Software Multiplexing: Share Your Libraries and Statically Link Them Too

WILL DIETZ, University of Illinois at Urbana-Champaign, USA
VIKRAM ADVE, University of Illinois at Urbana-Champaign, USA

We describe a compiler strategy we call "*Software Multiplexing*" that achieves many benefits of both statically linked and dynamically linked libraries, and adds some additional advantages. Specifically, it achieves the code size benefits of dynamically linked libraries while eliminating the major disadvantages: unexpected failures due to missing dependences, slow startup times, reduced execution performance due to indirect references to globals, and the potential for security vulnerabilities. We design Software Multiplexing so that it works even in the common case where application build systems support only dynamic and not static linking; we have automatically built thousands of Linux software packages in this way. Software Multiplexing combines two ideas: *Automatic Multicall*, i.e., where multiple independent programs are automatically merged into a single executable, and *Static Linking of Shared Libraries*, which works by linking an IR-level version of application code and all its libraries, even if the libraries are normally compiled as shared, *before* native code generation. The benefits are achieved primarily through deduplication of libraries across the multiplexed programs, while using static linking, and secondly through more effective unused code elimination for statically linked shared libraries. 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. We have implemented Software Multiplexing in a tool called `allmux`, part of the open-source ALLVM project. Example results show that when the LLVM Compiler Infrastructure is optimized using allmux, the resulting binaries and libraries are 18.3% smaller *and* 30% faster than the default production version. For 74 other packages containing 2–166 programs each, multiplexing each package into one static binary reduces the aggregate package size by 39% (geometric mean) compared with dynamic linking.

CCS Concepts: • **Software and its engineering → Compilers**; *General programming languages*;

Additional Key Words and Phrases: Code deduplication, Link-Time Optimization, LTO, LLVM, IR

ACM Reference Format:
Will Dietz and Vikram Adve. 2018. Software Multiplexing: Share Your Libraries and Statically Link Them Too. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 154 (November 2018), 26 pages. https://doi.org/10.1145/3276524

1 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 [Agrawal et al. 2015; Collberg et al. 2005; Orr et al. 1993]. 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

Authors' addresses: Will Dietz, University of Illinois at Urbana-Champaign, USA, wdietz2@illinois.edu; Vikram Adve, University of Illinois at Urbana-Champaign, USA, vadve@illinois.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2018 Copyright held by the owner/author(s).
2475-1421/2018/11-ART154
https://doi.org/10.1145/3276524

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 [MITRE Corporation 2017].

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 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.

1.1 Motivating Example

As a concrete example of the size vs. performance tradeoffs, consider Figure 1 and Table 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 2.5x 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

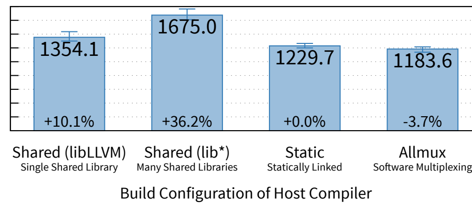

Fig. 1. Seconds to compile LLVM 4.0.1

Table 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

is 1.2x smaller and 30% faster than the Shared version, or 2.7x 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.

1.2 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 manual effort or high cost, or both. For example, Google is known to configure and build software almost entirely statically [Moore 2017] 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* [Collberg et al. 2005], 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 Sinky with Sofware Multiplexing would additionally enable redundant code pages across the multiplexed sets of applications, while preserving the extra benefits of Sofware Multiplexing within each set.)

1.3 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 9 in an Appendix 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[1] 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 [Schwab 2005]. These problems are almost entirely avoided before code generation, and so we solve this problem with a compiler-based strategy: we *export every component 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 apppropriate 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.

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

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 6.1.

1.4 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[2]. 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:

$$\text{Disk} \leq min(static, shared) \tag{1}$$

$$\text{Memory} \leq min(static, shared) \tag{2}$$

$$\text{Run time} \leq min(static, shared) \tag{3}$$

$$\text{Startup latency} \approx static \leq shared \tag{4}$$

The results in Figure 1 and Table 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 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.

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

- 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.

2 BACKGROUND

2.1 Multicall

A few program collections today, e.g., busybox and coreutils, are (optionally) organized as multicall programs (defined in Section 1). 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.

