that I needed to be to see this through.

I thank Andrew Lenharth for letting me work with him as an undergraduate student in the LLVM research group. Working with you inspired me to go down this crazy winding road.

I thank my advisor Professor Vikram Adve for your support and for allowing me the freedom to explore and find my way through what it means to be a researcher. Not many advisors would put up with the tumultuous path I took nor the emphasis on philosophy I placed on "doctor of philosophy" while pursuing it in computer science. Thank you.

I thank my coworkers, coauthors, colleagues, and peers Arushi Aggarwal, Sean Bartell, Joshua Cranmer, Assistant Professor John Criswell, Sandeep Dasgupta, Assistant Professor Nathan Dautenhahn, Theo Kasampalis, Maria Kotsifakou, John Mainzer, and Scott Moore for your collaboration, ideas, and insights on projects and for your camaraderie throughout this journey. Thank you for our great conversations about everything from computer science to life.

I thank the LLVM group undergraduate students, and now alumni, Tom Chen, Matthew Plant, Tarun Rajendran, and Matthew Wala for their hard work on the various projects that contributed to work included in the dissertation.

I thank the administrative staff at the Department of Computer Science in the Grainger College of Engineering, especially Viveka and Maggie for all their help navigating paperwork and deadlines. I also thank Vikram's former faculty office support specialist Michelle for helping with travel, reimbursements, and more.

I thank my committee members Professor John Regehr, Professor Tao Xie, and Assistant Professor Sasa Misailovic.

From the bottom of my heart, I want to thank my best friends Brian and Noëlle James for always helping me, whether it was related to grad school or not. You two believed in me at times I could not believe in myself. Thank you for not giving up on me and for carrying me across the finish line.

I thank my Mom, brother Nat, and sister Angel for all their unconditional love and support during my grad school experience. Nat, thank you for our unforgettable ice fishing trips and weekly FIFA nights ("*SHOOOOSH!*"). Angel, thank you for the always-fun weekends at your place, and especially for my *The Little Engine That Could* book, which I still cherish. "I thought I could. I thought I could".

I thank my girlfriend Clare for more than I can say. Thank you for your love, unwavering support, and for always believing in me. Thank you for putting up with my attempts to compare my research in computers with bridges and concrete. Hey, it kinda worked, way better than anyone would have expected it to.

Finally, I thank my cat Chex whose companionship and presence have been essential to my happiness and sanity while completing this dissertation. I do not thank you for standing on my laptop, but admit it was an effective way to remind me of many things such as ensuring I saved often and that it's important to take breaks. Most especially you always brought me to the here and now, dragging out of my nonsense into partaking in your joy of the present. Life is good and happiness comes easily if you let it, you would demonstrate, as long as there's food, play, and snuggles. Thank you. To you and to `##uiuc-llvm` I say: *Meow*.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Simple things should be simple,
complex things should be possible
—Alan Kay

Software is often slower, consumes more resources, and has undesired behavior and bugs; in summary we find our software doesn't do what we want it to do. Complexity is the root cause of problems regarding software performance, size, and security. Attacking this overwhelming complexity is the fundamental challenge I seek to address.

In the seminal paper "No Silver Bullet — Essence and Accidents of Software Engineering" [1], Brooks argues that, fundamentally, any approach of writing software captures both intentional and accidental complexity. Moseley and Marks [2], in contrast, suggest that declarative programming can capture only the essential complexity, without side effects. Certainly, to the extent possible, encapsulating only the intended complexity and avoiding all accidental complexity is ideal.

However, this is not how software is currently distributed. Programmers write and distribute programs. End-users run programs. At the program level, where all the attention is focused, it is impossible to identify complexity that, at the system level, is unnecessary. This "accidental complexity" [1] includes everything from unused features to independent implementations of common algorithmic tasks. Especially as we zoom out to a whole-system perspective the lack of over-arching design becomes more apparent.

The problem has grown much worse over the decades, in 1995 drove Wirth to write "A plea for lean software" [3] which offers as partial explanation that its growth was tolerated only because exponential growth in hardware performance and capacity made it less apparent. That was 25 years ago and was already an old call to arms; today we continue to see the same techniques driving the success of software practices (new frameworks, layers, and so on) simultaneously increasing the distance between what is intended by a person – developer or manager and especially the user – and what the executing code does in practice.

Linking is a useful example. The act of linking, while conceptually simple, actually invokes a complicated concrete implementation that defies succinct description [4]. Low-level implementation quirks which are merely a means to an end instead end up defining the resulting binaries, which in turn are a huge source of accidental complexity in terms of size, redundancy, etc. Both "The Missing Link" [4] and "Why do software packages conflict?" [5] are further evidence of this problem.

In a recent keynote, titled "The Mess We're In," Armstrong [6] of Erlang fame, revisits this problem and more generally the, well, mess that we've created for ourselves. It is in the context of these ideas I present this dissertation containing my research to improve our ability to understand the software we run and to find ways to help align intent and behavior.

I focus primarily on the world of open-source Linux software, as both what I know best and because open-source systems are surely the first place such an alignment would be possible.

While commercial interests are often at odds with the open-source ideology, many commercial technologies are built on open-source foundations. Thus, hopefully, improvements made to the open-source public infrastructure will have trickle-up effects. Within the world of open-source software, there are many “flavors” of Linux-based distributions. In NixOS [7], a specialized Linux distribution, the entire operating system – the kernel, applications, system packages, configuration files, and so on – is built declaratively. This makes NixOS a great starting point for minimizing accidental complexity.

The world of open-source software is today surprisingly opaque. Our collective grasp across community and industry on the software artifacts packaged and installed has grown increasingly inadequate – viewing behavior of software systems as fundamentally unpredictable, unreliable, and unknowable. This shapes the solutions today, which try to control and contain (e.g., containers) instead of understand and improve.

In contrast, I propose that part of the solution is to store programs in a form that preserves the declarative intent of the programmer – information that gets lost in the traditional process of compiling and linking and building software. Within this dissertation I present a system and a set of solutions for improving the nature of software by focusing on programmer's desired outcome (intent). My research targets specific instances of this gap between intent and execution at all stages of software's development and lifetime. I have identified ways in which our software is not what we intend:

- in the way programs communicate (Slipstream, Chapter 3)
- in the way programs are compiled, packaged, and distributed (ALLVM, Chapter 4)
- in the way we organize programs and collections of programs (Software Multiplexing, Chapter 5)

Thesis Statement

The Slipstream, ALLVM, and software multiplexing methods automatically reduce accidental complexity of programs and software systems while retaining intended functionality, without altering source code but operating on a whole-systems level, resulting in improved performance, reduced size, and lower memory usage. To retain functionality for some programs with edge case behaviors, developers and users can optionally choose to not use my tools. My initial work focused on solving these problems “locally” with Slipstream; however, it became apparent that a broader approach was necessary. Thus, my more comprehensive approach taken by ALLVM and multiplexing (`allmux`). My approach has been to develop solutions that help developers and users to detect and to reduce or eliminate unnecessary complexity interfering with ideal software function. As a recurring theme this often involves improving multiple aspects of the involved software ecosystems.

1.1 DISSERTATION CONTRIBUTIONS

1.1.1 Slipstream

Slipstream improves communication performance by detecting and transparently removing unnecessary complexity between programs. Use of TCP is ubiquitous, due to widespread support and location transparency. TCP provides a notion of location transparency to software, enabling the same code to communicate across a network and to the process next to it. Locality may not be determined until deployment, making TCP very compelling for programmers of distributed systems. However, faster interprocess communication (IPC) exists for local communication. Slipstream automatically capitalizes on these faster IPC mechanisms for other software. It is easy to deploy, language agnostic, and highly compatible. Our testing showed 2× bandwidth improvement for host-local communication. When optimization is not possible, the imposed overhead is negligible.

1.1.2 ALLVM

ALLVM is the umbrella name given to the ideas, insights, techniques, tools (see Table 1.1), software collection, and resulting system that use allexes, an executable containing LLVM IR. ALLVM is a new, whole-system approach toward responsibilities currently held by compilers, linkers, package managers, deployment infrastructure, and runtime systems. It enables delivery and analysis of large collections of software through a simple, novel executable format. An overview of the system and tools are provided in Chapter 4. Fundamentally, this is a paradigm shift systematically altering how software is represented throughout its lifetime. ALLVM leverages the intelligibility of the LLVM IR to preserve programmer intent.

Software represented as allexes are faster, smaller, and portable, compared to traditional representations. Allexes aren't dynamically linked, which slows traditional programs and libraries. In addition, the resulting binaries contain no unused code. Static linking ensures easy distribution of allexes.

By building on NixOS, ALLVM provides "reproducible builds" which allow anyone to recreate bit-by-bit identical copies of all specified artifacts [8]. This movement[1], which is backed by Google Open Source, Open Technology Fund, Software Freedom Conservancy and others, is important for improving privacy and security in software [9–12]. For ALLVM, these practices also help enable reasoning about programs and their behavior by ensuring what you test is what is executed.

No other technology today makes it possible to analyze and transform software in the context it is executed. ALLVM makes this possible and straightforward. ALLVM today puts thousands of packages at your fingertips, and run on any Linux environment. ALLVM enables functional treatment of analysis and transformation operations, making it easy to express for example `map andersonAA allexePkgs` to refer the results of running the classic Anderson alias analysis over the

[1]https://reproducible-builds.org/

Table 1.1: ALLVM Tools.

Tool	Description
Front-end, libs	clang, libc++, libc++abi, libunwind
bc2allvm	wrap bitcode file(s) into allexe format
alltogether	linker to resolve external name references
libnone	musl libc for simplicity and analyzability
allready	AOT optimizer and code generator
alley	execution engine with optional JIT

thousands of programs available as allexes. This is especially appealing because the results are cached, deduplicated, easily shared, traced, and reproducible. Analysis "steps" are automatically distributed over available builders making the process cloud-friendly and easily scalable.

1.1.3 Software Multiplexing: `allmux`

Software Multiplexing is a technique [13] for *re-organizing* collections of software into a single unified multiple-entry or *multicall* binary. Conventionally impractical or impossible, this technique is both simple and effective in ALLVM (see Chapter 5). Compared with equivalent dynamically linked programs, allmux-optimized programs start more quickly and even have slightly lower memory usage and total disk size. Compared with equivalent statically linked programs, allmux-optimized programs are much smaller in both aggregate size and memory usage, and have similar startup times and execution performance. The ALLVM tool `allmux` implements the two "Software Multiplexing" algorithms detailed in the paper, the latter of which performs library-granular "deduplication" across the input programs.

CHAPTER 2: RELATED WORK

As the area of our knowledge grows,
so too does the perimeter of our ignorance
—Neil deGrasse Tyson

This chapter gives background and review of the state of the art relating to this dissertation. The content has been organized into a section for conceptual background followed by sections for each chapter topic of Slipstream, ALLVM, and software multiplexing. Sections 2.2 and 2.4 are reproduced with minor modification from their corresponding publications.

2.1 BACKGROUND

2.1.1 Accidental Complexity

Managing the overwhelming complexity of software is a fundamental challenge because complexity is the root cause of problems regarding software performance, size, and security [2]. Complexity is what makes software hard to understand, and our ability to understand software in whole or in part is essential to being able to address these problems effectively. This complexity is partially intrinsic to the problems being solved, the rest is classified as *accidental* and in theory can be eliminated. In his seminal paper, Brooks proposed these concepts of accidental and essential complexity. These terms are more clearly defined in Moseley and Marks's rebuttal "Out of the Tar Pit" [2]. Accidental complexity is sometimes called side-effects [14]. In practice much of this complexity remains: it is rarely obvious how to go about the task of removal, the costs are hard to predict and often hard to justify, and of course sometimes we just write bad code because we're human.

Linking is a useful example. The act of linking, while conceptually simple, actually invokes a complicated concrete implementation that defies succinct description [4]. Low-level implementation quirks which are merely a means to an end instead end up defining the resulting binaries which in turn are a huge source of accidental complexity in terms of size, redundancy, etc. In particular, Kell, Mulligan, and Sewell note that linking is so poorly understood that many linker-unaware analyses are unsound. This is specifically due to systems software being unexpectedly dependent on linker behavior. Linkers are semantically critical and linker features can actively undermine language semantics. Finally, they reach a similar conclusion: much of this complexity is unnecessary. Therefore, I seek to remove it.

Package managers are another attempt to contain the complexity involved in software installation [15]. Artho *et al.* [5] found that 30% of installation incompatibilities, (also known as "conflict defects") could be easily eliminated with more detailed meta-data. An additional 30% could be eliminated by installation-time testing for common or shared resources to prevent overwriting. This highlights how pervasive the problem of complexity is throughout Linux and how it negatively impacts users. Installation is only a task necessary because programs are not self-contained, leading to the design decisions of allexes.

Balakrishnan and Reps note in "WYSINWYX: What You See is Not What You eXecute" that binaries, which contain the actual computer implementation details, are not what is typically analyzed. They present CodeSurfer/x86 as a tool for extracting an IR from executables rather than source code [16]. This work highlights the disconnect between source-code which is much close to programmer intent and the actual programming details in binaries. We take the opposite approach though for the same reasons, addressing the gap between human intention and machine execution.

2.1.2 Nix

NixOS [7] is a specialized Linux distribution in which the entire operating system – the kernel, applications, system packages, configuration files, and so on – is built declaratively. Nix [17] is a pure functional package manager. Nix utilizes a functional language to specify build instructions in the form of Nix expressions. For a more comprehensive introduction to Nix, see [18]. The Nix Package collection (Nixpkgs) is a set of over communally maintained 40,000 packages and is widely available[1]. Nix and NixOS are the technical and conceptual foundation for ALLVM and its ideas give context to this work. The author of Nix and NixOS, Eelco Dolstra, observed: "a binary distribution can be considered conceptually to be a source distribution that has been partially evaluated with respect to a target platform using, e.g., a compiler" [19]. Nix has numerous benefits. It features: (1) reproducibility; (2) atomic updates; (3) exact dependencies; (4) trivially supports many versions of a library; (5) declarative configuration of system and services in NixOS; and (6) binary caching when desired.

The use of functional expressions for packages is what empowers Nixpkgs to be extremely flexible and extensible, and is what makes it ideal for use in compiler-based research such as ALLVM, or of any kind interested in open-source software. Interaction with Nix is through invoking the `nix` tool on recursive sets of expressions (Nixpkgs), which contain a great number of "package expressions" and all the glue that keeps the expressions simple yet powerful. Package expressions are generally written as short functions that take as arguments a standard build environment and all build and runtime dependencies, which are used in the expression to describe how it should be built. The expression also specifies simple metadata such as name and version, as well as how to obtain the source and a trusted checksum of its contents.

2.1.3 Declarative Programming

Declarative programming is where you state what you want to happen not how it happens. In common usage, declarative programming is often defined in this somewhat vague but intuitive

[1]https://nixos.org/nixpkgs

way [20–22]. An excellent and thorough classification of declarative programming can be found in [22, Section 3.1.1]; using their terminology, ALLVM and NixOS are declarative using the definitional, functional view at the level of Nix expressions and derivations, but the build tasks they compose are better categorized as observationally declarative. In this dissertation, both views are included when referring to "declarative", as in [22] when discussing it generally. It is a conceptual approach to programming that focuses on the author's intent (output, properties of resulting state, etc.) rather than including all the "details" and methodology. Without delving too much into the philosophy of programming [23–26] it takes a more top-down approach, discussed extensively in *The cathedral & the bazaar: musings on Linux and open source by an accidental revolutionary* [27]. Some examples of declarative-style programming include NixOS, logic-programming (e.g. Prolog [21], Mercury [28]), and SQL.

Declarative programming overlaps heavily with functional programming. Functional programming, specifically, has a high barrier to entry [21, 29, 30]. The imperative style, in contrast, is friendly to corporation-style programming (pipelined and piecemeal) [31–33]. Functional is powerful but can be opaque to anyone but its author [32]. Additionally, it leaves the compiler and runtime system in charge of selecting the specific evaluation method which is not always advisable. A slower, or less ideal, method of evaluation may be selected due to lack of specification [34]; commonly this is mitigated by means of annotations, as available in many Datalog implementations.

2.1.4 Whole-System Approach

"Whole-systems" approaches are based on designing a top-down environment where you can program, execute, and analyze programs. The goal is to cut out layers of complication i.e. abstract them away by looking the bigger picture. Previously, Lisp [35], Smalltalk [36], and Pharo [37] also took a top-down approach. However, the world of computation has undergone an explosive growth during the last 60 years. Because times were simpler, they were able to design a contained system from the top down, but it does not align with how code is actually written. This is opposed to the "worse is better" [38, 39] approach which supports early adopters by allowing transferring of already written code and not requiring leaning a new language. Within this dissertation, I endeavor to provide the benefits of a top-down approach without eschewing early-adopter strengths.

There are many notable build systems (Adept [40], Bazel [41], BuildSome [42], CloudBuild [43], Hadrian [44], Pluto [45], Shake [46], Tup [47]) closely related in spirit to Nix which underlies ALLVM. These are build systems that address the complexity of storing packages and their dependencies. It is worth noting that all the major players in software (Microsoft, Google, etc.) are pushing in this direction. Many of these build systems take a rules-based approach to wrangle the complexity of program and dependency installation, branding themselves as replacements for GNU Make [48].

The whole-systems approach has been trumpeted by others [49, 50]. This problem of complexity has been noted in hardware designs [51], especially when considering cloud-based systems, that are complicated by "build non-determinism" or partial builds [43]. In summary, by changing the frame of reference [52] away from the program and towards how the smaller systems interact, it becomes apparent that a top-down approach is necessary to remove complexity.

2.2 SLIPSTREAM

A number of previous systems aim to optimize local interprocess communication. A brief summary and a feature comparison are presented in Table 2.1, and are discussed in more detail below.

In-Kernel Solutions: Recently, operating systems such as Windows [53], AIX [54], and Solaris [55], have made available localhost TCP optimizations. In general, they all bypass the lower levels of the kernel networking stack, only forming TCP and not IP packets. Performing these optimizations within the existing networking stack simplifies identifying local-only traffic and provides a fast-path for local streams.

Unfortunately, performing these optimizations within the OS has several drawbacks: the implementations are kernel-specific and other systems cannot benefit (e.g., Linux does not support it, although there have been efforts to add it [56]); OS upgrades are often slow to be adopted widely; and applications have no control over whether or not the optimization is available on a given system. In contrast, Slipstream is relatively easy to port, at least across Unix-like systems; it is easy to deploy (e.g., it does not even require superuser privileges to install); and application developers that choose to do so can incorporate the system fairly easily.

VMM solutions for inter-VM communication: Several approaches to improving performance of communication between co-located virtual machines have been described [57–59], all focusing on Xen. These solve similar communication inefficiencies as Slipstream, but either require application modification [58], guest kernel modification [57–59], are not fully automatic [57, 58], or operate at the IP layer so TCP overheads are not eliminated [59].

Language-Specific Solutions: The interfaces provided by languages for IPC are often at a much higher level than the basic operations provided by the system. For example, Java Fast Sockets [60] is able to greatly improve communication of Java applications with techniques such as avoiding costly serialization in situations in which the data can be passed through shared memory. While these optimizations are difficult with a language-agnostic solution like Slipstream, Slipstream is able to optimize applications that use sockets regardless of source language, as our results illustrate for C/C++ and Java.

Transparent Userspace Libraries: The FABLE library, which is only described in a position paper [61], provides automatic transport selection and dynamic reconfiguration of the I/O channel. FABLE provides a socket compatibility layer that uses a new system call for looking up a name mapping (implying that FABLE is not a pure userspace solution) to identify communication with hosts for which it may be able to provide a more efficient transport. Without any information about an implementation, it is unclear how well this compatibility layer works. However, the dynamic switching of transports in Slipstream is very similar to their reconfigurable I/O channels.

Fast Sockets [62] is a userspace library that provides a sockets interface on top of Active Messages [63]. This is superficially similar to Slipstream, but Fast Sockets assumes it can determine which transport to use by inspecting the address, which requires static configuration and a rigid network topology. In contrast, Slipstream focuses on automatic detection of inefficient communication without relying on network topology details and switches transports on-the-fly.

Universal Fast Sockets (UFS) [64] is a commercial solution to optimize local communication transparently. Like Slipstream, UFS uses a shared userspace library to interpose on application activity, but other details of how it operates are proprietary and unclear.

Explicit Userspace Solutions: Many software libraries provide *explicit* messaging abstractions for application use [65–67]. Without the limitations of existing interfaces, impressive performance results are possible, but applications need to be modified to use these frameworks, and seeing the best results may require deep changes to the fundamental structure of an application.

Several researchers have explored moving the network stack out of the operating system and entirely into userspace, citing many performance benefits [68–70]. Userspace networks stacks are components of popular OS designs including the microkernel [71] and exokernel [72] approaches. Some work goes further and collapses the entire network stack into the application [73], providing a specialized stack entirely in userspace. More recent work refactors the OS network stack into a control plane (which stays in the kernel) and separate data planes (which run in protected, library-based operating systems) [74]. All of these efforts redesign the networking stack from the ground up and require kernel modification, application modification, or both. These solutions do not directly aim to optimize local communication, but similar to the in-kernel approaches described above this could likely be added in a straightforward if not portable manner.

Table 2.1: Slipstream Prior Work: Categories and Features.

Category	Prior Work	Application Transparency[1]	OS Transparency[2]	Sockets[3]	Misc.
OS Impl.	Win. FastPath [53]	✗ Opt-in	✗ Included in OS already	✗	
	Solaris TCP Fusion [55]	✓ Opt-Out	✗ Included in OS already	✓	
	AIX fastlo [54]	✗ Opt-in	✗ Included in OS already	✓	
	Linux TCP Friends [56]	?	✗ Floating Kernel patch	✓	
VM-VM	XWay [57]	✓	✗ Guest kernel patch		
	XenSocket [58]	✗ `AF_XEN`	–	✓	
	XenLoop [59]	✓	✗ Guest kernel module		
User. Stack	mTCP [70]	✗ Similar API	✗ NIC driver for packet library	✗	
	Sandstorm [73]	✗ Specialized for App.	✗ requires netmap [69] kernel support	✗	
User. Shim	Fable [61]	✓ Limited, ns-based	✗ System call for name service	✓	
	Java Fast Sockets [60]	✓ Java.net.Socket	✓	✗	Java-Only
	Universal Fast Sockets [64]	✓	✓	✓	Commercial
	FastSockets [62]	✓	✓ Uses Active Messages	✓	
	Slipstream	✓	✓	✓	

[1] Application Transparency (no application modifications required)
[2] OS Transparency (no OS modifications required, no use of OS-specific tech)
[3] Provides Socket Interface (POSIX, Linux)

2.3 ALLVM

ALLVM refers to the ideas and programming approach as well as the techniques, tools, software collection, and resulting system that use allexes. These technologies make it possible to easily analyze and transform software in the context it is executed. The ideas central to ALLVM are discussed above in "Declarative Programming" (Section 2.1.3). Below focuses on the implementation.

Built Using: ALLVM is built using, leveraging, and extending a number of other important projects and accordingly has characteristics derived from each. A thorough explanation of ALLVM's construction and components is detailed in Chapter 4 with an abbreviated summary of projects key to ALLVM given below.

ALLVM is based on the LLVM compiler infrastructure [75] and extends it by adding the ALLVM tools (see Section 4.5), introducing a runtime and representations for full programs (including libraries) as well as collections of programs making possible techniques such as software multiplexing. ALLVM additionally combines, integrates, and extends these projects: (1) WLLVM [76] compiler wrapper; (2) the Nix [17] language and package manager; (3) Nixpkgs collection of software packages; and (4) the purely-functional NixOS [7] Linux distribution built using Nix and Nixpkgs.

ALLVM's libnone is built from musl libc [77]. Musl is popular in the realm of compiler-based analyses. I particularly capitalize on its static linking suitability and simple standards-compliant interface. A more detailed discussion is given in Sections 4.1.5 and 4.1.6.

Shipping software as LLVM IR: LLVM provides pieces to ship pieces of software in its intermediate representation (IR), as reportedly is done in industry by Apple and other companies. ALLVM is different in that it addresses representing commodity open-source software used in Linux ecosystems and introduces representations for software executables including their libraries, as well as collections of software. As details are unavailable for the industrial applications of LLVM related to ALLVM, further comparison is not possible. Even so, there appears to be strong interest in the subject; at the 2016 LLVM Developer's Meeting I led a "Birds of a Feather" session titled "Shipping Software as LLVM IR"[2] [78] which was well-attended [79].

LLVM-based Languages and Ecosystems: Numerous modern languages such as Rust[3] [80], Swift[4] [81], Zig[5] [82], and others provide experiences that are close to ALLVM's goals, at least for the scope of their language ecosystem, with tooling for building and deployment enabling varying degrees of cross-component analysis and transformations. This is increasingly possible as languages provide and improve upon their own preferred well-integrated tools for building and managing

[2]Slides available at https://wdtz.org/files/bof-2016.pdf
[3]https://rust-lang.org
[4]https://swift.org
[5]https://ziglang.org

dependencies, acting as unified replacement for traditional build system and package management approaches. This is similar to ALLVM's approach, which leverages Nix expressions driving software build processes to make possible cross-module optimizations and other improvements. ALLVM works at a slightly higher level, in theory supporting the many languages representable as LLVM IR and amenable to allexe generation. Such integration into language ecosystems appears promising, especially for those already integrated and suited for Nixpkgs but at time of writing this has not been pursued beyond prototypes (Rust, Go, Haskell).

2.4 SOFTWARE MULTIPLEXING

The problems addressed by our work are long-standing issues that have been addressed in a variety of ways in the past. We focus here on the most relevant related work, which falls into two broad categories: reducing dynamic linking overheads; and reducing code duplication.

2.4.1 Reducing Dynamic Linking Overheads

To reduce the cost of dynamic linking, a number of solutions have been developed, some of which are in use today. One of the earliest, OMOS server [83] describes a novel shared library implementation that speeds up dynamic linking by caching executables and libraries after symbol resolution and relocation. SpringOS [84] achieves a similar caching effect for applications and shared libraries by caching them after applications exit. Red Hat's `prelink` [85] tool precomputes relocation information and address assignment at link time instead of doing these at run time. Later work [86] extends this to allow use of Address Space Layout Randomization (ASLR). Prelinking and Preloading [87] extends this technique to also preload a predetermined set of shared libraries on embedded systems. Software Multiplexing achieves all these overhead reductions (and more), but also obtains the full benefits of static linking, while preserving the space savings of dynamic linking.

IRIX shared libraries from SGI used three techniques to mitigate negative impact of shared libraries: optimizations to reduce indirect references, a quick start scheme similar to prelinking, and layout optimization for procedure locality [88]. The latter is orthogonal to our work, while the first two achieve only a part of the benefits of Software Multiplexing, similar to prelinking.

Recent work [89] has even proposed hardware support to reduce dynamic linking overheads, focusing on the indirect function calls but not on initial startup overheads (they report "up to 4%" speedups with the approach).

Finally, a number of current and past tools [90–93] work by combining dynamically linked executables into a statically linked single-file equivalent. These tools rely on application checkpointing techniques, creating a snapshot of the program early in its execution for replay later. Similarly freezing a dynamically linked application is sometimes part of *checkpoint-restart* solutions such as MCR [94]. None of the tools are able to remove unused code from dynamic objects, which yields substantial code size reduction benefits (e.g., upper chart in Figure A.1 or N=1 case for `memcached` in Figure 5.7). These tools have the benefit that they do not require compiler support. A serious concern, however, is that these tools must support and emulate complicated semantics of binary formats, whereas `allmux` is better able to reason about high-level intent instead of low-level implementation details.

2.4.2 Reducing Code Duplication

There have been both compiler and system-level solutions to eliminating duplicated code in applications and systems. Shared libraries – which can be static or dynamic – were invented mainly to address this problem [95], and are widely used. Position-independent code (PIC) was invented to enable more flexible code layout, including dynamic linking. Operating systems, linkers, loaders and compilers all have evolved to support these mechanisms, but current practice suffers from all the widely known tradeoffs [95] between static and shared libraries described in the Introduction.

A few systems explicitly try to reduce or eliminate these tradeoffs. VMWare ESX server used a hashing technique to identify memory pages with identical contents [96] and share such pages between virtual machine instances on a single host. Kernel Same-page Merging (KSM) [97] modifies the Linux kernel to scan through main memory and find duplicate memory pages between processes; such pages are then mapped into multiple process address spaces and marked copy-on-write to detect page modifications. This technique has also been shown to be especially effective at increasing memory sharing between VM instances. Such approaches do not address offline code size, do not reduce dynamic linking complications and startup overheads, and do not enable better compiler optimizations, all of which are achieved by either static linking or Software Multiplexing.

Slinky [98] is perhaps the most effective previous solution, and is discussed in Section 5.2 and Section 5.6.9. In comparison, Software Multiplexing achieves better code size reduction and lower overheads, as discussed in Section 5.6.9, and does not require changes to the OS kernel, system linker or loader, but does require changes to the compiler while Slinky doesn't.

CHAPTER 3: SLIPSTREAM: AUTOMATIC OPTIMIZATION OF INTERPROCESS COMMUNICATION

Systems programmers are the high priests of a low cult

—R. S. Barton

3.1 CONTEXT

The following chapter describes the Slipstream project, as previously published[1] in Usenix ATC [99]. This work was done jointly with Joshua Cranmer and Nathan Dautenhahn.

Slipstream's ability to automatically optimize interprocess communication between programs and is an important baseline for further ALLVM research across communication boundaries. Indeed this is how the project came to be – early experiments with compiler-based approaches made it clear there were significant opportunities for improvement even without leveraging ALLVM, and until these were explored and published it would be difficult or awkward to clearly separate in design, or especially in evaluation, how useful ALLVM was to achieving these benefits. This also allowed Slipstream freedom to target situations where ALLVM would harder to deploy today.

Future research under the ALLVM umbrella will be exploring ways to improve communication further. Slipstream's pairing or transport replacement techniques are likely useful although that remains to be seen.

3.2 INTRODUCTION

TCP has become one of the most commonly used communication protocols because of its ubiquity on standard platforms (e.g., Windows, Android, Linux) and its *location transparency*: instead of creating one communication channel for host-local and another for remote communications, which would reduce portability and increase complexity of the application, developers use TCP because it works for all cases. Unfortunately, by using TCP, developers eschew faster host-local transport mechanisms (e.g., Unix domain sockets, pipes, or shared memory) resulting in missed performance opportunities: a claim supported by a comprehensive study providing clear evidence for the *potential* improvements to be had by replacing the TCP transport with other local IPC mechanisms [61].

Using TCP for its *location transparency* to reduce programming burden and enhance portability is common in several application domains. Web sites are commonly deployed on a single system, yet the Web server front-end communicates with database engines and application servers over TCP, hurting both latency and throughput of Web requests (e.g., LAMP). In some instances, application logic depends on TCP parameters [100], e.g., Memcached uses the IP address and TCP port to

[1]The author holds permission to reprint here material previously published in conference proceedings as: Dietz, Cranmer, Dautenhahn, and Adve. "Slipstream: Automatic Interprocess Communication Optimization." 2015 USENIX Annual Technical Conference (USENIX ATC 15), (ISBN: 978-1-931971-225, pages: 431–443).

implement consistent hashing [101], thereby requiring TCP addressing to function correctly, but *not* necessarily TCP data transport.

Perhaps the most compelling future need for optimizing local TCP communication is the increasing popularity of lightweight virtualized environments like Docker. Docker strongly encourages separating communicating services into distinct Linux containers, using TCP to communicate with each other because portability is a key goal [102]. For example, Yelp developers created a service discovery architecture in which client applications communicate with remote endpoints through a host-local proxy (`haproxy` [103]) via TCP [104]. One of the primary deployment scenarios for this architecture is to run a client application and its local proxy in separate containers, which puts the TCP latency on the critical path for every client request and response. Optimizing local TCP communication over Docker can therefore provide important performance gains (as our experiments with other containerized Docker services show).

In fact, the problem is important enough that there are several approaches that optimize TCP communication between communicating processes within single operating system environments. This includes commercial operating systems like Windows [53], AIX [54] and Solaris [55] and several userspace libraries [60, 62, 64]. However, these approaches either require changes to the operating system [53–55] or application code [60, 62, 64]—thus eliminating one of the key benefits of using TCP in the first place—or they are only applicable to specific language runtimes. In legacy deployments, it might not be possible to modify the OS or application, and furthermore, in cases where modifications are possible, they may not be feasible: any modifications may cost too much to make to existing application logic.

