Guided Linking: Dynamic Linking without the Costs

SEAN BARTELL, University of Illinois at Urbana–Champaign, USA
WILL DIETZ, University of Illinois at Urbana–Champaign, USA
VIKRAM S. ADVE, University of Illinois at Urbana–Champaign, USA

Dynamic linking is extremely common in modern software systems, thanks to the flexibility and space savings it offers. However, this flexibility comes at a cost: it's impossible to perform interprocedural optimizations that involve calls to a dynamic library. The basic problem is that the run-time behavior of the dynamic linker can't be predicted at compile time, so the compiler can make no assumptions about how such calls will behave.

This paper introduces *guided linking*, a technique for optimizing dynamically linked software when some information about the dynamic linker's behavior is known in advance. The developer provides an arbitrary set of programs, libraries, and plugins to our tool, along with *constraints* that limit the possible dynamic linking behavior of the software. By taking advantage of the constraints, our tool enables *any* existing optimization to be applied across dynamic linking boundaries. For example, the NoOverride constraint can be applied to a function when the developer knows it will never be overridden with a different definition at run time; guided linking then enables the function to be inlined into its callers in other libraries. We also introduce a novel code size optimization that deduplicates identical functions even across different parts of the software set.

By applying guided linking to the Python interpreter and its dynamically loaded modules, supplying the constraint that no other programs or modules will be used, we increase speed by an average of 9%. By applying guided linking to a dynamically linked distribution of Clang and LLVM, and using the constraint that no other software will use the LLVM libraries, we can increase speed by 5% and reduce file size by 13%. If we relax the constraint to allow other software to use the LLVM libraries, we can still increase speed by 5% and reduce file size by 5%. If we use guided linking to combine 11 different versions of the Boost library, using minimal constraints, we can reduce the total library size by 57%.

CCS Concepts: • **Software and its engineering → Compilers**; *Software libraries and repositories*; *Software performance*.

Additional Key Words and Phrases: Code Deduplication, Link-Time Optimization, Dynamic Linking, Shared Libraries, Plugins, LTO, LLVM, IR

ACM Reference Format:
Sean Bartell, Will Dietz, and Vikram S. Adve. 2020. Guided Linking: Dynamic Linking without the Costs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 145 (November 2020), 29 pages. https://doi.org/10.1145/3428213

1 INTRODUCTION

1.1 Motivation

Dynamic linking is ubiquitous on modern Linux, Windows, and MacOS computers. There are three primary motivations for using it. The first is to prevent the duplication of library code that occurs with static linking. The reductions in disk and memory usage can be enormous; compared with

Authors' addresses: Sean Bartell, smbarte2@illinois.edu, University of Illinois at Urbana–Champaign, USA; Will Dietz, wdietz2@illinois.edu, University of Illinois at Urbana–Champaign, USA; Vikram S. Adve, vadve@illinois.edu, University of Illinois at Urbana–Champaign, USA.

2475-1421/2020/11-ART145
https://doi.org/10.1145/3428213

static linking, using dynamic linking in a distribution of the Clang and LLVM compiler infrastructure can reduce the total binary size from 1800 MB to 260 MB. Dynamic linking also makes it possible to upgrade a library without waiting to recompile every program that uses it; for basic libraries like OpenSSL, this makes security updates orders of magnitude faster. Finally, dynamic linking allows software to be split into modular components that can be distributed separately and loaded only when necessary. Many popular programs rely on dynamically loaded modules; for example, when Python is run interactively, it loads its `readline` module with `dlopen`.

However, dynamic linking is not without its downsides. There is significant size and speed overhead involved in storing symbol tables, resolving symbols, and calling dynamically linked functions; Agrawal et al. [2015] showed that the indirection added to function calls can make the program 4% slower. But the greatest downside of dynamic linking is that it prevents the compiler from optimizing across the boundaries of programs and dynamic libraries. There are two key problems preventing optimization:

(1) The compiler only works on a single program or dynamic library at a time, preventing any analysis of external function calls to a dynamic library. Even if the compiler supports link-time optimization, it can optimize function calls across different modules *within a single program or library*, but not across different programs and libraries.
(2) The compiler can make no assumptions about the runtime behavior of the linker, which depends on many environment settings outside the control or even knowledge of the compiler, such as `LD_PRELOAD`, `LD_LIBRARY_PATH`, and `/etc/ld.so.cache`. With no way to be certain what code will actually be dynamically linked together, the compiler is forced to take the conservative option and assume nothing about the runtime environment can be known.

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

1.2 Our Approach

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

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

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

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

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

1.3 Use Cases

Here are several specific examples to illustrate how developers could make use of guided linking. Our evaluations in Section 8 are based on each of these scenarios.

Optimizing a Docker Container. Typical Docker containers are excellent targets for guided linking because the entire set of software to run in a container is normally known in advance. In addition, containers are often built once and then deployed to hundreds of machines, multiplying the benefit of our optimizations. All programs and libraries in the container can be included in a single optimized set, and the developer can specify strong constraints based on the knowledge that no external programs will ever be used on the system. Section 8.1 shows we can increase speed significantly in such a scenario.

Distributing Clang and LLVM. Without compression, the stripped binaries for the Clang compiler, LLVM tools, and associated libraries can take up more than 200 MB of disk space. If a developer wants to redistribute Clang while minimizing the amount of space needed, they can apply guided linking to the entire set of LLVM programs and libraries. They can use strong constraints because LLVM is a self-contained project. In this situation, Section 8.2 shows we can decrease code size by 5% while also increasing compilation speed by 5%.

Merging Multiple Versions of Libraries. Some libraries, such as the Boost libraries, often need multiple versions installed at once in order to satisfy the dependencies of different programs. We can reduce code size with guided linking by combining all necessary versions of the library and using our function deduplication technique, which is effective even if compatibility concerns mean that only weak constraints can be provided. Section 8.3 demonstrates a 57% size reduction by combining 11 versions of Boost into an optimized set and applying guided linking.

1.4 Benefits and Contributions

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

There are several practical benefits of such a system:

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

Reducing code size within programs: We can reduce code size by applying compiler- and linker-based techniques across program / shared library boundaries, including both basic optimizations

like dead function elimination and interprocedural constant propagation, and advanced techniques like partial specialization.

Enabling novel optimizations across programs: When components of multiple programs are available to the compiler as a single unit, new types of optimization may become possible. In this paper, we demonstrate a new technique enabled by guided linking that can detect and merge duplicate functions, even when the functions originate from different software packages. This is particularly useful to save space for software that uses static libraries, duplicated code, and C++ template instantiations. For example, the OpenWRT Linux distribution for wireless routers runs on devices with as little as 8MB flash [OpenWRT Community 2020]; a size reduction of 1MB could make it possible to install entirely new applications.

The technical contributions of this paper are as follows:

(a) We identify a set of constraints that can usefully be placed on dynamic linking behavior in order to enable optimizations.
(b) We explain how optimizable code can be generated for each possible set of constraints, without breaking compatibility with existing software that relies on the dynamic linker.
(c) We show how our system enables techniques like function deduplication to be applied across dynamic linking boundaries.
(d) We implement our proposed system on top of the LLVM compiler infrastructure.
(e) We show experimentally (Section 8) that our tool can transparently compile large, real-world Linux software packages. As part of this demonstration, we show that we can easily identify sets of constraints that can safely be applied to real-world software. We show that our system can improve speed by more than 9% or reduce code size by more than 50% on widely used software packages, depending on the optimized set and constraints used; the code size improvement is much more than any previous compiler-based technique we are aware of.

1.5 Paper Organization

Section 2 gives a more detailed overview of the guided linking process. Section 3 explains how dynamic linking works. Sections 4 and 5 discuss the optimized set and constraints respectively. Section 6 describes the guided linking process itself, and Section 7 describes the function deduplication process. Section 8 presents our evaluation and results, Section 9 discusses related work, Section 10 discusses future work, and Section 11 concludes the paper.