2.2 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 LLVM, GCC, Intel's ICC, IBM's XL compiler family, 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 [Lattner and Adve 2004] 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 [Adve et al. 2016] 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.

3 GENERATING MULTICALL PROGRAMS

Algorithm 1 Basic Allmux

```
1: function MuxBasic(A)                              ▷ Multiplex set of allexe programs A
2:     M ←GenDispatchMain(A)                          ▷ Described in Section 3.1.1
3:     for a ∈ 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, . . .)
10: end function
```

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

- generate new entry point that runs the selected program;
- transform each input program to use a uniquely named entry point, and execute only its own constructors and destructors; and
- 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 3.1) and then the full algorithm (Section 3.2).

3.1 Allmux Basic

The basic allmux transformation is presented in Algorithm 1, and a graphical overview is shown in Figure 2.

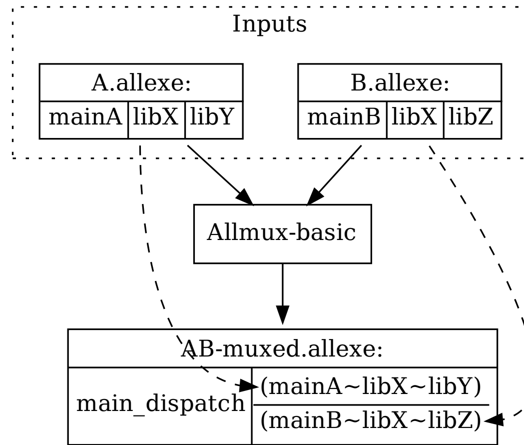

Fig. 2. Allmux Basic

3.1.1 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`[3].

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.

3.1.2 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.

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.

3.1.3 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`

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

or dlsym (this limitation is discussed further in Section 6.1). Unwinding still works properly using eh_frame information as usual on Linux, as does dl_iterate_phdr.

3.1.4 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.

3.1.5 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).

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 6.1). Our evaluation in Section 5.8 shows the impact of this code duplication on resulting executable sizes.

3.2 Multiplexing with Library Deduplication

Algorithm 2 Allmux w/Library Deduplication

1: **function** MuxLibDedup(A) ▷ Multiplex set of allexe programs A
2: $M \leftarrow$GenDispatchMainLibs(A) ▷ Described in Section 3.2.2
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 2 and an updated graphical overview is shown in Figure 3.

3.2.1 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

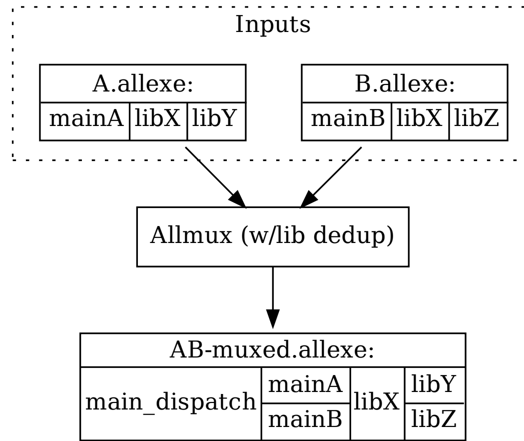

Fig. 3. Allmux with Library Deduplication
Only one copy of each library is emitted into the final allexe. Compare with Figure 2 where two copies of libX are included.

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).

3.2.2 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.

4 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 [openSUSE 2017]. 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— surprisingly effective approach that enables us to statically link many (but not all, see Section 6.1) applications consisting of shared libraries.

4.1 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 [Balakrishnan and Reps 2010; Meng and Miller 2016].

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.

4.2 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 [Drepper 2011], 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.

4.3 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 EVALUATION

5.1 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 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*[Collberg et al. 2005] (which is available on their website[Collberg et al. 2004]), 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 5.9.

5.2 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 (1)–(4) in Section 1.

