(Ab)using LTO plugin API for system-wide shrinking of dynamic libraries

Vladislav Ivanishin
vlad@ispras.ru

ISP RAS

GNU Tools Cauldron 2018

A Self-referential Slide

https://gcc.gnu.org/wiki/cauldron2018?action=AttachFile&do=get&target=lto-plugin-api-abuse.pdf
1

[1] It was https://compilers.ispras.ru/materials/plug.pdf during the actual talk.

Problem Statement

Given: a distribution with **shared libraries**, **immutable** once it's built (i.e. no package manager).

Slim it down by eliminating unused code/data

Problem Statement

Given: a distribution with **shared libraries**, **immutable** once it's built (i.e. no package manager).

Slim it down by eliminating unused code/data

assuming "closed world" full-distro rebuilds

- No packages bypass the toolchain we control
- Nothing is added afterwards; no "potential future uses"

Aside: Elimination in Static Linking

For static linking, already available in practice:

1. Compile with `gcc -ffunction-sections -fdata-sections`:

Per-function sections

```
        .section        .text.foo,"ax",@progbits
        .globl  foo
        .type   foo, @function
foo:
        movl    $42, %eax
        ret
```

2. Link with `--gc-sections`
   Linker omits sections not reachable by relocations from the entry point

--gc-sections for Dynamic Modules

Can we use `--gc-sections` for shared libraries?
For dynamic linking, entrypoint is not the only GC root

- The `.dynamic` section is another root
  Points to dynamic symbols and global library constructors/destructors
- Most code is reachable from dynamic symbols (the library's interface)
- Reducing the API surface (changing symbol's *visibility* to "hidden") allows GC

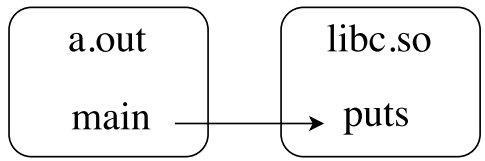
Dependency Types

Want to compute reachability on dynamic symbol set

- Link-time dependencies

Direct Call

```
int main()
{
  puts("Hello World");
}
```



Dependency Types

Want to compute reachability on dynamic symbol set

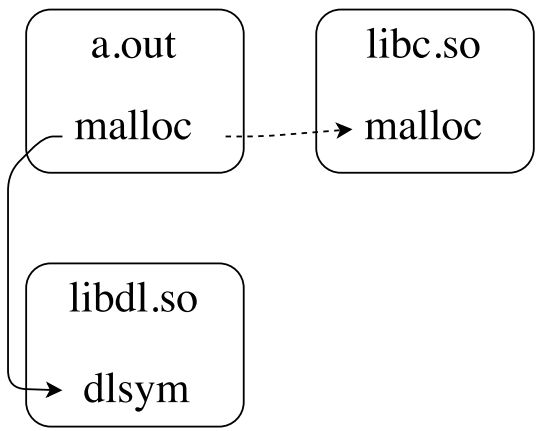
- Link-time dependencies
- Run-time dependencies via `dlsym()`

Dynamic `dlsym` Lookup

```
#include <dlfcn.h>

void *dlsym(void *handle,
            const char *name);

void malloc(size_t n)
{
  void *real_malloc =
      dlsym(RTLD_NEXT, "malloc");
  ...
}
```



Dependency Types

Want to compute reachability on dynamic symbol set

- Link-time dependencies
- Run-time dependencies via `dlsym()`
- Other run-time dependencies
  - interpreters expose `dlsym` to other languages (e.g. Perl)
  - manual parsing of ELF data structures

Dependency Types

Want to compute reachability on dynamic symbol set

- Link-time dependencies ← this talk only covers this kind
- Run-time dependencies via `dlsym()` ← described in [1]
- Other run-time dependencies ← only manual annotation
  - interpreters expose `dlsym` to other languages (e.g. Perl)
  - manual parsing of ELF data structures

High-level Approach

1. Record link-time dependencies (requires whole system rebuild)
2. Analyze system-wide symbol dependency graph
3. Eliminate unused symbols (another whole system rebuild)

Implementing steps 1 and 3 as plugins and/or wrappers allows to avoid patching the toolchain.

