Guided Linking:
Dynamic Linking Without the Costs

Sean Bartell, Vikram Adve
University of Illinois
LLVM 2020
Motivating Example

// foo.c  // bar.c
void foo(T* p) {
    int n = bar(p);
    // ...
}

int bar(T* p) {
    return p->size;
}

Can be optimized (w/ LTO)
Motivating Example

// plugin.so
void foo(T* p) {
    int n = bar(p);
    // ...
}

// library.so
int bar(T* p) {
    return p->size;
}

Can’t be optimized
Motivating Example

// plugin.so
void foo(T* p) {
    int n = bar(p);
    // ...
}

// library.so
int bar(T* p) {
    return p->size;
}

Can't be optimized
Can be optimized
w/ Guided Linking
Guided Linking

- Works on existing software
- Requires no code changes
- Python 9% faster
- Boost 57% smaller (multiple versions optimized together)
Overview
Problem #1

Each program/library is optimized separately

program | plugin.so | library.so
Solution #1

Optimize multiple programs/libraries at once

- program
- plugin.so
- library.so

Optimized Set
Dynamic linking is **unpredictable**

**Problem #2**

- $LD\_DYNAMIC\_WEAK$?
- `/etc/ld.so.cache`?
- Set-user-ID?
- Modified libraries?
- Interposing definitions?
- $LD\_PRELOAD$?
- $LD\_LIBRARY\_PATH$?
Developer provides **constraints**

“This bar() will never be overridden by a different bar().”
Guided Linking

“This bar() will never be overridden.”

Constraints

Optimized Set

program

plugin.so

foo(){}

library.so

bar(){}
Step 1: move code to a merged library

“This bar() will never be overridden.”
Step 2: static resolution (when possible)

“This bar() will never be overridden.”

program

plugin.so
foo()

library.so
bar()

Merged Library

foo_body()

bar_body()
Step 3: LTO does the rest

“This bar() will never be overridden.”

```
opt -O3
```

```
This bar() will never be overridden.
```

```
foo_body()
```

```
Merged Library
```

```
bar_body()
```

```
program
```

```
plugin.so
```

```
library.so
```

```
bar()
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
Choosing the Optimized Set
Choosing the Optimized Set

- Arbitrary set of programs, libraries, plugins
- Must be in bitcode form
- Must be distributed as a single unit
  - Slows down upgrades
Optimized Set Possibilities

- One package
- Entire Docker container
- Entire desktop computer
  - Ship software in bitcode form
  - Re-optimize whenever programs are added
Constraints
Available Constraints

**NoOverride**  No external overrides

**NoUse**  No external uses

**NoPlugin**  No use in plugin

**NoWeak**  No weak uses or external definitions
Specifying Constraints

default: NoOverride, NoUse, NoPlugin, NoWeak

[Use]
fun:PyInit_*
NoOverride: No external overrides
What We Want

We want to inline bar().

Optimized Set
Without the Constraint

Unsafe to inline. What if `bar()` is overridden?

Optimized Set

```
Without the Constraint
```

```
program
bar(){}
```

```
plugin.so
call bar()
```

```
library.so
bar(){}
```
With the Constraint

NoOverride guarantees this won’t happen. We can safely inline `bar()`.

Optimized Set
NoUse: No external uses
What We Want

We want to internalize `bar()` in the merged library.

```
// plugin.so
long bar();

// library.so
long bar() {
  // Implementation
}
```
Without the Constraint 1

Unsafe to internalize. Used outside the set.

Optimized Set
Without the Constraint 2

Unsafe to internalize. Used through dynamic linker.

Optimized Set
With the Constraint

**NoUse** guarantees neither will happen. Safe to internalize.

Optimized Set
NoPlugin: no use in plugin
Background

Each plugin has its own lookup scope (RTLD_LOCAL)

program


Optimized Set

plugin.so

call bar()

library.so

bar(){}

Moving code that uses external functions requires extra work.
Applying the Constraint

Apply **NoPlugin** to non-plugins

- **program**
- plugin.so
- library.so

Optimized Set
Applying the Constraint

Or, reduce optimized set to only cover one plugin

- program
- plugin.so
- library.so

Optimized Set
NoWeak: no weak uses or external definitions
What We Want

- program1
  - bar(){}
- program2
- library.so

Optimized Set
What We Want

We want to move defs to the merged lib

```
program1
bar(){

Merged.so
bar(){

library.so

program2
```
Without the Constraint

Unsafe with weak uses
Without the Constraint

Unsafe with other defs

program1
bar(){}

Merged.so
bar(){}

program2

library.so
bar(){}
With the Constraint

Safe with **NoWeak**
Results
Python

- Optimized set: Python, plus plugin modules
- Constraints: **NoOverride+NoUse+NoPlugin+NoWeak**
  - NoPlugin exceptions for plugins & their deps
  - NoUse exceptions for functions loaded with dlsym()
- Baseline: O3+LTO+PGO
- Benchmarks: pyperformance suite
Python

Average: 9.2%
Python Example 1

// _pickle.so
int save(...) {
    ...
    while (_PyDict_Next(...))
        ...
    ...
}

// libpython3.7m.so
int _PyDict_Next(...) { ... }

Guided Linking enables inlining here.
Python Example 2

// libpython3.7m.so
PyTypeObject PyFrame_Type = {
    ... (destructor)frame_dealloc,
    ...
};

- Guided Linking internalizes this variable
- LLVM determines it's never modified
- frame_dealloc() can be inlined.
Boost and Protobuf

- Optimized set: multiple versions of the same library
- Constraint: NoPlugin
- Function deduplication
  - Normalize functions
  - Merge bodies of identical functions
- Baseline: clang -Oz -flto
Size reductions

57%
11 versions of Boost

31%
8 versions of Protobuf
Clang

- Optimized set: all of LLVM and Clang
  - Built with BUILD_SHARED_LIBS (94 programs, 227 libs)
  - Caveat: LLVM already has better options
- Constraints: 4 different levels
  - Closed: NoPlugin+NoOverride+NoWeak+NoUse
  - Open-Spurious: NoPlugin+NoOverride+NoWeak
  - Open: NoPlugin+NoOverride
  - Interposable: NoPlugin
- Baseline: O3+LTO+PGO
- Benchmark: compile SQLite
Future Work
Automatic Multicall

- Automatically make a program like Busybox
- Get rid of dynamic linker entirely
- Will Dietz’ Allmux did this in limited cases
  - No libraries left out
  - No plugins or dlsym
  - No multiply-defined symbols
Conclusion

- Guided Linking can optimize dynamically linked code
- Use an optimized set and constraints
- Speed up Python by 9%
- Combine Boost/Protobuf versions and shrink by 31-57%
- Speed up Clang+LLVM by 5% and shrink by 13%
Conclusion

- Guided Linking can optimize dynamically linked code
- Use an **optimized set** and **constraints**
- Speed up Python by **9%**
- Combine Boost/Protobuf versions and shrink by **31-57%**
- Speed up Clang+LLVM by **5%** and shrink by **13%**

Questions? Ideas? smbarte2@illinois.edu