Binary Size (Claim 1 ⟶ Sections 5.4.2 and 5.8). : 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 2 ⟶ Section 5.4.1): To explore the memory usage of multiplexed applications we use a collection of graphical programs a "typical" user might execute concurrently (Section 5.4.1). 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 [PSS 2016]:

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

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

Runtime Performance (Claim 3 ⟶ Section 5.5): 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 4 ⟶ Section 5.6): A handful of applications were selected and startup latency was measured with and without background I/O loads, methodology adopted from Phoronix

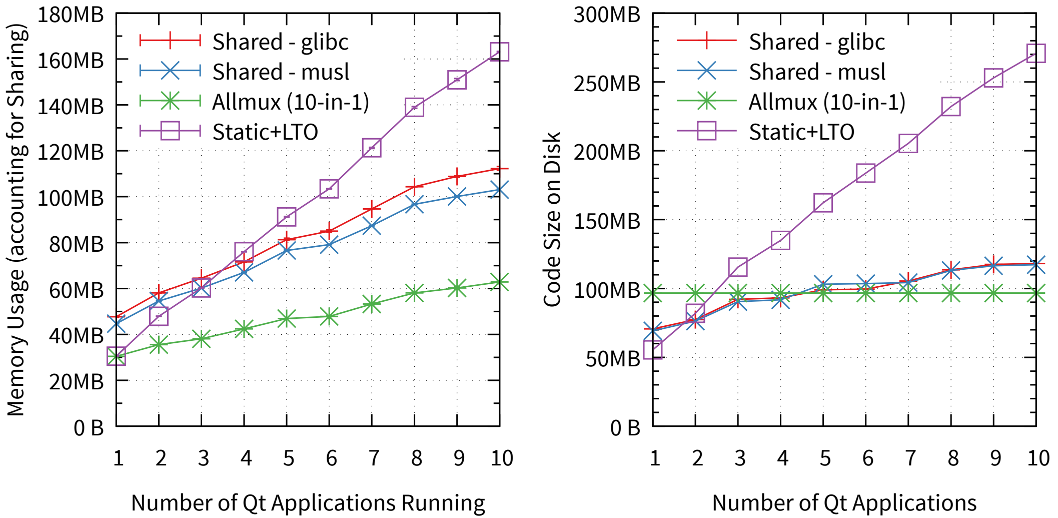

Fig. 4. Memory Usage and Disk Usage for Increasing Number of Qt Applications

Table 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

Test Suite's "Application Start-Up Time 1.0.0" modified to run our executables (Section 5.6)[4]. The benchmark was conducted for 10 iterations with consistent results.

5.3 Experimental Hardware

Performance and startup latency experiments (Section 5.6 and Figure 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).

5.4 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[Jasmin 2008] applications in the various configurations described in Section 5.1; the applications chosen

[4]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

and a brief description of each is given in Table 2. These were used to measure the effectiveness of software multiplexing in terms of reduced footprint, including both memory and disk usage.

5.4.1 Memory Usage. To measure memory usage across processes sharing code (Claim 2), we measured the `PSS`[PSS 2016] 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 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 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.

5.4.2 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.

5.5 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 1 in Section 1. 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 1 in Section 1 showed that *Allmux* is far smaller than *Static* and also smaller than both *Shared* versions: the best of both worlds.

5.6 Startup Latency vs I/O Load

The performance results for compiling LLVM 4.0.1 shown in Figure 1 are caused, to a substantial extent, by lower startup latency of the `allmux` version of Clang. This is because Clang must be

Table 3. Startup Latency of Applications (seconds)

IO 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 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 IO load is simulated and are slightly slower when heavy background IO is performed (but much faster than the dynamic versions).

5.7 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.

5.7.1 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. 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%.

5.7.2 Across Versions. 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[5].

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

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

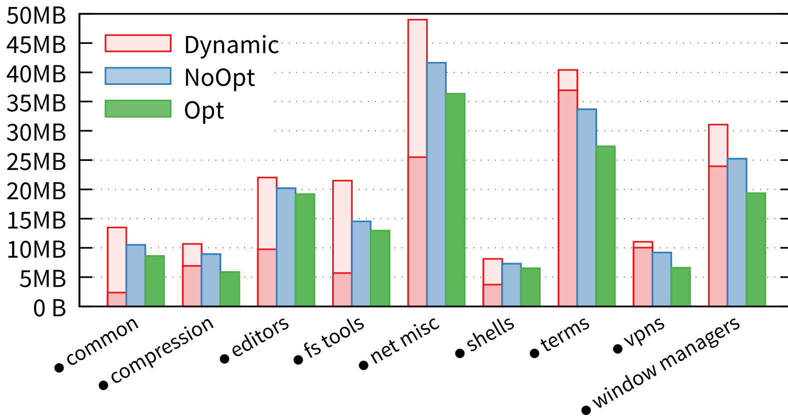

Fig. 5. Sizes of "themed" collections of utilities (dark pink portions represent shared libraries).