2 OVERVIEW OF GUIDED LINKING

This section gives a high-level overview of Guided Linking, before describing the full details of the system in Sections 4 to 7. The full guided linking process is shown in Figure 1. It begins with the optimized set and constraints provided by the developer, continues with the guided linking tool that creates the merged library and applies function deduplication, and concludes with the existing compiler optimizations and code generator.

2.1 Developer Choices

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

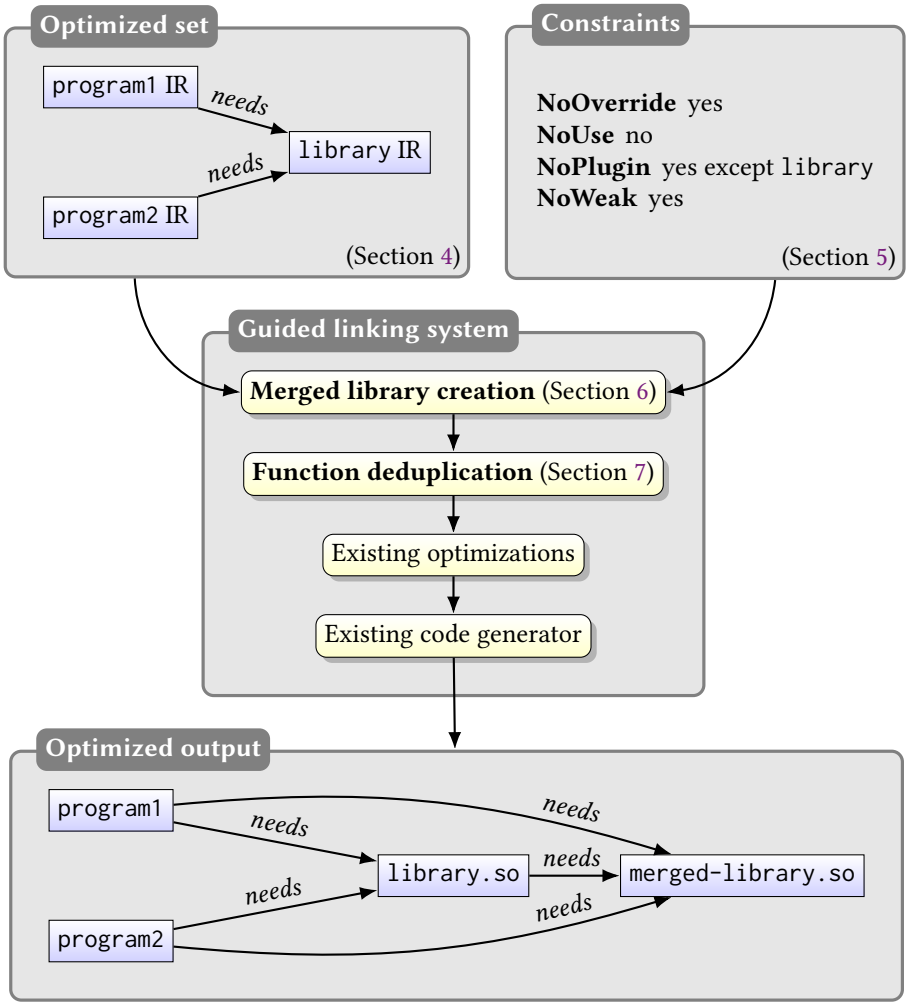


Fig. 1. The guided linking process.

Table 1. The constraints developers can choose to apply to the runtime behavior of their code.

Constraint	Usually applied to...	Enables...
NoOverride (Section 5.1)	Definitions that will not be overridden by another definition outside the optimized set.	Resolving references to these definitions statically.
NoUse (Section 5.2)	Definitions that are never used outside the optimized set, and never loaded with `dlsym`.	Making definitions private and eliminating dead definitions.
NoPlugin (Section 5.3)	Code that will never be loaded as a plugin or a dependency of one.	Moving function bodies to the merged library without extra indirection.
NoWeak (Section 5.4)	Symbols that are never used with weak references and are never defined anywhere outside the optimized set.	Moving symbol definitions to the merged library for optimization.

set of closely related libraries, an entire computer system, or any combination of components in between these extremes.

Like standard LTO, our approach requires all software in the optimized set to be compiled into the compiler's intermediate representation (IR), rather than directly into machine code. If the compiler already supports LTO, this may be as simple as adding a flag to the compiler command line. Most production compilers we know of support LTO today, including GCC, LLVM [Lattner and Adve 2004], IBM's XL family [IBM Corporation 2018], and Intel C++ Compiler [Intel Corporation 2020]. We have used LLVM to build thousands of Linux packages in IR form, mostly automatically.

Aside from providing the IR for each program and library, the developer also chooses what *constraints* to place on the runtime behavior of the system. The exact set of relevant constraints may vary depending on the dynamic linker in use. In this paper, we focus on target systems that use the ELF format, such as Linux; other systems are generally simpler [Kell et al. 2016]. In ELF-based systems, developer can choose from the possible constraints in Table 1. In many cases a constraint can be applied across all symbols in the entire optimized set, so the developer need only provide an appropriate flag to our tool. In more advanced situations, the developer can choose to apply these constraints on a symbol-by-symbol basis. For instance, they might specify that NoUse applies to all symbols except plugin entry points.

2.2 Our Optimizations

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

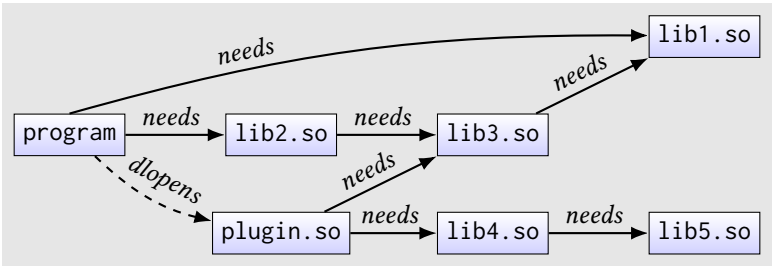
Simply by combining everything into a merged library, we enable certain optimizations even without relying on any constraints. Specifically, we demonstrate a new technique enabled by our system that can detect and merge duplicate functions, even when the functions originate from different software packages.

When constraints are provided, we can do more kinds of optimizations on top of the deduplication. We need to handle each symbol differently depending on what constraints the developer applied to it. At one extreme, if no constraints are provided, we can't assume anything. Each reference to the symbol must go through its definition in the wrapper library, which can be overridden at run time just like it could in the original library. At the other extreme, if we can assume all the constraints listed above, we may be able to move the symbol definition into the merged library entirely and even inline it into its callers in other libraries. Table 1 gives an overview of the types of optimization enabled by each constraint, and Section 5 gives the full details.

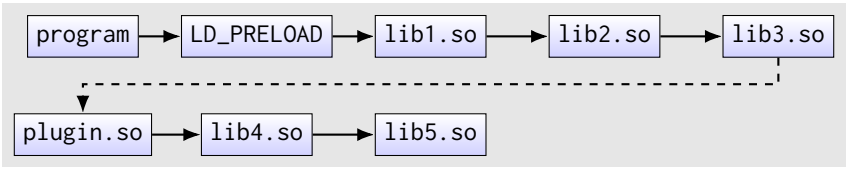
3 BACKGROUND

This section provides an overview of the dynamic linking process, as implemented on ELF-based systems such as Linux. We focus on the aspects of ELF linking behavior most relevant to guided

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



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



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

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

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

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

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

Fig. 2. A dynamic linking example.

linking; readers should consult Drepper [2011], Kell et al. [2016], and the Linux *man-pages* project [2020] for a more thorough exploration.

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

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

3.1 Producing ELF Files That Can Be Dynamically Linked

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

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

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

For dynamic references to functions, the address of the function is placed in the GOT just as for global variables. But when the function is called, an extra layer of indirection is used. The caller actually calls a stub function in the procedure linkage table (PLT), which in turn loads the correct address from the GOT and jumps to it. The PLT allows the caller to use a regular call instruction, which is generally smaller than the code that would be needed to use the GOT directly. The PLT also enables lazy binding (see below).