We present **Slipstream**, a userspace solution that identifies and optimizes the use of TCP between two host-local communicating endpoints without requiring changes to the operating system or applications (except for the use of a shim library above `libc`). Slipstream transparently reduces latency and increases throughput without requiring modifications to either the kernel or the application by interposing on TCP related events to detect and optimize the communication channel.

We built a Linux prototype of Slipstream that detects TCP-based host-local communications by inserting an optional shim library above `libc` to intercept TCP communication operations.[2] Our solution is portable across UNIX-like systems. Slipstream uses this vantage point to collect information on connections in order to apply a general detection algorithm that relies only on observable characteristics of TCP, without knowledge of the underlying network topology, to detect host-local communication endpoint pairs. Once a host-local TCP communication stream is detected, Slipstream transparently replaces TCP with a faster host-local transport while emulating all TCP operations to maintain application transparency. The primary complexity in the design and implementation of Slipstream arises in replicating kernel-level TCP state at the user-level and preserving the interface semantics of the TCP sockets API on top of the host-local transport.

[2]The source code for Slipstream and the evaluation scripts are available at: https://wdtz.org/slipstream.

Our results indicate significant performance benefits for applications using Slipstream: throughput improves up to 16-100% on server applications and 100-200% with Docker, and microbenchmarks show that latency is cut in half. Our results also show that when Slipstream tries but fails to optimize a connection (e.g., because one of the two endpoints is not using Slipstream), the throughput is impacted by only 1-3% on average. Moreover, Slipstream is an opt-in system that imposes *zero* overhead for applications that do not explicitly request the optimizations.

Our work makes the following contributions:

- We describe a novel backwards-compatible, transparent algorithm for classifying communication between two endpoints as host-local or remote.
- We describe a fully automatic optimization to replace TCP with Unix domain sockets as the transport layer, while preserving the interfaces, reliability guarantees, and failure semantics of the original transport.
- We show by use of microbenchmarks and server applications that Slipstream achieves significant improvements in throughput, without requiring manual tuning, custom APIs, or special configuration.

There are certain system configurations that will cause our system to mismatch connections; violations of our correctness conditions, while unlikely in practice, must be avoided during system setup.

Overall, our experience suggests that the improvements achieved automatically by Slipstream are comparable in terms of performance (as we show in the Netperf results) to those that can be achieved by modifying applications to explicitly use Unix domain sockets for host-local connections.

3.3 SLIPSTREAM OVERVIEW

Slipstream transparently identifies and dynamically transforms TCP streams between local endpoints into streams that employ more efficient IPC mechanisms, such as Unix domain sockets (UDS). This optimization improves communication performance for many existing applications, and alleviates the burden on programmers to manually detect and select the fastest transport mechanism. To accomplish this task, Slipstream interposes on TCP interactions between the application and the operating system to track TCP endpoints, detect local TCP streams, switch the underlying transport mechanism *on-the-fly*, and emulate TCP functionality on top of the local transport mechanism.

Performing all of these in userspace means that Slipstream must replicate critical stream state at the userspace level and it must adequately emulate TCP on a non-TCP-based transport mechanism. In this section we describe the key design goals we aim to meet, discuss the challenges presented by TCP, and then provide a high-level description of Slipstream.

3.3.1 Design Goals

We specify three key design goals for the optimization, which are desirable for real-world use and present new design challenges. None of the previous systems meet all three requirements. First, we aim to preserve application transparency, i.e., requiring *no changes* to application code in order to perform this replacement. However, application end-users can choose whether or not to enable the optimization. Second, we aim to avoid any operating system changes, not even through dynamically loadable kernel modules, i.e., the optimization should be implemented entirely via userspace code (in fact, we do not even require root privileges). Although aspects of the optimization would be simpler to implement within the OS, those extra challenges are solvable in userspace as well (as we show), and a solution that does not require kernel changes is far easier to deploy. Third, unoptimized communication (i.e., between an optimized component and an unoptimized one) must continue to work correctly and, again, with no application changes.

Of course, the system must also meet essential correctness requirements: the reliable stream delivery guarantees of TCP must be preserved, and the semantics of socket operations must be implemented correctly.

3.3.2 TCP Optimization Challenges

Within an OS, a *TCP endpoint* is a kernel-level object represented as a 2-tuple, ⟨local IP address, TCP port number⟩. A TCP stream is a pair of *TCP endpoints*, and is also referred to as a *socket pair*. The kernel provides userspace access to TCP via the standard socket interface, which applications use for creating and managing connections and for sending and receiving data. The socket interface represents instances of streams in the form of a socket descriptor, a special case of a file descriptor. These file descriptors are the only way in which userspace code can access TCP endpoints. Since Slipstream is a userspace mechanism but must emulate details of the TCP protocol, it operates by replicating the notion of a TCP endpoint in userspace.

TCP, the POSIX socket interface, and their implementation in modern operating systems present some key design challenges:

- Although a socket pair uniquely identifies TCP connections in a network, a single host can be part of multiple networks with overlapping network name spaces. For example, via use of virtual environments, it is possible to have the same IP address assigned to multiple network interfaces within the same system. This allows duplicate combinations of IP address / TCP port pair to be used for distinct TCP streams within the same host. Each such combination would be a distinct TCP endpoint in the kernel.
- It is common for the kernel to map a single TCP endpoint into multiple process address spaces, thereby creating several userspace file descriptors for a single kernel-level TCP endpoint. This feature is widely used in applications, such as Apache. Consequently, Slipstream must also

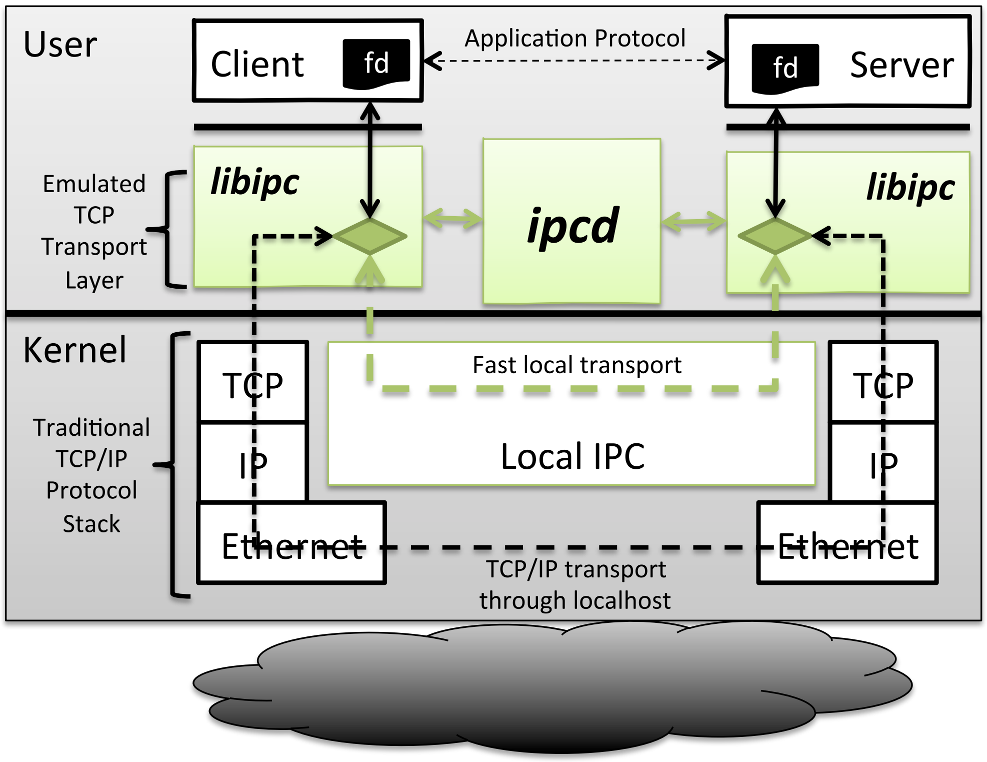


Figure 3.1: Network layers and local IPC within Slipstream.

track all application interactions with the kernel that might create or delete multiple instances of such endpoints.

- The TCP protocol does not support any reliable mechanism for transferring extra data "out-of-band" between the endpoints[3]. On the other hand, injecting any such data into the stream "in-band" would break an application that isn't using Slipstream and so cannot filter out the extra data. This significantly complicates the task of detecting when two endpoints are on the same machine and capable of using Slipstream.
- POSIX file descriptors support a large number of non-trivial functional features through numerous system calls, which must be correctly emulated to preserve application functionality transparently. Some are specific to sockets (e.g., `bind`, `connect`, `listen`, `accept`, `setsockopt`) while others are generic to file descriptors (e.g., `poll`, `dup`, `dup2`, `fcntl`, `fork`, `exec`, and the various send/receive operations). Slipstream supports all of these system calls and other less common ones that interact with TCP.

3.3.3 Slipstream Overview

Figure 3.1 shows a high-level diagram of a typical OS network stack, enhanced with Slipstream. Slipstream inserts a shim library we call `libipc` that interposes on all TCP interactions between the

[3]There is a TCP urgent data feature, but its use to communicate lengthy amounts of data is unreliable.

application and the kernel. `libipc` is responsible for tracking TCP endpoints at all TCP-relevant system calls and for reporting all TCP stream identifying information to a system-wide process, `ipcd`. `ipcd` collects and records all stream information and analyzes all existing streams using a stream matching algorithm.

Once Slipstream detects that both endpoints of a stream are local, `libipc` modifies the underlying communication channel to use a local IPC transport (in the case of our implementation, UDS). The use of emulation also indicates one of the major contributions of our efforts: emulating a sufficient subset of the TCP protocol in userspace to correctly support real applications, as demonstrated in our evaluation.

Overall, this sequence of steps ensures that (a) Slipstream can replace TCP with a local IPC transport without requiring any changes to application code; (b) Slipstream does *not* break remote streams or local streams that cannot be identified by Slipstream; and (c) the protocols used by Slipstream *never* introduce new errors in communication for identified local streams. In this sense, the optimizations are *transparent* to application code, are *backwards-compatible* with non-participating endpoints, and do not require kernel modification.

3.4 DESIGN AND IMPLEMENTATION

The functionality of Slipstream has three major aspects: (1) identifying TCP communication streams in which both endpoints are local; (2) replacing TCP with an alternative local transport; and (3) emulating most of the functionality of the TCP sockets interface. The first two subsections below describe preliminary design aspects for interposing on and tracking TCP events. Sections 3.4.3 to 3.4.5 then discuss the three major aspects of the design.

3.4.1 Interposing on TCP Events

In order to identify local streams and emulate TCP on optimized transport, we monitor applications and track the creation and use of TCP endpoints. We do this using our per-process library, `libipc`, which intercepts all TCP-related calls exposed by the system. In our implementation, we insert `libipc` into a client application by using the `LD_PRELOAD` environment variable or by specifying the full path to `libipc` in `/etc/ld.so.preload`. We prefer dynamic interposition in favor of replacing `libc` so as to avoid requiring modifications to the system libraries and, importantly, to enable applications to choose whether or not to use our technology.

3.4.2 TCP Endpoints in Userspace

In order to track TCP streams, Slipstream assigns a unique *endpoint identifier* (*EPid*) to each TCP endpoint created and used by the application. Each *EPid* represents an in-kernel TCP endpoint. To manage the optimization state, `libipc` assigns a state to each *EPid* to track its optimization status:

Pre-opt	Optimization not yet attempted.
No-opt	Optimization attempted and failed.
Opt	Optimization successful.

As explained in Section 3.3.2, in order to fully track streams, Slipstream must replicate endpoint state at the user level (in `libipc`) by tracking TCP state at critical system calls, such as `fork`, as well as traditional TCP modifying operations. For each *EPid*, `libipc` maintains a reference count representing how many processes have at least one file descriptor open to this endpoint. `libipc` updates this reference count on events that affect it, such as `fork`, `exec`, and `exit`. Moreover, instead of communicating with `ipcd` on every use of a socket, as many details as possible about file descriptors and *EPid*s are retained by `libipc`.

In our implementation, `libipc` state information is maintained in two tables, one tracking file descriptors and the other tracking endpoints. The file descriptor table tracks much of what the kernel also tracks at a per-file descriptor granularity, such as the `CLOSE_ON_EXEC` flag or `epoll` state. In addition, this table also tracks the mapping of file descriptors to endpoint identifiers. The *EPid* table tracks the optimization state for each *EPid*, explained above. It also tracks information that is relevant for the optimization procedure, such as the handle for local transport if optimized and running hashes of sent and received data.

3.4.3 Identifying Host-Local Flows

The first major step of Slipstream is to identify when two endpoints of a TCP stream are located on the same host. As noted in Section 3.3.2, the combination of IP address and TCP port is insufficient to do so because it is possible to have access to two network domains on a single host, even though this may be rare. Instead, to identify local TCP streams, Slipstream augments the usual IP address and port pairs with extra information passively obtained by watching the initial TCP conversation. This information consists of hashes of the first N bytes of the stream and precise timing of the connection creation calls. Together, these components are sufficient to pair endpoints in the vast majority of situations. When they are not sufficient, Slipstream detects such situations and does not attempt to optimize the socket.

More specifically, the steps Slipstream takes are as follows. When a new TCP socket is connected, `libipc` immediately records the time of the connection attempt and forwards it to `ipcd` along with the IP and port information. `ipcd` uses this information (all but the hashes) to identify endpoints that are likely candidates for pairing. By receiving this information *immediately*, without waiting for the hashes, `ipcd` can eagerly detect if multiple pairings are possible due to overlapping address/port pairs and timing information. After N bytes have been sent in one direction on the stream, `libipc` contacts `ipcd` to attempt to find a matching endpoint. Since the N-byte transfer almost certainly takes significantly longer than reporting the connection information to `ipcd` for reasonable[4] values

[4]In our prototype, we use $N = 2^{16}$.

of N, this ensures that if a mispairing is possible *it is detected before the optimization happens*. In this case, the stream is conservatively switched to the 'No-opt' state, and optimization is aborted.

If a single matching endpoint is found, `ipcd` initiates the optimization procedure, explained in Section 3.4.4. If a matching endpoint is not found, `ipcd` records the current endpoint in a list and waits for a match, while `libipc` tries again several times after which it declares the procedure has failed. In this case, `libipc` changes the state of the *EPid* to the 'No-opt' state. The last request by `libipc` for a matching endpoint is sent with a flag telling `ipcd` to remove the list entry if it is not matched. This removal serves two purposes: first, it helps eliminate matching errors by preventing stale endpoint data from being matched; second, the atomic "request-or-removal" avoids the issue of having only one endpoint aware of a pairing: `ipcd` only pairs endpoints if they have both indicated they will check again for a pair.

3.4.4 Transparent Transport Switching

Once `ipcd` has determined that two local endpoints are communicating with each other over TCP, `ipcd` generates a pair of connected sockets using a faster transport (in our implementation, Unix domain sockets) and passes them to the `libipc` instances controlling the communicating endpoints. Generating the new sockets in `ipcd` and not `libipc` allows the procedure to work even when the two `libipc` instances cannot directly communicate, such as between separate Docker containers. Upon receiving its side of the faster socket, `libipc` copies appropriate flags from the old socket to the new socket and then changes the state of the *EPid* to 'Opt'.

`libipc` then migrates communication to the new channel for improved throughput and latency. The primary challenge in doing this correctly is ensuring that both endpoints will switch to the new channel at the same position in the data streams.

To make this switch, `libipc` ensures that a `send` or `recv` request ends at exactly N bytes. For a non-blocking `send` or `recv` operation on the TCP socket, `libipc` merely truncates this request, returns a short write and lets the application issue the next request as normal. For blocking operations, in which a short write could be misinterpreted by the application, `libipc` instead splits the request into two pieces and processes them internally as two requests but only returns control to the application after both requests have been processed. After splitting the request, `libipc` attempts to optimize the endpoint as detailed above, and subsequently transfers the remaining bits of the request using the selected transport, except that the request is handled in a non-blocking manner to better emulate what would have occurred if the entire request had been processed without `libipc` intervention.

3.4.5 Emulating TCP in User-Space

The bulk of the socket API has a straightforward implementation in `libipc`: whenever a file descriptor that has an underlying endpoint identifier is mentioned, the real API is called with either the optimized transport's file descriptor (if in the 'Opt' state) or the original TCP socket's file descriptor (otherwise). Some system calls, however, require a more complex implementation.

A global connection table is maintained by `ipcd` to coordinate the matching process. Entries are created in this global table whenever functions like `socket` and `accept` request to create a TCP socket and are initialized with properties about the socket such as the local and remote IP address. Removal from the global table only happens when a `libipc` instance determines that all file descriptors within that process that refer to a single endpoint have been closed. Since it is possible to share endpoints across multiple processes via use of `fork`, `ipcd` maintains a count of the number of processes using a single endpoint identifier.

The basic read and write functions (`send`, `recv`, `recvmsg`, etc.) require more work when the endpoint identifier is in the 'Pre-opt' state, where the tracking of the first N bytes and the seamless transition steps described in Section 3.4.4 need to be executed in addition to the normal I/O.

Emulating `fork` requires more care due to the introduction of multiple processes that could potentially race on communication to `ipcd`. This race is resolved by having `libipc` inform `ipcd` that all of its endpoint identifiers are about to be duplicated before making the call to `fork`; should the call to `fork` fail, `libipc` tells `ipcd` to close the duplicated file descriptors. If `libipc` were to notify `ipcd` of the endpoint identifier duplication after the call to `fork`, then a child process that immediately calls `close` on its copy of the file descriptor would cause `ipcd` to prematurely clean up the connection.

While `fork` is emulated by `libipc`, the implementation is only sufficient to support sharing file descriptors across multiple processes if only one process at a time communicates on it. This level of support is sufficient to support most forking server applications, in which the parent creates a socket, forks, and then closes the socket while the child process does all of the communication on said socket.

The `exec` family of functions implicitly refer to file descriptors by the need to clean up those that have the `CLOSE_ON_EXEC` flag, but they also pose a challenge to support since the internal memory, including both the code and data segments of `libipc`, is completely wiped. `libipc` retains its memory across the `exec` call by copying it to a shared memory object tied to a file descriptor that is retained across the `exec` call. If the new process uses `libipc`, the initialization process first reads this table and initializes its current state. If the new process does not use `libipc`, the tie to the file descriptor at least ensures that the eventual death of the process will clean up the system resources allocated by `libipc`.

3.5 DISCUSSION

The current design of Slipstream assumes no ability to communicate protocol data using packets within the original stream, i.e., to add reliable "out-of-band" (OOB) signaling between two `libipc` instances connected in a stream. This has implications for performance, correctness, and security.

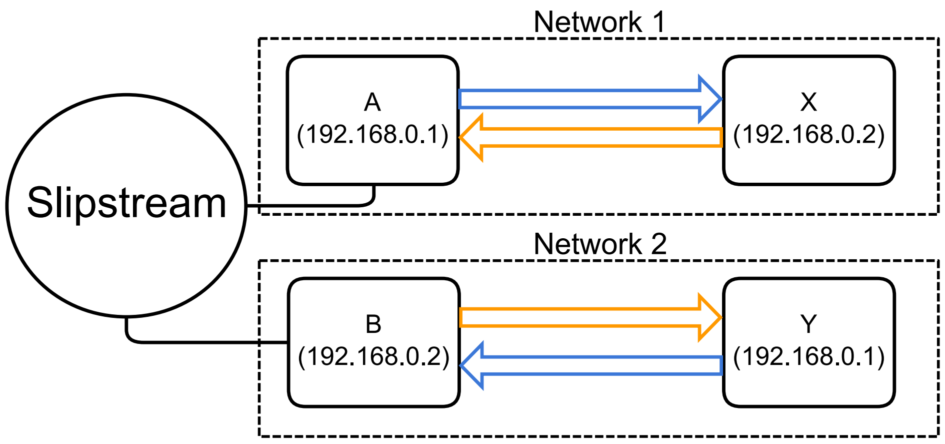


Figure 3.2: Parallel Network Configuration Example.

3.5.1 Performance

The implication for performance is that Slipstream may fail to optimize some instances of host-local TCP communication, i.e., we may have *false negatives*. For example, Slipstream may conservatively decide not to optimize a stream if it exchanges fewer than N bytes, or if the match is not identified within a short time interval.

More generally, we focus on optimizing using observations external to the TCP stream itself that are readily available on any system. Moreover, we design the protocol conservatively to avoid *false positives*, which represent correctness violations. To avoid such errors, Slipstream must be able to disambiguate TCP streams in fairly complicated scenarios such as multiple identical virtual networks with endpoints having identical conversations.

3.5.2 Correctness

It is virtually impossible *through an accidental misconfiguration* for Slipstream to incorrectly match and optimize endpoint communication. A "mispairing" (or "false positive" or "incorrect match") can only occur if *all* of the following are true:

1. There are multiple TCP streams described by the same 4-tuple $\langle$SrcIP, SrcPort, DstIP, DstPort$\rangle$, and there must be at least one common host system running more than one stream.
2. These streams were established with overlapping timings.
3. The first N bytes of these streams are identical.
4. Slipstream is deployed to some, but not all, of the endpoints described in these streams.

An example of a scenario that would be needed to cause false positives is shown in Figure 3.2. In this example, A and B are using Slipstream but X and Y are not. The ports used by A and Y (not shown in figure) must be the same, and so must be the ports used by B and X. This scenario would then satisfy condition 1 as the same 4-tuple of the IP addresses and ports identifies both concurrent TCP streams, A–X and B–Y. In addition, these endpoints must establish connections at approximately the same time (within the time window used by a `libipc` instance to poll its associated `ipcd` for a match; not greater than 100ms), and both connections must communicate the same first N bytes of data. Only if all these conditions hold will Slipstream erroneously attempt to pair endpoints A and B.

The need for the first three conditions is obvious from our endpoint matching protocol. The first condition cannot occur within a single well-behaved network; for the same IP address to occur in multiple distinct streams, there must be local endpoints that reside on distinct networks making use of the same IP addresses. Even in such scenarios, the ports used must match as well, which is unlikely because it is very common for clients to use randomly generated port numbers (called *ephemeral ports*) when setting up connections with servers, by using a range of dynamic port numbers set aside for this purpose [105].

Condition 4 is required because if Slipstream is deployed to all endpoints, the possibility of mispairing will be detected before optimization is attempted, and Slipstream will conservatively avoid the pairing.

While these four conditions are possible, they are unlikely to occur accidentally. A well-configured system would not assign identical IP addresses to different interfaces. The use of ephemeral ports, which are drawn randomly from a fairly large range (e.g., 32768-61000 by default on recent Linux kernels, for IPv4) makes condition (1) even more unlikely. The two connections using those two ports must both be started within a very small window of time. Finally, the connections must send exactly the same N bytes of data, for moderately large values of N (e.g., 2^{16}). As a result, Slipstream is well-suited for most real-world applications and is only unsafe when deployed to applications known to intentionally violate one or more conditions (e.g., regularly sending the same first N bytes).

3.5.3 Security

The key new security risk posed by Slipstream is that an attacker (any unauthorized third party) could try to force a mispairing, i.e., that the attacker is given read or write access to a TCP stream to which she did not have access previously. The threat model we assume is that the attacker must have local access to a machine where either endpoint of some local stream lives (necessary to talk to `ipcd`), *and* does *not* have root privileges on that machine (with root, more powerful attacks are possible even without Slipstream).

In the absence of root access, it is impossible for the attacker to forge IP headers or to misconfigure a second network to obtain duplicate IP addresses as an existing network. In our current *implementation*, however, we trust that each `libipc` is being honest in describing the information about its socket connection. A `libipc` controlled by a local attacker could simply "lie" about its IP address and port number and could potentially construct the remaining information, including the N-byte hash, in order to fool `ipcd` into giving it an optimized endpoint incorrectly. A simple solution is to give `ipcd` sufficient privileges to verify the IP address sent to it. On a well-configured

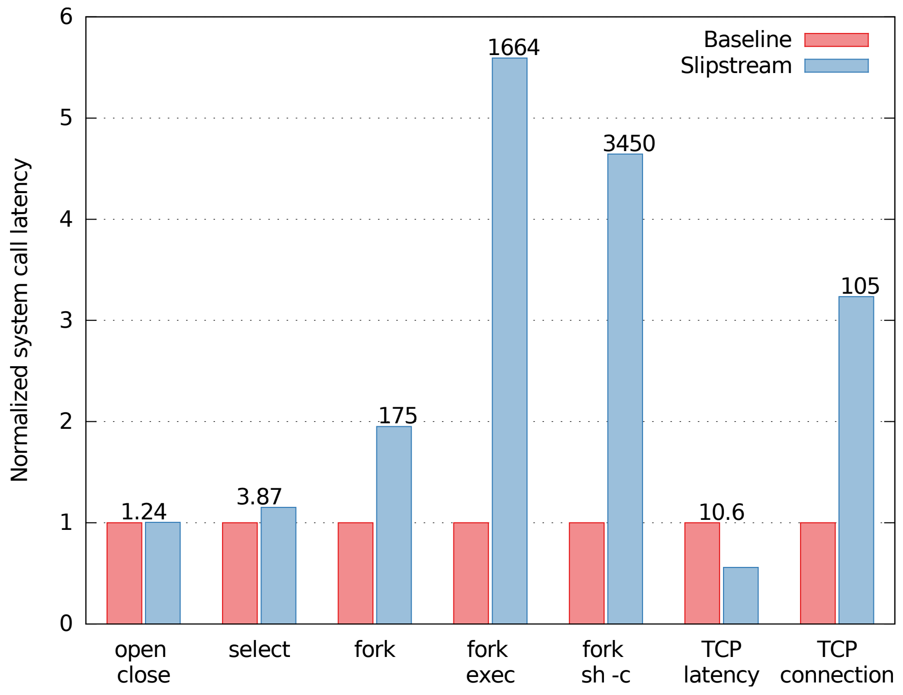


Figure 3.3: System call overheads. Values for Slipstream in microseconds are shown above the bars. Lower is better.

system that is not running a Docker-like environment, this is sufficient to prevent a non-root attacker from impersonating another endpoint.

On a system running a Docker-like environment, it is plausible that multiple containers are assigned the same IP address in a virtual network configuration. The default Docker configuration, however, is to assign all containers unique IP addresses from a single virtual network, which prevents condition 1. Restricting Slipstream to use only in the default configuration therefore eliminates any security risk from untrusted containers. We leave it to future work to support non-default Docker configurations with duplicate IP addresses securely. For example, it might be possible to abandon our initial assumption and extend Slipstream to exchange data reliably "out-of-band" but within the TCP stream by building on existing approaches in the literature, e.g., through covert use of various TCP/IP headers [106]. This would add some complexity to `libipc`, but would be justified in large installations (e.g., a data center) in which the one-time cost of enhancing `libipc` would benefit many customers.

3.6 RESULTS

To evaluate Slipstream, we use a suite of microbenchmarks and applications that measure the performance of various aspects of networking. We have two primary goals in this evaluation. First, we aim to measure the performance impact of Slipstream for network microbenchmarks and for networked applications. Second, to investigate the performance impacts in more detail, we measure the performance *overheads* incurred by common system calls due to the extra bookkeeping necessary for Slipstream.

We perform all of our experiments on a workstation with a 4-core Intel x86-64 processor with 16GB of DDR3-1333 RAM. This workstation runs Ubuntu 14.04 using stock packages, including most notably Linux 3.13.0-36 as the base kernel, Docker 1.0.1, and OpenJDK 1.7.0_75 for the Java VM. All of the networking configuration parameters for the Linux kernel have been left set to their default values.

3.6.1 Microbenchmarks

We use the NetPIPE and Netperf microbenchmarks to measure total networking throughput under different networking communication patterns. We use two variants of NetPIPE, one in C and one in Java, to show the impact for two different programming languages; in fact, Slipstream is able to optimize Java code running on OpenJDK as transparently as it does for C code, as explained below. Another microbenchmark, lmbench [107], measures the overhead of Slipstream on common system calls.

3.6.1.1 lmbench

lmbench [107] is a microbenchmark that measures the overhead of various system calls, which gives an indication of the penalty incurred by using Slipstream in code that may not benefit from its improvements. Selected results are shown in Figure 3.3; the other numbers are omitted because they are unaffected by Slipstream.

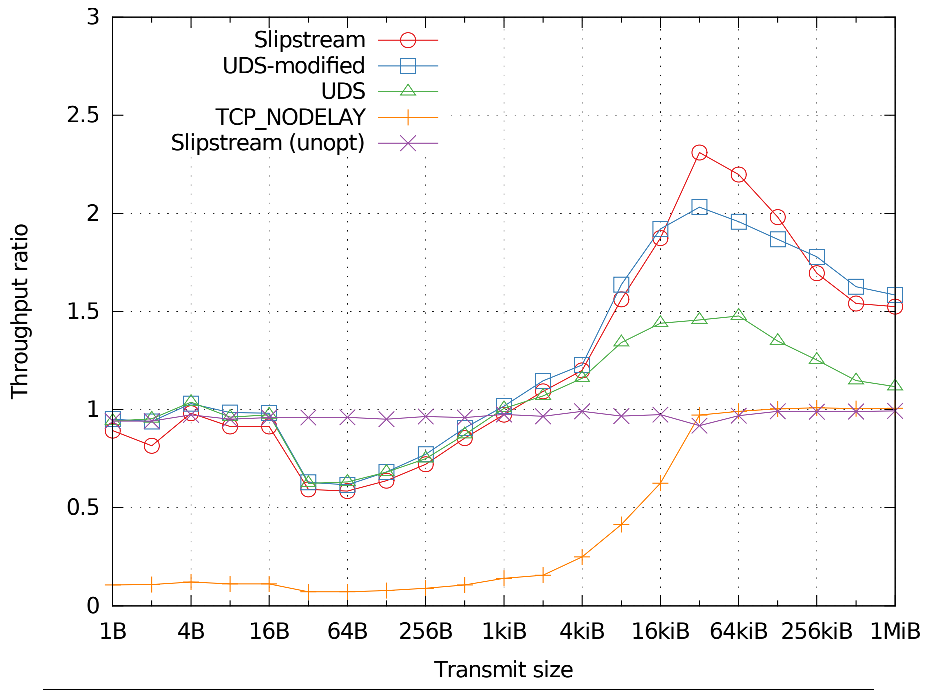
Due to the tracking of file descriptors within userspace, Slipstream naturally adds a small amount of overhead to most system calls that interact with file descriptors. For example, an `open` and `close` pair is 5% slower with Slipstream, while a `select` over 100 socket descriptors is 15% slower.

The tracking of file descriptors imposes the largest overheads on `fork` or `exec`. For `fork`, this is due in large part to synchronous communication between `libipc` and `ipcd` that blocks the actual system call. In contrast, the overhead in `exec` is due to loading `libipc` in the memory space of the new process, as well as the overhead of setting up the shared memory object to retain `libipc` state across the call (see Section 3.4.5).

TCP latency, when the connections have been optimized, is brought down to the same latency as UDS: about 10 microseconds, about half the original TCP latency. However, the initial connection latency is greatly increased due to our need to register the new connection to `ipcd`.

3.6.1.2 Netperf

Netperf [108] is a microbenchmark to measure total throughput of network connections. Netperf sends data unidirectionally, creating a new socket for each transfer size. The sizes transferred are chosen in logarithmic fashion. Results are presented in Figure 3.4.



Experiment	1 B	32 B	1 KiB	32 KiB	1 MiB
Baseline	2.06	98.28	1625.83	3891.92	5193.10
TCP_NODELAY	0.22	7.11	228.34	3785.77	5232.23
Slipstream	1.84	58.30	1585.19	8988.88	7917.70
Slipstream (unopt)	1.94	94.32	1585.71	3576.18	5159.05
UDS	1.94	61.36	1636.47	5670.26	5801.53
UDS (modified)	1.96	61.93	1655.32	7905.42	8226.89

Figure 3.4: Throughput as measured by Netperf, with a baseline of TCP without Slipstream or `TCP_NODELAY` specified. The table contains a subset of the throughput results, measured in MB/s.