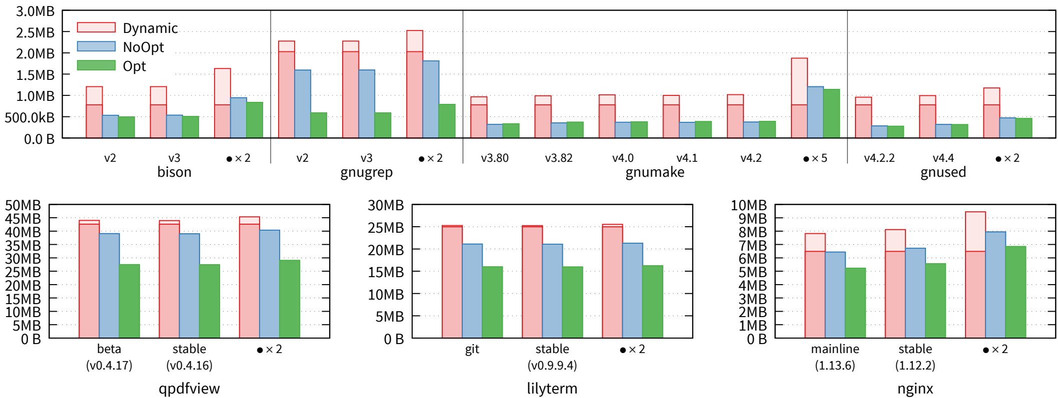

Fig. 6. Multiplexing multiple versions of software together, binary sizes (dark pink represents shared libraries)

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 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 6.2.

5.8 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 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 3x 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!

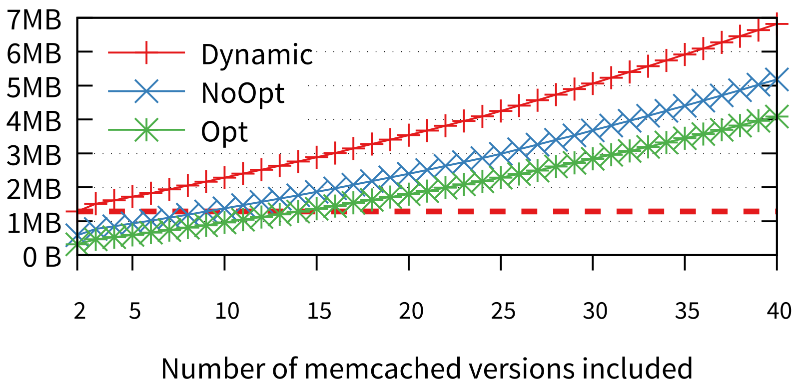

Fig. 7. Up to 40 Versions of Memcached at Once: Binary Sizes

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 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.

5.9 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

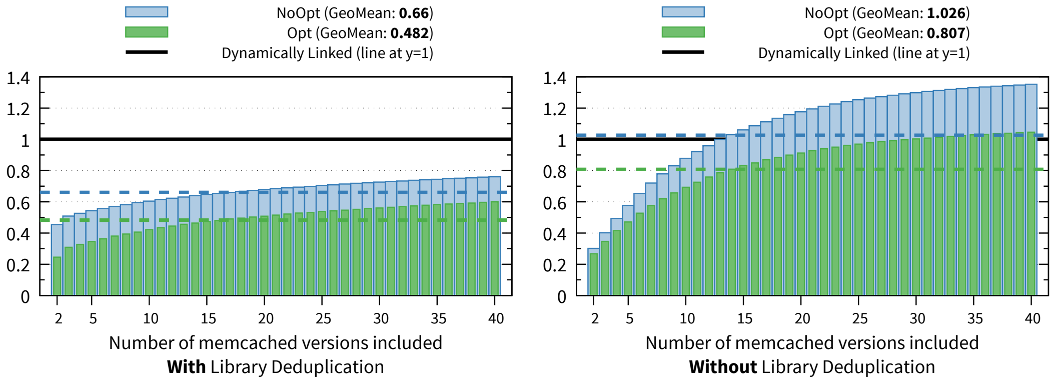

Fig. 8. Multiplexing up to 40 versions of Memcached, Relative to *aggregate* size of dynamically linked versions. Dashed lines show geometric mean reductions for Opt and NoOpt cases.