The relocation table must be used not only for references to symbols that are defined in another library, but **also for references to symbols that are defined in the same library**, when those definitions are publicly exported. This allows a definition of that symbol in another library to *interpose*—that is, to override—a definition in the current library. The dynamic linker's symbol resolution behavior (Section 3.3) ensures that only one exported definition of a given symbol will actually be in use. This is important for static member variables of C++ templates, for example, which should only have one active definition across the entire program and all its libraries.

3.2 Finding and Loading Dependencies

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

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

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

Figure 2a shows an example program that depends on several libraries and also dynamically loads a plugin. Figure 2b shows the order in which the program and its libraries will be loaded by the dynamic linker. Note that `lib1.so` is loaded before `lib3.so`, even though the latter depends on the former, because the dynamic linker sees `lib1.so` first. Also note that the plugin does *not* cause `lib3.so` to be loaded twice.

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

3.3 Relocating Symbol References

After loading all required libraries, the dynamic linker must resolve all external symbols used in any dynamic object to point to the correct address. For each library, it reads the relocation table, finds the address of the symbol referenced by that relocation, and stores the address into the GOT. If no definition of the symbol can be found, the linker normally aborts the program with an error; however, if the relocation is marked as *weak*, the linker instead uses the null address.

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

Given that there may be multiple libraries loaded that have definitions of the same symbol, the dynamic linker must search the libraries in the correct order. When resolving references from the program and its dependencies, it searches in the *global scope*, which consists of the program and all its dependencies in the order they were initially loaded. The first definition found is used. For the example program in Figure 2a, the global scope is shown in Figure 2c.

When a plugin is loaded with dlopen, the behavior depends on flags given to dlopen. When the RTLD_GLOBAL flag is used, the plugin and its dependencies are added to the global scope, just as if they were dependencies of the main program. However, if the RTLD_LOCAL flag is used, which is the default, a new *local scope* is created that consists of the plugin and its dependencies in breadth-first order. References made by the plugin and any newly loaded libraries are first searched in the global scope, and then, if no definition was found, in the new local scope. References made by the original program and libraries are still searched in the global scope, as before, ignoring the plugin and new libraries. Dynamic lookups made by dlsym use the same scope as the module calling dlsym. For the example program in Figure 2a, the full search order for a reference in the plugin is shown in Figure 2d.

3.4 Calling Initializers and Constructors

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

4 SELECTING PROGRAMS AND LIBRARIES TO OPTIMIZE

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

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

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

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

Whenever an upgrade is made to one of the programs and libraries included in the optimized set, the entire guided linking process must be performed again. Depending on the size of the set and the compiler optimizations used, this could take several hours, which is undesirable for libraries that must be upgraded quickly (such as cryptography libraries). These libraries can be omitted from the optimized set, so they can be upgraded the normal way by simply replacing the library file with a new version.

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

5 CONSTRAINING THE DYNAMIC LINKER

In order to optimize dynamically linked code, we make use of constraints provided by the developer that bound the possible behavior of the dynamic linker. The list of all constraints is shown in Table 1. This section will describe the constraints in detail, and give some intuition about how we can use them for optimization. Details of optimizations enabled by these constraints will be given in Section 6; in brief, we move as much code as possible into a new library called the merged library, and leave the original programs and libraries in place as wrappers that refer to code in the merged library.

By default, our tool assumes that no constraints are available. The developer can enable a constraint by providing a command-line option, such as `--no-override`, to indicate that the constraint applies to all symbols unless otherwise specified. If there are exceptions for which the constraint does not apply, the developer can list the exceptional symbol names and library names in a separate file, such as the one in Listing 1, using wildcards to match multiple symbols with one exception. In most cases, developers will need at most a few exceptions per dynamic object.

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

We will now explain each of the available constraints.

5.1 NoOverride: No External Overrides

When a program refers to an external symbol, it's usually obvious what definition of the symbol the developer *intended* to be used. Given the example on the left of Figure 3, the developer

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

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

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

Listing 1. The constraint exception list we use for Python.

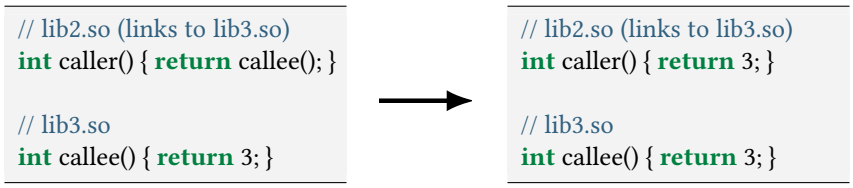


Fig. 3. An optimization that is only valid when the NoOverride constraint is applied to `callee`.

almost certainly wants `caller` to call the definition of `callee` in `lib3.so`. However, there are various ways the dynamic linker could provide a different definition of `callee`. The user could use `LD_PRELOAD` to load a library containing a different definition, or `LD_LIBRARY_PATH` could lead to a different version of `lib3.so` than expected. The program could also have its own definition of `callee` that interposes the one in `lib3.so` (see Section 3.1). The compiler can't be sure which definition of `callee` will be used, so it must optimize `lib2.so` without making any assumptions about `callee`; for example, it can't inline `callee` into `caller`.

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

When to use it. This constraint, when applied to any exported definition, specifies that the definition will never be overridden at run time by another definition outside the optimized set.[3] This is trivially true if there are *no* definitions outside the set, but it is also true if the definition inside the set overrides all others, or if the definitions are in different libraries that will never be loaded simultaneously. In the example in Figure 3, if the developer applied this constraint to the definition of `callee`, it would guarantee that this is the *only* possible definition of `callee` that could be used by `caller`.

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

Enabled optimizations. When a piece of code refers to a symbol defined in a different program or library, and the symbol has this constraint applied, we can statically determine which definition is used. This allows us to perform arbitrary interprocedural optimizations across multiple programs and libraries, including inlining as shown in Figure 3.

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

5.2 NoUse: No External Uses

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

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

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

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

5.3 NoPlugin: No Use in Plugin

As described in Section 3.3, plugins that are loaded with the `RTLD_LOCAL` flag (which is the default) add an extra complication to the symbol lookup process. When resolving a symbol used by the plugin library, or by a library that was loaded as a dependency of the plugin, the dynamic linker searches not only the global scope used by all the libraries but also a local scope specific to the plugin. This plugin-specific lookup scope makes it more difficult to move plugin code into the merged library.

As an example, consider the libraries on the left side of Figure 4, supposing these are part of the larger set in Figure 2a. When the original program is run and the dynamic linker resolves the

```
// plugin.so (links to lib4.so)
int plugin_func() {
    return lib4_func(4);
}
```

```
// plugin.so (links to lib4.so & merged.so)
int plugin_func() {
    return plugin_func_actual_body();
}

// merged.so
int plugin_func_actual_body() {
    return lib4_func(4);
}
```

Fig. 4. A transformation that is only valid when the NoPlugin constraint applies to the use of `lib4_func`.

reference to `lib4_func`, it will search through the libraries shown in Figure 2d, and eventually find the definition in `lib4.so`. However, if we attempted to move the body of `plugin_func` into `merged.so`, as shown on the right side of Figure 4, the reference to `lib4_func` might be resolved differently. In particular, if `merged.so` was loaded earlier as a dependency of the program, it will use global scope; `lib4.so` may be missing from global scope, and only present in the local scope of `plugin.so`. In that case, the reference to `lib4_func` from `merged.so` will be undefined.

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

Even when these scope conflicts are possible, our system still relies on the ability to move all code into the merged library. We can do this by adding an extra layer of indirection, described in Section 6.5. The NoPlugin constraint indicates that these conflicts are impossible, allowing us to avoid this extra indirection.

When to use it. Unlike other constraints, which apply to symbol definitions, the NoPlugin constraint applies to external references, such as the reference to `lib4_func` from `plugin.so` in Figure 4. It is normally applied to every reference in a program or library. The NoPlugin constraint should be applied to all programs, and all libraries that will never be loaded as part of a plugin. In addition, if the entire optimized set belongs to a single plugin, the NoPlugin constraint can be applied to the entire set (despite the name). This constraint can also be applied to the whole set if it can be guaranteed that no two objects in the optimized set will be loaded simultaneously, as when merging multiple versions of the same library.