At certain smaller buffer sizes (e.g., 32B-256B), both Slipstream and UDS perform roughly 25-50% worse than the TCP baseline in terms of total throughput. This effect is due to synchronization overhead inside the Linux kernel, which is not observed by the baseline because `TCP_NODELAY` is disabled and TCP buffer coalescing occurs. To validate this, we also compared TCP performance with `TCP_NODELAY` enabled; that curve clearly shows the negative impact of eliminating TCP buffering. At higher data sizes, the relative overhead of the synchronization vs. data transfer is reduced, and Slipstream and UDS sockets both perform better than the baseline, mimicking the results for other benchmarks.

Surprisingly, we observed an increase in throughput with Slipstream compared to using UDS for some of the larger data sizes. On further investigation, we found that this extra speed is primarily due to Netperf using a very small socket receive buffer size (2304 bytes) for the UDS tests. When we changed the Netperf code to not set a socket buffer size (labeled "UDS-modified" in the graph), the apparent effect largely disappears.

Finally, to measure the impact of using Slipstream when optimization is not possible, we ran Netperf using Slipstream only on the client. As shown in the figure, performance was generally very close to that without using Slipstream, on average 3.5% slower than baseline.

3.6.1.3 NetPIPE

NetPIPE [109] is another microbenchmark that measures the throughput of TCP. NetPIPE differs from Netperf in that it transfers its data bidirectionally and that it reuses the same socket for all of the transfer sizes.

Both variants of NetPIPE use the same basic networking structure, in which the client socket sends data of a given buffer size that the server receives, and then the server sends back that data; the process repeats until sufficient measurements are taken to reliably estimate the throughput. They also both use the idiomatic synchronized socket functions for the language—`send` and `recv` in C, and `java.io.InputStream` and `java.io.OutputStream` in Java.

The results of running NetPIPE-C are presented in Figure 3.5, and the results for NetPIPE-Java are presented in Figure 3.6. For sizes less than about 64KB, Slipstream is able to consistently achieve around 70-150% more throughput compared to baseline TCP. At higher sizes, Slipstream does not provide as much improvement, but is still able to produce at least a 40% increase in throughput.

We also observe that Slipstream optimizes Java networking performance transparently, with no changes to the JVM or the application. Slipstream is able to achieve this because it interposes on `libc`, which is also used by OpenJDK, providing essentially the same benefits to Java programs as to C programs.

3.6.2 Application Benchmarks

Memcached and PostgreSQL are two example applications that are sometimes used in ways that put the client and server on the same host. We evaluate these applications with representative workloads to (a) demonstrate that Slipstream is indeed fully transparent for important real applications; and (b) to determine the impact of improving TCP performance for real applications. In addition to measuring performance benchmarks, we used Slipstream on a set of applications—ZeroMQ, OpenSSH, Jenkins (Java), Apache, iperf, simple python TCP client/server, nepim—to informally evaluate compatibility when local to remote TCP communications operate through Slipstream: all of these functioned correctly, and all except OpenSSH were successfully optimized. OpenSSH writes to a socket from multiple processes in a way that we do not currently support in our implementation.

3.6.2.1 Memcached

Memcached [101] is a distributed, in-memory key-value store that is primarily intended to be used to cache database queries for web applications. While Memcached can be configured to listen

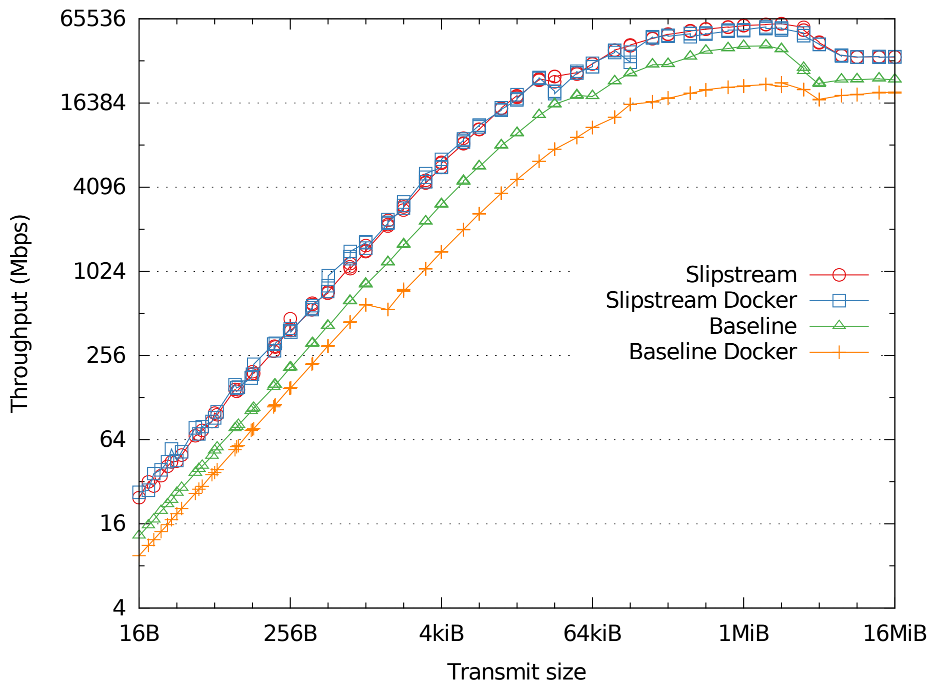


Figure 3.5: NetPIPE-C performance both with and without Docker containers.

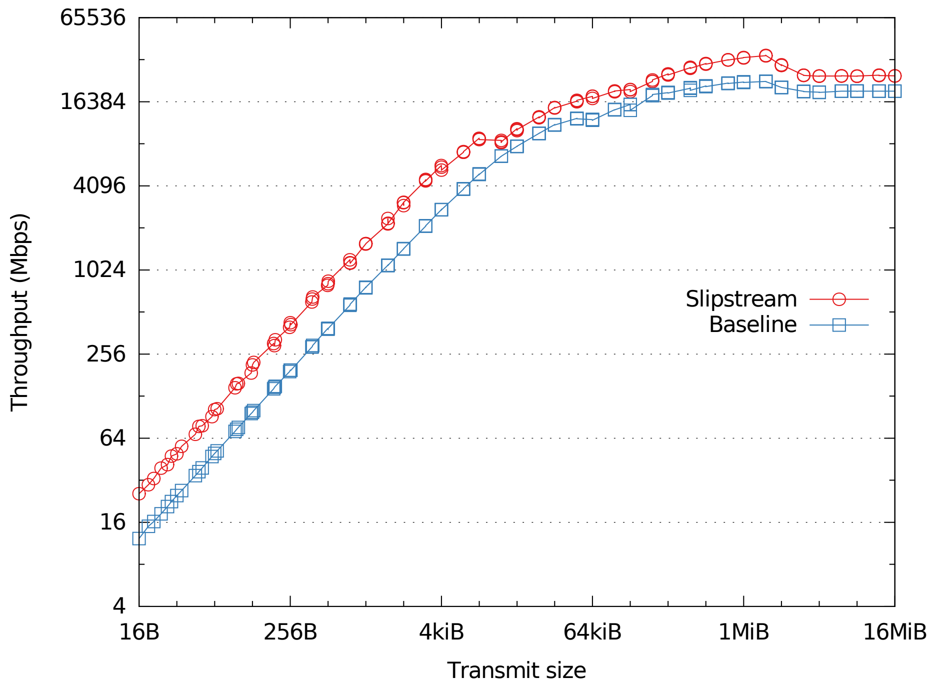


Figure 3.6: NetPIPE-Java, run only outside of Docker.

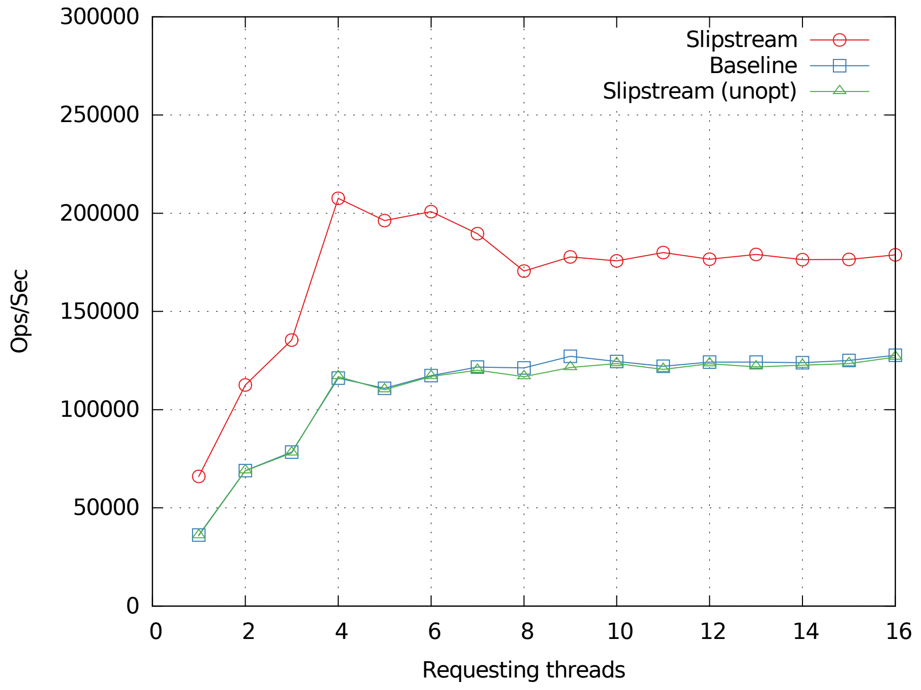


Figure 3.7: Memcached throughput on host system.

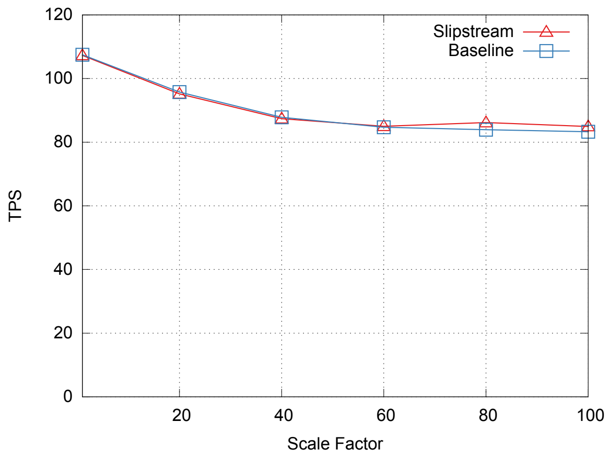
on Unix domain sockets instead of TCP, feature requests to allow it to listen on both have been rejected since the particulars of the socket it listens on are used in the distributed hash function [100].

For testing, we run Memcached using a single server and 2GB of storage. Queries are executed against a pool of 10000 items each between 1 byte and 4KB in size. The number of connections is varied, and the average number of operations executed per second is observed. Results are presented in Figure 3.7. Slipstream provides a 25%-to-100% improvement in throughput, which would be a substantial benefit for small Web sites where most of the traffic is local. When using Slipstream on the client but not the server, we measured on average 1-3% slowdown.

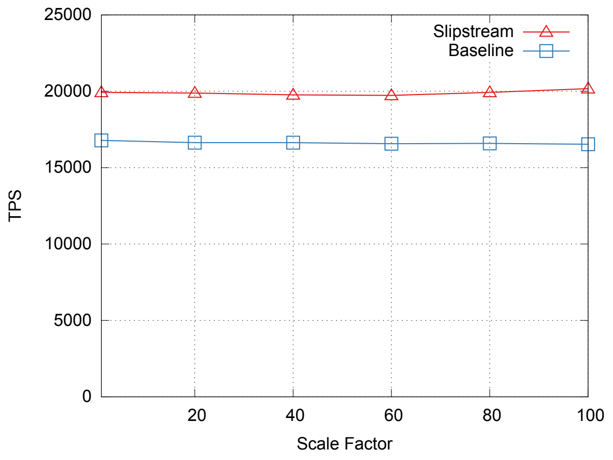
3.6.2.2 pgbench

pgbench [110] is a benchmark for PostgreSQL, a widely used open-source relational database. We run pgbench with two separate workloads, one based on the industry-standard TPC-B benchmark and the other based on a `SELECT`-only benchmark, which spends more time in communication. Slipstream successfully optimizes all TCP communication in both cases. A benchmark scale factor, N, creates $100000N$ rows in the database or about $N \cdot 16$ MB of total database size [111]. Figures 3.8a and 3.8b show the results (in database transactions per second, or TPS). Each data point represents the average of multiple runs; the variance observed was negligible.

The TPC-B workload shows little improvement, which is not surprising because the workload is designed to stress the database's internals (primarily disk access) [112]: communication changes have little impact.



(a) pgbench transactions per second, TPC-B.



(b) pgbench transactions per second, Select.

Figure 3.8: pgbench performance benchmarks.

In contrast, the `SELECT` workload shows 16-23% improvement in database throughput for all scale factors. This workload performs simple queries that are processed by the database at a much higher rate and, as a result, benefits significantly from communication optimization.

3.6.3 Docker

Slipstream is able to detect and optimize local TCP traffic within a single network (i.e., to localhost), but also across virtualized networks executing on the same machine such as those created by Docker. To demonstrate this functionality, and evaluate the performance characteristics of Slipstream in this environment, we conducted additional experiments across Docker containers on the same host. Rerunning our experiments within the same Docker container do not produce different results than by running them on a regular Linux setup.

As different Docker containers have distinct filesystems from the host system, using Slipstream from within a container requires an extra configuration step. If `ipcd` is installed into a Docker container of its own, then the directory containing the UDS that `ipcd` listens on can be also installed on other containers by leveraging the volumes feature of Docker. Alternatively, it is also possible to bind this directory from the host system to a Docker container. Either way, using Slipstream with Docker only requires ensuring that `libipc` is installed within the container and adding a single flag when running the container. *This flexibility is a key benefit of the routing-oblivious design of Slipstream.*

3.6.3.1 Docker Microbenchmark

We reran our NetPIPE-C microbenchmark using Docker to illustrate the basic performance of Docker networking, also shown in Figure 3.5. The graph shows that Slipstream is not slowed down when used across Docker containers, whereas normal TCP is, magnifying the TCP throughput

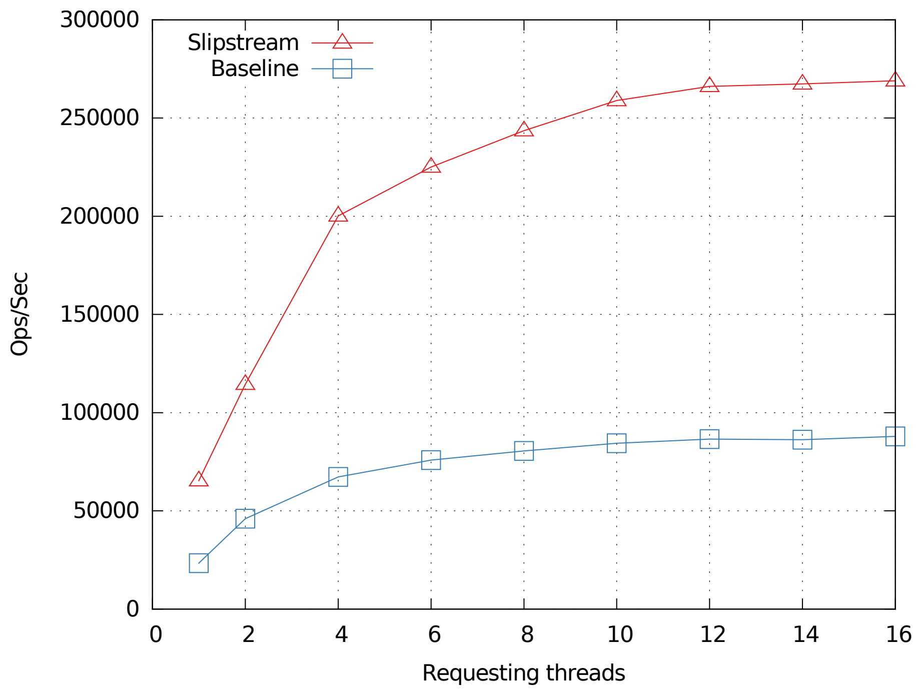


Figure 3.9: Memcached throughput with Docker.

improvement to between 150% and 350%. Since Docker containers use separate networking namespaces, the kernel layers need to swap packets between different interfaces, which imposes an extra overhead on TCP transfer, layers which are bypassed by Slipstream. Thus, Slipstream's benefits are only enhanced when Docker is in use.

3.6.3.2 Docker Application Benchmark

In addition to the basic TCP throughput benchmark, we also evaluate the performance of Memcached across Docker containers on the same host. This experiment is identical to the Memcached experiment without containers, with the sole difference that the server and client live in separate containers. These results are shown in Figure 3.9. Comparing with Figure 3.7, we again see substantially greater improvements due to Slipstream with Docker than without, ranging from about 100% to 200% speedup. These are very large *application-level* improvements, showing that Slipstream can be a valuable and transparent way to improve the overall performance of services that use Docker containers.

3.7 ACKNOWLEDGEMENTS

This work was supported in part by the Office of Naval Research grant numbers N00014-12-1-0552 and N00014-4-1-0525, and by the AFOSR under MURI award FA9550-09-1-0539.

3.8 CONCLUSION

Slipstream is a novel system for the optimization of TCP communication that requires neither OS nor application modification, which allows it to be easily and rapidly deployed. Our evaluations show that our system is capable of achieving significant performance benefits, at least 16% more throughput than TCP, and up to 200% if Docker is involved, both on real applications in real usage scenarios. Slipstream's minimal assumptions allow it to be used in a variety of network topologies and to use a variety of faster local transports, capabilities we plan to explore in future work. We believe that Slipstream provides an excellent base for reducing the overhead of IPC in applications that is usable across a wide variety of applications and setups.

CHAPTER 4: ALLVM

We don't know ourselves except through listening to our echoes

—Alan Watts

4.1 INTRODUCTION

To resolve the real-world open-source "mess we're in", this chapter introduces a new approach for improving software systems through the use of compiler-based techniques.

The foundation of this work is **ALLVM** - an environment and tool suite in which all software is represented in the LLVM IR, making possible analyses and transformations across various traditional software boundaries. No system today allows compiler analyses and transformations across arbitrary software boundaries; ALLVM makes this natural.

4.1.1 Allexe

Programs in ALLVM are provided in a novel format called *allexe*. An allexe is, at its most basic level, a collection of code represented in the LLVM bitcode format. The individual elements within an allexe are LLVM modules, including the primary program's code as well as code from requisite libraries. By storing code in the LLVM bitcode format, it is trivially convertible to LLVM IR and its associated textual representation. This format, then, means allexes benefit from the properties considered inherent to the LLVM IR, namely, human-readability and tool-friendliness. All of the analyses, optimizations, transformations, and capabilities provided by LLVM itself and its rich ecosystem can be used with allexes.

Importantly, what differentiates allexes from other executable formats is that allexes store **all** of the code needed for the execution of the program. This fundamental design principle allows allexes to effectuate the WYSIWYX ideal [16, 113]. This principle is more significant in its implications than the simplicity of its presentation may suggest; every piece of the ALLVM architecture either indirectly or directly serves this goal. In fact, the ALLVM project's entire contribution is to realize this principle – of representing, in a readable and knowable form, the entirety of the program as it will actually be executed.

Allexes are however still normal executables, which is to say they can be invoked as directly as other executables native to the platform. ALLVM is based on Linux, where ELF is the usual format for executables. Similar to the role played by the ELF interpreter (`ld-linux.so`), allexe execution is handled by the `alley` tool. In fact, by virtue of having all the code immediately available in a single location, execution is very straightforward. Uncertainties, such as in library selection, are completely eliminated by the ALLVM design as an allexe is built. The resulting allexe is statically-bound, even if not always explicitly fully statically linked.

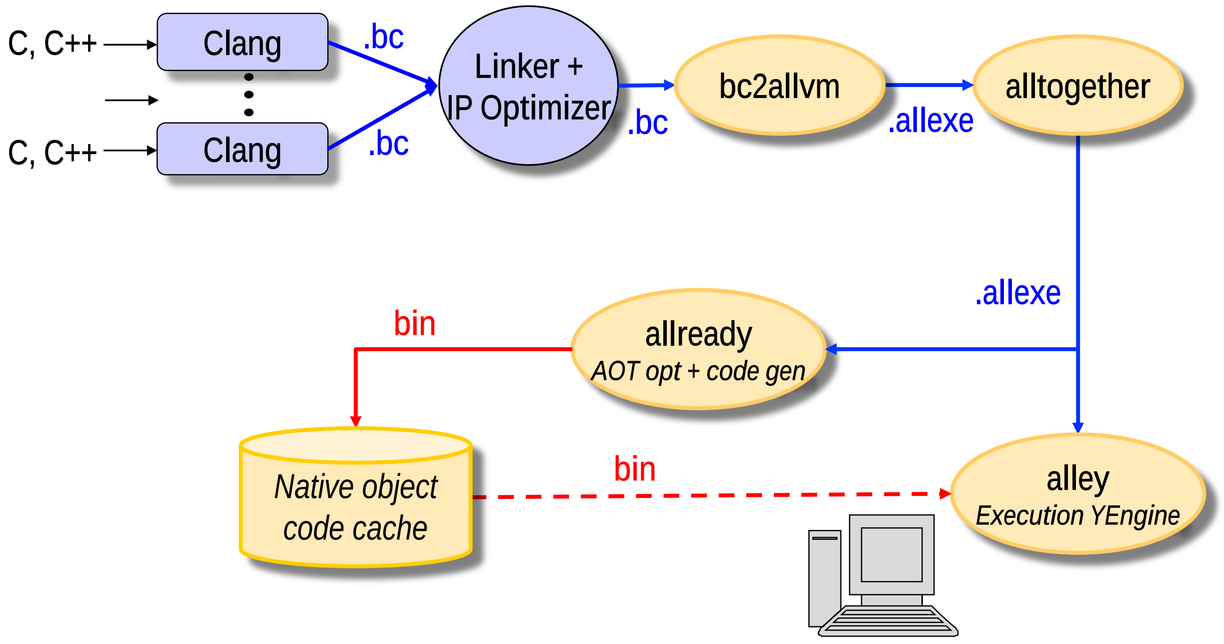


Figure 4.1: *ALLVM tools and software development workflow.* ALLVM uses the allexe file format. Few or no changes to automated build systems are needed to generate and ship applications or libraries as allexes; thousands of Linux packages are routinely shipped and run this way.

An overview of ALLVM components is provided in Figure 4.1.

4.1.2 Knowable Code

There exists no universally agreed-upon definition of "complete program", and scholars are still working on refining what is even meant by "program", which presently varies among fields [24]. These difficulties are made worse by the lack of a common-ground vocabulary to precisely express definitions. Acknowledging these challenges, only informal explanations of terms such as "knowable" and "complete" are presented here.

In short: ALLVM endeavors to be "complete" by including all portions of the program, i.e. all the code that will possibly run during a program's execution, with one caveat discussed later. Similarly, ALLVM intends that the program is "knowable" in that its representation can be meaningfully and directly examined, particularly by any party performing execution of the code.

Regardless of formalisms used, further clarification is provided by describing what is certainly *not* meant, as this is more readily discussed.

4.1.3 Libraries & Dynamic Linking

Abstracting away code, data, and services is an encouraged practice, but the flexibility and modularity gains have a cost when it comes to identifying precisely what constitutes a piece of software.

This is especially true when considering the details of how library dependencies are specified, discovered, and the messy unspecified particulars involved at the low-level. A notable high-level source of problems is that executables do not and can not specify the precise pieces of code required or even exactly which libraries these pieces are to be selected from. Similarly, the handling of transitive dependencies is a source of contention – often leading to “over-linked” or “under-linked” programs and libraries, both with numerous uncertainties in how their constituents are found and processed, what scope their contents are exposed, and so on.

Fundamental to these problems are the use of strings to define and bind interfaces to implementation. Library discovery, to the extent discovery is required at all, occurs through executables and libraries including mention of suggested nicknames for files that may have contents containing the intended pieces [114]. These names are explored in a recursively computed hierarchy of search paths read from files encountered thus far in the search process itself. Further complicating details, such as library versioning, symbol versioning, alternative or competing implementations are discussed in [4, 95, 114].

Indeed, the challenges of sorting out these difficulties directly motivates the development and popularity of “containers” such as Docker [102]. While these technologies provide other benefits, they do not actually directly address this problem. Instead, they sidestep it by simply wrapping programs and the entire environment, so that uncertainty from outside influences is less common.

It suffices to say that there are many obstacles that make libraries, in practice, a significant source of uncertainty about a program’s behavior and contents. The usefulness of dynamically linking and loading new code leads unfortunately to the modern status quo: programs are neither directly portable nor precisely predictable.

The ALLVM platform, in light of this, takes a direct approach to addressing these challenges. Dynamic code loading, including dynamic linking, is eliminated. No new code appears at runtime, at least not through traditional means. Exotic techniques that go out of their way to dynamically load code through unexpected mechanisms, e.g. scribbling into a buffer and then treating it as executable, would not be eliminated. In choosing static behavior, the ability to know and reason about code is restored. Modularity is by no means a casualty; all the options that libraries, for example, present are still available. However, in ALLVM, the specific choice of option is explicitly known.

4.1.4 The C Standard Library, System Intermediary

Though allexes contain “all” the code needed to run the program within, they still rely on a platform. The allexe interacts with the platform exclusively through the library `libnone` which contains the only portion of the executable in userspace that is not represented by IR nor contained within the allexe. Derived from `libc`, the C Standard Library, this platform is essential for the creation of most non-trivial or useful programs: without it, an allexe would be unable to input, output, or otherwise interact in any conventional way with the world outside its closed domain.

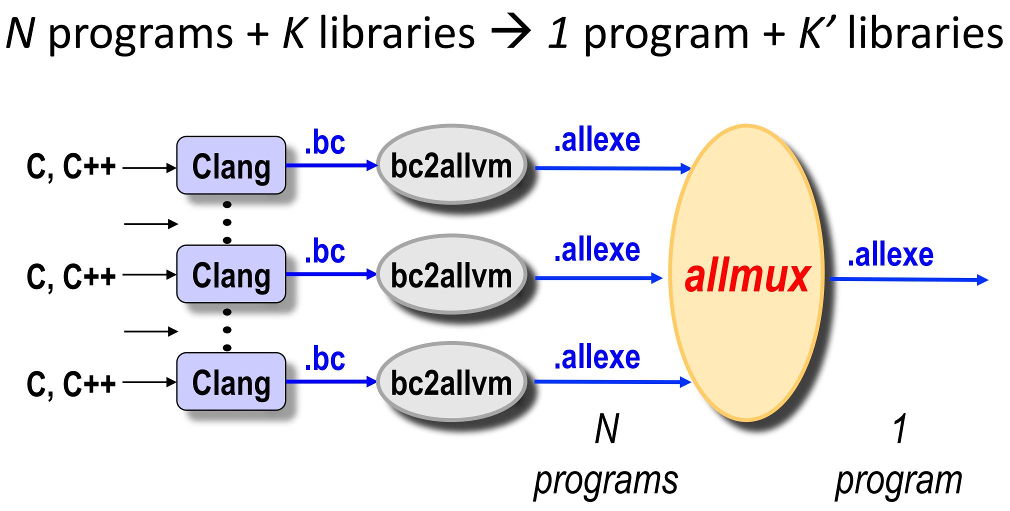


Figure 4.2: By compiling all the bitcode for every user desired component, the duplicated code between programs and libraries is automatically exposed. Additionally, by including all dynamic libraries before code generation enables static linking without rewriting the build system. For more information on `allmux`, see Chapter 5.

Traditionally, libc serves this role of "system intermediary" [115], providing all the conventional interfaces to OS/kernel resources and services expected by modern applications following POSIX standards. These include but aren't limited to file-related operations, I/O, processes, threads, locks, networking, IPC, memory, and so on.

Additionally, libc is responsible for executing programs, starting with the dynamic linker/loader (ELF interpreter). As such, libc handles responsibilities including, among many others, parsing ELF headers of program and libraries, mapping segments of libraries into memory as appropriate, searching for and processing library dependencies, initializing some basic data-structures and other setup such as invoking static constructors, and finally calling the program's `main()` entry point. As part of dynamically loading and linking libraries, libc processes entries in the relevant relocation tables, patching the program replacing placeholders with resolved and appropriately modified addresses as indicated by the relocation type.

4.1.5 Musl - Designed for Static Linking

ALLVM's libnone is largely based on musl libc, which its authors describe as "lightweight, fast, simple, free, and . . . correct in the sense of standards-conformance and safety" [77]. These values and other specific details make musl an attractive foundation for ALLVM's pursuit of knowable code.

In particular, musl emphasis on rigorous guarantees and standards-compliance are especially valuable, as they make it possible for program analyses, and optimizations based on them, to reason about behavior of all of libnone and thereby the entire execution domain of an allexe. For example, musl ensures that all error conditions are detectable. Importantly these results can be trusted without need to inspect observed runtime behavior, as such deviations are bugs in musl or ALLVM.

Unlike nearly any other libc implementation, musl was “designed from the ground up for static linking” [116]. Static linking is conceptually aligned with ALLVM, but practically speaking, has the benefit of making it possible to generate standalone binaries with only the portions of the already intentionally minimal musl included. Because of this, ALLVM produces smaller binary sizes, accordingly reducing footprint on disk and in memory, as well as launching more quickly. The magnitude of these benefits varies, influenced by a number of factors. However, even without these benefits, musl’s ability to create feature-complete, fully-statically linked, standalone binaries is critical for ALLVM’s allexe format.

Musl is popular in the realm of compiler-based analysis, partially because its intentionally simple source and source tree allow easy customization. Most mature analysis frameworks already use the libc boundary in their models and abstractions. In the same way, many details not capturable in LLVM IR are abstracted behind a function call at this same libc boundary. As a result, most of these tools and techniques will remain easily applicable, while many analysis frameworks would struggle to deal with fully-exposed very low-level details involved in the implementation of libc. By basing the ALLVM platform upon this same natural boundary, we preserve the already-familiar relationships used in existing analyses.

4.1.6 Libnone - ALLVM’s platform

ALLVM’s libnone platform, in terms of implementation, is essentially musl libc, with a few modifications toward satisfying ALLVM’s fundamental design principle of having allexes store all the required code. In particular, it was necessary to disable some undesirable behaviors that would otherwise be possible with musl libc.

The most important of these modifications is to prevent conventional methods of direct program introspection. ALLVM treats the specific implementation as an artifact, not the actual design. For example, an allexe executed using JIT or static AOT strategies will appear different when introspected, and more sophisticated techniques such as software multiplexing further discourage support of this feature, at least as traditionally implemented. In ALLVM, the executed code and the knowable allexe are technically distinct yet critically represent precisely the same program entity. In practice, then, preventing introspection has fewer consequences than it would in a traditional environment. That is to say that many questions answerable by currently typical low-level introspection can instead be answered at a much higher level.

Libnone disables the `dlopen` and `dlsym` behaviors, returning appropriate error messages indicating these are not supported. Other introspection methods are also disabled or otherwise rendered ineffective, such as reading from `/proc/self/exe`, inspecting contents of `argv[0]`, or attempting to find or parse ELF headers on filesystem at locations currently containing allexes. This is not a significant limitation, as other than explicit debugging tools, these methods are not common.

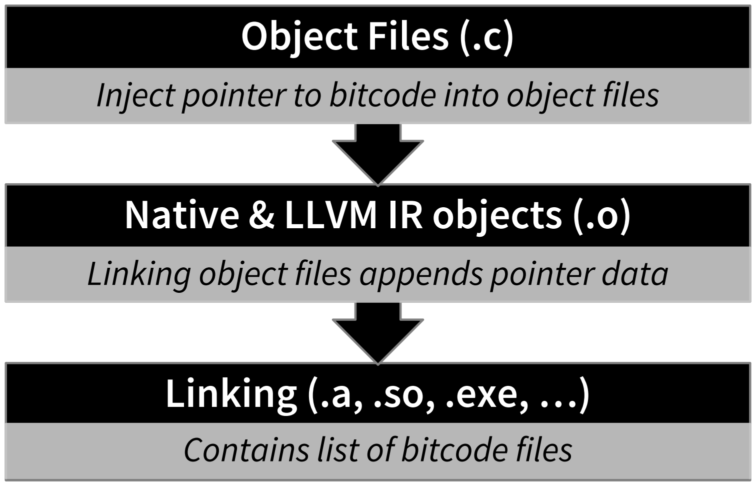

Object Files (.c)
Inject pointer to bitcode into object files
Native & LLVM IR objects (.o)
Linking object files appends pointer data
Linking (.a, .so, .exe, …)
Contains list of bitcode files

Figure 4.3: Using WLLVM, we build allexes transparently and automatically. The native code is the primary format for build compatibility. Pointer data is null-terminated for delimitation upon completion.