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.

6 DISCUSSION

6.1 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.

6.2 Opportunities

Finer-grain Code Deduplication: So far, `allmux` only eliminates duplicate copies of libraries shared by two or more applications that are muxed 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 [De Bus et al. 2003; De Sutter et al. 2005, 2003; Edler von Koch et al. 2014; Johnson 2004]. 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 [Dietz et al. 2015]) optimizing across the communication boundaries between those programs by analyzing and transforming the programs *collectively*.

6.3 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 [MITRE Corporation 2017]. It also makes it harder to create attacks through improper updates to dynamic libraries.

7 RELATED

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.

7.1 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 [Orr et al. 1993] describes a novel shared library implementation that speeds up dynamic linking by caching executables and libraries after symbol resolution and relocation. SpringOS [Nelson and Hamilton 1993] achieves a similar caching effect for applications and shared libraries by caching them after applications exit. Red Hat's `prelink` [Jelinek 2003] tool precomputes relocation information and address assignment at link time instead of doing these at run time. Later work [Yoon et al. 2014] extends this to allow use of Address Space Layout Randomization (ASLR). Prelinking and Preloading [Jung et al. 2007] 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 [Ho et al. 1995]. 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 [Agrawal et al. 2015] 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 [CryoPID 2006; Reznic 2016, 2018; scrut 2003] 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 [Giuffrida et al. 2017]. 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 9 or N=1 case for `memcached` in Figure 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.

7.2 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 [Levine 1999], 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 [Levine 1999] 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 [Waldspurger 2002] and share such pages between virtual machine instances on a single host. Kernel Same-page Merging (KSM) [KSM 2009] 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 [Collberg et al. 2005] is perhaps the most effective previous solution, and is discussed in Section 1 and Section 5.9. In comparison, Software Multiplexing achieves better code size reduction and lower overheads, as discussed in Section 5.9, and does not require changes to the OS kernel, system linker or loader, but does require changes to the compiler while Slinky doesn't.

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.