Enabled optimizations. This constraint allows us to move code to the merged library without adding a potentially expensive layer of indirection, as shown in Figure 4. Moving the code is not an optimization in and of itself, but it's important to enable our other optimizations.

Constraint check. Whenever a wrapper library is loaded with this constraint applied, we check to ensure it's in the same scope as the merged library and raise an error if it is not. If the constraint were violated without this check, it would generally cause an "undefined symbol" error as symbols were looked up in the wrong scope, but could potentially cause incorrect behavior if a symbol has multiple definitions and the wrong one is used.

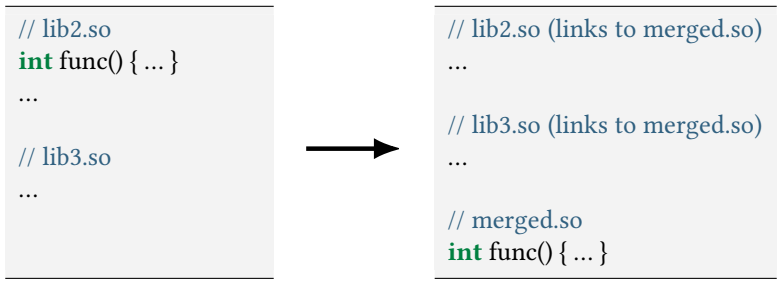


Fig. 5. A transformation that is only valid when the NoWeak constraint is applied to `func`.

5.4 NoWeak: No Weak Uses or External Definitions

Suppose the optimized set includes the libraries on the left side of Figure 5. Assuming none of the other programs and libraries being optimized exports a symbol named `func`, we might like to move the definition of `func` into `merged.so` for better optimization; because the new `lib2.so` links against `merged.so`, `func` will still be available in any program that uses `lib2.so`. However, this will create a *spurious export* of `func`—any program that links against `merged.so` (perhaps indirectly, via `lib3.so`) will see the definition of `func`, even if it never actually links against `lib2.so`.

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

(1) Some code behaves differently depending on whether or not `func` is defined. For instance, it uses a weak declaration of `func` and expects the symbol to resolve to `NULL`, or it calls `dlsym("func")` and only behaves correctly if the result is `NULL`. We refer to these uses collectively as *weak uses*. The spurious export could cause `func` to be defined when it was previously undefined, breaking the code.
(2) There is a definition of `func` in an external library or plugin, not included in the optimized set. If the external code is loaded by the dynamic linker after the merged library, its definition of `func` could be interposed by the spurious export, causing the wrong `func` to be used.

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

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

Enabled optimizations. This constraint allows us to move the definition of an exported symbol into another library (specifically, the merged library), as demonstrated in Figure 5. This allows extra interprocedural optimizations to be applied when other programs and libraries refer to the symbol.

Constraint check. We need to check every external program and library that gets loaded and raise an error if any of them refer to a NoWeak symbol. We also need a check in `dlsym` to determine if the result of the call would be different without the constraint.

5.5 Special-Purpose Constraints

There are a few special constraints needed for specific symbols. These constraints are applied automatically by the guided linking tool, and are not expected to be used manually by developers.

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

There is also an "*always defined elsewhere*" constraint, which should be used for symbols weakly defined in static libraries that cannot be compiled to IR. We automatically apply this constraint to all symbols defined in the static parts of the C standard library and C++ standard library. In other cases, developers should either use dynamic libraries or compile their static libraries into IR form, rather than linking in unoptimized static libraries.

There are also certain functions, specifically `dlopen` and `dlsym`, which can behave differently depending on which dynamic library they are called from (Sections 3.2 and 3.3). We add a special constraint to these functions which ensures they are always called from the correct library.

6 OPTIMIZING DYNAMICALLY LINKED CODE

This section explains how our system actually optimizes a set of modules (programs and libraries), given the constraints provided by the developer. As noted in Section 2, we move as much code as possible into a new *merged library*. We replace each of the original programs and libraries with a *wrapper program* or *wrapper library* which contains a minimal set of symbol definitions, and links against the merged library to access the rest of the code.

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

In order to move as much code as possible into the merged library, we separate each function into a *body function*, which contains the actual code for the function, and a *stub function* which simply jumps to the body function. All other references to the function use the stub function. An example is shown in Figure 4. The body function can be moved into the merged library even when the public definition in the stub function cannot. This extra indirection could add overhead to the program, but Section 6.4 explains how the stub functions can be optimized away in many cases. The separation between body functions and stub functions is especially useful because the body functions can be deduplicated, as described in Section 7.

The rest of this section explains in detail the process of creating the merged library, wrapper programs, and wrapper libraries. We expect the same overall process to work for any compiler that supports LTO and dynamic linking, although the details may differ.

6.1 Resolving Symbols

The first step is to attempt to resolve all dynamic references made in the input modules. We would like to resolve references statically, so that we can optimize them, but some references must be left dynamic to ensure correctness. Specifically, we use the following algorithm for each reference:

(1) If there is a definition in the same module as the use, and LLVM can determine based on the linkage information in the module that this definition is the exact one that will be used at runtime, we resolve to it. This happens particularly for references to private symbols.
(2) If we can look through the dependencies of the module containing the use within the optimized set and determine which definition is used, and the NoOverride constraint applies to this definition, we resolve to it.[4]
(3) Otherwise, the reference will be resolved at run time, by the dynamic linker, just as it would be in the original code.

6.2 Choosing Where to Define Symbols

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

(1) The symbol is one of the few "unmovable" symbols, such as `main` (Section 5.5). The stub function must be kept in the wrapper library.
(2) The symbol is exported publicly by more than one of the modules being merged. We must keep the stub functions in their respective wrapper libraries and let the dynamic linker decide which definition to use.
(3) The symbol is exported publicly, but one of the modules being merged has a reference to a symbol with the same name which could not be resolved statically. We must keep the stub function in the wrapper library and let the dynamic linker decide whether the reference should be resolved to our definition or to an external definition.
(4) The symbol is exported publicly, the symbol may be used dynamically (Section 5.2), and it may have weak uses or external definitions (Section 5.4). That is, neither the NoUse nor NoWeak constraints apply. We must keep the stub function in the wrapper library, because defining it in the merged library might break other programs that attempt to use it.

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

(5) When a global variable initializer uses a reference to a symbol that may need to be resolved in local scope (Section 5.3), and the reference cannot be replaced with a static reference, we need to define the global variable in the wrapper library. For more details, see Section 6.5.
(6) Sometimes there are limitations of the system's compiler or linker that require two symbols to be placed in the same module.[5] In these cases, if one of the symbols is kept in the wrapper library, we must do the same with the other symbol.

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

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

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

6.3 Merging and Linking

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

For references that were statically resolved, we update the reference to make sure it points to the corresponding definition. In certain cases we may move a private symbol to the merged library but leave its callers/users in a wrapper library, or vice versa. In these cases we export the formerly-private symbol so that the other library can reference it, giving it a new name such as `__merged_func.0` to avoid conflicts.

At this point, the merged library contains code that came from various different programs and libraries, each of which may still have dynamic references to other symbols. In any given execution, only a subset of the programs and libraries may actually have their code used; for the others, there's no guarantee their dynamic references will actually have a definition available. We need to prevent these missing references from causing a dynamic linker error. Therefore, we make all the references *weak*, so if no definition is found, the reference is simply resolved to NULL rather than causing an error.[6] If necessary, we can check at run time whether any of these references would have caused an "undefined symbol" error in the original code, and raise a corresponding error ourselves.

Finally, for each symbol publicly exported by the merged library which has the NoUse constraint (Section 5.2), and which is not used by any of the wrapper libraries, we make the symbol private.