Debug information is currently stripped due to bugs in the incorporated version of LLVM, leading to extraordinarily poor scaling behavior when combining modules containing debug information. This limitation only affects the current ALLVM implementation; it is not inherent to ALLVM's fundamental architecture.

Unwinding is important in many programs, particularly for exception handling support. Much of LLVM's libunwind library can and should be included in allexes requiring this support. Toward a design goal of confining all non-IR code into a single place, libnone includes the two particular assembly functions needed for fully saving and restoring register state, or “context”.

4.2 FORGING AN ALLEXE

4.2.1 Leveraging Nix

Allexes are built leveraging the Nix package manager's build process. Recall that Nix utilizes a pure functional language to specify build instructions in the form of Nix expressions. The build process evaluates a Nix expression, in fact a composition of expressions, with the basic expression for building the package requested at the base. The evaluated results, or *derivations*, are sent to builders which reproducibly actually produce the package from the inputs, operating in a carefully constructed clean sandboxed environment.

Most interaction with Nix is done by authoring package expressions and invoking `nix` tooling on trees of expressions communally maintained (Nixpkgs), which contain a great number of package expressions and all the glue that keeps the expressions simple yet powerful. Package expressions are generally written as short functions that take as arguments a standard build environment (`stdenv`, expected to contain a C compiler among other basic utilities) and all build and runtime dependencies, with the primary portion of the expression describing how to build the desired software in terms of these locally unbound names. The expression also specifies simple metadata such as name and version, as well as how to obtain the source and a trusted checksum of its contents. This use of functional expressions for packages is what empowers Nixpkgs to be extremely flexible and extensible, and is what provides appeal to ALLVM and compiler-based research of any kind interested in open-source software.

ALLVM employs modifications to the `stdenv` abstraction (to change the compiler used, for example) to forge allexes. Using the Nix language itself, ALLVM concisely and elegantly describes transformations across sets of packages, individual packages, and the transitive closure of their dependencies, producing iterative stages that ultimately are entirely allexes. A critical advantage to this approach is that ALLVM is immediately and automatically able to create thousands of allexes using the existing Nixpkgs framework.

Despite the additional complexity introduced by ALLVM, the user experience for building, e.g. vim as an allexe (`nix build allexePkgs.vim`) is little changed from building it traditionally in Nix (`nix build vim`). Indeed, a variety of sets of packages built in different ways or with different flags are available and easy to generate. For example, `allexePkgs` is the Nix expression for "create the `allexePkgs` set by mapping `pkgToAllexe` over the attributes in the `wllvmPkgs` set".

The changes required for ALLVM support in Nixpkgs were written with intention for eventual inclusion upstream, with multiple parties expressing interest in helping see this through. Already these efforts produced substantial submitted and accepted improvements to Nixpkgs: support for Clang, the addition of musl as a semi-supported platform, and a multitude of changes to the build abstractions used by ALLVM as well as cross-compiling scenarios. This approach also means ALLVM's allexes receive updates and improvements naturally as the tree is updated to follow upstream. In this way, ALLVM has been architected to be robust across changes for at least the near future and hopefully beyond, without extensive maintenance.

4.2.2 Build Expressions

The most important details regarding the Nix build expressions used by ALLVM are discussed below. True minutiae are omitted, the majority having been addressed and fixed in upstream Nixpkgs.

In the path toward allexes, the first waypoint is the construction of the package set `wllvmPkgs`. This is an attribute set where $W(a)$ for every attribute a in the `wllvmPkgs` attribute set, and for the transitive closure of its build dependencies. The property W is true for a iff a has been built using the `wllvmStdenv` and all inputs are W.

Creating `wllvmPkgs` requires at least three stages, coarsely explained as each stage building an iterative wavefront of new packages starting with a minimal (often carefully manually constructed) bootstrap set. These packages are built in a manner usually used only for cross-compilation, since ALLVM is not recognized as a proper build target and because, when the project started, ALLVM tools and allexe execution were not mature enough to support treatment as a proper platform. Instead, in a hybrid approach, the custom `clangStdenv` is used to create the `wllvmStdenv`.

4.2.3 Standard Environments

When using `clangStdenv`, packages are built in the normal manner but with an ALLVM-friendly compiler stack instead of the usual Nixpkgs default. Happily, Nix is well-equipped for this sort of change and many such diverse compilation stacks can easily coexist and concurrently execute without issue. Significant engineering effort, including modifications to package expressions upon occasion, were required before `clangStdenv` could support the majority of Nixpkgs. Much of this is thanks to Nix and Nixpkgs' engineering, as well as building on work in the reproducible-builds.org project. Some assistance in this effort came from Nixpkgs users interested in its use on Darwin as well, and the entirety of these changes are now part of Nixpkgs proper. Reusing the `vim` example, this stdenv and a multitude of others (especially those used for cross-compilation) are readily available via invocations such as `nix build pkgsMusl.vim`. Recent work adds support for using entirely LLVM-based tools, particularly replacing traditional binutils with LLVM's equivalents as well as using LLVM's linker (`lld`) instead of `ld.bfd` or `ld.gold`.

This `clangStdenv` is used to create the `wllvmStdenv` (the stdenv referenced in the definition of W). This standard environment primarily changes the compiler, linker, and other build tools, in a reliable way that's conceptually akin to `CC=custom-cc` but more robust. For example, all of the default tools are simply not available nor will they be looked for in the standard way by auxiliary build drivers as may otherwise occur. This environment specifies the WLLVM project as the compiler.

4.2.4 WLLVM Compiler

WLLVM is essentially a clang wrapper that, when used as the "real" compiler, executes the wrapped clang twice, producing both the output normally generated by a direct invocation of clang as well as LLVM bitcode alongside. After both are created, WLLVM notes the path of the bitcode in a special format-specific location in the normally-generated output (often a `.o`). When linking, by virtue of WLLVM's clever design these paths are obligatorily concatenated – providing a list of bitcode in the linked output. Later, WLLVM's provided tool or ALLVM's mostly-equivalent `wllvm-extract` can be used to detect and parse such lists, gathering all mentioned bitcode files, and combine them into a single module. The generation of traditional compiled machine code with corresponding bitcode alongside is essential to the success of the allexe build sequence. The presence of the machine code minimizes headaches arising from the traditional build recipes, and is essential for the later step of determining the complete transitive closure of required components for a given program.

WLLVM itself is written in python, which has side effect. Python, as well as its runtime closure, are problematically pulled in as dependencies when they shouldn't be. In ALLVM, this tension is resolved by using a minimal python for WLLVM, by tolerating resulting impurities in early bootstrapping stages and finally by replacing the WLLVM used for third and future stages with one suitably built by WLLVM itself. In this way, the original problematic dependencies are, at a minimum, replaced with WLLVM-enabled versions. Further impurities are explicitly scanned for, and any inputs not from WLLVM are refused inclusion when constructing flags for compilers and linkers. This follows patterns in Nix's own handling and similar mechanisms for ignoring impure paths not from the Nix store.

The WLLVM project has created a new project GLLVM with the goals of providing WLLVM's behavior but without Python, instead using Go which enables statically linking into utilities that don't introduce such impurities. GLLVM is also significantly faster, and is largely straightforward to switch to. However, differences in what flags are detected, supported, and how they are treated, cause sufficient build breakage to relegate the integration of GLLVM to the future.

4.2.5 Build Phase

The build phase begins, proceeding as specified by the base build recipe, but using the modified `wllvmStdenv` instead of the generic default. Having embedded the paths of corresponding bitcode into each generated `.o`, the build phase eventually, in almost all cases, invokes the system linker, often through a linker driver (generally `$CC`); the linked result contains a concatenation of these paths by virtue of the details and flags set when originally added to the `.o`. The bitcode paths are, like the `.o` files they correspond to, generally intended for only transient storage with only linked combinations being part of the final "package".

4.2.6 Install Phase

After the `buildPhase` completes, the `installPhase` commences which invokes `make install` or the equivalent for this package. As a universal convention, this install places all of the important artifacts (binaries, supporting libraries, man-pages, development headers), with the implicit notion that anything not so "installed" is not part of the packaged software and can be removed as convenient. While the mechanism of how to do so varies a little, the directory installed into is configurable and the contents of this directory will ultimately be the nix path representing the package. Documentation and other components in some but not all scenarios are stored in sibling nix paths through the Nixpkgs notion of "multiple outputs".

4.2.7 Pre-Fixup Phases

As of this point, the installation steps taken by the package will have created a package with machine code containing paths of corresponding bitcode – paths that reside alongside their object file brethren, and will be distinctly uninstalled and pending deletion once the build environment is cleaned (sandbox deleted). Since the bitcode is the actual output of importance to ALLVM, a setup-hook is used to add a `WLLVM_extract` phase to the list of "preFixupPhases" (see Nixpkgs manual [117] for overview of phases and their usage).

4.2.8 WLLVM-Extract

The high-level notion of the `wllvm-extract` utility is to gather all the referenced bitcode and move it somewhere it will be preserved for future usage. This tool reads the mentioned bitcode paths from an executable, gathers them, and outputs the collection in the specified way. Linking the bitcode is done relatively naively, with no attempt to emulate any linker flags or options. Only flags specifying unusual linker behavior would actually be relevant here, not the bulk of common flags which serve to indicate what libraries to link and where they might be found. In a sense this is unsound, but upon consideration it may be truer to the original intent, should it be the case that the flags were not carefully chosen incantations of linker details – the behaviors of which are paramount – but rather added for some other reason (perhaps a too-clever or conservative build system, or to work around a bug no longer present). Study may be warranted; in addition to helping better glean the intent of programmers, interesting results would suggest significant implications for projects defining linker semantics.

Regardless of intent, the `wllvmStdenv` happily ignores these flags when constructing bitcode equivalents for linked objects (shared libraries and executables). It is with quite some surprise and to the author's continuing astonishment that such disregard for these flags is not catastrophic. Experiments specifically intended to demonstrate that such seeming carelessness would promptly produce errors or extreme breakage instead quite contrarily revealed that this practice creates only a few rare issues, even as the number of supported packages continues to grow past multiple thousand. This property is as convenient as it is worthy of further investigation.

The `wllvm-extract` utility, in addition to combining all the reference bitcode files into a single bitcode file also does other processing tasks. It strips debug information, a precaution taken for disk space reasons but primarily to avoid a bug present in the LLVM version currently used in ALLVM, leading to extraordinary resource usage when linking multiple files with debug info. It also executes a practical but novel transform, DeInlineAsm.

4.2.9 DeInlineAsm

The DeInlineAsm LLVM pass searches for all instances of inline assembly, in particular attempting to identify instances where inline assembly was used to express a concept that had an LLVM IR equivalent. Generally, such uses of assembly originate from times when the corresponding language features or compiler intrinsics were unavailable or not available in a portable manner.

Very commonly, inline assembly found its way into early allexes for C-based software wishing to have the sort of atomics that weren't available until C11. Another frequent use is using inline assembly to express a memory barrier of some kind, which is the first and motivating example for this pass.

This transformation is valuable, as many analysis passes "give up" when encountering any form of inline assembly. By replacing the assembly with equivalent IR instructions, the resulting IR (and the resulting allexe containing it) became more amenable to analysis and more aligned with ALLVM's fundamental design principle of knowable software. This pass is also used by the `bc2allvm` tool, whose output is also an allexe.

4.2.10 Pruning

Finally, during the pre-fixup phases, all detected code lacking bitcode information is removed, as well as all static archives. This helps ensure only code with bitcode equivalent is ever used. In the future, it would be preferable for ALLVM to check that the bitcode obtained both a) contains equivalent or superset of what reaches the machine code and particularly b) replaces the machine code entirely with a version obtained leveraging (a). This would help catch missed assembly bits or embedded data earlier in the process, as well as give higher confidence in their correspondence. It is not, however, a serious limitation of the current version of ALLVM.

4.2.11 Finalizing `wllvmPkgs`

As introduced above, the `wllvmPkgs` set is built iteratively, each iteration proceeding through the phases above. These iterations are named `wllvmS1Pkgs`, `wllvmS2Pkgs`, and so on, until the properties detailed earlier are true for all attributes in the package set. Once this final iteration of `wllvmPkgs` is complete, the task of actually generating allexes is possible.

4.2.12 Converting Bitcode to Allexes

The traditional way that executables are loaded requires, at every runtime, the discovery of the list of directly-needed libraries, and then recursively their needed libraries, to ensure all code reachable is included. This is handled by glibc's standard dynamic linker `ld-linux.so`.

To create the allexe for a program naturally requires this same step, except that the search is done a single time at this point in the allexe creation process. Furthermore, this cements the list to a "known" set which is critical to carrying out ALLVM's primary design principle. In this way the creation, itself, of an allexe eliminates uncertainty.

Straightforwardly, at this point in the creation process, ALLVM uses a tool similar to ldd to obtain the full list of required libraries. The tool navigates the hierarchies of locally-defined search paths, environmental variable paths, inheritance from parents, and other involved complexities, ensuring correctness and completeness of the list.

This list is mapped into the list of bitcode equivalents of those required libraries. At this point in forging the allexe, we now have bitcode for specified program and for all code reachable from that representation. The program's bitcode will be treated as the 'main' module of the allexe. It must contain an entry point named "main".

Using the `bc2allvm` tool, this collection of bitcode is zipped together. The zip format allows for random access, unlike several other archival formats. Additionally, zip files can patched to be executable within environments such as the Linux shell, while remaining a valid zip. In particular, `bc2allvm` prepends the normal zip content with a "shebang". This shebang contains the path to the `alley` that will be used to run this allexe. Shebangs are widely used to specify the interpreter to use for executing the invoked file. Accordingly, the zip, now an allexe, is executable through natural and transparent means.

An alternative to the shebang approach is to make use the `binfmt_misc` capabilities of Linux, which make it possible to register allexe files with the kernel and specify how they should be executed using `alley`. This is similarly possible to provide easy execution of popular file formats such as Java `.jar` files or even to transparently execute programs for other architectures by automatically running them with, for example, `qemu-aarch64`. Doing this often requires root or high-level capabilities usually reserved for administration purposes, and for good reason. To avoid requiring anyone to place system-level trust in ALLVM's research tooling, and to ensure allexes could be easily executed by most anyone, allexes instead use shebang approach. Though slightly less elegant, it works nearly as well in practice.

4.2.13 Discussion

The allexe format has a downside in that collections of allexes often are storage-inefficient. Research is underway to developed improved formats, importantly overcoming this while retaining ALLVM's important properties regarding "known code".

A careful reader may be curious why machine code is kept at all after we have bitcode equivalents, since only the bitcode reaches the allexes ultimately produced. The reason for this is subtle. By leaving the "expected" code alone builds can proceed in default non-cross configurations, despite effectively performing a cross-compilation to the ALLVM platform. Notably, libraries are in the expected locations with expected contents – which are used to guide the eventual creation of allexes. While this technique isn't ideal, it allowed earlier prototyping without hurdle of a fully defined platform with binutils support and so on.

Cross-compilation is generally poorly supported by most software, and only recently has cross-compilation support in Nixpkgs matured to its current state, generally able to overcome many of these obstacles in stride. Were ALLVM's build process to be reimplemented today it would likely leverage these improvements and take quite a different form. For example, it is now possible to bootstrap a system from a compatible `alley` and allexes of a traditional bootstrap toolchain. This was impossible at the start of the project and so was not considered. It would also significantly reduce the size and evaluation time of ALLVM's current approach, which spans several stages before reaching the end result.

While allexes are built in a manner that successfully meets their design goals of "full and knowable code", allexes are not software packages. As such, sometimes proper execution of an allexe requires files or resources not alongside the allexe itself, where the program may expect them (relatively or absolute path). For this reason the "bin" output of the corresponding element in `wllvmPkgs` (the "base" for a given `allexePkgs` element) is propagated which is enough for vast majority of cases. This is mostly a limitation for attempts to run allexes "standalone" and is largely why the mass availability of such has not been published despite being poised to do so.

When used in a NixOS environment, such as cloud deployment via NixOps, locally as NixOS instance or in container generated declaratively, this is rarely (anecdotally never) a problem. Nixpkgs services, and whatever they're constructed into, strongly prefer explicitly specifying paths, configuration, and so on; an inability to find needed resources "automatically" appears unnecessary.

4.3 SPECIFIC CHALLENGES ENCOUNTERED

While ALLVM's approach to building allexes has proven successful for many packages, a number of packages faced hurdles. Some of these were relatively specific, while others were of a recurring pattern. Often, the problems ultimately stemmed from package build procedures containing esoteric operations dependent on aspects of the prior build steps that are not true with ALLVM's approach. Several of these challenges are examined here, including when extant their solutions.

4.3.1 Embedding Data

Programs are rarely only code. In practice, software commonly includes a variety of data embedded in the executables or libraries quite intentionally. A trivial, though universally recognizable example is in the storage of strings.

More complicated data, especially data generated during the build process, can be awkward to portably "bake" into the program using conventional tools. A popular solution is to leverage a tool such as `objcopy` to essentially directly inject data into a specified section or segment. Other more opaque approaches are of similar form: some tools generate C files containing large arrays with the data encoded as literals, typically hex. Excepting this last approach, generally these processes are not recognized by ALLVM's build environment. This has build-breaking end results of the data missing from the bitcode versions of these objects.

Many programs making use of such techniques, however, offer safer alternatives specifiable in their build recipe. When provided, these alternatives are generally compatible with ALLVM and make this common pattern easy to resolve with confidence. As a specific example, the ICU package exhibits this behavior. It uses the problematic method, by default, for build-time efficiency reasons, but offers a compatible alternative.

4.3.2 Dynamic Code Loading as Primary Purpose

Predictably, software that employs dynamic code loading as integral part of its design, or more problematically as its explicit purpose, does not work with ALLVM. Resolving this is tricky, as such usage is nearly directly opposite the core principles of ALLVM and allexes. In particular such behavior, as conventionally viewed, directly makes it impossible to have “knowable code”. One such program with the explicit purpose of dynamically loading code is `ld-linux.so` itself, which cannot be represented as an allexe. Note that the use of plugins is discussed separately, as extending the program is a distinct behavior (even if such extensions are arguably required for reasonable use).

4.3.3 Dynamic Injection

For a variety of reasons, some programs/libraries rely on the ability to inject themselves between dynamically linked components of software. The environment variable `LD_PRELOAD` is often used to request a particular library be loaded before others. A number of pieces of software leverage this variable to provide intercepting or “hooked” implementations of symbols, exporting them so as to ensure their variants are used and not the normal definitions.

One popular use of this approach is to replace the normal system allocation routines (e.g., `malloc`) with alternative versions for debugging problems or to provide behavior optimized for a particular workload or application. Other uses are to implement one variant of a “tracer”, by overriding functions of interest with variants that log their invocation and forward to the normal definition so as to provide the expected behavior.

This behavior can be emulated with an allexe constructed to reflect the desired symbol resolutions explicitly. This process is potentially automatable in future versions of ALLVM. Other more sophisticated techniques of tracing or debugging program execution may not be fully supported or function at all. However, with `alley`, a variety of powerful debugging and tracing techniques from the literature are newly available which help overcome this limitation and may offer capabilities not available with conventional execution strategies.

4.3.4 Ambiguous Resolution

It is unfortunately commonplace for programs and their libraries to have ambiguous resolutions, with multiple different definitions available and without any clear indication which should be preferred. This is generally easily handled by properly defining symbol visibility, although in particularly messy situations this is insufficient.

Virtually every instance of this encountered in the ALLVM project has been an oversight by the package developer. For this reason, and lacking ideas how to sanely model such, any program exhibiting these failings is simply considered malformed. Eventually, ALLVM tools should outright reject these with consistent and helpfully informative description of the detected problem, but this is not yet fully realized in all cases across all tools.

Though ALLVM classifies these as invalid programs, it should be noted that this type of issue does not reflect developer skill. Such ambiguities are easy to introduce and may remain unnoticed for quite some time, as many platforms tolerate some of these silently, or at most with a warning. When ambiguous resolution occurs, it may not even produce any discernible problem, having resolved favorably by happenstance.

In the interesting case of PulseAudio, an ambiguous resolution was clearly unintentional yet in practice entirely harmless. Specifically, all possible resolutions were distinct addresses, but each was the beginning of an identical constant data structure. These data structures were, in turn, exported by each of three libraries that had been provided separately to allow selective usage of various features. Similar circumstances arose in several other packages. In PulseAudio's case, this behavior was raised as a bug[1]. From that issue, ALLVM was able to use a suitable patch which directly prevented the duplicate data structures from being loaded, resolving the ambiguous resolution.

4.3.5 Assumptions and Introspection

Software occasionally makes assumptions, intentionally or otherwise, about the nature of the artifacts produced by the compiler. When these assumptions are not valid, errors almost certainly result. Problems of this type occur at all stages of the build process. These problems may even appear only when the problematic package is used as a dependency. More rarely, at runtime some programs attempt introspection and fail when they encounter unexpected results.

In many cases, these are defensive checks or parts of test-suites and can be safely omitted or ignored by ALLVM.

PulseAudio serves as a useful example again in this context. This software assumes the presence of a number of modules which it will be able to `dlopen`. Instead, utilizing PulseAudio's configure flags, the adjusted ALLVM build recipe can force preopening of modules which bypasses this problem. This a good example of the type of small, simple changes sometimes required to build software as an allexe.

4.3.6 Graphics

A critical library with unique needs is the graphics library, `libGL`. This library is notoriously vendor-specific, and of course the underlying hardware (graphics cards) used varies especially in capability and features. In theory, some amount of interface standardization exists, such as the various versions of OpenGL and its multitude of extensions.

In practice, however, programs never actually link against `libGL` themselves, dynamically or otherwise, nor even attempt to find or `dlopen` this library. Instead, a series of "loading libraries" act

[1] https://bugs.freedesktop.org/show_bug.cgi?id=55180

as a combination of compatibility layer, provider of convenience helpers, and general shim to keep `libGL` and its dependencies (often including LLVM and various C++ components) from interfering with the program wishing to make use of graphical functionality. These "loading libraries" are rarely simple, often involving code generation tricks to mask indirections and help bridge the gap between software and the hardware on the particular machine. The most recent of these "loading libraries" is NVIDIA's `libglvnd` ("Vendor-Neutral Dispatch"), which has a daunting architecture for ALLVM to model.

Owing to these myriad complexities and indirections, ALLVM does not currently attempt to support `libGL` as conventionally used. Efforts to replace indirections with direct calls to `libGL` when known may be successful, but remain as future work.

4.3.7 Plugins and related notions

The use of plugins is a common mechanism to allow software to be extended. Sometimes software provides a set of plugins which can be optionally selected. Other software uses plugins to allow extensions separately (e.g. in time, or by another party) from the construction of the software itself. Regardless of purpose, plugins are specifically separate components, even when packaged along with the software. Specifically problematic for ALLVM is that plugins are loaded dynamically. In this discussion, plugins which are particularly "shared libraries" or "loadable libraries" are examined.

4.3.7.1 Unused Dynamism

The dynamic loading aspect is actually frequently unimportant, merely a means selected for the task. A recurring pattern is for plugins to be loaded immediately during startup, and the list of plugins is knowable before execution and will never change without action by the administrator (or similar trivially-visible changes). In such cases the use of a plugin mechanism is of little utility in its actual deployment.

When the plugins are effectively statically knowable before execution, ALLVM can handle them relatively straightforwardly. Simply creating a copy of the base allexe, with the plugins included, will suffice. Minor "plumbing" work may be required to ensure that the included plugins are actually loaded. Often, however, no intervention is needed. Some plugin architectures use a base registry populated by static constructors executed when a plugin is loaded. For these types of plugins, such as those used by LLVM, simply including them in the allexe is sufficient; no further work is needed.

This static inclusion is particularly appropriate when plugins are themselves unchanging, without activity or additions from maintainers or community. Similarly, perhaps due to historical or portability concerns, many programs support a plugin ecosystem of, for example, exactly 1 plugin. Despite implementation as a “plugin”, its inclusion is typically fundamental for all use cases and not optional.

One specific example concerns I/O modules used by GTK’s glib. In GTK2, it is possible to use configuration flags `--disable-modules` and `--with-included-modules=yes` to statically include the input modules normally optionally loaded dynamically. This has the expected limitation of no longer supporting *ad hoc* additional input modules, such as East Asian Language support [118, 119].

4.3.7.2 Constrained Bundling

Plugins are sometimes developed separately from the software they extend, but for reasons not strictly technical. Examples include separation based on social organization (“only plugins from official developers go in the main tree”), licensing reasons (plugin and project can be used together but not easily shipped under a unified license), or simply because the plugins are so specialized to a particular niche that their general inclusion would be unwarranted. Occasionally, functionality is deferred to dynamic loading simply to avoid a dependency cycle that would otherwise be awkward to resolve. SVG support in `gdkpixbuf` via `librsvg` is one such example.

In all of these cases, often the solution is, again, to simply bundle the plugin with the project and use the appropriate techniques for the plugin style employed by the software to ensure it is correctly loaded. Though thus-far sufficient for individual use, systematic application of this technique would be problematic. Licensing incompatibilities in particular are completely unresolvable, and at-scale, an endeavor to bundle these pieces in single allexes swims against the currents that separated them in the first place.

4.3.7.3 Limited Dynamism

Static inclusion can also be appropriate in handling scenarios where the set of all possible plugins is known, and only the choice of which are used varies from execution to execution. Some plumbing work may be required, the amount larger for plugin systems built around `dlopen` and `dlsym` which would need special implementations. In these situations, it is possible to never actually dynamically load libraries, but instead pretend to do so and provide appropriate responses such as the requested address of a plugin’s primary function. This approach is taken by libtool [120, 121] in its “dlpreopening” technique [122].

Another frequent circumstance is one in which plugin choice is invariant throughout program execution, reliant perhaps on a configuration. Frequently, such a configuration is used for quite some time between changes to the selection of plugins. Popular container-based deployments often exhibit this scenario. Management of configuration is, for that reason, already a well-understood part of the deployment process; suitable allexes with statically-included plugins can be selected at deployment time accordingly.

Still other software is built around dynamic loading, generally to provide configuration to specify which modules to include. Deferring loading past the conventional point of build and install is important in these cases. However, the modules are often only dynamic insofar as a configuration must be parsed to discover the fixed set of modules that are otherwise essentially static w.r.t. the program execution. The technique of **program specialization** uses known configurations for purposes of optimizing size and performance as well as safety reasons, concerns similar to those addressed by ALLVM. This technique provides a means to build appropriately specialized allexes to address the varying use cases.

4.3.8 Code Running Other Code

The line between code and data is not always clear. Collectively, programs like interpreters, browsers, and virtual machines pose unique difficulty for ALLVM's fundamental design principle of of representing, in a readable and knowable form, the entirety of the program as it will actually be executed. When code becomes data, this principle breaks down: it becomes impossible to collect "all the code" into the allexe.

Handling these types of software elegantly within ALLVM is a significant challenge, and while in general it is not "solved" an outline of specific details and resolutions is presented below.

4.3.8.1 Interpreters

Interpreters are by definition tasked with execution of code provided as input, generally unknown until runtime. For many common interpreters (e.g. CPython) the interpreter itself can readily be made an allexe, but is not fully functional yet. In the case of CPython, modules can optionally be built-in to the core interpreter, but not all are by default. Most of the ecosystem's modules are elsewhere (viz. pypi), many of which require building or loading native code. These will not be functional, though modules that are pure python will. A python-aware extension to ALLVM, or conversely an ALLVM-aware extension to python are possible approaches to this sort of fundamental difficulty.

4.3.8.2 Browsers

Modern browsers are difficult to map into appropriate ALLVM concepts and artifacts, for multiple compounding reasons. First, by their core nature, browsers fetch resources and operate on them. Processing of data fetched remotely vs locally is not inherently a concern, but browsers in popular use such as Chrome or Firefox are expected to act as interpreters or compilers themselves for JavaScript and more recently WebAssembly.

All mainstream browsers today are of such phenomenal complexity that even attempting to bring one within the ALLVM architecture is presently untenable. Despite presenting as a single software component or package, browsers "vendor" the vast majority of their dependencies. This is the practice of including copies of these dependencies as part of their own source tree. This practice generally affords control of these dependencies including the specific version used and how they're built, and facilitates modifications that are either not yet included upstream or unsuitable for use elsewhere. This practice is especially natural given that much of the active development of these dependency libraries is driven by the same developers and organizations. Google, in particular, is known for being a strong proponent of the one-giant-monorepository approach, particularly evinced by its Chrome browser (and by extension other Chromium derivatives).

Beyond these complications, browsers in common usage are unabashedly platforms in their own right. A salient demonstration of this is found in GitHub's open-source Electron framework, the main GUI component of multiple open-source projects (e.g. Atom, Light Table, Visual Studio Code), which combines the Chromium rendering engine with the Node.js JavaScript platform.

ALLVM does support a number of browsers that are less fully-featured, including modern browser projects with features more compatible with ALLVM, such as Arora, Dillo, or surf, as well as older terminal-based browsers including links2 and lynx.

4.4 RUNNING AN ALLEXE

ALLVM executables, allexes, are executed by the `alley` tool. Serving a similar role as the traditional ELF interpreter (e.g. `ld-linux.so`), `alley` is automatically invoked through the allexe's embedded shebang. These interpreters take the executable as an argument and are responsible for setting up the initial execution environment by loading required resources, and then they pass execution to the application starting point.

The interpreter and execution engine, `alley`, has two alternative execution strategies that depend on whether or not the specified allexe has already been prepared by the `allready` tool. That tool transforms allexes into fully statically-linked binaries, which it caches for later use by `alley`. If the allexe does not have a corresponding ready-to-execute binary in the statics cache, `alley` follows a JIT strategy. Objects created and used during that JIT execution are similarly stored in a persistent cache.

4.4.1 Statics Cache & JIT Cache

Both the statics cache and the JIT cache use a simple lookup scheme, designed to not require full unpacking and loading for use. The lookup key is computed by combining the module name, often containing the originating nix store path, with the CRC32 conveniently present in the zip index. By default, both are created in accordance to XDG base directory specification [123], respecting the value of `XDG_CACHE_HOME` in most situations. When this is unset, the cache will next try `~/.cache/` as its base (`~/.cache/allvm`), with the final location simply `$PWD/.cache`. The cache location can be specified using the environment variable `ALLVM_CACHE_DIR`.

Queries against the caches do not return paths, but rather sealable file descriptors containing the requested contents. “File sealing” is a mechanism which recent kernel development has iterated toward to allow components to share data without providing access to the underlying file itself or even exposing a file namespace at all. These technologies were not yet mature during ALLVM’s development. As a result, ALLVM does not currently leverage them, though the JIT cache is designed to take advantage of these features as soon as they are available for general use.

4.4.2 Execution Sequence

Upon execution of an allexe, `alley` is invoked by the embedded shebang; in this manner, each allexe points to its preferred `alley` – the one from the tool suite version used to create or modify it.

When handed an allexe, `alley` first opens it without unpacking its contents or the modules contained within, handling these lazily. Regardless of which execution strategy is ultimately used, `alley` adjusts the argument vector to omit `alley`’s intermediary role, a detail usually taken care of by the kernel. This is a minor but important change for usability and surprising amounts of correctness. For example, `argv[0]` is often used in help messages and to identify invocation name. The latter is particularly important for correct handling of multicall programs, which are discussed in detail in Section 5.3.1.

To determine the execution strategy, the ALLVM statics cache is queried. As alluded to above, in the case of a cache hit, execution immediately proceeds using the returned file descriptor and the adjusted argument vector. Otherwise, `alley` takes a JIT strategy using LLVM’s MCJIT infrastructure, similar at a high level to `lli`. While some of the steps below may be common to LLVM-based JIT solutions, several are of course unique to ALLVM.

4.4.3 JIT Execution Strategy

To start, `alley` must unpack the modules from the allexe and parse them into their LLVM in-memory representations. These operations can be relatively expensive, one reason this step was delayed until this point. To minimize computational cost, `alley` endeavors to avoid unnecessary code generation.