A ADDITIONAL RESULTS

A.1 Impact of Multiplexing on Compression

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

A.2 Binary Sizes for Various Software

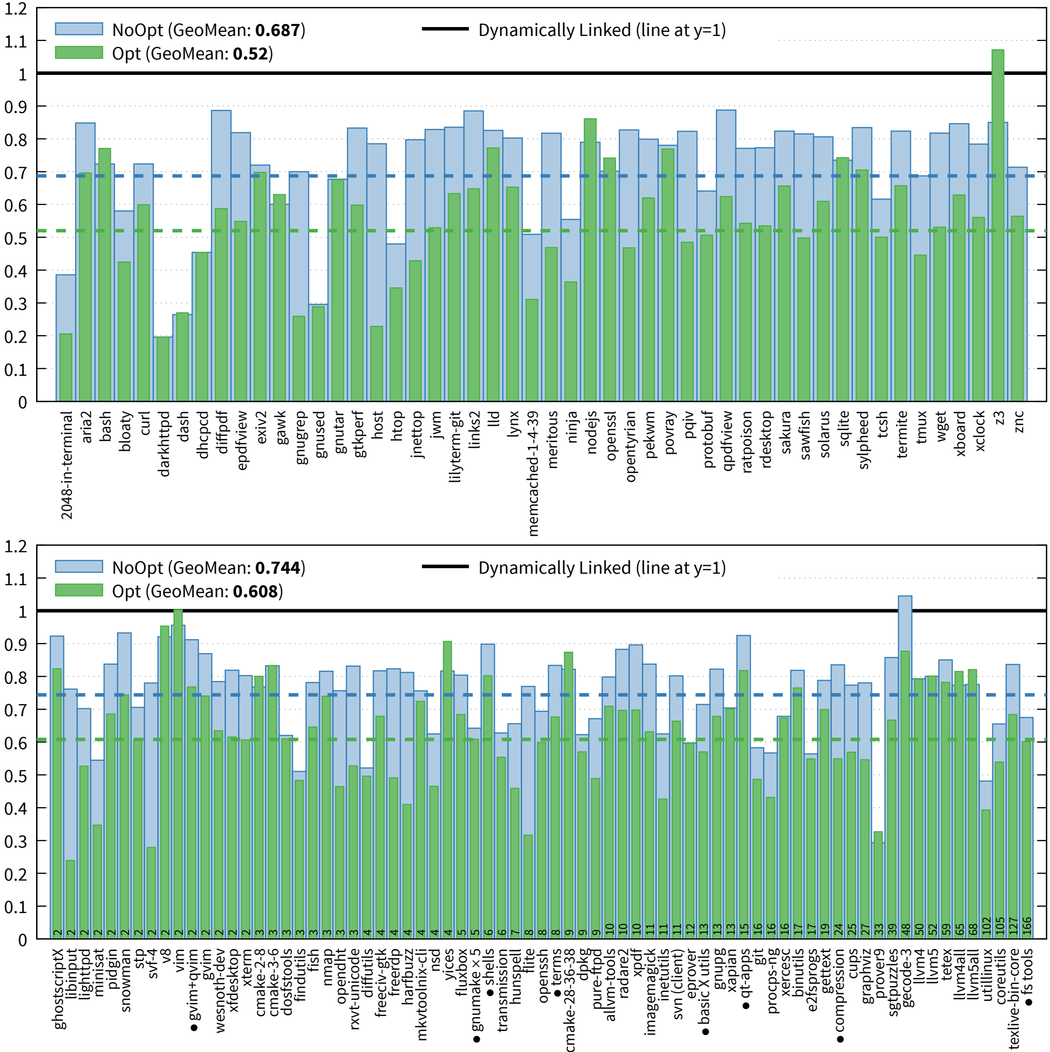

Fig. 9. Relative binary size of multiplexing vs dynamically linking, single programs (top) and multiple-program (bottom) sets. Program count shown when greater than one.