The merged library, wrapper programs, and wrapper libraries can now be optimized, compiled, and executed using an existing LTO implementation. Because almost all code from the original dynamic modules has been moved to the merged library, the LTO implementation can optimize across the boundaries between the modules just as if they had originally belonged to a single library (albeit with the extra indirection we add depending on the constraints). In addition to standard LTO optimizations, we also apply a code deduplication technique described in Section 7, which can cross these boundaries thanks to the merging process.

6.4 Additional Optimizations

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

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

[6] As an exception, symbols with the "always defined elsewhere" constraint (Section 5.5) are not made weak, to ensure that the external definition is always used.

6.5 Handling Local Scope

We have a problem in cases where a body function without the NoPlugin constraint refers to an external symbol—the symbol may need to be resolved using local scope (Sections 3.3 and 5.3). We put all body functions in the merged library, but the external symbol can only be resolved correctly if it is loaded from the wrapper library. We solve this problem by adding a layer of indirection to these references. We add a constructor to the wrapper library that runs when the library is loaded, taking the address of each such external symbol and storing it in a global variable defined in the merged library. Then, whenever a body function in the merged library needs to use one of these external symbols, we modify it to load the correct address from the new global variable.

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

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

7 FUNCTION DEDUPLICATION

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

Our function deduplication optimization is integrated into the optimization system described in Section 6. Before we link a body function into the merged library, we check whether an equivalent function has already been linked in, as described in Section 7.1. If there is one, we reuse the existing function rather than adding a new copy to the merged library. If many objects are added that consist mostly of duplicate functions, such as many slightly different versions of the same program, the merged library will grow only slowly because of this deduplication. Various additional checks are needed to make sure deduplication is a legal operation, and full details are described in Section 7.2.

7.1 Duplicate Function Detection

Code equivalence checking has a rich literature, with sophisticated techniques that can verify equivalence of complex code [Churchill et al. 2019; Dahiya and Bansal 2017; Lim and Nagarakatte 2019]. Our initial prototype only uses a simple check for exact syntactic equivalence, which is sufficient to get very good results in some cases (see Section 8.3). Whenever a body function is linked into the merged library, we try to find any equivalent functions that are already present. We first extract the body function into a separate piece of IR, and then calculate a 256-bit cryptographic hash of the exact representation of the IR. The function's hash is then compared against the hashes for functions that have already been linked into the merged library, and if a match is found, we have detected a duplicate function. With a 256-bit hash, the risk of a hash collision is astronomically low; just in case, we can perform a bytewise comparison to verify that the functions are identical.

This hashing comparison technique can be easily confused by incidental differences between functions, even if they are semantically equivalent. We use a few normalization transformations

to reduce these spurious differences. First, identical constant strings may be assigned different names in different modules for a variety of reasons, causing functions that use those strings to produce different hashes. We rename all constant strings using a hash of their contents to ensure that identical strings get identical names. Second, many functions that use pointers to struct types do not actually dereference the pointer, so the details of the struct type are irrelevant. We replace such types with pointer-to-void (effectively making the struct type opaque), which means that functions that only operate on the pointer and not the target can be detected as equivalent. Third, we remove all struct type names because type names appear in the IR for readability, but are ignored by compiler passes (LLVM uses structural type equivalence to unify identical struct types). Finally, we delete the name of the function itself from the module, so that two functions with different names can still be detected as equivalent.

7.2 Duplicate Function Merging

When we create the merged library, in addition to the process described in Section 6.3, we also merge duplicate functions when possible. Whenever duplicate functions are detected, the basic technique is to create only *one* copy of the body function, and make all stub functions call the merged body function.[7]

However, it's possible for merging two body functions to be incorrect, even if they have the same hash value. That's because the hash uses the *names* of external symbols used by the function, without considering what the names will resolve to. For example, given the following input:

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

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

Both versions of `caller` will have the same hash value, but they can't be merged because they need to call different definitions of `callee`.

In order to detect such cases, our tool has an extra phase after symbol resolution is performed (Section 6.1), in which we find functions that have identical hash values, check whether the external symbols in all copies of the function have been resolved to the same definitions, and if so, merge the bodies of the functions.

8 EVALUATION

In order to evaluate our proposed system, we have implemented a prototype based on the LLVM 10.0 compiler [Lattner and Adve 2004]. Our prototype implements the full optimization system described in Section 6, along with the function deduplication technique described in Section 7, with certain exceptions. First, the special behavior for `dlopen` and `dlsym` described in Section 5.5 is not yet implemented. Second, the constraint checks have not yet been implemented, so if an invalid constraint is specified, the optimized program may crash unexpectedly or exhibit incorrect behavior. Note that the constraint checks would only be run when a new library is loaded or when `dlsym` is called, so we expect their impact on program speed to be very limited.

We obtained LLVM IR for the test programs and libraries by compiling them with the Clang compiler and using the `-fembed-bitcode` option. All file size evaluations were performed after applying the `strip --strip-unneeded` command to reduce file size. All performance evaluations were run on a machine with two 24-core Xeon Platinum 8259CL CPUs, NixOS Linux version

[7] We can't safely merge stub functions generated for duplicate functions—the program may compare the addresses of two different functions and expect them to be different, even if the functions have identical bodies. But all references to the function refer to the stub function, including recursive calls, so it's safe to merge the body functions.

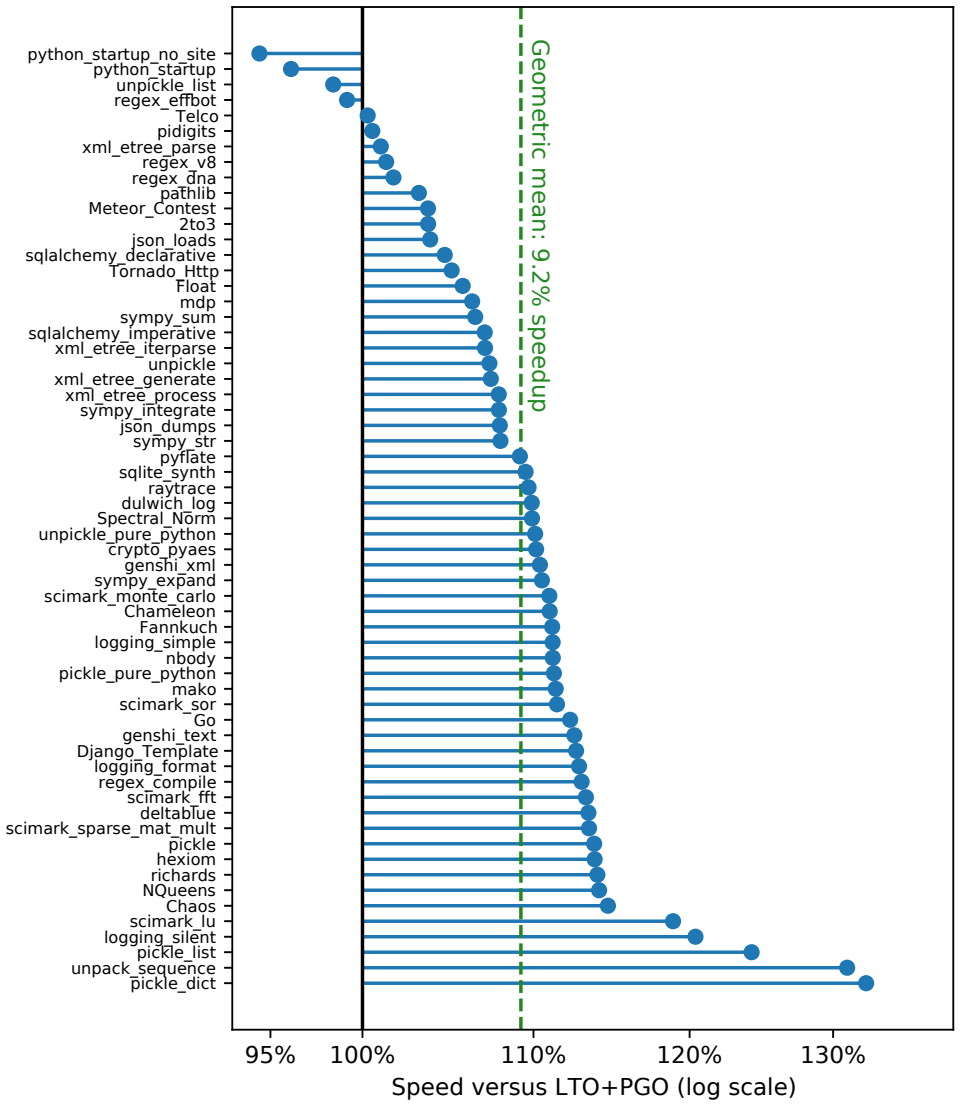


Fig. 6. Speed of each of the 61 `pyperformance` benchmarks with our optimized build of Python, relative to standard LTO+PGO (higher is better).