The first module in the allexe contains the primary entry point and will therefore be executed in all but exceptional circumstances. Accordingly, `alley` always fully materializes that module, then immediately incorporates libnone. In the current ALLVM architecture, every module contained within an allexe is expected to be referenced in some capacity. Therefore, the remaining modules are all extracted and loaded so that the persistent JIT cache can be queried for their corresponding machine code. When no cached version exists, the module is fully materialized, built, and stored in the cache. In this manner, the JIT cache enables modules common to multiple allexes to be reused in a manner similar to the role of shared libraries in a traditional system.

Because `alley` and the hosted allexe exist in same process, `alley` needs to carefully carve out a distinct environment exclusively for the allexe. The paramount concern is symbol resolution. Critically, the hosted program must only refer to its own code and to libnone and never have symbols resolve to those in `alley` itself. Accomplishing this requires two steps. The first is simply setting the flag in LLVM's API that disables attempts to search for symbols, to avoid mistakenly referencing portions of `alley` and because allexes are by definition self-contained (needing only libnone). The second step is to patch LLVM itself (as of 4.0) to remove an attempt to `dlopen(nullptr, ...)` – which will fail when `alley` is itself an allexe, and is intended to ensure symbols from the current executable are found. Next, memory for the allexe's stack is allocated and initialized. As per convention, the stack is populated with the arguments, environmental variables, and aux vector.

Modules have zero or more static constructors or destructors, which must execute before and after main, respectively. Each module's list of constructors and destructors are executed sequentially, and it's important for `alley` to process these in the expected order in and across these lists. `alley` itself only addresses the constructors, instead leveraging libnone's (libc's) normal handling of destructors by synthesizing a composite list of all destructors. The addresses of the JIT-compiled functions are gathered for static constructors and destructors, for main, and for the `__libc_init`.

As the final step, `alley` transfers execution to the allexe by invoking the early initialization function, whose address was obtained earlier, with requisite parameters such that all of the above details are used by the execution of the allexe. In particular, the address of the entry point to be used, i.e. the address of main from above, is provided because the loaded libnone's libc will invoke it, not `alley` itself.

4.4.4 Commentary

The implementation of `alley` described above uses a JIT strategy that lacks the on-the-fly compilation often associated with JIT technologies. This is not an inherent limitation of `alley`'s design; it largely results from details of LLVM's JIT implementation when this project began. LLVM has vastly improved in this regard and we expect the next iteration of `alley` to have this feature.

4.5 ALLVM UTILITY SUITE

There are a variety of tools in the ALLVM tools suite, the most important of which are listed below, detailing the role they play in the growing ALLVM ecosystem. Each tool operates on allexes as input or output. Upon opening, basic integrity is checked of the allexe which primarily helps differentiate an empty allexe from an empty zip, but also simplifies logic elsewhere in the code. Output allexes are only written when a change has been made, in which case the shebang is updated to point to the `alley` from the set of tools being used.

4.5.1 all-info

This simple tool prints to stdout information about the contents of an allexe. In particular it presents prints module name and CRC32 hash.

4.5.2 alld

This default implementation of the linker abstraction class is used throughout the ALLVM tool suite. It is a wrapper around an embedded copy of LLVM's linker, `lld`, handling any difficulties of using lld as a library. ALLVM abstracts the specific choice of linker, enabling flexibility, such as using commonplace BFD or gold linkers provided by binutils and used by most systems. However, `alld` provides reliable linking behavior expected by ALLVM. Importantly, `alld` works independently of environment or host build tools. As an integral part of the ALLVM platform, `alld` must always be invokable. Accordingly, `alld` is entirely self-contained, without even dependencies on libnone or a dynamic linker.

4.5.3 allready

To make an allexe more immediately executable, the `allready` tool can process a single-module allexe and produce a fully statically-linked binary, which it places into the statics cache. The primary benefit is that, going forward, the allexe can be executed immediately without further work. Machine code generation facilities from LLVM are leveraged to transform the module in the allexe into a rough equivalent to an object file. This is combined with libnone and the appropriate pieces of the C runtime (e.g. `crt?.o`) using `alld`. Multiple-module allexes must first be processed by `alltogether` before using `allready`.

4.5.4 alltogether

A multi-module allexe can be combined into a single-module allexe containing the combined contents of the individual modules using the `alltogether` tool. More specifically, `alltogether` links the modules at the LLVM level, using the same mechanisms as `llvm-link`. During this process, `alltogether` optionally performs Link-Time Optimizations (LTO). In this way, `alltogether` acts analogously to the linker in a more traditional build process. Pruning trivially dead code is in fact unavoidable when representing the code as LLVM IR. However, at this stage large swathes of code may newly be seen to be entirely dead enabling more thorough pruning.

Because LLVM does not have a notion of a shared library at the IR level, ALLVM must handle some of these concerns. During the linking step, we must be careful to address name collisions related to scope and visibility. Programmers use visibility to control scope of their code and data. When we remove the module boundaries that scopes are defined by, problems like name collision naturally arise. We deal with similar ambiguities at other levels of the ALLVM design, such as in `ldd-recursive`. Here, while collapsing module boundaries, `alltogether` preserves the intent of hidden visibility by internalizing hidden symbols.

4.5.5 allopt

Using `allopt`, users can transform and analyze allexe programs. This utility serves a similar role to LLVM's `opt`, providing the usual large set of LLVM passes and analyses, bringing its power and robustness to ALLVM users easily. This tool handles allexes as input and output, streaming the bitcode contents to the specified command. If not performing only analysis, the output bitcode is repackaged into the output allexe.

Another key role for tools like LLVM's `opt` is support for loadable plugins, typically used to allow external developers to create custom LLVM passes and analyses. Naturally, providing such dynamic extensibility in ALLVM is not trivial, and as a result `allopt` requires a unique approach. Unlike LLVM's `opt`, `allopt` utilizes inter-process communication which, in addition to its natural simplicity, has the effects of increasing both isolation and, unfortunately, overhead. In this manner, `allopt` is a useful proof-of-concept demonstrating how to get plugin-like behavior in a non-dynamic environment.

CHAPTER 5: SOFTWARE MULTIPLEXING

Solving a problem simply means representing it so as to make the solution transparent

—Herbert Simon

5.1 PREFACE

This chapter is an adapted reproduction of the original publication[1] on software multiplexing [13]. Additional results are provided in Appendix A.

5.2 INTRODUCTION

On most modern desktop and server systems, the vast majority of applications are dynamically linked, because it reduces network, disk and memory consumption for libraries that are shared across applications. Dynamic linking, however, has significant disadvantages [83, 89, 98]. Application installation sometimes fails because necessary libraries are missing from the end-user's environment. Applications are slower to start because they must discover what code to use, and resolve memory layouts and indirection tables. Execution performance is also negatively impacted by introducing indirection on external references. Even compiler optimization is impacted by preventing cross-module optimizations that are possible when statically linking. Additionally, the ability to load executable code at run-time creates exploitable vulnerabilities; e.g., the recently discovered Samba exploit allows a malicious remote client (with access to a writable share) to cause the server to dynamically load and execute a shared library containing arbitrary code [124].

In this paper, we describe a compiler strategy we call "*Software Multiplexing*" that combines a predetermined set of applications into a single statically linked executable, while achieving the code size benefits of dynamically linked libraries and eliminating all the above disadvantages. Put another way, Software Multiplexing achieves many benefits of both statically and dynamically linked libraries, and adds some additional advantages. Briefly, the executables shipped this way are statically linked and fully self-contained (if all components are included when linking); are *much smaller than* the equivalent statically linked versions in aggregate, and *also smaller than* the equivalent dynamically linked versions in aggregate; start up immediately because no load-time overhead is incurred; execute without run-time indirection overheads because they are statically linked; are fully amenable to link-time optimization across all application/library boundaries; and avoid vulnerabilities due to dynamic loading of library components (as long as all libraries are included via static linking). Moreover, these programs enable optimizations *across multiple distinct*

[1]The author holds permission to reprint here material previously published in conference proceedings as: Dietz and Adve. "Software Multiplexing: Share Your Libraries and Statically Link Them Too." Proc. ACM Program. Lang. 2018, (DOI: 10.1145/3276524, pages: 154:1–154:26).

applications, e.g., when such applications may share code not captured by shared libraries (we briefly describe this new opportunity, but exploiting it and evaluating the benefits are subjects for future work).

A key part of the technical contribution is enabling Software Multiplexing to work without requiring a major rewrite of existing application build systems, which would be impractical. In particular, the build systems of most applications are designed for dynamic linking, while a few allow more flexibility for individual libraries. Rewriting such build systems to enable static linking if they don't already support it can be onerous and even impractical. Software Multiplexing works transparently without requiring modifications to the build system in most cases; we have built thousands of Linux packages using Software Multiplexing, with only a small number requiring minor build system changes.

5.2.1 Motivating Example

As a concrete example of the size vs. performance tradeoffs, consider Figure 5.1 and Table 5.1 which show size and performance of the set of executables and libraries comprising the LLVM compiler system when built using static vs. using shared libraries. The performance metric used is the total time to compile the full LLVM 4.0.1 system from source. Note that although this example is itself a compiler system, the size and performance impacts should be similar to those in other large systems (at least those written using C++).

Using shared libraries results in a much smaller footprint overall, but negatively impacts performance by 36%, compared with Static. LLVM developers prefer the statically linked variants, while OS distributions and users build using shared libraries.

To mitigate the overheads of LLVM's many libraries, they provide a special option that combines all the libraries into a single shared `libLLVM.so` which is the recommended way to build LLVM suited for dynamic linking. This is much faster than using separate shared libraries, but is also about 2.5× larger.

Our approach (`allmux`) combines all the executables and libraries of LLVM into one single statically linked executable, which is significantly smaller than all the other versions and significantly faster than both the Shared versions (and as fast as the Static one). In particular, the `allmux` version is 1.2× smaller and 30% faster than the Shared version, or 2.7× smaller and 13% faster than the `libLLVM` version preferred by distributions.

In other words, `allmux` is significantly better than either static or dynamic linking, without any significant disadvantages.

5.2.2 Existing Solutions

These observations are not new: the software community has attempted several approaches to balance these tradeoffs, although none of them are optimal, and the best ones require extensive

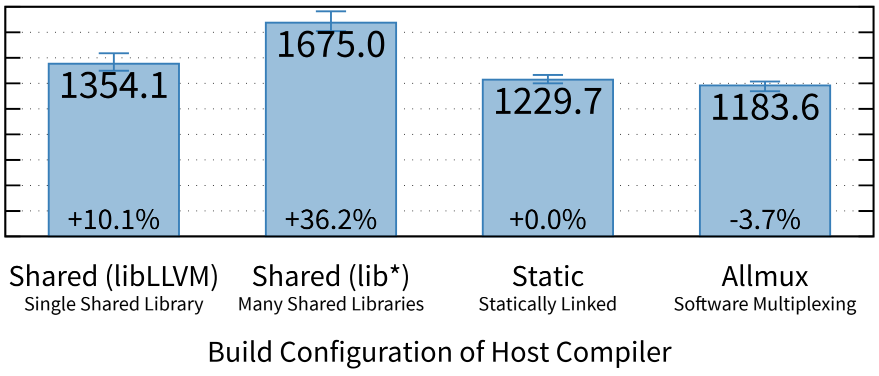


Figure 5.1: Seconds to compile LLVM 4.0.1.

Table 5.1: Sizes of LLVM Binaries in various build configurations, including Clang and lld.

Build Config	Bins	Libs	Total	Sharing
Static (Default)	590M	2.1M	593M	None
Shared (libLLVM)	231M	38M	268M	Coarse
Shared (lib*)	11M	93M	104M	Fine
Allmux	85M	0M	85M	Best

manual effort or high cost, or both. For example, Google is known to configure and build software almost entirely statically [125] essentially choosing to pay the cost of higher storage and memory to obtain better performance and reliability. LLVM, as described above, provides a unified library, `libLLVM`, but this is neither as fast as static linking nor as small as separate dynamic libraries.

The most explicit solution, *Slinky* [98], uses SHA-1 digests of code pages to identify identical pages across statically linked executables. It modifies the Linux kernel and uses a novel executable format to identify shared code, and to transmit, store and load it into memory only once. Slinky executables are larger than dynamically linked executables because of the additional hashes, they incur some load-time penalty, and the authors report a 20% storage space increase and a 34% network bandwidth increase. They also do not enable compiler optimization across application and library boundaries. Software Multiplexing is superior in all these ways, and avoids requiring OS kernel changes, but does require explicit (though simple) compiler support, and also requires identifying the sets of applications that should be multiplexed together. (Combining Slinky with Software Multiplexing would additionally enable redundant code pages across the multiplexed sets of applications, while preserving the extra benefits of Software Multiplexing within each set.)

5.2.3 Overview of Software Multiplexing

Software Multiplexing is intended to be used for software systems, packages, or sets of packages that are expected to be installed on a system, and which share one or more libraries of code. Some examples include the set of programs in a compiler (like LLVM or GCC); applications built using a common windowing framework like `libQtGui`; collections of applications with common themes, such as editors, shells, or window managers; separate versions of the same application or library, because different users on a system may use different versions; etc. The bottom half of Figure A.1 shows a large number of software packages containing multiple applications (ranging from 2 to 166 programs per package), yielding a geometric mean 39.2% reduction in aggregate binary size for these packages, *compared with dynamically linking*. Most importantly, while the benefits of multiplexing depend on the chosen set of applications to multiplex, *every case we have examined* – including a very large number of widely used software packages – shows benefits, and these are often substantial.

The Software Multiplexing compiler transformation has two parts:

(1) Automatic Multicall: This is a conceptually simple transform that has to deal with subtle but well-understood correctness issues. A *multicall* program combines multiple programs into one executable, and dispatches them based on the name used to invoke the program, or a predefined argument. Some packages, e.g., Busybox or coreutils, are designed to do this manually, but otherwise, introducing this feature retroactively is complicated and inflexible. Our work automates this step: a compiler pass takes N application programs as input and combines them into a single combined multicall program. Carrying out these steps after individual executables (e.g., ELF format binaries) have been generated can be quite complex; we instead export the compiler's internal representation (IR) (this is usually a feature in compilers that support Link-time Optimization) for all applications into individual files, merge the files into a single IR file, and use 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 pass produces a single merged IR as the output. Note that this is purely a compiler transform: no link editing occurs in this step.

(2) Statically shared libraries: The second part of the transform takes the multicall program and as many of the necessary libraries as possible – static and dynamic libraries – and links them into a single program. If all the applications' build systems are designed to use static linking, this step would be straightforward, but of course this is rarely true. Unfortunately, dynamically linked libraries have substantially different semantics from statically linked ones; a message on the binutils[2] mailing list asserts that simply linking in machine code for dynamic libraries using static linking was not just difficult but "isn't a sane idea" because the information needed to do so is "irretrievably gone" at this stage [126]. These problems are almost entirely avoided before code generation, and so we solve this problem with a compiler-based strategy: we *export every component*

[2]binutils is used by all known Linux distributions and contains the implementations for the linkers most commonly used

in the form of the compiler's IR, before code generation, including the multicall program and all necessary libraries (although some libraries could be omitted, if necessary), then link the IR versions of all components together, then generate native code for the fully linked multicall program.

Software Multiplexing achieves two kinds of code reduction. Like dynamic linking, it eliminates library duplication between (a predetermined set of) applications that are multiplexed into a single binary. Like static linking, it also eliminates unreferenced functions and global variables, which is *not achieved during dynamic linking*. This is why we are able to achieve binary sizes that are smaller than either statically linked or dynamically linked binaries (in aggregate) for any given set of applications.

Limitations: Multiplexing is not appropriate for all software, and by its nature (statically linking all your code together) is not suited for situations where what code is executed constantly changes. For example, one would not want to try to multiplex the dynamic loader itself as the entire purpose is to load an unknown program upon request. There are three specific limitations to multiplexing, at present. First, the benefits of multiplexing are limited to a predetermined set of applications combined together, unlike either shared libraries or Slinky, both of which share code across arbitrary applications on an end-user's system. As noted earlier, combining multiplexing with Slinky would get both kinds of benefits. As a direct consequence, sets of applications to multiplex must be predetermined, cannot be varied from one end-user to another, and adding new applications to an existing set is difficult (short of replacing the entire multiplexed binary for the set). Second, multiplexing makes it difficult or more cumbersome to update software by upgrading or patching dynamic libraries. Third, the current design of multiplexing disallows introspection techniques like the use of `dlopen` and `dlsym`. We discuss these further in Section 5.7.1.

5.2.4 Implementation and Results

We have implemented Software Multiplexing in the LLVM compiler infrastructure as a tool called `allmux`. This tool is part of the open-source ALLVM project available on GitHub[3]. Allmux allows arbitrary sets of applications, along with their library dependencies, to be merged into a single statically linked executable. The basic usage looks like: `allmux arora djview -o combined`. The output allexe (essentially, a zip archive of one or more LLVM bitcode files) can be executed using either AOT or JIT compilation using ALLVM tools.

Our results can be summarized as follows: For any particular set of one or more applications, `allmux` results in a single statically linked binary that has the following properties, compared with the same set of applications using either conventional shared libraries or statically linked individually:

[3]https://github.com/allvm/allvm-tools

$$\text{Disk} \leq min(static, shared) \tag{5.1}$$

$$\text{Memory} \leq min(static, shared) \tag{5.2}$$

$$\text{Run time} \leq min(static, shared) \tag{5.3}$$

$$\text{Startup latency} \approx static \leq shared \tag{5.4}$$

The results in Figure 5.1 and Table 5.1, above, illustrate all four of these conclusions for LLVM. As another example, for a set of 10 applications using Qt, the disk size of the multiplexed version is 17% smaller than shared and 66% smaller than static, in aggregate, and the memory usage (when all 10 run concurrently) is 40% less than shared and 63% less than static. A number of other examples are presented in Section 5.6 and in Appendix A.2.

Our research contributions are the following:

- We present a novel compiler strategy, "Software Multiplexing," that achieves many benefits of both statically linked and dynamically linked libraries.
- We show how to make Software Multiplexing automatic, even for programs that do not support static linking, by exporting and linking programs and all libraries in terms of the compiler IR.
- We implement Software Multiplexing in the LLVM Compiler infrastructure as the tool `allmux`. We use `allmux` to create self-contained fully static executables for a large variety of software and collections of software.
- We evaluate Software Multiplexing and find it creates programs that take less space and use less memory than both statically and dynamically linked versions, start and execute faster than dynamic versions (matching static versions), and are more secure and portable.
- We share our tools and infrastructure with the community as part of the open-source ALLVM project.

5.3 EXTENDING ALLVM

5.3.1 Multicall

A few program collections today, e.g., busybox and coreutils, are (optionally) organized as multicall programs (defined in Section 5.2). Busybox is organized as a collection of optional "applets" that are built into a single multicall binary. To execute one of the applets, one of two methods can be used: directly invoke the multicall binary giving the name of the applet as the first argument, or more commonly indirectly invoking the multicall binary using symlinks or hardlinks. While busybox supports building applets as separate binaries, this is not encouraged and, in fact, is accomplished with a script that iterates through selected applets and builds a one-applet busybox for each.

On the other hand, coreutils is organized in the more conventional manner: each utility provided has a unique `main` defined in a source file with matching name. Coreutils can optionally built into a single multicall binary, which is accomplished by extra build system support added by the developers that leverages application knowledge and uses the preprocessor to transform the program and insert declarations.

In both cases the source code organization and build system reflect the expected use case, using ad hoc techniques to build in either the multicall or separate configuration. For self-contained projects designed in this way from the start, the manual approach works well, but larger projects and their dependencies quickly become too complex to repurpose their build systems using these methods. Moreover, independently developed applications cannot be multiplexed in this way.

As a less similar example, compilers like GCC and Clang employ a limited form of multicall: each of these compilers (specifically, the driver program of each) is a *single* program that invokes different code paths based on the program name used (e.g., gcc vs. g++). Clang goes so far as to allow the invocation name to indicate the desired target triplet, essentially converting the invocation name into an argument. These driver programs go beyond ordinary multicall: they add additional semantics based on built-in knowledge of the intent of the selected program names and options.

In our work, we *automate* the process of constructing a multicall program from an *arbitrary set* of separate programs and their libraries, without requiring any changes to the individual programs or build systems, and without adding any new semantics to any of the individual programs.

5.3.2 Compiler Requirements and ALLVM

The Software Multiplexing approach presented in this paper depends on two compiler capabilities:

1. **Exporting IR:** The ability to export a self-contained IR for a source file, application or library.
2. **IR linking:** The ability to link multiple IR modules into a single one, either an application or a library.

These capabilities are available in many production compilers today, including GCC, LLVM [75], Intel's ICC [127], IBM's XL compiler family [128], and others, because these capabilities are also the key ingredients for link-time optimization (LTO), which is widely available in production compilers. We use the LLVM IR [75] as the basis for our work. Note that the final statically-linked binaries created by `allmux` (and used in our evaluation) are ordinary ELF executables suitable for execution on any reasonably-modern Linux system.

For this work, we also use a file format called an allexe, which is essentially just an ordinary zip archive of LLVM IR modules, e.g., a single shared library, or an application and some or all of its libraries, or (after multiplexing) multiple applications and their libraries.

The allexe file format and tools that operate on it have been developed as part of a broader project called ALLVM. ALLVM [129] is a strategy for system construction in which *all* (native) software components are represented in a virtual instruction set (in our case, LLVM IR) instead of native machine code. In particular, the `allmux` tool was developed as an exploration of how code sharing could be made possible in a way that was visible at the LLVM IR level and be naturally analyzed and optimized by compiler techniques across non-traditional boundaries, including across application/shared library boundaries, and across multiple programs. Both these capabilities are made possible by `allmux`.

In ALLVM, we have extended the LLVM tools (the IR linker, back-end static code generator, JIT compiler, etc.) to work with the allexe file format. We only use the linker and code generator in this work. The ALLVM linker, in particular, merges a multi-module allexe into an equivalent single-module allexe.

It is important to note that the use of the `.allexe` format and the ALLVM toolchain have negligible influence on the performance results presented here: the file format and the ALLVM tools and are essentially a repackaging of LLVM IR and LLVM tools for greater convenience and reproducibility, and flexibility, with no direct performance impact.

5.4 GENERATING MULTICALL PROGRAMS

Algorithm 5.1 Basic Allmux

```
 1: function MuxBasic($A$)                                ▷ Multiplex set of allexe programs $A$
 2:     $M$ ←GenDispatchMain($A$)                         ▷ Described in Section 5.4.1.1
 3:     for $a \in A$ do
 4:         $N$ ← Name($a$)                               ▷ unique invocation name for $a$ (e.g. bash)
 5:         $a'$ ← alltogether($a$)   ▷ Link components in $a$ into a single component; return as allexe $a'$
 6:         Rename entrypoint in $a'$ to main_<$N$>       ▷ (e.g. main_bash)
 7:         Generate functions ctors_<$N$>, dtors_<$N$>   ▷ make static constructors/destructors explicit
 8:     end for
 9:     return NewAllexe($M, a'_1, a'_2, \ldots$)
10: end function
```

Combining multiple programs into a single multicall executable is, at a high-level, a simple transformation:

1. generate new entry point that runs the selected program;
2. transform each input program to use a uniquely named entry point, and execute only its own constructors and destructors; and
3. merge programs into a single program, binding each program to correct dispatch entry in the new main.

A key addition is to use only one copy of each library in the final program, which requires a less obvious strategy. We first describe the simpler version, "Allmux Basic," which does not deduplicate libraries (Section 5.4.1) and then the full algorithm (Section 5.4.2).

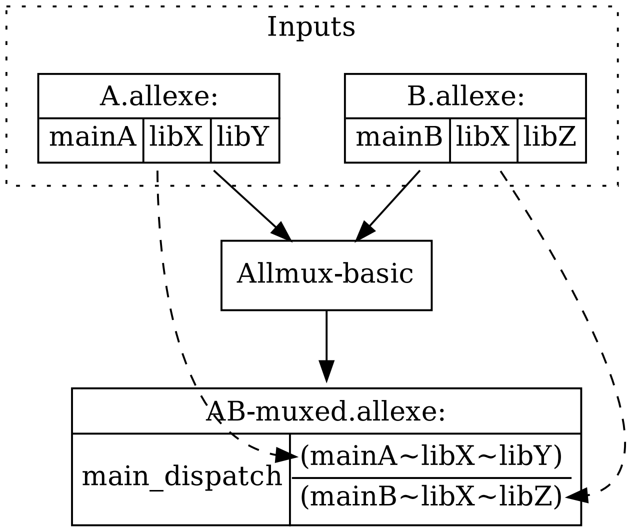


Figure 5.2: Allmux Basic.

5.4.1 Allmux Basic

The basic `allmux` transformation is presented in Algorithm 5.1, and a graphical overview is shown in Figure 5.2.

5.4.1.1 Generating Dispatch Main

The entry point is a generated function that determines which program is being invoked by comparing the filename portion of `argv[0]` with the names of supported programs, dispatching when a match is found. Once a match is found, the static constructors are executed by a call to the generated `ctors_`<*N*> and the static destructors are registered for execution by using `cxa_atexit`[4].

It is an error to attempt to use `allmux` on programs with the same basename, since the dispatch main would not be able to distinguish which program was invoked. This has not proven to be a problem in our experience and is easily avoided if an alternative dispatch mechanism was desired. Matching is implemented as a sequence of calls to `strncmp` but any appropriate lookup technique may be used.

5.4.1.2 Static Constructors and Destructors

Static constructor and destructor functions are required by some programs such as those using certain C++ features or by manually marking functions with special attributes in C. Static constructors must be executed before `main` and destructors should be executed on successful program termination or normal exit. Normally these are handled through the use of either `.init_array` or `.ctors` sections created by the linker and used by the libc implementation.

[4]in musl this is the same machinery as `atexit`

When multiplexing, care must be taken to only run the constructors and destructors of the program selected for execution. In an LLVM module, constructors and destructors are stored in special arrays of function pointers called `llvm.global_ctors` and `llvm.global_dtors`. The `allmux` Basic tool replaces these arrays with functions (two per input program) that invoke each listed function in the appropriate order and exports those functions as `ctors_`<*N*> and `dtors_`<*N*> making their execution explicit for use by the generated dispatch main once a program has been selected.

5.4.1.3 Merging and Binding

After the above steps, the transformed modules undergo a final modification before being linked together: All symbols are internalized other than `main`, `ctors_`<*N*>, and `dtors_`<*N*> which are explicitly exported (there are programs that give their `main` hidden visibility!). This ensures no conflicts or interference when linking, and is not a problem since symbol names are not significant at this point as ALLVM programs are not allowed use of `dlopen` or `dlsym` (this limitation is discussed further in Section 5.7.1). Unwinding still works properly using `eh_frame` information as usual on Linux, as does `dl_iterate_phdr`.

5.4.1.4 Compiling and Running the Multiplexed Program

The multiplexed allexe can be built into a fully static binary using a standard LLVM native code generator. The resulting binary can be deployed to any Linux machine. Symbolic or hard links should be created for each multiplexed input program, usually in a directory that contains other files the program may require such as configuration or data.

5.4.1.5 Discussion

The basic `allmux` algorithm automatically creates multicall programs that dispatch between effectively statically linked versions of each program. The transform is straightforward and the basic behavior of each input program is clearly modeled in the result, making it straightforward to reason about the preservation of program behavior. A few low-level details present in some programs require attention, such as use of `/proc/self/exe`, but few programs overly rely on such non-portable functionality and when they do they can be addressed as part of porting to the ALLVM program model. In our experience this has worked very well, and we demonstrate a number of examples of this success in the evaluation (Section 5.6).

The automatic multicall is complete, but there is a problem: duplicated libraries like `libX` produce multiple copies of code and data in the resulting allexe. Current optimizations in LLVM are unable to identify and eliminate this code duplication, because they cannot identify equivalent functions (we discuss this further in Section 5.7.1). Our evaluation in Section 5.6.8 shows the impact of this code duplication on resulting executable sizes.

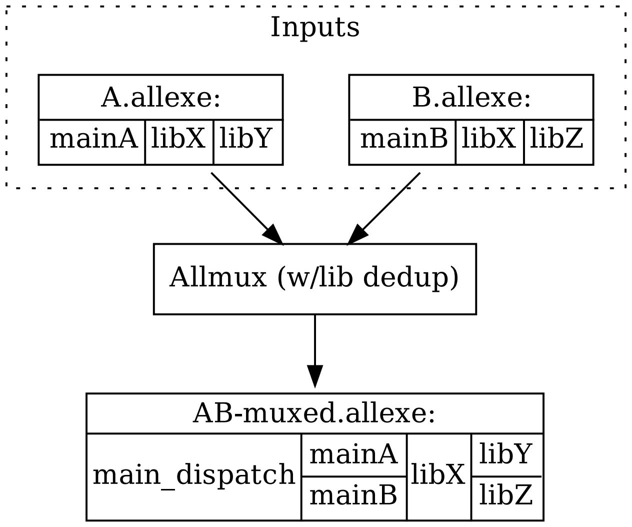


Figure 5.3: Allmux with Library Deduplication.
Only one copy of each library is emitted into the final allexe. Compare with Figure 5.2 where two copies of libX are included.

5.4.2 Multiplexing with Library Deduplication

Algorithm 5.2 Allmux w/Library Deduplication

```
 1: function MuxLibDedup($A$)                          ▷ Multiplex set of allexe programs $A$
 2:     $M \leftarrow$GenDispatchMainLibs($A$)          ▷ Described in Section 5.4.2.2
 3:     $L \leftarrow \emptyset$
 4:     for $a \in A$ do
 5:         $N \leftarrow$ Name($a$)                    ▷ unique invocation name for $a$ (e.g. bash)
 6:         $\{m', L_a\} \leftarrow a$                  ▷ module $m'$ contains entry point, $L_a$ contains set of libraries
 7:         Rename entrypoint in $m'$ to main_<$N$>     ▷ (e.g. main_bash)
 8:         Generate functions ctors_<$N$>, dtors_<$N$> into $m'$     ▷ make static constructors/destructors explicit
 9:         $L \leftarrow L \cup L_a$                   ▷ Track set of unique libraries used
10:     end for
11:     for $l \in L$ do
12:         $N_l \leftarrow$ genLabel($a$)              ▷ unique identifier for library $l$ (used by generated main)
13:         Generate functions ctors_<$N_l$>, dtors_<$N_l$> into $l$     ▷ make static constructors/destructors explicit
14:     end for
15:     $P \leftarrow$ NewAllexe($M, m'_1, m'_2, \ldots, m'_n, l_1, l_2, \ldots, l_k$)     ▷ $n$ entry points with $k$ unique libraries
16:     return Alltogether($P$)                     ▷ Statically link all components into a single bitcode module
17: end function
```

To address the code size increase limitation discussed above, we extend the basic allmux algorithm to treat the libraries of input programs specially and to avoid duplication of exact copies in the common case where a shared library is actually shared. The modified algorithm is presented as Algorithm 5.2 and an updated graphical overview is shown in Figure 5.3.

5.4.2.1 Key Modifications

There are two key modifications to the earlier algorithm. First, step 5 of the basic algorithm ran `alltogether` on each input allexe individually to link the application and all the libraries into a statically linked IR module, *before adding it to the output allexe*. The revised algorithm cannot do that because it needs to identify libraries shared between the input programs. We therefore skip the linking step on individual programs, and instead track the *set* of unique libraries used by the various programs. These libraries are emitted into a combined allexe, along with all the modules with the renamed entry points (step 15). We now run `alltogether` on the resulting allexe to generate a fully statically linked IR module, and return the resulting allexe, which contains a single module (like the one returned by the basic algorithm).

5.4.2.2 Generating Dispatch Main with Libraries

The second modification is in generating the dispatch main and handling constructors and destructors. Construction of `main` is modified slightly to handle the static constructors and destructors for the libraries (and only the libraries) included in the selected program. This is handled in much the same way as the constructors and destructors are handled already but in addition to invoking `ctors_`<*N*> and registering `dtors_`<*N*> the constructors and destructors of the libraries are also invoked and registered.

