Guided Linking: Dynamic Linking without the Costs

Motivation

Optimizations don't cross dynamic linking boundaries

```
// plugin.so                    // library.so
void foo(T* p) {                int bar(T* p) {
    int n = bar(p);                 return p->size;
    // ...                      }
}
```

We want to inline `bar` into `foo`—but how?

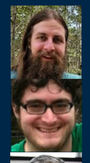
Guided Linking

- A new extension of LTO
- Works on existing, **unmodified software**
- Uses existing, **unmodified optimizations**
- Enables new deduplication technique
- Code: **https://github.com/yotann/bcdb**

Related Work

- Ziegler et al. 2019; Agadakos et al. 2019
  - Performs dead code elimination for shared libraries
  - Can't apply any other optimizations
- Dietz & Adve 2018 (Software Multiplexing)
  - Converts dynamic linking to static linking
  - Inflexible and incompatible

Authors



Sean Bartell

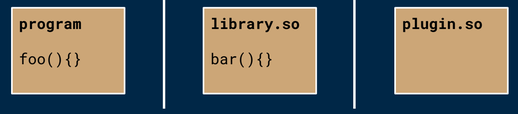
Will Dietz

Vikram S. Adve

University of Illinois

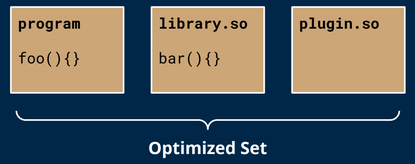
Problem #1

Compilers optimize each dynamic object **separately**

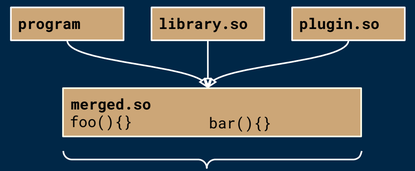


Solution

Developer provides **multiple objects** at once in the form of compiler IR



We move code to a **merged library**



And apply **existing optimizations** and **deduplication**

Optimized Set

- Arbitrary set of programs, libraries, plugins
- Must be compiled into IR form
- Must be distributed as a single unit
- Could be one package, an entire Docker container, ...

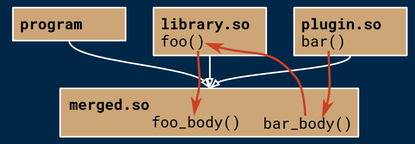
Problem #2

LD_DYNAMIC_WEAK? *LD_PRELOAD?* *Modified libraries?* */etc/ld.so.cache?*

Dynamic linking is **unpredictable**

Set-user-ID? *LD_LIBRARY_PATH?* *Interposing definitions?*

We can maintain perfect compatibility using **indirection** through the dynamic linker, but this prevents optimizations

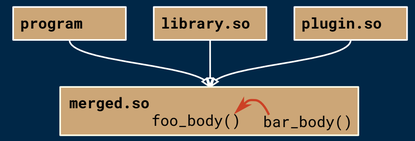


Solution

Developer provides **constraints**

e.g.: "This `foo()` will never be overridden by a different `foo()`."

Constraints let us **avoid indirection**



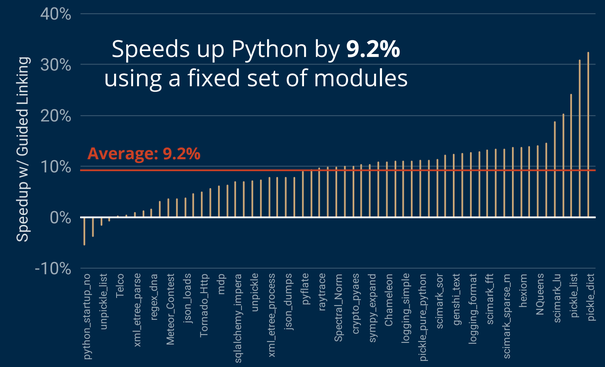
Constraints

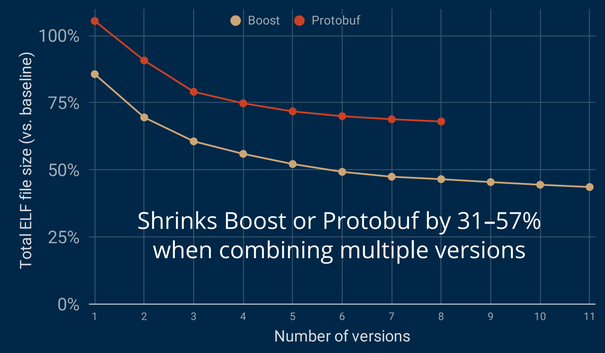
Four constraints can be applied to each symbol:

NoOverride: definition will not be overridden by an external definition.
NoUse: definition will not be used externally or with dlsym().
NoPlugin: code will not be used in a plugin.
NoWeak: no weak uses or external definitions.

With all four, we can almost optimize the code as if it were statically linked.

Results





Shrinks & speeds up Clang+LLVM (depending on constraints)