20.09pre239228.bd0e645f024, and pyperf version 2.0.0 [Python Software Foundation 2020b]. In order to reduce variability, we disabled simultaneous multithreading and used the system settings recommended by pyperf.[8]

8.1 Optimizing Python in a Closed System

Our system can achieve the maximum performance improvement in cases where the entire set of program and libraries on the machine are optimized together. This is possible on closed systems,

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

such as Docker containers, which do not have new software installed after they are originally created. We evaluate the performance of our system in this case by optimizing Python, and assuming that no external programs will use any of the Python libraries.

We perform guided linking on the Python interpreter itself, its dynamically loadable plugins (including several external plugins used by our benchmarks), and several libraries the plugins depend on. Because we know there are no external programs left out of the optimized set, we can apply all four of our constraints to everything being optimized, with certain exceptions needed for plugins to work as given in Listing 1.[9] We compiled the merged library using Clang with the `-O3` option to maximize performance; for our baseline, LTO is applied separately to each library and plugin with Clang `-O3`. We also use profile-guided optimization (PGO) for both our optimized version and the baseline;[10] this is important because guided linking introduces many new inlining possibilities, and PGO helps to determine which possibilities are profitable.

We evaluate our optimized version of Python on version 1.0.2 of `pyperformance`, the Python Performance Benchmark Suite [Python Software Foundation 2020a], which was developed to test the performance of the Python interpreter.[11] Results are shown in Figure 6. The geometric mean speedup across all benchmarks is 9.2% compared to the LTO+PGO baseline.

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

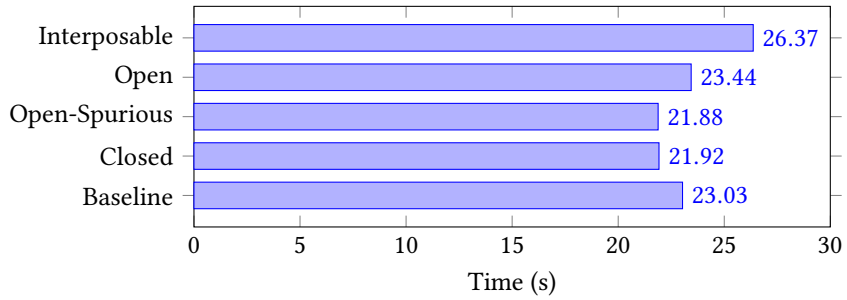
Guided linking achieves these improvements by enabling a variety of LLVM optimizations. Not only can library functions be inlined into a plugin or a program, but inlining can be performed within a single library that was impossible before. There are many different optimizations in play depending on the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

- The main Python library defines several public functions, such as `_PyObject_GetMethod`, that are only used once by the library. If LLVM inlined this code, it would normally have to leave the original functions in place in case another library calls them, creating duplicate code. Even with LTO and PGO enabled, LLVM's heuristics decide not to inline the functions because duplicating the code is too expensive. Guided linking determines that no other code will call these functions and makes them private to the merged library, enabling LLVM to inline them and delete the originals.
- The main Python library has a global variable `PyFrame_Type` containing a pointer to function `frame_dealloc()`. With a normal LTO build, LLVM must conservatively assume that some other library might modify the variable. Guided linking makes the variable private to the merged library, so LLVM can determine that `PyFrame_Type` is never modified, allowing it to inline `frame_dealloc()` into code that calls the function pointer.
- Functions in one library can be inlined into functions in another, even if the second library is a dynamically loaded plugin. For example, in benchmark `pickle_dict`, most of the execution time is spent in the `_pickle.so` plugin's `save()` function, which makes many calls to the main Python library's `PyDict_Next()` function. Guided linking moves both of these functions into the merged library and statically resolves the call, allowing LLVM to inline it.

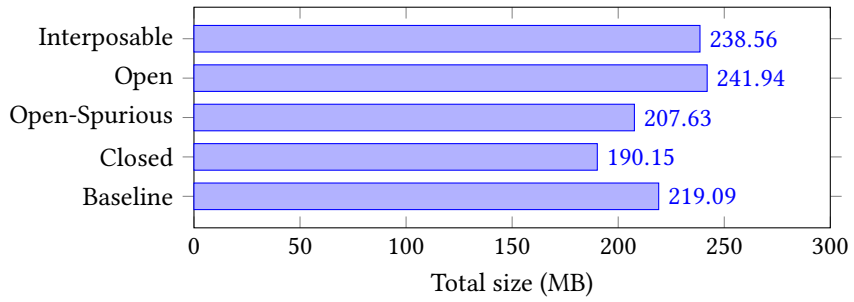
[9] Note that the NoWeak constraint only affects the few symbols given in Listing 1 under `[use]`. Removing NoWeak affects the speed by less than 0.5%.

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

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



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



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

Fig. 7. Evaluation results for PGO-optimized versions of Clang.

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

8.2 Optimizing Clang and LLVM

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

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

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

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

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

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

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

After building each version of the set, we evaluate its performance by running the optimized version of the `clang` compiler to compile and link the SQLite library with the `-O3` option. The `clang` compiler program will invoke our optimized version of `llvm-ld` in order to link the library.[13]

Results are shown in Figure 7. Compared with the baseline, our Closed configuration of Clang is 5.1% faster and 13.2% smaller. Our Open-Spurious configuration, which allows external programs to use the LLVM libraries, is 5.3% faster than the baseline and still 5.2% smaller, despite the fact that symbols cannot be made private. (While it may seem surprising that "open-spurious" is faster than "closed", the difference is very small and may be caused by variation in code layout.) PGO and guided linking do not combine as well as they did in the Python experiments; the Closed configuration without PGO is 5.5% faster than the baseline without PGO.

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

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

8.3 Merging Multiple Versions of the Same Library

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

We evaluate these space savings on `boost` and `protobuf`, using guided linking to optimize together several different versions of each library. We enable the NoPlugin constraint; we only expect one version of the library to be used in any given dynamic execution, so plugin scope conflicts are impossible. We do not enable any other constraints, because we want to ensure maximum compatibility with any possible users of the library.

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

Table 2 shows the results. We optimized the merged library with Clang's `-Oz` option to minimize code size. The baseline is the total size of all versions when each library is optimized separately at link time with Clang's `-Oz` option. We get a very significant 31% size reduction when merging 8 versions of `protobuf` together, and an even greater 57% size reduction when merging 11 versions of `boost`. Note that these reductions come from the ability to deduplicate identical functions between multiple versions of the library, as described in Section 7.

9 RELATED WORK

9.1 Expanding Domains of Optimization

Over the history of compiler optimizations, the scope of code transformed by a single optimization has been growing larger and larger. The earliest optimizing compilers performed only local optimizations, which consider only a small part of a function. In the 70s and 80s, compilers began to use "global" optimizations, which worked on an entire function at a time [Auslander and Hopkins 1982]. This was soon followed by work on interprocedural optimization, particularly at Rice University [Cooper et al. 1986]. These optimizations are now standard in all common compilers. In the 90s, link-time optimization (then called "cross-module optimization") was becoming more common [Srivastava and Wall 1992], allowing optimizations to cross the boundary between source code files. Guided linking is a natural extension to this history, allowing optimizations to expand further and cross the boundary between programs and shared libraries, and even between multiple distinct programs. Our system is the first we know of that can perform arbitrary optimizations across this barrier, without breaking compatibility with existing dynamically linked code.