5.5 STATICALLY LINKING SHARED LIBRARIES

A key pragmatic obstacle to Software Multiplexing is that most programs we would like to build this way are not easily obtainable in statically linked forms. The generation of shared libraries and dynamically linking against them is the prevailing and explicitly preferred way to build Linux software [130]. For example, we were surprised to find that no Linux distribution offers fully static or even mostly static executables for non-trivial applications such as `git` or `vim`.

This is a significant problem for the application of software multiplexing to commodity software, and we note this is also a significant barrier faced by the compiler community to the use of compiler-based cross-module tools on non-trivial applications. To address this general problem, we propose a simple but—as we show, in part, with our evaluation in Section 5.6—surprisingly effective approach that enables us to statically link many (but not all, see Section 5.7.1) applications consisting of shared libraries.

5.5.1 Insight

The key insight is a perhaps deceptively simple one and is driven by our own experiences with thousands of Linux applications: despite the possibility of relying on subtle linking semantics or obscure linker features, in general the vast majority of programs only use the fundamental, common features of symbol resolution and relocation.

Embracing this, we set out to try linking of shared libraries at the IR level using simple symbol resolution rules (relocation is irrelevant with IR because all references can remain symbolic). Doing so is far easier at the IR level than after generating binary code, which makes even simple tasks such as "what parts of this are code" famously difficult to answer [16, 131].

This approach proved much more effective in practice than we expected, only requiring a few minor details be addressed before being sufficient to support thousands of software packages. The most important detail to handle is that of symbol visibility, which we discuss next.

5.5.2 Visibility

The use of symbol visibility is important in many applications, allowing shared libraries to internally use symbols without exporting them globally. Beyond good interface hygiene, this can prevent linking problems or runtime errors when multiple objects use the same functions and allows programmers to freely name functions and globals without concern that someone somewhere else might also want to name their function, for example, "`print_usage()`". Furthermore visibility affects whether the symbol can be preempted by a definition elsewhere or if uses can be assumed to resolve to the local definition [114], which can be important for behavioral and performance reasons.

The scope of a symbol's visibility is at the level of the shared object that defines it, which means it must be addressed when linking the code statically. This behavior is handled when linking the main executable of an allexe with any included shared libraries: we use a straightforward pass to identify hidden definitions and convert them to have internal linkage.

5.5.3 Other Details

Additional consideration may be given to support some interactions involving "vague linkage" (COMDAT or weak symbols), which is used by many C++ implementations to provide a number of features such as the one-definition rule (ODR). Similarly thread-local-storage (TLS) is an important feature for some applications. Neither of these have required taking significant measures to support or emulate, but this has only been tested indirectly, not exhaustively.

5.6 EVALUATION

5.6.1 Goals and Software Variants

We claimed in the Introduction that, for any particular "deployment" (1 or more programs), `allmux` results in a single statically linked binary that has specific advantages when compared to the equivalent software using conventional shared libraries or statically linked individually. We evaluate these claims here, through a variety of software and use cases suitable for desktop, server, and developer environments.

For these experiments, we compare up to five different versions of each set of software applications. When statically linking, we use link-time optimization (LTO) to provide a better baseline.

Shared–Musl: (aka, *Shared* or *Dynamic*)

Each application is built to link with normal shared libraries, using the LLVM toolchain: `clang`, `libc++`, `libc++abi`, `compiler-rt`, and musl libc (which is used by the ALLVM tools, and so gives an apples-to-apples comparison against the `allmux` versions).

Shared–Glibc:

Same as above, but with GNU libc (since this is more widely used than Musl libc).

Static+LTO: (aka, *Static*)

The same software configuration as *Shared–Musl*, but with all components compiled into LLVM IR, then linked statically, optimized using LTO, and then compiled into native code, yielding a fully statically linked standalone executable for the application.

Allmux-NoOpt: (aka, *NoOpt*)

The executable for a set of applications created by Algorithm 5.2, *Allmux with Library Deduplication*, and no subsequent optimization. Individual applications (represented by the input allexes) are linked (but not optimized using LTO), before running the `allmux` pass and native code generation.

Allmux-Opt: (aka, *Opt* or *Allmux*)

Same as *Allmux-NoOpt*, except that LTO is run on all the applications and (deduplicated) linked libraries collectively, after running the `allmux` pass and before native code generation.

We are unable to compare directly against the state-of-the-art alternative, *Slinky* [98] (which is available on their website [132]), because we couldn't use it on any program we tried to feed it. We have attempted but so far failed to debug the exact cause. Qualitative comparisons are discussed briefly in Section 5.6.9.

5.6.2 Workloads Used

Not all questions are reasonable for all software: runtime performance is not easily quantified in a useful way for applications such as a torrent client, and memory usage is most naturally measured for long-running and sometimes concurrently executing applications. Accordingly we've selected collections of programs and used them to answer the questions that best match the common usage. A summary of these applications and the questions answered by each is given below. The Claim numbers refer to Claims 5.1 to 5.4 in Section 5.2.

Binary Size (Claim 5.1 → Sections 5.6.4.2 and 5.6.8): All collections of software are suited for reporting their disk usage, although the appropriate comparisons vary depending on the way the software is commonly deployed.

Memory (Claim 5.2 → Section 5.6.4.1): To explore the memory usage of multiplexed applications we use a collection of graphical programs a "typical" user might execute concurrently (Section 5.6.4.1). For these experiments memory usage reported is *Proportional Set Size* (PSS), which accounts for memory shared by processes and is calculated by the kernel using the following definition [133]:

$$\mathrm{M}(p) = \text{Set of memory regions mapped into process } p$$

$$\mathrm{PSS}(p) = \sum_{m \in M(p)} \frac{\mathrm{size}(m)}{\#\text{ processes using } m}$$

Runtime Performance (Claim 5.3 → Section 5.6.5): We used compilation of LLVM 4.0.1 with Clang as a reasonable aggregate benchmark likely influenced by a combination of startup latency, memory usage, and effectiveness of cross-module optimization.

Startup Latency (Claim 5.4 → Section 5.6.6): A handful of applications were selected and startup latency was measured with and without background I/O loads, methodology adopted from Phoronix Test Suite's "Application Start-Up Time 1.0.0" modified to run our executables (Section 5.6.6)[5]. The benchmark was conducted for 10 iterations with consistent results.

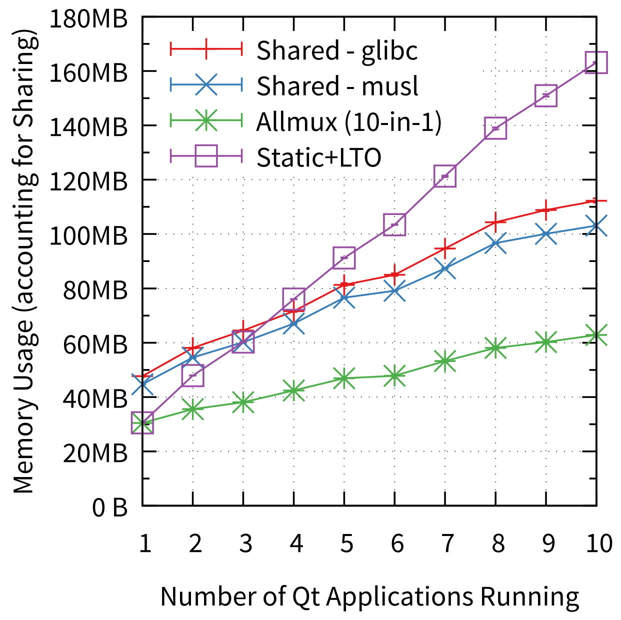
5.6.3 Experimental Hardware

Performance and startup latency experiments (Section 5.6.6 and Figure 5.1) were conducted on a Dell XPS 15 9560 laptop with an i7-7700HQ processor (6M cache), 16 GB DDR4, and a 512GB NVMe SSD. Turboboost was disabled to obtain consistent behavior across runs, and hyperthreading was enabled. The machine was running NixOS 17.09 (Linux).

5.6.4 Qt Applications

Graphical programs are classic examples of where shared libraries shine: many users will run a number of applications that all use the same toolkits and X11 support libraries. We built 10 Qt [134] applications in the various configurations described in Section 5.6.1; the applications chosen and a brief description of each is given in Table 5.2. These were used to measure the effectiveness of software multiplexing in terms of reduced footprint, including both memory and disk usage.

[5]Test suite 7.4.0m2 http://www.phoronix-test-suite.com/download.php?file=development/ and using the benchmark data http://phoronix-test-suite.com/benchmark-files/S-20170810.zip



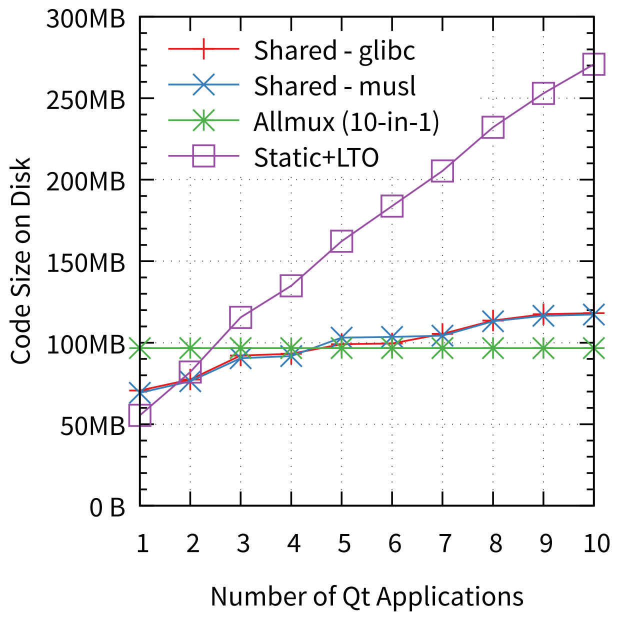


Figure 5.4: Memory Usage and Disk Usage for Increasing Number of Qt Applications.

Table 5.2: Description of Qt Applications.

App Name	Description
arora	Cross-platform browser using QtWebKit
djview	A portable DjVu viewer
qbittorrent	Free Software alternative to µtorrent
qgit	Graphical Front-end to Git
qpdfview	A tabbed document viewer
qscreenshot	Screenshot creation and editing
qtikz	Editor for the TikZ language
qvim	Qt GUI for Vim
snowman	Decompiler
wpa_gui	GUI for secure wireless networks

5.6.4.1 Memory Usage

To measure memory usage across processes sharing code (Claim 5.2), we measured the PSS [133] of programs after launching one application, after launching two applications, and so on, with each group running in a single virtual machine with no other programs other than the X server. The X server was not included in the PSS items used. (This may cause the PSS numbers shown for Shared libraries to be lower than they should be, i.e., biasing the results in their favor; this would happen if the X server and the applications shared any libraries.) Results are presented in Figure 5.4, which displays the results of 5 runs with error bars (due to low variance they are not visible).

As can be seen in Figure 5.4 the `allmux` variant consistently uses significantly less memory than the next best configuration – *Shared–Musl* – despite containing functionality for all 10 applications in a single binary. When using configurations that share memory across processes the growth is sublinear, but when launching applications that are individually statically linked, the memory usage is roughly linear (as expected).

Note that when running a single application, the static configuration uses less memory than the shared library configurations, which matches the `allmux` configuration.

`allmux` consistently uses no more than any other configuration, and often much less, e.g., about 40% less than the next best (*Shared–Musl*) when running all 10 applications, and just 1/4 of *Static+LTO*.

5.6.4.2 Disk Size

For each configuration we recorded the number of bytes on disk required to store the binary code (program and closure of library dependencies) for the first application, the first two applications, and so on – this was done to facilitate comparison with memory usage for concurrent execution of the processes as evaluated above. Note that the `allmux` series has a fixed size since the binary is fixed and includes all 10 applications. As a result, for a small number of applications, the `allmux` version is larger than the shared and static configurations. We consider this disk size increase a relatively small cost to pay for the fairly large gains in memory consumption.

5.6.5 Compiler Performance

We use Clang running time (when compiling LLVM 4.0.1) as a metric of software performance, since Clang is modern and widely used software, its performance is important to many application teams, and its use of libraries is carefully designed and flexible. The results of Clang compiling LLVM 4.0.1 were shown in Figure 5.1 in Section 5.2. The Clang/LLVM software is organized as a set of libraries that can be linked into a number of programs (tools), such as the Clang program itself. Alternatively, all libraries can be prelinked into one shared library (`libLLVM`), which is then linked into the separate programs. *Shared (lib*)* and *Shared (libLLVM)* show the performance of Clang linked in these two ways, both using dynamic linking. *Static* and *Allmux* correspond to the *Static-LTO* and *Allmux-Opt* versions defined above.

The results show that both shared library versions are much slower than the two static versions, with `libLLVM` yielding a large speedup because of the prelinking. More importantly, *Allmux* matches the statically linked version and strongly outperforming the two dynamically linked versions. Moreover, Table 5.1 in Section 5.2 showed that *Allmux* is far smaller than *Static* and also smaller than both *Shared* versions: the best of both worlds.

5.6.6 Startup Latency vs I/O Load

The performance results for compiling LLVM 4.0.1 shown in Figure 5.1 are caused, to a substantial extent, by lower startup latency of the `allmux` version of Clang. This is because Clang must be

Table 5.3: Startup Latency of Applications (seconds).

I/O Load	App	Static	Dynamic	Allmux-Opt
None	clang4	0.02	0.190	0.022
None	nocode	0.02	0.081	0.020
None	qbittorrent	0.24	0.369	0.243
None	termite	0.32	0.436	0.318
Read	clang4	0.33	1.242	0.401
Read	nocode	0.35	1.172	0.399
Read	qbittorrent	5.57	6.852	5.559
Read	termite	5.77	7.012	5.853

invoked once for each input C++ or C source file, paying much of the startup cost every time, and there are over 17,000 such files in LLVM 4.0.1.

To quantify this effect more precisely, we measured the startup costs for a set of programs, for the static, dynamic and `allmux` versions. Startup latency is also important for interactive computer use such as launching a terminal while system is under heavy load. We repeated this with no background I/O activity and with a background I/O load of 10 sequential readers from large files. Table 5.3 shows the measured results.

The table shows that dynamically linked program versions are often far slower than their statically linked counterparts, sometimes by an order of magnitude. The `allmux` versions are virtually identical to the statically linked versions when no I/O load is simulated and are slightly slower when heavy background I/O is performed (but much faster than the dynamic versions).

5.6.7 Software Collections

Because the benefits of `allmux` are highly dependent on the set of applications that is multiplexed into each binary, we evaluated different possible scenarios that motivate different such groupings.

5.6.7.1 Themed Collections

First, we evaluate the effectiveness of multiplexing over collections of software grouped by theme or purpose. As the multiplexed final output binary is a statically linked multicall program that contains the functionality of *all* the input programs, it would potentially be useful to have prepared to enable themed tasks in a self-contained and efficient manner. The contents of each collection are listed in Appendix B.1. The results are shown in Figure 5.5. Statically linked version sizes are not shown because they would be large enough that this approach does not make sense.

The results show that both unoptimized and optimized (LTO) `allmux` versions are always significantly smaller than the dynamically linked version, and the latter significantly so. The overall reductions range from roughly 10% up to (often) 30-50%.

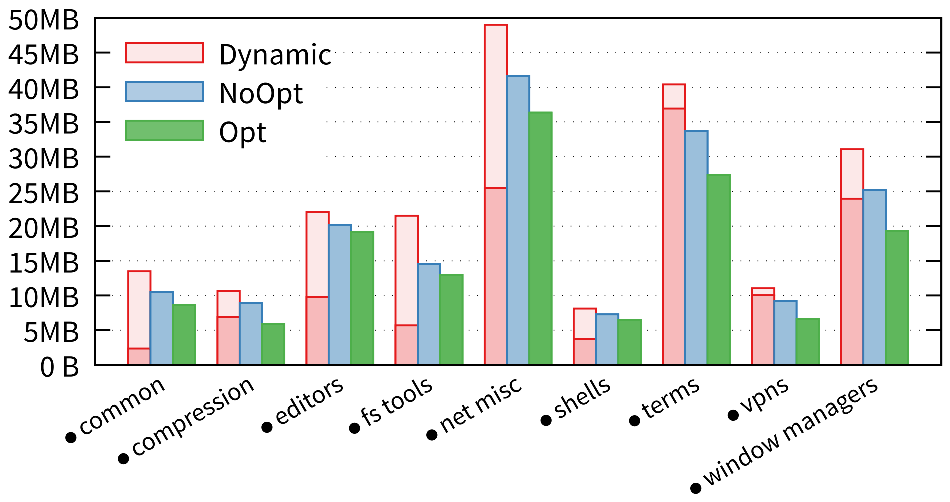


Figure 5.5: Sizes of “themed” collections of utilities (dark pink portions represent shared libraries).

5.6.7.2 Across Versions

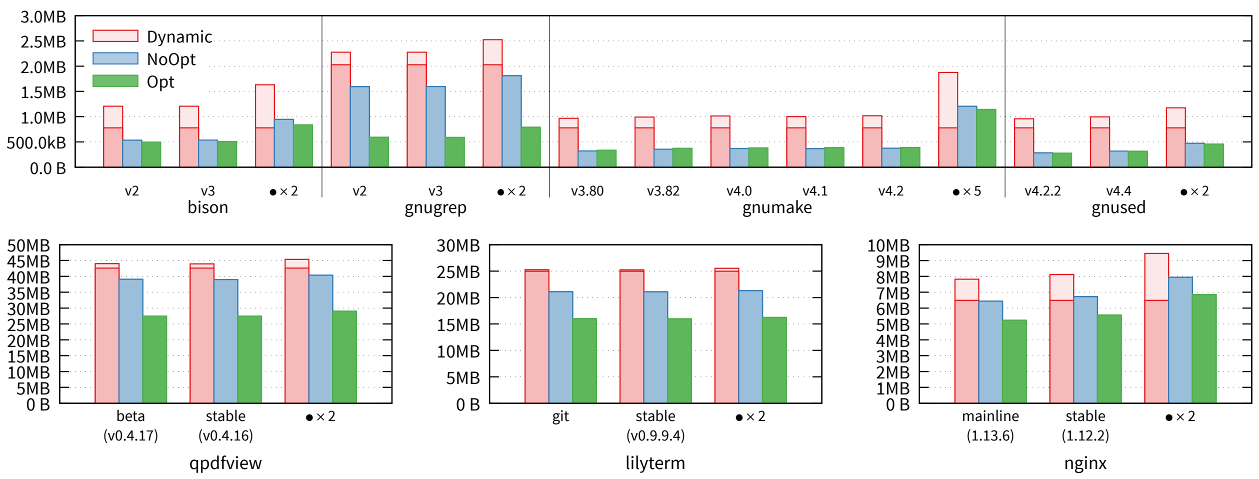


Figure 5.6: Multiplexing multiple versions of software together, binary sizes (dark pink represents shared libraries).

It is common for multiple versions of an application or library to exist on an end-user’s system. To determine how effective multiplexing is across versions of software, we selected a number of applications and measured their sizes across the three primary configurations: dynamic, noopt, and opt. The software was selected from programs which already had multiple versions present in meta-build repository we used (based on Nixpkgs), indicating the distinct versions were considered useful and not simply an upgrade or bugfix[6].

We consider examples of server, desktop, and command-line applications. We selected the following software in each category:

[6]The default policy is to only have a single version.

server: two versions of `nginx` (1.12.2, 1.13.6) representing "stable" and "mainline" versions.
desktop: stable and latest git variants of a pdf viewer, `qpdfview`, and graphical terminal emulator, `lilyterm`.
command-line: for compatibility reasons it is often useful to have multiple versions (or a specific version) of utilities such as the ones included here: `bison`, `gnugrep`, `gnumake`, and `gnused`.

The results for these applications are shown in Figure 5.6. For all of these applications, multiplexing significantly reduces binary size individually and even more so when applied across multiple versions. Notice that the sizes of the multiplexed programs with two or more versions are not much larger than those with a single version, whereas for the baseline (dynamically linked) programs, the shared libraries stay fixed (lower portions of the bars) but the application sizes (upper portions) are simply the sum of the individual versions. By exposing shared code to optimizations (opt), size is further reduced; in many cases, the result is even smaller than the size of the dynamic libraries alone (without the executables in question).

Future work on function-level deduplication techniques are likely to be especially effective for these experiments, as discussed briefly in Section 5.7.2.

5.6.8 All the Memcached Versions

An unusual and ambitious idea is to combine *all available versions of a given application or library into a single multiplexed executable* and ship that to all end-user systems! If there is substantial common code across versions, this may not be as impractical as it sounds, whereas today, it's something that simply would not be practical at all because of increased binary size. This is also useful for answering questions about the effectiveness of traditional compiler optimizations in taking advantage of highly redundant code.

We evaluated this idea on all 40 versions of `memcached` available at time of writing. We multiplexed together N of these versions in chronological order, up to and including $N = 40$. The sizes of the resulting programs, as well as comparisons to the dynamically linked equivalents, are shown in Figure 5.7.

The key point we see in the figure is that the size of the single, unified binary (Opt) with all 40 versions is only about 3× the size of the dynamically linked binary containing only one application version. In fact, the multiplexed binary can hold over 16 complete, fully statically linked versions of memcached in the same size as the single, dynamically linked version!

Importance of Library Deduplication To demonstrate the impact of not performing library deduplication we repeated the memcached multiplexing experiment above without using library deduplication. A comparison of the binary sizes produced is shown in Figure 5.8 with (on the left) and without (on the right) library deduplication enabled. As shown when not using library deduplication the multiplexed binaries can become larger than the dynamically linked equivalents.

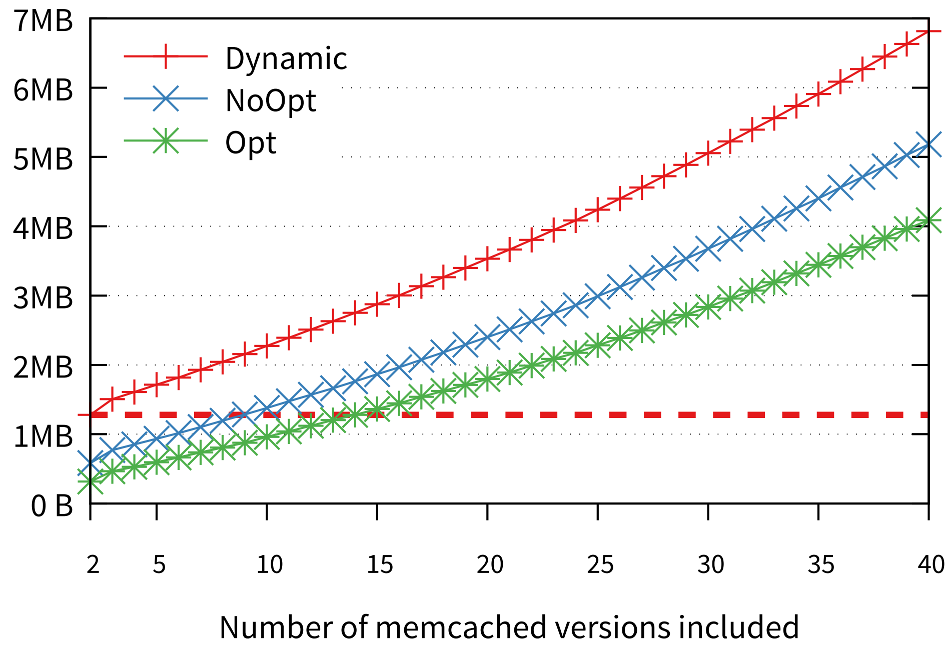
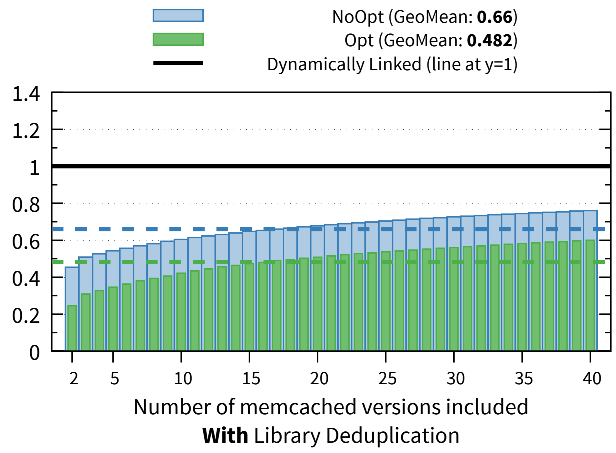


Figure 5.7: Up to 40 Versions of Memcached at Once, binary sizes.



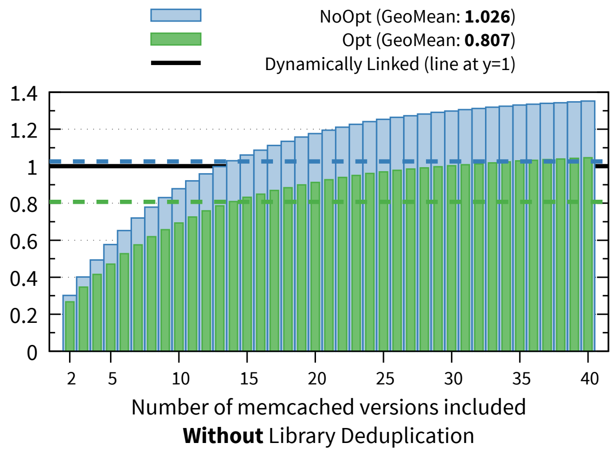


Figure 5.8: Multiplexing up to 40 versions of Memcached, Relative to *aggregate* size of dynamically linked versions. Dashed lines show geometric mean reductions for Opt and NoOpt cases.

5.6.9 Summary and Discussion

In all cases, multiplexed applications are smaller in size, use less memory, and start up faster than their dynamically linked equivalents, often with quite large improvements. Moreover, they are fully self-contained and require no external dependencies, or dynamic loading functionality. It is also worth noting that although the benefits of multiplexing depend on the chosen set of applications to multiplex, *every case we have examined* – including a very large number of widely used software packages – shows benefits, and these are often substantial.

It is instructive to compare these results qualitatively with those reported for Slinky. The major advantage of Slinky is that it can deduplicate arbitrary pages across arbitrary sets of applications, without predetermining groups to optimize, as with Allmux. In practice, we expect the benefits are orthogonal and the two could be combined for the best outcomes. In particular, Slinky results in larger code sizes than dynamically linked libraries (reported to be 20% higher), and they incur a non-trivial load-time penalty when programs start up. Also, Slinky cannot take advantage of code sharing at a finer granularity than individual pages (e.g., for redundant functions), or in cases where identical code exists but is not identically page-aligned, whereas `allmux` works – or can work – in both these cases. For example, the deduplication for multiple versions of software is less likely to work with Slinky because ensuring identical page alignments is more difficult for arbitrary code than for dynamic libraries (which are the main focus of Slinky). From a deployment perspective, our system does not require changes to the OS kernel, system linker or loader, but we do require changes to the compiler and Slinky doesn't.

5.7 DISCUSSION

5.7.1 Limitations

The Software Multiplexing approach has some limitations. First, as discussed in the Introduction, it requires ahead-of-time selection and processing of collections of programs, and is not well-suited for sharing code across dynamically changing collections of applications. A closely related weakness is that the sharing benefits of multiplexing are confined to the set of applications combined in each package: arbitrary applications cannot share libraries. Another related weakness is that the multiplexed sets are likely to be the same for all or most users, and would not be easy to customize for different systems with different user requirements. A major improvement that addresses many of these problems would be to allow software – including multiplexed applications – to be distributed in IR form (e.g., as allexes), so that new applications and libraries could be added to a multiplexed program *in the field*. This would also enable per-system customization of multiplexed packages.

Second, our approach disallows (or limits the benefits of) updating shared libraries. This can slow down the distribution of bug fixes or security patches through libraries, for example. Some large enterprises like Google compile and distribute a lot of their software statically linked, indicating that this issue may not be of importance to them. Interestingly, distributing software in IR form would mitigate this problem as well, because a new library version could be multiplexed in with other components in a package on the end-user's system.

Third, our approach disables explicit symbol lookup and other forms of process introspection such as the use of `dlsym`, `dlopen`, and others. These features are infrequently used and are rarely important; for example we discovered that seemingly benign programs such as `gnumake`, `bash`, and `gawk` all have functionality enabled by default that allows for loading arbitrary native code. In future work we plan to optionally support such introspection, as at least basic support for `dlsym` would not be overly difficult. We originally expected to find this a more serious limitation than it has proven in practice.

5.7.2 Opportunities

Finer-grain Code Deduplication: So far, `allmux` only eliminates duplicate copies of libraries shared by two or more applications that are multiplexed together. Significant additional code duplication could be eliminated by identifying other duplicated fragments of code, e.g., functions or smaller code regions. LLVM lacks a pass to identify identical functions or code fragments, but adding that is one of our goals for the near future. Several previous papers have presented sophisticated program analysis techniques to identify duplicate code fragments within programs at various granularities, ranging from functions down to a variety of small code regions [135–139]. They all focused on individual applications, and we hope to see bigger benefits when applying some subset of those techniques across multiplexed applications and their libraries.

Optimizations Across Novel Software Boundaries: Another opportunity is that a multiplexed program exposes much more code to optimizations, including applications together with their shared libraries, and even multiple related applications. This could enable new optimization opportunities, e.g., inlining code from shared libraries into application-level callers, or (when a set of locally communicating programs are multiplexed together [99]) optimizing across the communication boundaries between those programs by analyzing and transforming the programs *collectively*.

5.7.3 Security

A common misunderstanding is that software multiplexing exposes all programs in a multicall binary to the vulnerabilities of other programs. In actuality, the only code paths exercised for a program are those that already existed in the original, separate version; any code from some other program in the set will not be executed *in the same process* at all!

The only way security could be *harmed* is because it's likely that a significant amount of code not originally included in a program is now part of its executable code. This may allow more opportunities for code reuse attacks, such as composing gadgets for Return-oriented Programming (ROP). Except for small programs, this situation is unlikely to be much worse than the original. In the future we plan to address this by modifying the generated dispatch main to mark unrelated code as neither readable nor executable, similar to what is currently done by the runtime loader.

On the other hand, eliminating the ability to load code dynamically can significantly *improve* software security, by preventing attacks such as the recent Samba exploit [124]. It also makes it harder to create attacks through improper updates to dynamic libraries.

5.8 CONCLUSIONS

We have presented a compiler approach called Software Multiplexing that combines the code size and sharing benefits of dynamic linking and the benefits in code size, fast startup, more efficient execution, and better cross-module compiler optimization enabled by static linking. Our results show that our implementation, `allmux`, achieves smaller startup times than dynamically linked program versions with far smaller code sizes and memory usage than statically linked versions. Moreover, Software Multiplexing opens up new opportunities for novel future compiler research, including fine-grain code deduplication across application boundaries, and optimizations across non-traditional boundaries, such as application/shared-library and application/application.

CHAPTER 6: CONCLUSIONS

I would rather have questions that can't be answered than answers that can't be questioned
—Richard Feynman

6.1 FUTURE DIRECTIONS

6.1.1 Fine-Grain Deduplication: allmux++

As discussed in Chapter 5, software multiplexing operates on collections of programs and can dramatically reduce the disk and memory footprints. This is primarily accomplished through the deduplication of libraries used by the programs, with cross-collection optimizations further improving code size. A natural extension of `allmux` is to instead consider sets of functions rather than simply libraries. There is a large amount of code reuse not only between versions of the same program (Figure 6.1) but also between programs within a package (Figure 6.2). Two versions of a program, or copies of a program using different versions of a library, are likely to have a significant amount of code in common (many sources files may even be identical) but would not be deduplicated by `allmux` and remain as a source of bloat. Even when not built from the same source, it is common for functions or modules to be copied between codebases and may be a source of significant bloat unaddressed with the existing techniques. The resulting multiplexed programs would share a single copy of each identical function.