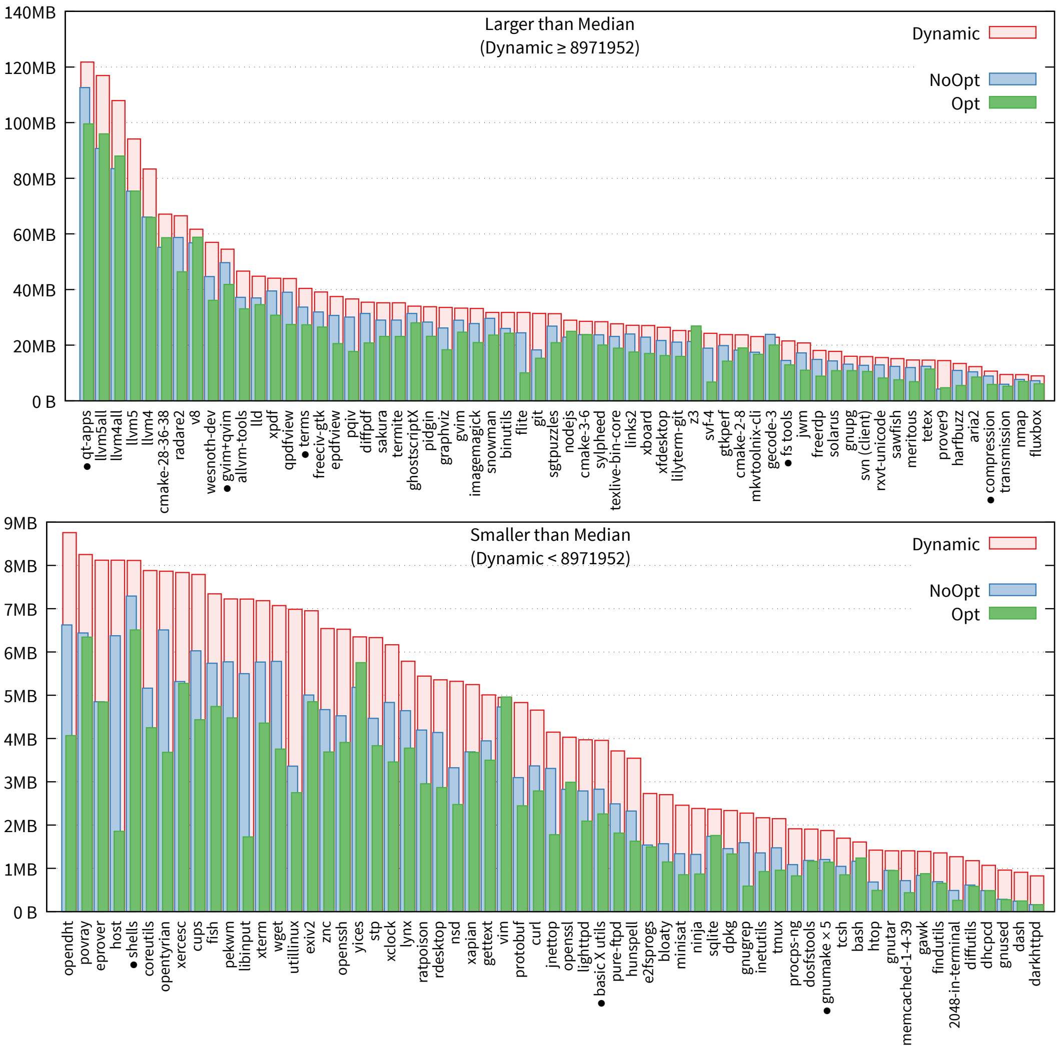

Fig. 10. Allmux vs Dynamically Linked: Absolute Binary sizes for same data shown in Figure 9. Graph partitioned at median binary size.

B 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

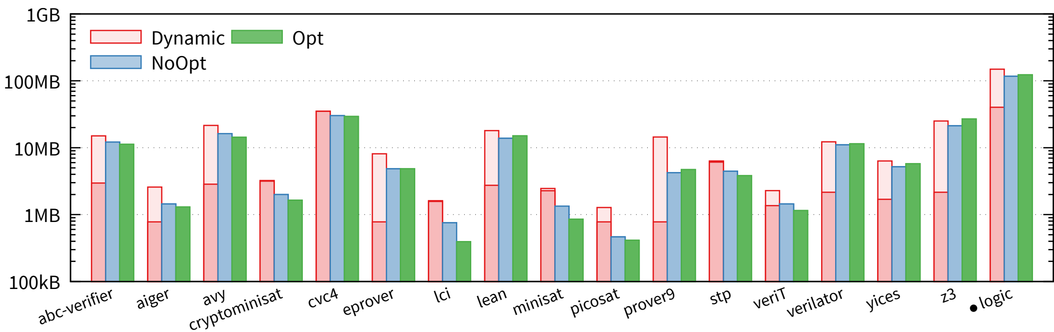

Fig. 11. Binary Sizes for Logic-related programs (**y-log**)

ACKNOWLEDGMENTS

This material is based upon work supported by the Office of Naval Research under Grant Nos. Navy N00014-4-1-0525 and Navy N00014-17-1-2996.

REFERENCES

Vikram Adve, Will Dietz, et al. 2016. ALLVM Project. http://allvm.org

Varun Agrawal, Abhiroop Dabral, Tapti Palit, Yongming Shen, and Michael Ferdman. 2015. Architectural Support for Dynamic Linking. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 691–702. https://doi.org/10.1145/2694344.2694392

Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst.* 32, 6, Article 23 (Aug. 2010), 84 pages. https://doi.org/10.1145/1749608.1749612

Christian S Collberg, John H Hartman, Sridivya Babu, and Sharath K Udupa. 2004. Slinky - Static Linking Reloaded. Retrieved 2018 from http://slinky.cs.arizona.edu/

Christian S Collberg, John H Hartman, Sridivya Babu, and Sharath K Udupa. 2005. SLINKY: Static Linking Reloaded.. In *USENIX Annual Technical Conference, General Track*. 309–322.

CryoPID 2006. CryoPID. http://freecode.com/projects/cryopid/. Original homepage is no longer available.

Bruno De Bus, Daniel Kästner, Dominique Chanet, Ludo Van Put, and Bjorn De Sutter. 2003. Post-pass Compaction Techniques. *Commun. ACM* 46, 8 (Aug. 2003), 41–46. https://doi.org/10.1145/859670.859696

Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. 2005. Link-time Binary Rewriting Techniques for Program Compaction. *ACM Trans. Program. Lang. Syst.* 27, 5 (Sept. 2005), 882–945. https://doi.org/10.1145/1086642.1086645

Bjorn De Sutter, Hans Vandierendonck, Bruno De Bus, and Koen De Bosschere. 2003. On the Side-effects of Code Abstraction. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '03)*. ACM, New York, NY, USA, 244–253. https://doi.org/10.1145/780732.780766