9.2 Optimizing Dynamically Linked Code

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

Reducing Dynamic Linking Overhead. Much of the existing work on optimizing dynamically linked code focuses on speeding up the dynamic linking process itself. This is generally done by caching the results of the relocations done by the dynamic linker [Jelínek 2004; Jung et al. 2007; Nelson and Hamilton 1993; Orr et al. 1993]. Several tools go as far as taking a snapshot of the address space after relocations have been processed, then saving the snapshot as a new executable file, which can be run without any further relocation [Reznic 2016, 2018; scut 2003]. These techniques do not enable any compiler optimizations across dynamic linking boundaries. Guided linking can sometimes replace dynamic relocations with static references; in other cases, caching can be combined with our system for an additional performance gain.

Table 2. File size reduction when combining multiple versions of the same library.

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

Dead Code Elimination. A number of tools can debloat shared libraries by analyzing the programs that use them and removing unneeded code [Agadakos et al. 2019; Davidsson et al. 2019; Mulliner and Neugschwandtner 2015; Ziegler et al. 2019]. The primary goal is to improve security by reducing the attack surface, with code size reduction as a secondary goal. In general, these tools use a custom-built analysis to construct a call graph that encompasses multiple libraries and programs, which is then used to determine which code may be safely eliminated. Some of these tools work directly on binary code, which allows them to debloat libraries that our guided linking tool does not support (such as `glibc`, which cannot be built in the form of LLVM IR).

The biggest limitation of these tools is that they cannot reuse existing implementations of compiler analyses and optimizations. Call graph construction and dead code elimination are already implemented in standard compilers, but the debloating tools have to introduce entirely new implementations that are explicitly aware of dynamic linking. In contrast, guided linking allows the compiler's existing analysis and optimization code to be reused without modification. This benefit extends to all possible optimizations that would be worth performing across dynamic linking boundaries; for example, guided linking can easily reuse the compiler's support for inlining, whereas extending these other tools to support inlining would require an entirely new implementation with explicit awareness of dynamic linking.

Another major limitation of these tools is that they do not explicitly specify what assumptions they make about the runtime configuration of the dynamic linker. For instance, none of them explain how they interact with `LD_PRELOAD`. Most of them also have limited support for dynamically loading symbols with `dlsym`; however, Davidsson et al. [2019] use a developer-provided whitelist to indicate which symbols are needed by `dlsym`, which is very similar to using our NoUse constraint with an exception list. Finally, unlike guided linking, these tools do not allow optimizing libraries when not all of the programs that use them are known in advance.

Arbitrary Optimizations. The most aggressive previous technique to optimize across the dynamic linking boundary is the Allmux tool by Dietz and Adve [2018]. Allmux takes the compiler IR for a set of programs and all the dynamic libraries they depend on, and statically links the IR together into a single module, including only one copy of each library. The module can be optimized using arbitrary existing optimizations, including inlining and dead code elimination. Then the module is compiled into a single *multicall program*, which can be invoked to run any of the programs that were linked in. Allmux enables arbitrary compiler optimizations and completely eliminates dynamic linking overhead, but it has severe limitations which make it difficult to use in practice:

(1) Allmux requires the entire set of libraries to be optimized together; unlike our system, no libraries can be left out of optimization.
(2) Unlike our system, Allmux prohibits external programs from using the libraries that are optimized together.
(3) Unlike our system, Allmux prohibits programs and libraries from using `dlopen` and `dlsym`.
(4) Allmux uses a static linker to combine programs and libraries that were designed to be dynamically linked, which can lead to incorrect behavior. For instance, when combining two libraries that define the same symbol, Allmux will refuse to run (if both definitions are strong) or arbitrarily delete one definition (if both definitions are weak). Guided linking handles these cases by falling back to the dynamic linker.

9.3 Optimizing Using Constraints

Our system is the first we know of that makes use of developer-provided constraints on dynamic linking behavior, but other optimization systems have used other kinds of constraints on run-time behavior. For example, when a program's configuration or input data is known at compile time,

it can be partially evaluated: code that depends only on known inputs is evaluated in advance, which may cause other code to become dead. Malecha et al. [2015] attained some improvements by partially evaluating programs using standard compiler optimizations already implemented in LLVM. Sharif et al. [2018] achieved better results by using more aggressive versions of certain optimizations, such as loop unrolling and interprocedural constant propagation. These tools can be a valuable way to easily customize applications for specific end-user scenarios. Partial evaluation takes advantage of optimization opportunities that are essentially orthogonal to the ones used by guided linking; in fact, guided linking could make partial evaluation more effective by allowing it to cross dynamic library boundaries.

9.4 Deduplicating Code

Many of the most effective and widely used techniques for reducing code size rely on code deduplication. There are a number of compiler- or linker-based techniques that eliminate duplication within a single program or library. At a coarse-grained level, equivalent functions can be identified and merged into a single function; techniques have been developed to verify equivalence of increasingly complex code sequences [Churchill et al. 2019; Dahiya and Bansal 2017; Lim and Nagarakatte 2019]. At a fine-grained level, outlining (also known as procedural abstraction) involves finding duplicate pieces of code and factoring them out into a new function, replacing all the duplicates with calls to the new function so only one copy of the code remains [Cooper and McIntosh 1999; Debray et al. 2000; Fraser et al. 1984; Komondoor and Horwitz 2001; Standish et al. 1976]. Link-time techniques can compact both code and data, achieving substantial size reductions for individual binaries [De Sutter et al. 2005, 2001]. Another approach is to find functions containing similar code sequences and merge them together, deduplicating those sequences in the process [Rocha et al. 2019; von Koch et al. 2014]. It would be straightforward to apply any of these techniques to the merged library created by guided linking, enabling them to deduplicate code across multiple programs and libraries.

Shared libraries are a practical way to reduce code duplication across independently developed software packages [Levine 1999]. They can ensure only one copy of the code is transferred over the network, stored on disk, and loaded in RAM. Shared libraries are nearly universally used today, but they have some key drawbacks [Agrawal et al. 2015; Collberg et al. 2005; Levine 1999; Orr et al. 1993]: compiler optimizations cannot cross dynamic linking boundaries, and shared libraries increase both startup costs (because the libraries must be linked at run time) and execution costs (because of indirect function dispatch). Guided linking is designed to reduce these costs.

An alternative way to apply deduplication across different programs is to detect identical memory pages in virtual memory and merge them so they occupy the same space in physical memory [Arcangeli et al. 2009; Waldspurger 2002]. However, these techniques can only deduplicate code at the granularity of whole pages. Slinky [Collberg et al. 2005] is a system based on this idea that allows static linking to be used with no more performance overhead than dynamic linking, and greatly reduces the storage and bandwidth costs of static linking; however, it still incurs a 20% storage space increase and a 34% network bandwidth increase over shared libraries.

10 FUTURE WORK

Security Implications. Guided linking is likely to have an effect on the security of the optimized software, particularly ASLR and return-oriented programming, because of the way it rearranges code among different libraries. Further research is needed to study this effect. There will also be new security features enabled by guided linking; for example, when we optimize a program together with its plugins, we can add a run-time check to the program to ensure only the intended plugins can be loaded, preventing certain types of attacks.

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

Replacing the Dynamic Linker. Our current system relies on the standard dynamic linker to perform relocations and symbol resolution at run time. In some cases, such as with scope conflicts as described in Section 6.5, we need to add extra layers of indirection in order to work with the dynamic linker. We also need to check each symbol ourselves to determine whether any constraints are violated. Future research should explore whether we can reduce this indirection by replacing the dynamic linker, either in whole or in part; special care will be needed to maintain compatibility with external programs and libraries.

Equivalent Function Detection. Our current system uses a very simple syntactic comparison to detect duplicate functions. It would be straightforward to incorporate more sophisticated function equivalence detection techniques, such as the one by Churchill et al. [2019]. If these techniques were added to our tool, the rest of the system would continue to work as-is but with improved deduplication.