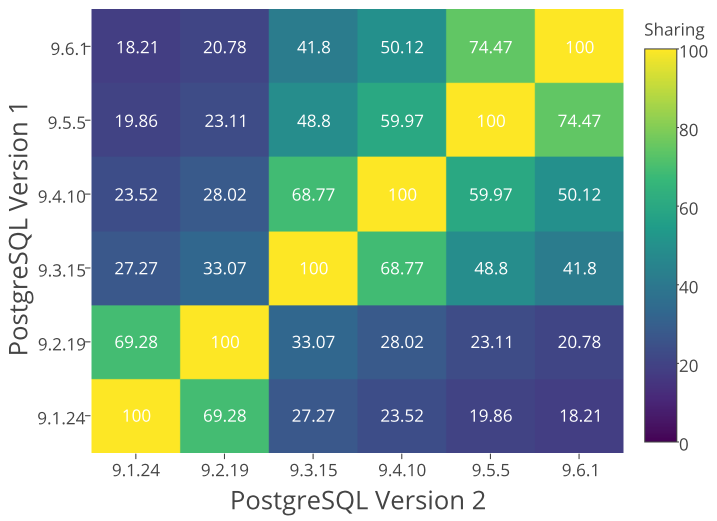


Figure 6.1: Heatmap Showing Sharing Levels Between PostgreSQL Versions.

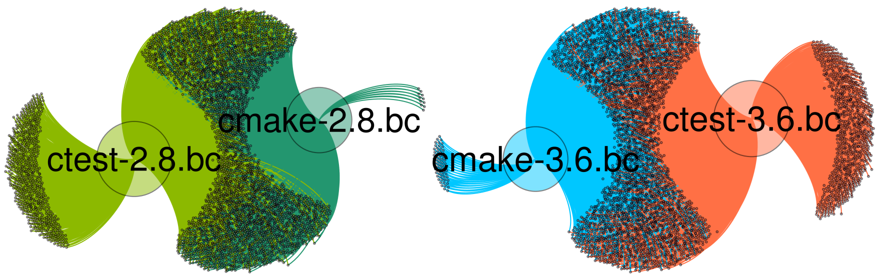


Figure 6.2: *Syntactic Hashes of Functions* contained in `ctest` and `cmake` from CMake 2.8 and 3.6 (not including libraries). Small nodes represent a hash value and edges indicate containing a function definition that hashes to that value. Colored regions are just large numbers of edges, too dense to be distinguishable. Notice the large amount of syntactically-similar code within executables of a version, identifiable as the large group of shared nodes in the center of each diagram.

Using fine-grain deduplication is expected to be able to significantly improve on code size reductions already observed, to extend benefits to include situations beyond library sharing such as across versions (released and during development) of software. Sean Bartell has developed a technique to break apart programs into functions for storage [140] which could help enable fine-grain deduplication.

6.1.2 Bitcode Database (BCDB)

ALLVM programs are stored in a database, the BCDB. Various prototypes for this have been constructed for purposes of efficient storage, hosting and delivering bitcode from trusted builders, and for conducting automated analysis over the thousands of programs contained in our primary BCDB. It also contains every version built since the project began, which are identified by a cryptographic hash of the unique expression used to create it. This database is currently publicly available for general research purposes, containing more than 5 terabytes of data. Future work includes moving to addressing the code by content and lookup by semantic equivalence.

When used in conjunction with graph databases, this data can be queried using a graph query language (e.g., Cypher). For indexing the data, I used the syntactic hash function that LLVM provides. This function computes a hash value for each component, using LLVM instruction opcodes in program order, but ignoring the operands. Ultimately, bitcode components are nodes in the graph and while edges represent symbol definitions and references (Figure 6.3). Queries on this database allow us to extract a wide variety of information, including information about approximate code equivalence between components. For example, Figures 6.1 and 6.2 show graphical representations of code similarity between functions, indicating the potential gains for `allmux++`. Other applications for this database include: (1) idiom recognition; (2) equivalence and translation validation; (3) program comprehension; (4) security vulnerability identification and propagation in supplemental programs [141]; and (5) inline assembly in programs commonalities [142].

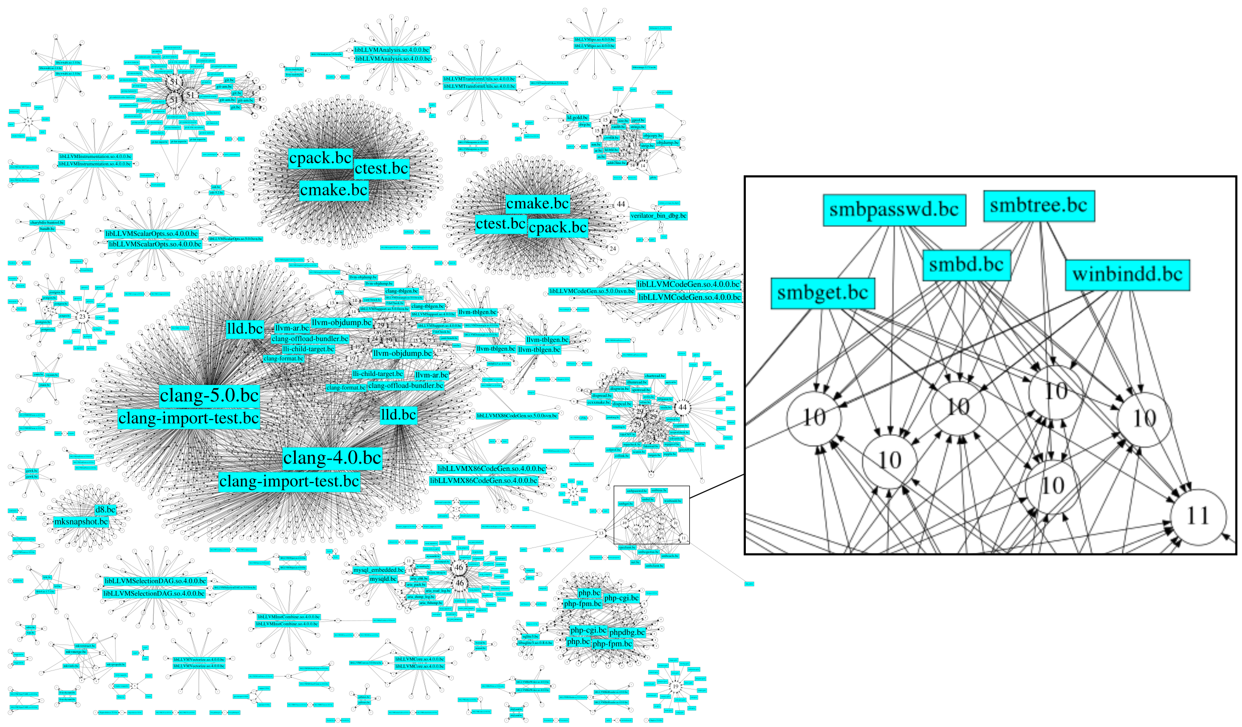


Figure 6.3: The rectangle nodes are bitcode modules from our database, the circle nodes represent hash values of functions, and edges indicate that a module contains the definition for a function that hashes to that node's value. The numbers within the hash nodes are counts of the number of functions that hash to that value. Note that this number is always at least the number of incoming edges but sometimes may be larger in the case that a module contains multiple functions that hash to the value represented by a node.
Left: Selection of programs from BCDB. Only functions with ≥ 2000 instructions shown.
Right: The callout shows programs from the Samba service and the nodes shared between them.

6.1.3 ALLVM as a platform for research

ALLVM is a simplified yet realistic system suitable for research. This is partially because it avoids the "long-tail" of features common in current research platforms. Additionally, it allows for reproducible builds [8] enhancing ALLVM's use as a platform for the analysis of software. For example, it is possible to run pointer analysis using ALLVM on an unprecedented scale. Typically these experiments are conducted over tens of programs. In contrast, using ALLVM increases this scope by two orders of magnitude, making analysis across thousands of packages trivial.

ALLVM would be a great foundation for an Integrated Development Environment (IDE) like LightTable or XCode's Playgrounds but at the whole-system level. I lay the groundwork to create a coherent set of tools integrating across programs. The use of bitcode for allexes preserves code readability. Furthermore, the use of allexes rather than binaries allows for: (1) optimize for your workload or specific machine; (2) verifying compilers have not been compromised; (3) compiler-based instrumentation; (4) auto-tuning; (5) specialize code for your needs (e.g. build

Table 6.1: Commands supported by `allplay`.

Subcommand	Description
asmscan	Search modules for module-level and inline asm
combine	Combine all modules into single bitcode file
cypher	Create cypher queries for allexe data
decompose	Decompose a bitcode file into linkable fragments
finddirectuses	Search modules for uses of function, direct calls only
finduses	Search modules for uses of function
functionhashes	Analyze and graph basic hashes of functions
graph	Produce bcdb graph
neocsv	Create CSV files for importing into neo4j
printsource	Print Source information from module flags
toml	Print allexe contents as toml
uncombine	Uncombine all modules from combined bitcode file

without optimizations, change configuration options, or removing functionality you don't need); and (6) conduct analyses like pointer analysis or program slicing.

As a research platform, ALLVM has several distinct benefits. It is universal, in that all code is represented as bitcode. This makes code easily searchable. ALLVM programs (allexes) are executed using just-in-time (JIT) compilation by default. JIT adaptive compilation allows for easy patching, transformation, and modification of code. Alternatively, optional ahead-of-time (AOT) translation can be used to transparently enhance performance. I provide a simple interface for complex analyses using `allplay`[1] (Table 6.1). It allows arbitrary code searching, decomposing and combining of allexes. Additionally, users can visualize and query across an entire collection of bitcode (BCDB, Section 6.1.2). In this way, `allplay` allows transformation and analysis on collections of allexes. Together, these capabilities open larger-scale cross-package experiments of particular interest to researchers in numerous fields especially compilers, software security, and software engineering.

6.2 DISCUSSION

Within this dissertation I have presented a system and a set of solutions for improving the nature of software by focusing on programmer's desired outcome, i.e. their intent. Slipstream improves the programmer's desired behavior of interprocess communication by eliminating unnecessary slowdown from TCP layers. ALLVM expanded my scope to the representation and distribution of programs. Using IR, allexes are able to preserve programmer intent. This is achieved by representing programs in a simple way, removing accidental complexity. ALLVM capitalizes on this, allowing for easy user interaction without programmer involvement. Software Multiplexing (`allmux`) removes accidental complexity born from separately packaged software including duplicated libraries and dynamic linking. The Slipstream, ALLVM, and software multiplexing methods

[1]https://github.com/allvm/allplay

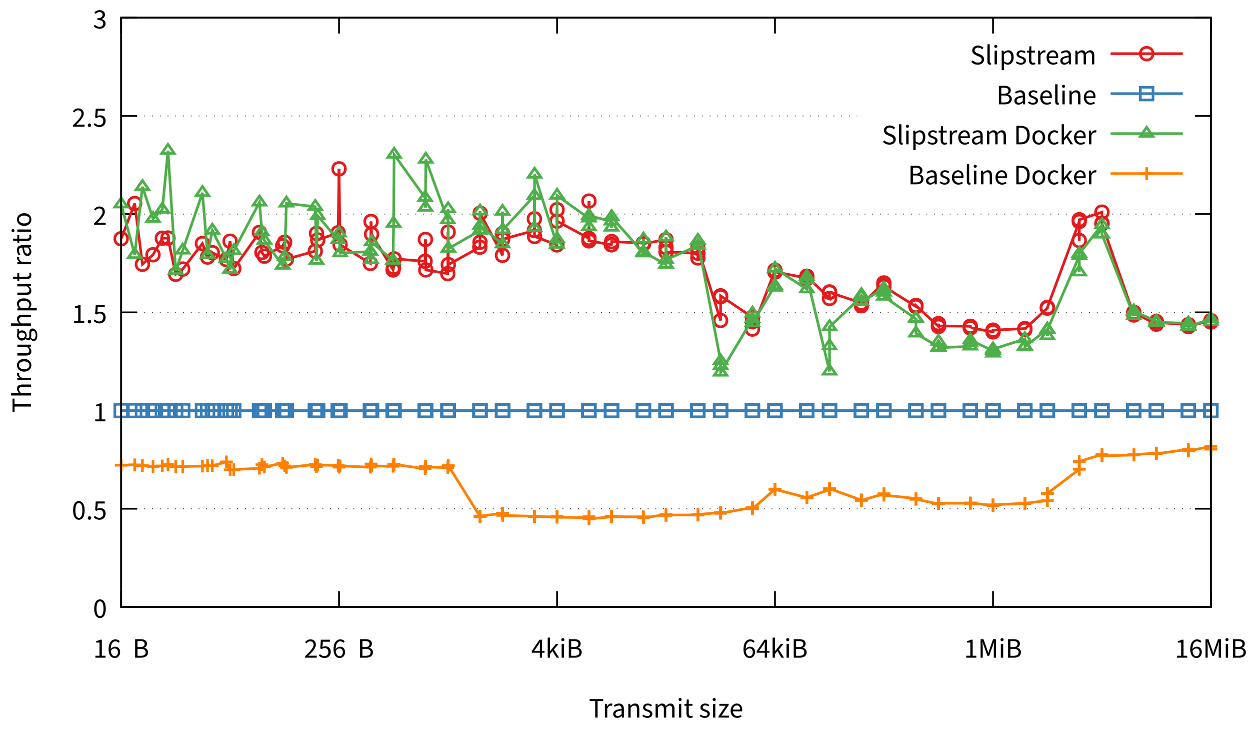


Figure 6.4: Slipstream Performance Improvements with and without Docker for the NetPIPE-C benchmark. Slipstream has up to 2× throughput compared to baseline. Note there is no slowdown when combined with Docker.

automatically reduce accidental complexity of programs and software systems while retaining intended functionality, without altering source code but operating on a whole-systems level, resulting in improved performance, reduced size, and lower memory usage.

Slipstream helps realize the desire for fast and efficient communication, regardless of how connections are made. In particular Slipstream allows programmers to continue to specify TCP, embracing location transparency, but allows for switching to faster communication transports when both endpoints are co-located. This frequently happens in cloud or container deployments which offer as their strength the freedom of not caring what physical machines are used.

Slipstream shows large speedups (up to 2× bandwidth) compared to baseline system performance. When used in conjunction with Docker, performance is unaffected – avoiding the overheads normally introduced by Docker's networking (see Figure 6.4). Slipstream is easy to deploy and highly compatible. It requires no modifications to the operating system or involved applications and is language-agnostic. Additionally, Slipstream is backwards-compatible and supports partial deployment.

ALLVM enables delivery and analysis of a large collection of software through a novel and simple executable format (allexes). Because this format is self-contained and does not use dynamic linking behavior, it is faster than traditional deployment solutions such as ELF binaries. For example, startup performance is consistently improved across tested applications (Table 5.3) as summarized

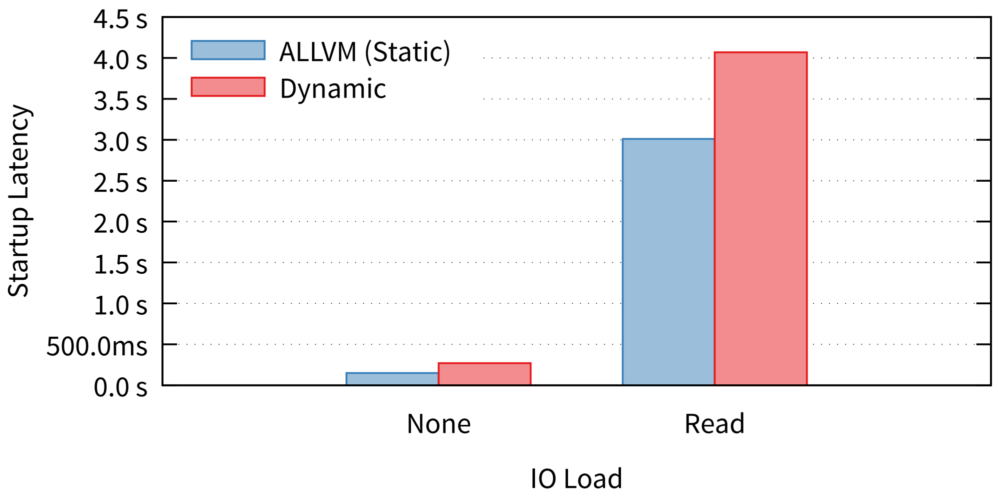


Figure 6.5: ALLVM's static linking resulted in faster loading times both with and without I/O load. ALLVM load times were consistently faster for all 4 programs tested - a compiler (clang), a terminal (termite), a decompiler (snowman/nocode), and a torrent client (qbittorrent).

in Figure 6.5. The resulting machine code is also smaller because it contains no unused code. Finally, static linking also ensures easy distribution and ensures that users "run the same thing" as was tested by the developers.

Constructing allexes systematically and reliably from thousands of programs, ALLVM is the foundation for techniques to leverage the simpler format. ALLVM also provides a suite of tools, "allvm-tools" (Table 1.1 and Section 4.5), which facilitate interaction, inspection, and manipulation of software represented as allexes. ALLVM is neither a compiler, linker, nor package manager, but takes responsibilities that are currently delegated to each of those players. This whole-system approach reduces complexity while retaining intended functionality resulting in faster and smaller programs.

Software Multiplexing (`allmux`) operates on collections of programs and produces results that are considerably smaller on disk and in memory usage, faster to start and execute, and more portable than the nearest conventional approaches. By including all the dynamic libraries before code generation, duplicate code between collections of programs and libraries is automatically exposed. Thus, it is possible to produce a self-contained collection with the "minimal"[2] code. For normal software collections, size grows linearly with the number of applications (see Figure 5.4), i.e. it grows in proportion to the amount of code included, with each application including all of its code. It is important to note that the reduction in complexity is not due to compression but actually eliminating redundancy (Table A.1). In addition, because we shifted the focus away from libraries, runtime footprint is reduced to contain only code used by installed applications. Even for a small collection of 10 programs, software multiplexing results in approximately 20% less disk size and more than 40% less RAM usage.

[2]See Section 6.1.1

Given the immense and overlooked (unappreciated) complexity in executing even a simple "Hello World" [4], it is unsurprising that gross complications emerge given the "building block" way in which "real-world" programs are written and combined into computer systems. It is possible to model software beyond the function or translation unit. In fact not only is it possible, it's natural and the whole-systems approach is actually much closer to the ideas that developers have when designing software. Here, I demonstrate that it is possible and practical for the vast majority of software to use this model to remove accidental complexity. While this approach has trade-offs, by design within this platform it is hard if not impossible to deliberately or even unintentionally introduce complexity. This makes packages easier to understand. In conclusion, I have shown that part of the solution is to store programs in a form that preserves the declarative intent of the programmer – information that gets lost in the traditional process of compiling and linking and building software.

REFERENCES

[1] F. P. Brooks Jr., “No silver bullet — essence and accidents of software engineering,” *Computer*, vol. 20, no. 4, pp. 10–19, Apr. 1987, ISSN: 0018-9162. DOI: 10.1109/MC.1987.1663532.

[2] B. Moseley and P. Marks, “Out of the tar pit,” in *SOFTWARE PRACTICE ADVANCEMENT (SPA)*, 2006.

[3] N. Wirth, “A plea for lean software,” *Computer*, vol. 28, no. 2, pp. 64–68, 1995.

[4] S. Kell, D. P. Mulligan, and P. Sewell, “The missing link: Explaining elf static linking, semantically,” in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016, Amsterdam, Netherlands: ACM, 2016, pp. 607–623, ISBN: 978-1-4503-4444-9. DOI: 10.1145/2983990.2983996.

[5] C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen, and S. Zacchiroli, “Why do software packages conflict?” *IEEE International Working Conference on Mining Software Repositories*, pp. 141–150, 2012, ISSN: 21601852. DOI: 10.1109/MSR.2012.6224274.

[6] J. Armstrong, “The mess we’re in,” in *Strange Loop Conference. Keynote*, 2014.

[7] E. Dolstra and A. Löh, “NixOS: A purely functional linux distribution,” in *ACM Sigplan Notices*, ACM, vol. 43, 2008, pp. 367–378.

[8] *Reproducible builds*, https://reproducible-builds.org.

[9] V. Cascadian and H. Levsen, “There and back again, reproducibly!” Linux Developer Conference Brazil 2019, 2019.

[10] C. Dong, “Bitcoin Build System Security,” Breaking Bitcoin, 2019.

[11] B. Hof, “Software transparency: package security beyond signatures and reproducible builds,” Annual Debian Developer’s Conference (DebConf), 2018, [Online]. Available: https://debconf18.debconf.org/talks/104-software-transparency-package-security-beyond-signatures-and-reproducible-builds/.

[12] B. Daroussin, “Reproducible builds in FreeBSD packages,” FOSDEM, 2016.

[13] W. Dietz and V. Adve, “Software multiplexing: Share your libraries and statically link them too,” *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 154:1–154:26, Oct. 2018, ISSN: 2475-1421. DOI: 10.1145/3276524.

[14] O. Kiselyov and C.-c. Shan, “Delimited Continuations in Operating Systems,” *Modeling and Using Context*, pp. 291–302, ISSN: 03029743. DOI: 10.1007/978-3-540-74255-5_22.

[15] E. Z. Yang, *The fundamental problem of programming language package management*, 2014. [Online]. Available: http://blog.ezyang.com/2014/08/the-fundamental-problem-of-programming-language-package-management/.

[16] G. Balakrishnan and T. Reps, “WYSINWYX: What you see is not what you execute,” *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, 23:1–23:84, Aug. 2010, ISSN: 0164-0925. DOI: 10.1145/1749608.1749612.

[17] E. Dolstra, M. De Jonge, E. Visser, *et al.*, “Nix: A safe and policy-free system for software deployment.,” in *LISA*, vol. 4, 2004, pp. 79–92.

[18] (2020). “Nix: The purely functional package manager,” [Online]. Available: https://nixos.org/nix.

[19] E. Dolstra, “Integrating software construction and software deployment,” in *Software Configuration Management*, Springer, 2001, pp. 102–117.

[20] R. W. Sebesta, *Concepts of programming languages*. Boston: Pearson, 2012.

[21] A. Egri-Nagy, “Declarativeness: the work done by something else,” pp. 1–12, 2017. arXiv: 1711.09197.

[22] P. Van Roy and S. Haridi, *Concepts, Techniques, and Models of Computer Programming*. 2003, ISBN: 0-262-22069-5. DOI: 10.1017/S1471068405002450.

[23] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, “The concept assignment problem in program understanding,” Tech. Rep. MSR-TR-93-03, May 1993, p. 18. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-concept-assignment-problem-in-program-understanding/.

[24] W. J. Rapaport, *Philosophy of Computer Science*. 2016.

[25] K. Gödel, “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I,” *Monatshefte für Mathematik und Physik*, vol. 38-38, no. 1, pp. 173–198, Dec. 1931, ISSN: 0026-9255. DOI: 10.1007/BF01700692.

[26] H. Samimi, C. Deaton, Y. Ohshima, A. Warth, and T. Millstein, “Call by Meaning,” *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software - Onward! ’14*, no. 310, pp. 11–28, 2014. DOI: 10.1145/2661136.2661152.

[27] E. S. Raymond, *The cathedral & the bazaar: musings on Linux and open source by an accidental revolutionary*. O’Reilly, 1999, ISBN: 1565927249.

[28] Z. Somogyi, F. Henderson, and T. Conway, “The execution algorithm of mercury, an efficient purely declarative logic programming language,” *The Journal of Logic Programming*, vol. 29, no. 1-3, pp. 17–64, 1996.

[29] K. Hinsen, “The promises of functional programming,” *Computing in Science Engineering*, vol. 11, no. 4, pp. 86–90, Jul. 2009, ISSN: 1558-366X. DOI: 10.1109/MCSE.2009.129.

[30] P. Wadler, “An angry half-dozen,” *SIGPLAN Not.*, vol. 33, no. 2, pp. 25–30, Feb. 1998, ISSN: 0362-1340. DOI: 10.1145/274930.274933.

[31] T. Bandt. (2019). “Who Cares About Functional Programming?” [Online]. Available: https://thomasbandt.com/who-cares-about-functional-programming.

[32] P. Wadler, “Why no one uses functional languages,” *ACM SIGPLAN Notices*, vol. 33, no. 8, pp. 23–27, Aug. 1998, ISSN: 03621340. DOI: 10.1145/286385.286387.

[33] P. Graham. (Apr. 2001). “Beating the averages,” [Online]. Available: http://paulgraham.com/avg.html.

[34] P. Krill, “Functional languages: What they are, where they’re going,” *InfoWorld*, Feb. 2016. [Online]. Available: https://www.infoworld.com/article/3033912/functional-languages-what-they-are-where-theyre-going.html.

[35] E. C. Berkeley and D. G. Bobrow, *The programming language LISP: Its operation and applications*, ISBN: 0262590042.

[36] A. Goldberg, *SMALLTALK-80: The Interactive Programming Environment*. USA: Addison-Wesley Longman Publishing Co., Inc., 1984, ISBN: 0201113724.

[37] A. P. Black, O. Nierstrasz, S. Ducasse, and D. Pollet, *Pharo by example*. Lulu, 2009, ISBN: 9783952334140. [Online]. Available: https://gforge.inria.fr/frs/download.php/25599/PBE1-2009-10-28.pdf.

[38] R. P. Gabriel. (1990). "Worse is better," [Online]. Available: http://www.dreamsongs.com/WorseIsBetter.html.

[39] A. Meyers. (Sep. 2014). "Worse is better vs. better is better," [Online]. Available: https://andrumyers.wordpress.com/2014/09/20/worse-is-better-vs-better-is-better/.

[40] F. Ekholdt, *Adept – the predictable dependency management system*. [Online]. Available: https://github.com/adept-dm/adept.

[41] Google, *Bazel*. [Online]. Available: https://bazel.build.

[42] E. Lotem, *Buildsome: The awesome build system*. [Online]. Available: https://buildsome.github.io/buildsome.

[43] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula, "Cloudbuild: Microsoft's distributed and caching build service," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 11–20.

[44] A. Mokhov, N. Mitchell, S. Peyton Jones, and S. Marlow, "Non-recursive make considered harmful: Build systems at scale," in *Proceedings of the 9th International Symposium on Haskell*, ser. Haskell 2016, Nara, Japan: Association for Computing Machinery, 2016, pp. 170–181, ISBN: 9781450344340. DOI: 10.1145/2976002.2976011.

[45] S. Erdweg, M. Lichter, and M. Weiel, "A sound and optimal incremental build system with dynamic dependencies," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 89–106, 2015.

[46] N. Mitchell, "Shake before building: Replacing make with haskell," in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '12, Copenhagen, Denmark: Association for Computing Machinery, 2012, pp. 55–66, ISBN: 9781450310543. DOI: 10.1145/2364527.2364538.

[47] M. Shal. (). "Tup build system," [Online]. Available: http://gittup.org/tup/.

[48] R. Stallman and R. McGrath, *GNU make: A program for directing recompilation*, 0.25 Beta for make, Version 3.57 Beta, Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA, Tel: (617) 876-3296, Oct. 1989, pp. vi + 118.

[49] E. Z. Yang, *The convergence of compilers, build systems and package managers*, 2015. [Online]. Available: http://blog.ezyang.com/2015/12/the-convergence-of-compilers-build-systems-and-package-managers/.

[50] S. Chaturvedi, *Systems, not programs*, Oct. 2018. [Online]. Available: https://shalabh.com/programmable-systems/systems-not-programs.html.

[51] A. Mokhov and V. Khomenko, "Algebra of parameterised graphs," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, Jul. 2014, ISSN: 1539-9087. DOI: 10.1145/2627351.

[52] S. Chaturvedi, *Meta idea - finding the frame*, Jun. 2018. [Online]. Available: https://shalabh.com/programmable-systems/finding-the-frame.html.

[53] E. Briggs, *Fast TCP loopback performance and low latency with Windows Server 2012 TCP Loopback Fast Path*, http://blogs.technet.com/b/wincat/archive/2012/12/05/fast-tcp-loopback-performance-and-low-latency-with-windows-server-2012-tcp-loopback-fast-path.aspx, Dec. 2012.

[54] D. Quintero, S. Chabrolles, C. Chen, M. Dhandapani, T. Holloway, C. Jadhav, S. Kim, S. Kurian, B. Raj, R. Resende, *et al.*, *IBM Power Systems Performance Guide: Implementing and Optimizing*. IBM Redbooks, 2013, ISBN: 9780738437668. [Online]. Available: https://www.redbooks.ibm.com/abstracts/sg248080.html.

[55] J. Mauro and R. McDougall, *Solaris Internals (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006, ISBN: 0131482092.

[56] J. Corbet, *TCP friends*, https://lwn.net/Articles/511254/, Aug. 2012.

[57] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim, "Inter-domain socket communications supporting high performance and full binary compatibility on Xen," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08, Seattle, WA, USA: ACM, 2008, pp. 11–20, ISBN: 978-1-59593-796-4. DOI: 10.1145/1346256.1346259.

[58] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin, "XenSocket: A high-throughput interdomain transport for virtual machines," in *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, ser. Middleware '07, Newport Beach, California: Springer-Verlag New York, Inc., 2007, pp. 184–203. [Online]. Available: https://dl.acm.org/citation.cfm?id=1516124.1516138.

[59] J. Wang, K.-L. Wright, and K. Gopalan, "XenLoop: A transparent high performance inter-VM network loopback," *Cluster Computing*, vol. 12, no. 2, pp. 141–152, Jun. 2009, ISSN: 1386-7857. DOI: 10.1007/s10586-009-0079-x.

[60] G. L. Taboada, J. Touriño, and R. Doallo, "Java fast sockets: Enabling high-speed java communications on high performance clusters," *Comput. Commun.*, vol. 31, no. 17, pp. 4049–4059, Nov. 2008, ISSN: 0140-3664. DOI: 10.1016/j.comcom.2008.08.012.

[61] S. Smith, A. Madhavapeddy, C. Smowton, M. Schwarzkopf, R. Mortier, R. M. Watson, and S. Hand, "The case for reconfigurable I/O channels," in *RESoLVE workshop at ASPLOS*, vol. 12, 2012.

[62] S. H. Rodrigues, T. E. Anderson, and D. E. Culler, "High-performance local area communication with Fast Sockets," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '97, Anaheim, California: USENIX Association, 1997, pp. 20–20. [Online]. Available: https://dl.acm.org/citation.cfm?id=1268680.1268700.

[63] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A mechanism for integrated communication and computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ser. ISCA '92, Queensland, Australia: ACM, 1992, pp. 256–266, ISBN: 0-89791-509-7. DOI: 10.1145/139669.140382.

[64] *Universal Fast Sockets*, http://torusware.com/product/universal-fast-sockets-ufs/.

[65] S. Vinoski, "Advanced message queuing protocol," *IEEE Internet Computing*, vol. 10, no. 6, pp. 87–89, Nov. 2006, ISSN: 1089-7801. DOI: 10.1109/MIC.2006.116.

[66] *ZeroMQ*, https://zeromq.org.

[67] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd, "The virtual interface architecture," *IEEE micro*, vol. 18, no. 2, pp. 66–76, 1998.

[68] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris, “Improving network connection locality on multicore systems,” in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys ’12, Bern, Switzerland: ACM, 2012, pp. 337–350, ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168870.

[69] L. Rizzo, “netmap: A novel framework for fast packet I/O,” in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC’12, Boston, MA: USENIX Association, 2012, pp. 9–9. [Online]. Available: https://dl.acm.org/citation.cfm?id=2342821.2342830.

[70] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, “mTCP: A highly scalable user-level TCP stack for multicore systems,” in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’14, Seattle, WA: USENIX Association, 2014, pp. 489–502, ISBN: 978-1-931971-09-6. [Online]. Available: https://dl.acm.org/citation.cfm?id=2616448.2616493.

[71] D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman, “Microkernel operating system architecture and Mach,” in *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 1992, pp. 11–30.

[72] D. R. Engler, M. F. Kaashoek, and J. O’Toole Jr., “Exokernel: An operating system architecture for application-level resource management,” in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’95, Copper Mountain, Colorado, USA: ACM, 1995, pp. 251–266, ISBN: 0-89791-715-4. DOI: 10.1145/224056.224076.

[73] I. Marinos, R. N. M. Watson, and M. Handley, “Network stack specialization for performance,” *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks - HotNets-XII*, pp. 1–7, 2013. DOI: 10.1145/2535771.2535779.

[74] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, “IX: A protected dataplane operating system for high throughput and low latency,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 49–65, ISBN: 978-1-931971-16-4. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay.

[75] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis and transformation,” in *Proc. Conf. on Code Generation and Optimization*, San Jose, CA, USA, Mar. 2004, pp. 75–88.

[76] T. Ravitch, *WLLVM: Whole-Program LLVM*, https://github.com/travitch/whole-program-llvm, 2011.