Will Dietz, Joshua Cranmer, Nathan Dautenhahn, and Vikram Adve. 2015. Slipstream: Automatic Interprocess Communication Optimization. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 431–443. https://www.usenix.org/conference/atc15/technical-session/presentation/dietz

Ulrich Drepper. 2011. How To Write Shared Libraries. Retrieved July 2017 from http://people.redhat.com/drepper/dsohowto.pdf

Tobias J.K. Edler von Koch, Björn Franke, Pranav Bhandarkar, and Anshuman Dasgupta. 2014. Exploiting Function Similarity for Code Size Reduction. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '14)*. ACM, New York, NY, USA, 85–94. https://doi.org/10.1145/2597809.2597811

Cristiano Giuffrida, Clin Iorgulescu, Giordano Tamburrelli, and Andrew S. Tanenbaum. 2017. Automating Live Update for Generic Server Programs. *IEEE Trans. Softw. Eng.* 43, 3 (March 2017), 207–225. https://doi.org/10.1109/TSE.2016.2584066

W. Wilson Ho, Wei-Chau Chang, and Lilian H. Leung. 1995. Optimizing the Performance of Dynamically-linked Programs. In *Proceedings of the USENIX 1995 Technical Conference Proceedings (TCON'95)*. USENIX Association, Berkeley, CA, USA, 19–19. http://dl.acm.org/citation.cfm?id=1267411.1267430

Blanchette Jasmin. 2008. *C++ GUI Programming with Qt4, 2/e*. Pearson Education India.

Jakub Jelinek. 2003. *Prelink*. Technical Report. Technical report, Red Hat, Inc., 2004. available at http://people. redhat. com/jakub/prelink. pdf.

Neil E Johnson. 2004. *Code size optimization for embedded processors*. Technical Report. University of Cambridge, Computer Laboratory.

Changhee Jung, Duk-Kyun Woo, Kanghee Kim, and Sung-Soo Lim. 2007. Performance Characterization of Prelinking and Preloading for Embedded Systems. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software (EMSOFT '07)*. ACM, New York, NY, USA, 213–220. https://doi.org/10.1145/1289927.1289961

KSM 2009. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium*. 19–28.

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. Conf. on Code Generation and Optimization*. San Jose, CA, USA, 75–88.

John R. Levine. 1999. *Linkers and Loaders* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Xiaozhu Meng and Barton P. Miller. 2016. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 24–35. https://doi.org/10.1145/2931037.2931047

MITRE Corporation. 2017. CVE-2017-7494. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7494

Matthew Moore. 2017. Distroless Docker: Containerizing Apps, Not VMs. (2017). https://swampup2017.sched.com/event/A6CW/distroless-docker-containerizing-apps-not-vms SwampUP.

Michael N. Nelson and Graham Hamilton. 1993. High Performance Dynamic Linking Through Caching. In *Proceedings of the USENIX Summer 1993 Technical Conference - Volume 1 (Usenix-stc'93)*. USENIX Association, Berkeley, CA, USA, Article 17, 14 pages. http://dl.acm.org/citation.cfm?id=1361453.1361470

openSUSE 2017. openSUSE:Shared library packaging policy. Retrieved 2018 from https://en.opensuse.org/openSUSE:Shared_library_packaging_policy

Douglas B. Orr, Jay Lepreau, J. Bonn, and R. Mecklenburg. 1993. Fast and Flexible Shared Libraries. In *Proceedings of the Summer 1993 USENIX Conference, Cincinnati, OH, USA, June 21-25, 1993*. 237–252.

PSS 2016. Proportional set size. Retrieved 2018 from https://en.wikipedia.org/wiki/Proportional_set_size

Valery Reznic. 2016. ELF STATIFIER MAIN PAGE. Retrieved 2018 from http://statifier.sourceforge.net/

Valery Reznic. 2018. Ermine: Linux Portable Application Creator. http://www.magicermine.com/.

Andreas Schwab. 2005. Re: Statically linking against a shared library. https://sourceware.org/ml/binutils/2005-03/msg00350.html Binutils mailing list.

scrut. 2003. reducebind.c - dynamic to static binary conversion utility. Retrieved 2018 from https://dl.packetstormsecurity.net/groups/teso/reducebind.c

Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 181–194. https://doi.org/10.1145/844128.844146

Hyungjo Yoon, Changwoo Min, and Young Ik Eom. 2014. 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), 1.