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

11 CONCLUSION

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

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

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1564274, and by the Office of Naval Research under Grant No. N00014-17-1-2996. We thank Hashim Sharif, Abdul Rafae Noor, Sibin Mohan, and the anonymous reviewers for their feedback.

REFERENCES

Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: Debloating Binary Shared Libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC '19)*. ACM, New York, NY, USA, 70–83. https://doi.org/10.1145/3359789.3359823

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

Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proc. Linux Symp.* (2009). 19–28. https://www.kernel.org/doc/ols/2009/ols2009-pages-19-28.pdf

Marc Auslander and Martin Hopkins. 1982. An Overview of the PL.8 Compiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN '82)*. ACM, New York, NY, USA, 22–31. https://doi.org/10.1145/872726.806977

Berkeley R. Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 1027–1040. https://doi.org/10.1145/3314221.3314596

Christian S. Collberg, John H. Hartman, Sridivya Babu, and Sharath K. Udupa. 2005. SLINKY: Static Linking Reloaded. In *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC '05)*. 309–322. https://www.usenix.org/conference/2005-usenix-annual-technical-conference/slinky-static-linking-reloaded

Keith D. Cooper, Ken Kennedy, and Linda Torczon. 1986. The Impact of Interprocedural Analysis and Optimization in the $\mathbf{R}^{\mathrm{n}}$ Programming Environment. *ACM Trans. Program. Lang. Syst.* 8, 4 (Oct. 1986), 491–523. https://doi.org/10.1145/6465.6489

Keith D. Cooper and Nathaniel McIntosh. 1999. Enhanced Code Compression for Embedded RISC Processors. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99)*. ACM, New York, NY, USA, 139–149. https://doi.org/10.1145/301618.301655

Manjeet Dahiya and Sorav Bansal. 2017. Black-Box Equivalence Checking Across Compiler Optimizations. In *Proceedings of the 15th Asian Symposium on Programming Languages and Systems (APLAS 2017) (Lecture Notes in Computer Science, Vol. 10695)*. Springer, Cham, 127–147. https://doi.org/10.1007/978-3-319-71237-6_7

Nicolai Davidsson, Andre Pawlowski, and Thorsten Holz. 2019. Towards Automated Application-Specific Software Stacks. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS 2019) (Lecture Notes in Computer Science, Vol. 11736)*. Springer, Cham, 88–109. https://doi.org/10.1007/978-3-030-29962-0_5

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, Bruno De Bus, Koen De Bosschere, and Saumya Debray. 2001. Combining Global Code and Data Compaction. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES '01)*. ACM, New York, NY, USA, 29–38. https://doi.org/10.1145/384197.384204

Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler Techniques for Code Compaction. *ACM Trans. Program. Lang. Syst.* 22, 2 (March 2000), 378–415. https://doi.org/10.1145/349214.349233

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

Ulrich Drepper. 2011. *How To Write Shared Libraries.* Retrieved Oct. 2020 from https://akkadia.org/drepper/dsohowto.pdf

Christopher W. Fraser, Eugene W. Myers, and Alan L. Wendt. 1984. Analyzing and Compressing Assembly Code. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction (SIGPLAN '84)*. ACM, New York, NY, USA, 117–121. https://doi.org/10.1145/502874.502886

IBM Corporation. 2018. Code optimization with the IBM XL compilers on Power architectures. Retrieved Oct. 2020 from http://www-01.ibm.com/support/docview.wss?uid=swg27005174&aid=1

Intel Corporation. 2020. Intel® C++ Compiler 19.1 Developer Guide and Reference. Retrieved Oct. 2020 from https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/interprocedural-optimization-ipo.html

Jakub Jelínek. 2004. *Prelink.* Technical Report. Red Hat, Inc. https://people.redhat.com/jakub/prelink.pdf

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 & IEEE International Conference on Embedded Software (EMSOFT '07)*. ACM, New York, NY, USA, 213–220. https://doi.org/10.1145/1289927.1289961

Stephen Kell, Dominic P. Mulligan, and Peter Sewell. 2016. 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 (OOPSLA 2016)*. ACM, New York, NY, USA, 607–623. https://doi.org/10.1145/2983990.2983996

Raghavan Komondoor and Susan Horwitz. 2001. Using Slicing to Identify Duplication in Source Code. In *Proceedings of the 8th International Symposium on Static Analysis (SAS 2001) (Lecture Notes in Computer Science, Vol. 2126)*. Springer, Berlin, Heidelberg, 40–56. https://doi.org/10.1007/3-540-47764-0_3

Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)*. 75–86. https://doi.org/10.1109/CGO.2004.1281665

John R. Levine. 1999. *Linkers and Loaders* (1st ed.). Morgan Kaufmann, San Francisco, CA, USA. https://linker.iecc.com/

Jay P. Lim and Santosh Nagarakatte. 2019. Automatic Equivalence Checking for Assembly Implementations of Cryptography Libraries. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE, 37–49. https://doi.org/10.1109/CGO.2019.8661180

Linux *man-pages* project 2020. *Linux Programmer's Manual: ld.so(8)*. Linux *man-pages* project. Retrieved Oct. 2020 from https://man7.org/linux/man-pages/man8/ld.so.8.html

Gregory Malecha, Ashish Gehani, and Natarajan Shankar. 2015. Automated Software Winnowing. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15)*. ACM, New York, NY, USA, 1504–1511. https://doi.org/10.1145/2695664.2695751

Collin R. Mulliner and Matthias Neugschwandtner. 2015. CodeFreeze: Breaking Payloads with Runtime Code Stripping and Image Freezing. In *Black Hat USA*. https://www.mulliner.org/security/codefreeze/

Michael N. Nelson and Graham Hamilton. 1993. High Performance Dynamic Linking Through Caching. In *Proceedings of the USENIX Summer 1993 Technical Conference (USENIX ATC)*. https://www.usenix.org/legacy/publications/library/proceedings/cinci93/nelson.html

OpenWRT Community. 2020. 4/32 Warning. https://openwrt.org/supported_devices/432_warning

Douglas B. Orr, John Bonn, Jay Lepreau, and Robert Mecklenburg. 1993. Fast and Flexible Shared Libraries. In *Proceedings of the Summer 1993 Usenix Conference*. 237–252. https://www.cs.utah.edu/flux/papers/shlibs.html

Python Software Foundation. 2020a. *The Python Performance Benchmark Suite*. https://pyperformance.readthedocs.io/

Python Software Foundation. 2020b. *Python pyperf module*. https://pyperf.readthedocs.io/en/latest/

Valery Reznic. 2016. *Statifier*. Retrieved 2018 from http://statifier.sourceforge.net/

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

Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2019. Function Merging by Sequence Alignment. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE, 149–163. https://doi.org/10.1109/CGO.2019.8661174

scut. 2003. reducebind.c. Retrieved Oct. 2020 from https://packetstormsecurity.com/files/30760/reducebind.c.html

Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 329–339. https://doi.org/10.1145/3238147.3238160

Amitabh Srivastava and David W. Wall. 1992. *A Practical System for Intermodule Code Optimization at Link-Time*. WRL Research Report 92/6. Digital Western Research Laboratory. Retrieved Oct. 2020 from https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-92-6.pdf

Thomas A. Standish, Dennis F. Kibler, and James M. Neighbors. 1976. Improving and Refining Programs by Program Manipulation. In *Proceedings of the 1976 Annual Conference (ACM '76)*. ACM, New York, NY, USA, 509–516. https://doi.org/10.1145/800191.805652

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

Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th ACM Symposium on Operating Systems Design and Implementation (OSDI '02)*. ACM, New York, NY, USA, 181–194. https://doi.org/10.1145/844128.844146

Andreas Ziegler, Julian Geus, Bernhard Heinloth, Timo Hönig, and Daniel Lohmann. 2019. Honey, I Shrunk the ELFs: Lightweight Binary Tailoring of Shared Libraries. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 102 (Oct. 2019), 23 pages. https://doi.org/10.1145/3358222