[77] R. Felker, *musl libc*, https://musl.libc.org/.

[78] M. Larabel. (Oct. 2016). “ALLVM: Forthcoming project to ship all software as LLVM IR,” [Online]. Available: https://www.phoronix.com/scan.php?page=news_item%5C&px=LLVM-IR-ALLVM.

[79] *LLVM developers’ meeting 2016*, Nov. 2016. [Online]. Available: https://sched.co/8Yzq.

[80] N. D. Matsakis and F. S. Klock, “The rust language,” *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014.

[81] J. Goodwill and W. Matlock, “The swift programming language,” in *Beginning Swift Games Development for iOS*, Springer, 2015, pp. 219–244.

[82] A. Kelley. (Feb. 2016). "Introduction to the zig programming language," [Online]. Available: https://andrewkelley.me/post/intro-to-zig.html.

[83] D. B. Orr, J. Lepreau, J. Bonn, and R. Mecklenburg, "Fast and flexible shared libraries," in *Proceedings of the Summer 1993 USENIX Conference, Cincinnati, OH, USA, June 21-25, 1993*, 1993, pp. 237–252.

[84] M. N. Nelson and G. Hamilton, "High performance dynamic linking through caching," in *Proceedings of the USENIX Summer 1993 Technical Conference - Volume 1*, ser. Usenix-stc'93, Cincinnati, Ohio: USENIX Association, 1993, 17:1–17:14. [Online]. Available: https://dl.acm.org/citation.cfm?id=1361453.1361470.

[85] J. Jelinek, "Prelink," Red Hat, Inc., Tech. Rep., Mar. 2004. [Online]. Available: https://people.redhat.com/jakub/prelink.pdf.

[86] H. Yoon, C. Min, and Y. I. Eom, "Dynamic-prelink: An enhanced prelinking mechanism without modifying shared libraries," in *Proceedings of the International Conference on Embedded Systems and Applications (ESA)*, The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2014, p. 1.

[87] C. Jung, D.-K. Woo, K. Kim, and S.-S. Lim, "Performance characterization of prelinking and preloading for embedded systems," in *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, ser. EMSOFT '07, Salzburg, Austria: ACM, 2007, pp. 213–220, ISBN: 978-1-59593-825-1. DOI: 10.1145/1289927.1289961.

[88] W. W. Ho, W.-C. Chang, and L. H. Leung, "Optimizing the performance of dynamically-linked programs," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, ser. TCON'95, New Orleans, Louisiana: USENIX Association, 1995, pp. 19–19. [Online]. Available: https://dl.acm.org/citation.cfm?id=1267411.1267430.

[89] V. Agrawal, A. Dabral, T. Palit, Y. Shen, and M. Ferdman, "Architectural support for dynamic linking," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15, Istanbul, Turkey: ACM, 2015, pp. 691–702, ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694392.

[90] V. Reznic. (2016). "ELF STATIFIER MAIN PAGE," [Online]. Available: http://statifier.sourceforge.net/.

[91] V. Reznic. (2018). "Ermine: Linux portable application creator," [Online]. Available: http://www.magicermine.com/.

[92] scrut, *Reducebind.c - dynamic to static binary conversion utility*, 2003. [Online]. Available: https://dl.packetstormsecurity.net/groups/teso/reducebind.c.

[93] *Cryopid*, http://freecode.com/projects/cryopid/, Original homepage is no longer available., 2006.

[94] C. Giuffrida, C. Iorgulescu, G. Tamburrelli, and A. S. Tanenbaum, "Automating live update for generic server programs," *IEEE Trans. Softw. Eng.*, vol. 43, no. 3, pp. 207–225, Mar. 2017, ISSN: 0098-5589. DOI: 10.1109/TSE.2016.2584066.

[95] J. R. Levine, *Linkers and Loaders*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, ISBN: 1558604960.

[96] C. A. Waldspurger, "Memory resource management in VMware ESX server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, Dec. 2002, ISSN: 0163-5980. DOI: 10.1145/844128.844146.

[97] "Increasing memory density by using KSM," in *Proceedings of the Linux Symposium*, Montreal, Canada, 2009, pp. 19–28.

[98] C. S. Collberg, J. H. Hartman, S. Babu, and S. K. Udupa, "SLINKY: Static linking reloaded," in *USENIX Annual Technical Conference, General Track*, 2005, pp. 309–322.

[99] W. Dietz, J. Cranmer, N. Dautenhahn, and V. Adve, "Slipstream: Automatic interprocess communication optimization," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, Santa Clara, CA: USENIX Association, 2015, pp. 431–443, ISBN: 978-1-931971-225. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/dietz.

[100] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '97, El Paso, Texas, USA: ACM, 1997, pp. 654–663, ISBN: 0-89791-888-6. DOI: 10.1145/258533.258660.

[101] B. Fitzpatrick, "Distributed caching with Memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[102] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014, ISSN: 1075-3583. [Online]. Available: https://dl.acm.org/citation.cfm?id=2600239.2600241.

[103] *HAProxy*, https://www.haproxy.org.

[104] T. Doran, *Building a smarter application stack*, Presented at DockerCon, 2014.

[105] Internet Assigned Numbers Authority, *Service Name and Transport Protocol Port Number Registry*, (Date last updated.), May 2015. [Online]. Available: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml.

[106] E. Y. Vasserman, N. Hopper, and J. Tyra, "SilentKnock: Practical, provably undetectable authentication," *International Journal of Information Security*, vol. 8, no. 2, pp. 121–135, 2009.

[107] L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ser. ATEC '96, San Diego, CA: USENIX Association, 1996, pp. 23–23. [Online]. Available: https://dl.acm.org/citation.cfm?id=1268299.1268322.

[108] R. Jones, *The Netperf benchmark*, http://www.netperf.org.

[109] Q. O. Snell, A. R. Mikler, and J. L. Gustafson, "NetPIPE: A network protocol independent performance evaluator," in *IASTED International Conference on Intelligent Information Management and Systems*, Washington, DC, USA), vol. 6, 1996.

[110] T. Ishii, *pgbench*, http://www.postgresql.org/docs/devel/static/pgbench.html.

[111] G. Smith, *Introduction to pgbench*, http://www.westnet.com/~gsmith/content/postgresql/pgbench-intro.pdf, Presented at PG East, 2009.

[112] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki, "From a to e: Analyzing TPC's OLTP benchmarks: The obsolete, the ubiquitous, the unexplored," in *Proceedings of the 16th International Conference on Extending Database Technology*, ser. EDBT '13, Genoa, Italy: ACM, 2013, pp. 17–28, ISBN: 978-1-4503-1597-5. DOI: 10.1145/2452376.2452380.

[113] G. Balakrishnan, "WYSINWYX: What you see is not what you execute," Ph.D. dissertation, 2007.

[114] U. Drepper. (2011). "How to write shared libraries," [Online]. Available: http://people.redhat.com/drepper/dsohowto.pdf.

[115] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel - from I/O ports to process management*, 3rd ed. O'Reilly, 2005, ISBN: 978-0-596-00565-8. [Online]. Available: http://www.oreilly.com/catalog/understandlk/.

[116] R. Felker, *Introduction to musl*. [Online]. Available: https://www.musl-libc.org/intro.html.

[117] NixOS/Nixpkgs Community. (2020). "Nixpkgs users and contributors guide." Version 19.09.2201.7d31bbceaa1, [Online]. Available: https://nixos.org/nixpkgs/manual/.

[118] *IBus - Intelligent Input Bus for Linux / Unix OS*. [Online]. Available: https://github.com/ibus/ibus/wiki.

[119] *fcitx*. [Online]. Available: https://fcitx-im.org.

[120] G. V. Vaughan, B. Elliston, T. Tromey, and I. L. Taylor, *GNU Autoconf, Automake and Libtool*. Carmel, IN, USA: New Riders Publishing, 2000, pp. xx + 390, ISBN: 1-57870-190-2. [Online]. Available: http://sources.redhat.com/autobook/.

[121] J. Calcote, *Autotools: a practitioner's guide to GNU Autoconf, Automake, and Libtool*. San Francisco, CA, USA: No Starch Press, 2010, ISBN: 1-59327-206-5 (paperback).

[122] *Libtool: Dlpreopening*, https://www.gnu.org/software/libtool/manual/html_node/Dlpreopening.

[123] W. Bastian, R. Lortie, and L. Poettering, *XDG base directory specification*, 2010. [Online]. Available: https://specifications.freedesktop.org/basedir-spec/latest/.

[124] MITRE Corporation, *CVE-2017-7494*, 2017. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7494.

[125] M. Moore, "Distroless docker: Containerizing apps, not vms," SwampUP, 2017, [Online]. Available: https://swampup2017.sched.com/event/A6CW/distroless-docker-containerizing-apps-not-vms.

[126] A. Schwab. (Mar. 2005). "Re: Statically linking against a shared library." Binutils mailing list, [Online]. Available: https://sourceware.org/ml/binutils/2005-03/msg00350.html.

[127] Intel Corp, *Intel(R) C++ Compiler 19.0 Developer Guide and Reference*, 2019. [Online]. Available: https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-interprocedural-optimization-ipo.

[128] IBM Corp, "Code optimization with the IBM XL compilers on Power architectures," White Paper, 2018. [Online]. Available: https://www.ibm.com/support/pages/code-optimization-ibm-xl-compilers-power-architectures.

[129] V. Adve, W. Dietz, *et al.* (2016). "ALLVM Project," [Online]. Available: http://allvm.org.

[130] *openSUSE: Shared library packaging policy*, 2017. [Online]. Available: https://en.opensuse.org/openSUSE:Shared_library_packaging_policy.

[131] X. Meng and B. P. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016, New York, NY, USA: ACM, 2016, pp. 24–35, ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931047.

[132] C. S. Collberg, J. H. Hartman, S. Babu, and S. K. Udupa, *Slinky - static linking reloaded*, 2004. [Online]. Available: http://slinky.cs.arizona.edu/.

[133] *Proportional set size*, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Proportional_set_size.

[134] B. Jasmin, *C++ GUI Programming with Qt4, 2/e*. Pearson Education India, 2008.

[135] B. De Bus, D. Kästner, D. Chanet, L. Van Put, and B. De Sutter, "Post-pass compaction techniques," *Commun. ACM*, vol. 46, no. 8, pp. 41–46, Aug. 2003, ISSN: 0001-0782. DOI: 10.1145/859670.859696.

[136] B. De Sutter, H. Vandierendonck, B. De Bus, and K. De Bosschere, "On the side-effects of code abstraction," in *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '03, San Diego, California, USA: ACM, 2003, pp. 244–253, ISBN: 1-58113-647-1. DOI: 10.1145/780732.780766.

[137] N. E. Johnson, "Code size optimization for embedded processors," University of Cambridge, Computer Laboratory, Tech. Rep., 2004.

[138] B. De Sutter, B. De Bus, and K. De Bosschere, "Link-time binary rewriting techniques for program compaction," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 5, pp. 882–945, Sep. 2005, ISSN: 0164-0925. DOI: 10.1145/1086642.1086645.

[139] T. J. Edler von Koch, B. Franke, P. Bhandarkar, and A. Dasgupta, "Exploiting function similarity for code size reduction," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '14, Edinburgh, United Kingdom: ACM, 2014, pp. 85–94, ISBN: 978-1-4503-2877-7. DOI: 10.1145/2597809.2597811.

[140] S. Bartell, W. Dietz, A. Fortunat, and V. Adve, "Breaking the Dynamic-Linking Barrier with the BCDB," (In Prep.)

[141] R. Fellker, *Twitter post: In light of CVE-2017-7494, a massive FOSS-wide code search for dlopen & analysis of source of its argument string might be a good idea...* (@richfelker), May 2017. [Online]. Available: https://twitter.com/RichFelker/status/867573725637599232.

[142] M. Rigger, S. Marr, S. Kell, D. Leopoldseder, and H. Mössenböck, "An analysis of X86-64 inline assembly in C programs," in *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '18, Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 84–99, ISBN: 9781450355797. DOI: 10.1145/3186411.3186418.

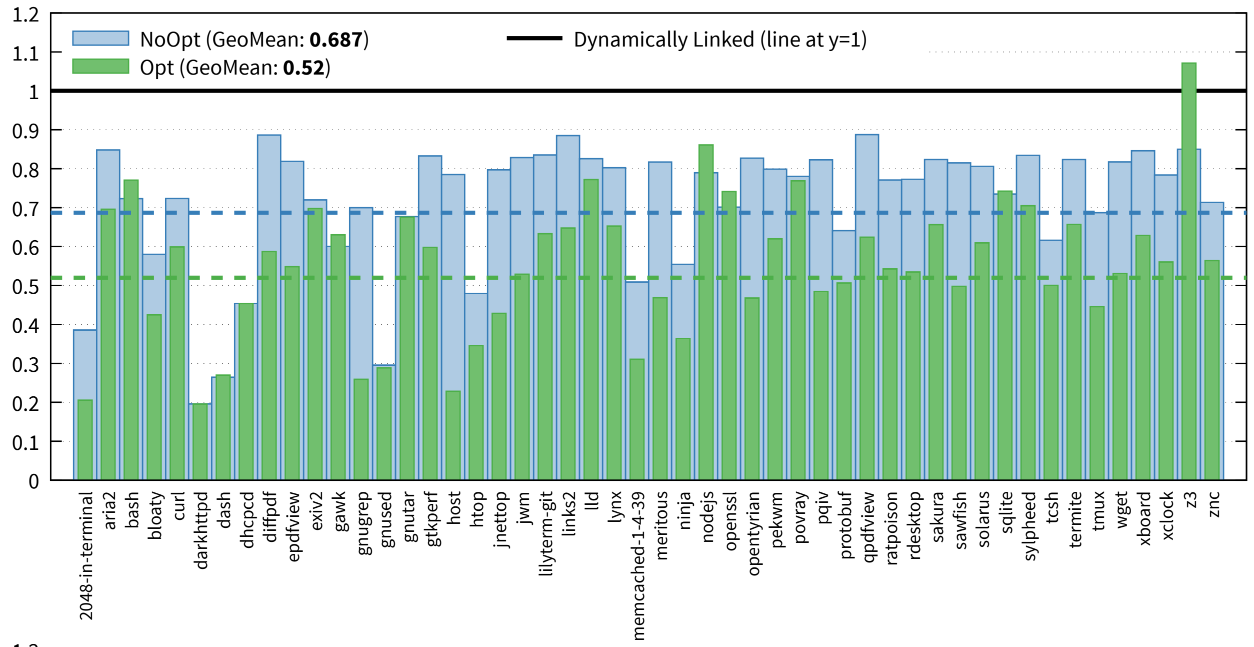
APPENDIX A: MULTIPLEXING: ADDITIONAL RESULTS

A.1 IMPACT OF MULTIPLEXING ON COMPRESSION

Table A.1: Comparing XZ Compression of Binaries.

Name	#	Dynamic			Allmux Opt			**Allmux**
	Bins	Size	XZ'd	**Ratio**	Size	XZ'd	**Ratio**	**/Native**
allvm-tools	10	46.6 MB	10.1 MB	**4.6** ×	33.1 MB	8.2 MB	**4.0** ×	**0.87** ×
cmake-28-36-38	9	67.1 MB	10.9 MB	**6.2** ×	58.6 MB	10.2 MB	**5.7** ×	**0.93** ×
• compression	24	10.7 MB	3.2 MB	**3.3** ×	5.9 MB	2.0 MB	**2.9** ×	**0.88** ×
• fs tools	166	21.5 MB	4.9 MB	**4.4** ×	12.9 MB	4.0 MB	**3.2** ×	**0.73** ×
git	16	31.4 MB	5.2 MB	**6.1** ×	15.3 MB	3.6 MB	**4.2** ×	**0.69** ×
• gnumake × 5	5	1.9 MB	0.6 MB	**3.0** ×	1.1 MB	0.3 MB	**3.4** ×	**1.16** ×
mkvtoolnix-cli	4	23.1 MB	4.3 MB	**5.4** ×	16.7 MB	3.4 MB	**4.9** ×	**0.90** ×
• qt-apps	15	121.8 MB	33.3 MB	**3.7** ×	99.6 MB	28.7 MB	**3.5** ×	**0.95** ×
radare2	10	66.5 MB	13.1 MB	**5.1** ×	46.4 MB	4.4 MB	**10.5** ×	**2.07** ×
• shells	6	8.1 MB	2.6 MB	**3.1** ×	6.5 MB	2.2 MB	**3.0** ×	**0.94** ×
snowman	2	31.8 MB	8.6 MB	**3.7** ×	23.6 MB	6.6 MB	**3.6** ×	**0.96** ×
• terms	8	40.4 MB	11.1 MB	**3.6** ×	27.3 MB	7.6 MB	**3.6** ×	**0.99** ×
• gvim+qvim	2	54.5 MB	16.9 MB	**3.2** ×	41.8 MB	13.2 MB	**3.2** ×	**0.99** ×
GeoMean				**4.1** ×			**4.0** ×	**0.97** ×

A.2 BINARY SIZES FOR VARIOUS SOFTWARE



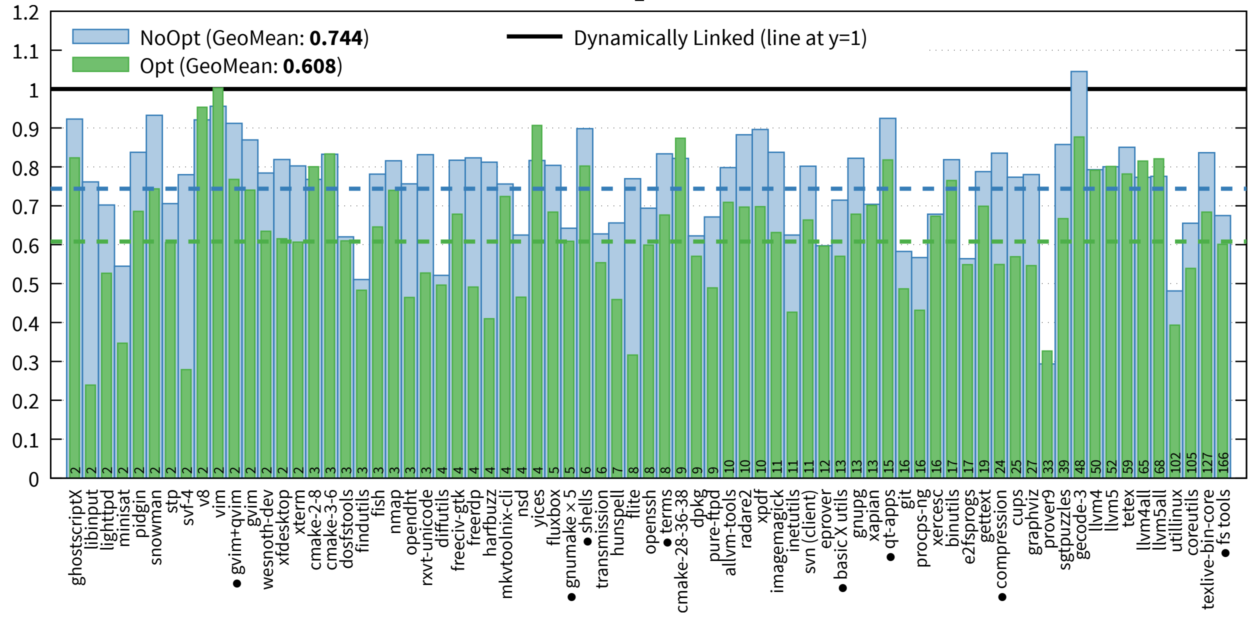


Figure A.1: Relative binary size of multiplexing vs dynamically linking, single programs (top) and multiple-program (bottom) sets. Program count shown when greater than one.

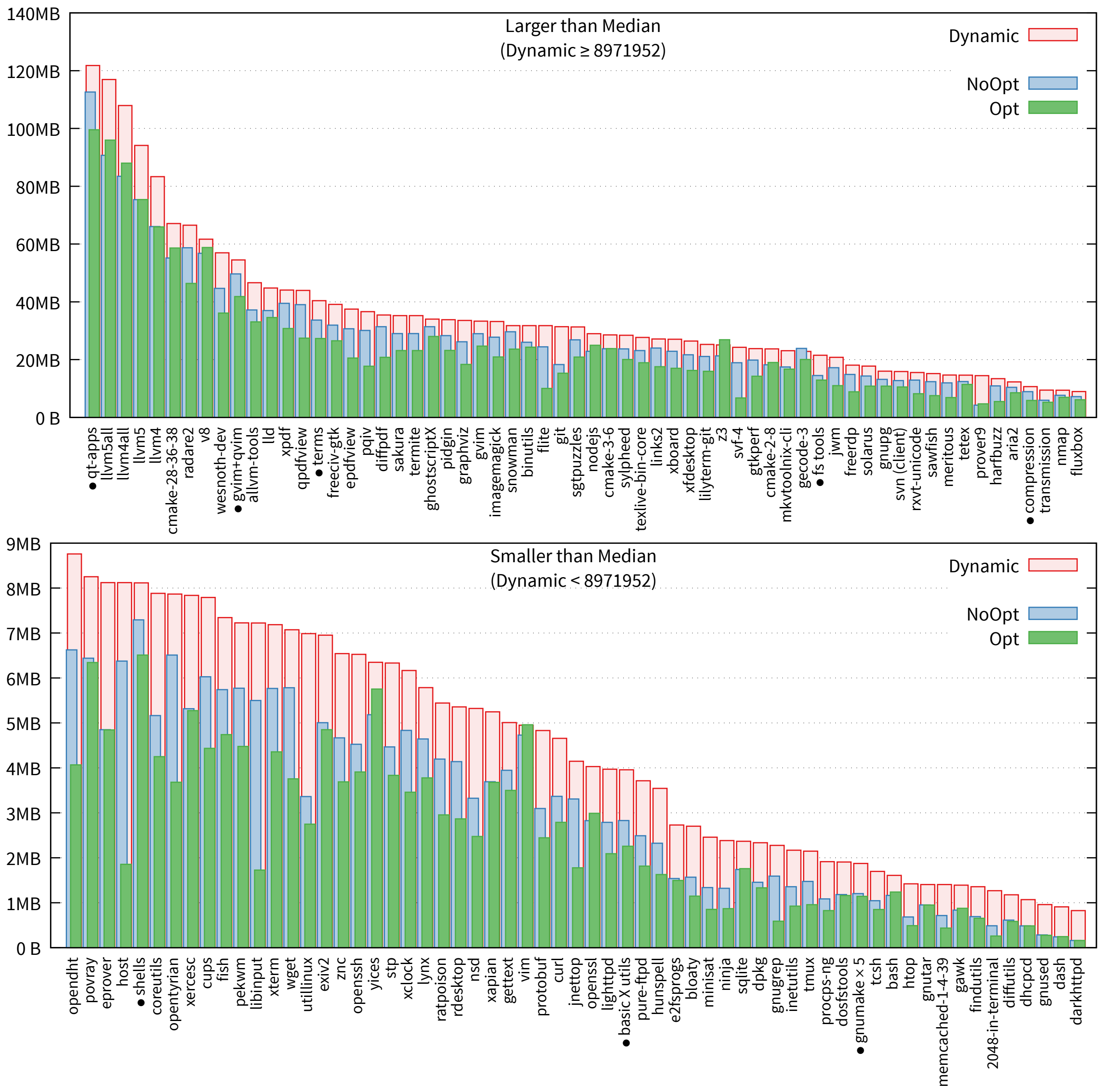
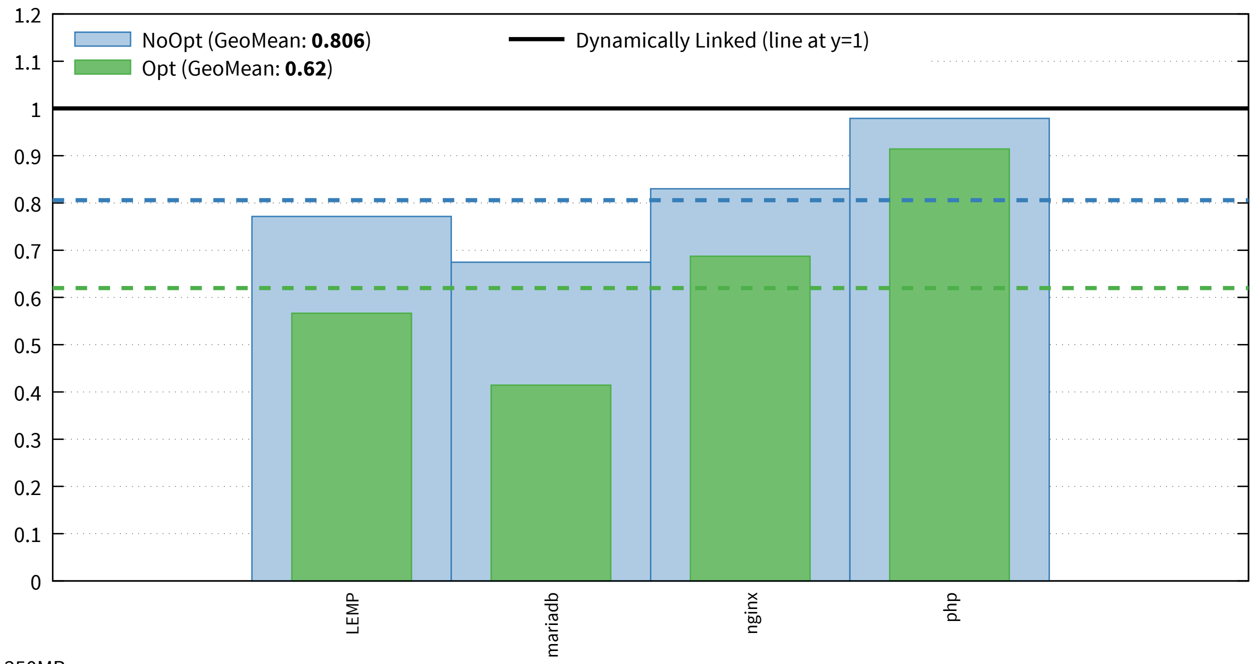


Figure A.2: Allmux vs Dynamically Linked, absolute binary sizes for same data shown in Figure A.1. Graph partitioned at median binary size.



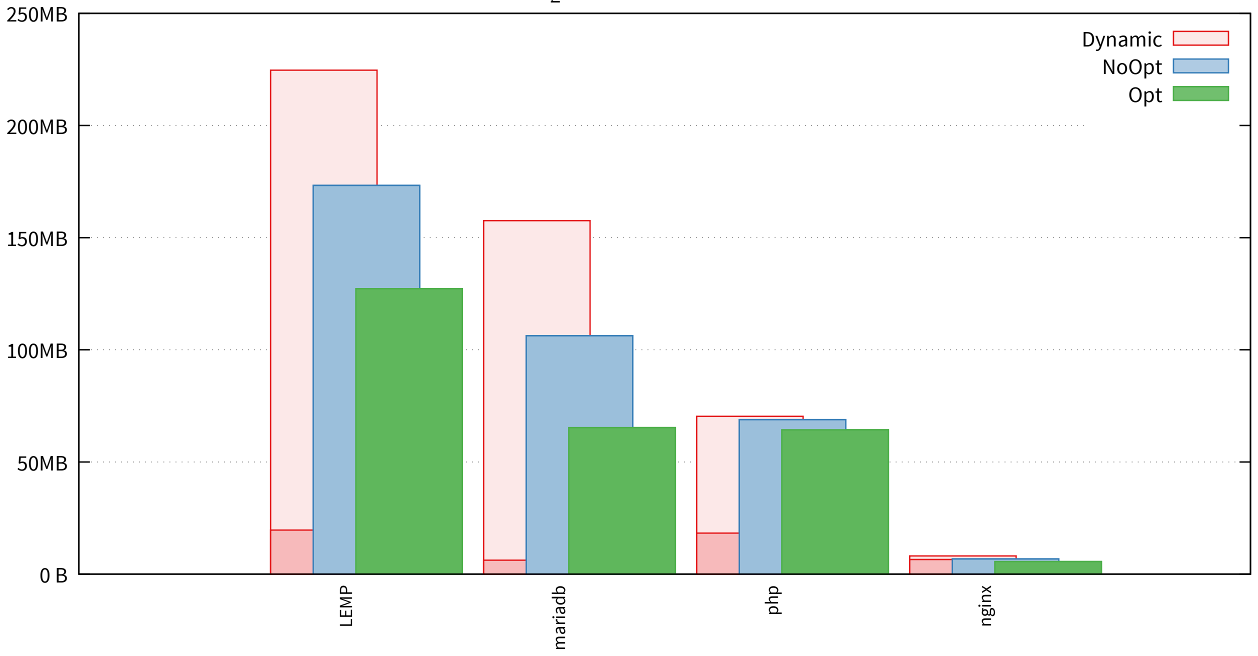


Figure A.3: Multiplexing applied to PHP, MySQL, Nginx as a collection ("LEMP") as well as each individually. Sizes relative to the dynamically linked version are shown on top, and the absolute sizes for the same data is presented in the bottom graph.

APPENDIX B: MULTIPLEXING: THEMED COLLECTIONS

B.1 LIST OF PACKAGES IN EACH COLLECTION

common:
bash, bzip2, coreutils, diffutils, findutils, gawk, gnugrep, gnumake, gnupatch, gnused, gnutar, gzip, xz

compress:
brotli, bzip2, gnutar, gzip, lrzip, lz4, lzip, rzip, unzip, xar, xz, zip, zstd

editors:
bvi, bviplus, ed, elvis, flpsed, ht, joe, kakoune, moe, nano, ne, nvi, vim, wily, zile

fs tools:
9pfs, btrfs-progs, cdrkit, dosfstools, e2fsprogs, e2tools, exfat, mtools, ntfs3g, squashfsTools, utillinuxMinimal, xorriso

net misc:
adns, aircrack-ng, aria2, arp-scan, chrony, curl, dhcpcd, dhcping, fping, httping, iperf, iproute, iputils, iw, jnettop, jwhois, miniupnpc, netcat-gnu, netperf, netrw, nettools, ngrep, nmap, ntp, openconnect, openssh, mosh, socat, tcptraceroute, traceroute, wget, wpa_supplicant

shells:
bash, dash, es, fish, tcsh, zsh

vpns:
openconnect, openfortivpn, openvpn, vpnc

window managers:
2bwm, bspwm, cwm, dwm, evilwm, fluxbox, icewm, jwm, lemonbar, matchbox, oroborus, pekwm, ratpoison, rofi, sxhkd, tabbed

B.2 BINARY SIZES FOR “LOGIC” COLLECTION

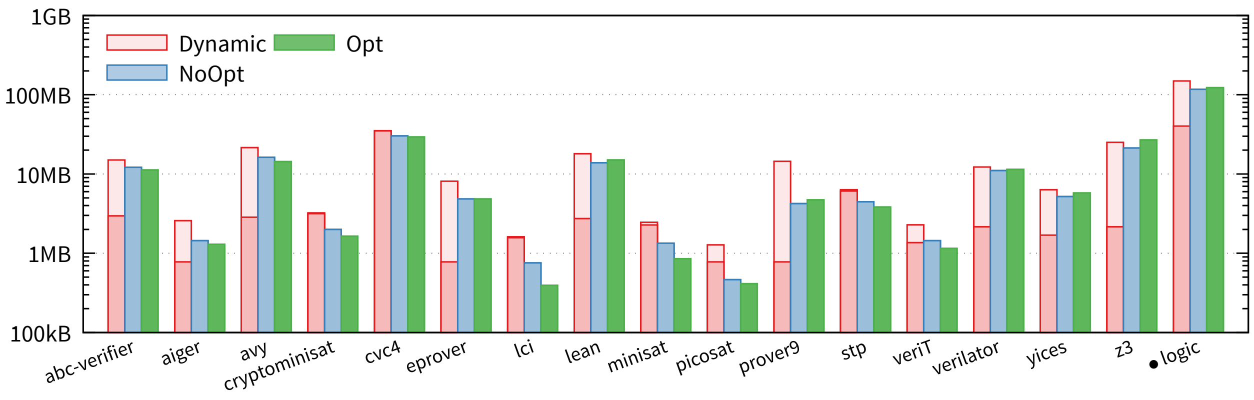


Figure B.1: Binary Sizes for Logic-related Programs (y-log).