Recording Link-time Dependencies

Use LTO plugin interface for introspection

- Avoid patching the linker
- Avoid duplicated work (search in static archives)
- Subtle point: disable `-flto` (if we are invoked on a file with IR we won't know what to do; otherwise, we won't see the file at all: plugin hooks are not called on files added from plugins with `add_input_file`)

The `claim_file_handler` API hook allows to inspect object files

- Find symbol tables
- Find relocation tables
- Resolved relocations give intra-DSO dependencies
- Cross-DSO deps are given by dynamic relocations

Analyzing System-wide Dependency Graph

- stand-alone tool
- takes dependencies collected at the previous step from all links
- merges them into one global graph
  $V =$ {sections and symbols}, $E =$ {relocations and definitions}
- traverses it from entry points

Aside: Simple Elimination with Version Script

Linking musl with a version script

```
gcc -Wl,--gc-sections -Wl,--version-script,./1.lds ...
  -o libc.so
```

```
/* 1.lds: */
{
  global:
    _dlstart;
    __dls3;
  local: *;
};
```

```
$ ls -lh libc.so{,.orig}
 64K libc.so
723K libc.so.orig
```

Aside: Simple Elimination with Version Script

Linking musl with a version script

```
gcc -Wl,--gc-sections -Wl,--version-script,./1.lds ...
  -o libc.so
```

```
/* 1.lds: */
{
  global:
    _dlstart;
    __dls3;
  local: *;
};
```

```
$ ls -lh libc.so{,.orig}
 64K libc.so
723K libc.so.orig
```

Limited to DSOs linked without a version/linker script:

- multiple linker scripts are not supported
- modification of existing linker script would be tricky

Eliminating Unused Symbols, Take 1

Two opportunities: compile time (in GCC), and link time

1. At compile time: optional, for optimization
   - Compiler doesn't process asm inputs
   - Constrained: can only eliminate symbols unneeded in *all* links
2. At link time: required
   - Arbitrary source language
   - Elimination on per-DSO basis

Implementation:

1. Force-enable `--gc-sections`
2. Set *hidden visibility* on eliminated symbols
   Linker plugin claims the input `.o` files and adds their copies with adjusted visibility info to the link (via `add_input_file`)

Eliminating Unused Symbols, Take 1: Bugs (bfd madness)

`ld.bfd` has a bug which stems from using a bfd representing a real file to hold phony sections.

```
/* The BFD used to hold special sections created
   by the linker. This will be the first BFD found
   which requires these sections to be created. */
bfd *dynobj;
```

- Manifested in 2016 with LTO plugin, fixed
- Unlike LTO plugin, we may claim `crt{1,i,begin}.o`
- H.J. Lu proposed a `[PATCH] Add a fake bfd to hold linker created dynamic sections`. Unfortunately, it was never applied.
- Workaround: offer a `dummy.o` in sacrifice to `ld.bfd`.

Eliminating Unused Symbols, Take 1: Bugs (`.symver`)

The linker provides `add_symbols` interface [4]:

```
enum ld_plugin_status
(*add_symbols) (const void *handle,
                int nsyms,
                const struct ld_plugin_symbol *syms);

struct ld_plugin_symbol
{
  char *name;
  char *version;
  int def;
  int visibility;
  uint64_t size;
  char *comdat_key;
  int resolution;
};
```

Eliminating Unused Symbols, Take 1: Bugs (`.symver`)

mode \ origin	`.s` file	`__asm__`
`linker w/o plugins`	sees versioned symbol in ELF, knows what to do	ditto
`LTO plugin`	does not claim such files: no IR	has no way of knowing what `version` to set, as it does not look inside toplevel asms
`other plugins`	can try to set `ld_plugin_symbol::version`, but the interface is untested and inconsistent across linkers	ditto

Eliminating Unused Symbols, Take 1: Bugs (common & .a)

[PR 23411] Different behavior when linking common symbol statically or to shared object

```
/* dyp.c: */
int main () {
  return bar();
}

/* dup.c: */
/* COMMON; w/o this decl the bug does not manifest */
int xre_syntax_options;
int xre_max_failures = 42;
int bar() {
  return xre_max_failures;
}

ar rc dup.a dup.o
gcc -Wl,-y,xre_max_failures -shared dyp.o dup.a dup.a
```

Eliminating Unused Symbols, Take 1: Bugs (c'tor order)

(This is not a toolchain bug.)

No mechanism for preserving constructor order.

- no guaranteed ordering for constructors with default priority taken from different object files
- some packages (erroneously) rely on default ordering, and will be broken when plugin claims such files

Eliminating Unused Symbols, Take 2 (`aux.o`)

Observation (per ELF spec [2]):

> *if **any reference** to or definition of a name is a symbol with a non-default visibility attribute, the visibility attribute must be propagated to the resolving symbol in the linked object . . . **the most constraining** visibility attribute must be propagated to the resolving symbol in the linked object.*

The approach:
For each link, generate an object file with references carrying hidden visibility for desired symbols on the fly and supply it to the linker.

- compatible with symbol versioning
- does not disturb constructor ordering
- and library ordering

Eliminating Unused Symbols, Take 2: Bugs

[PR gold/22566] --gc-sections preserves hidden symbols if initial definition has default visibility

For us it means the aux file must go first on the command line.

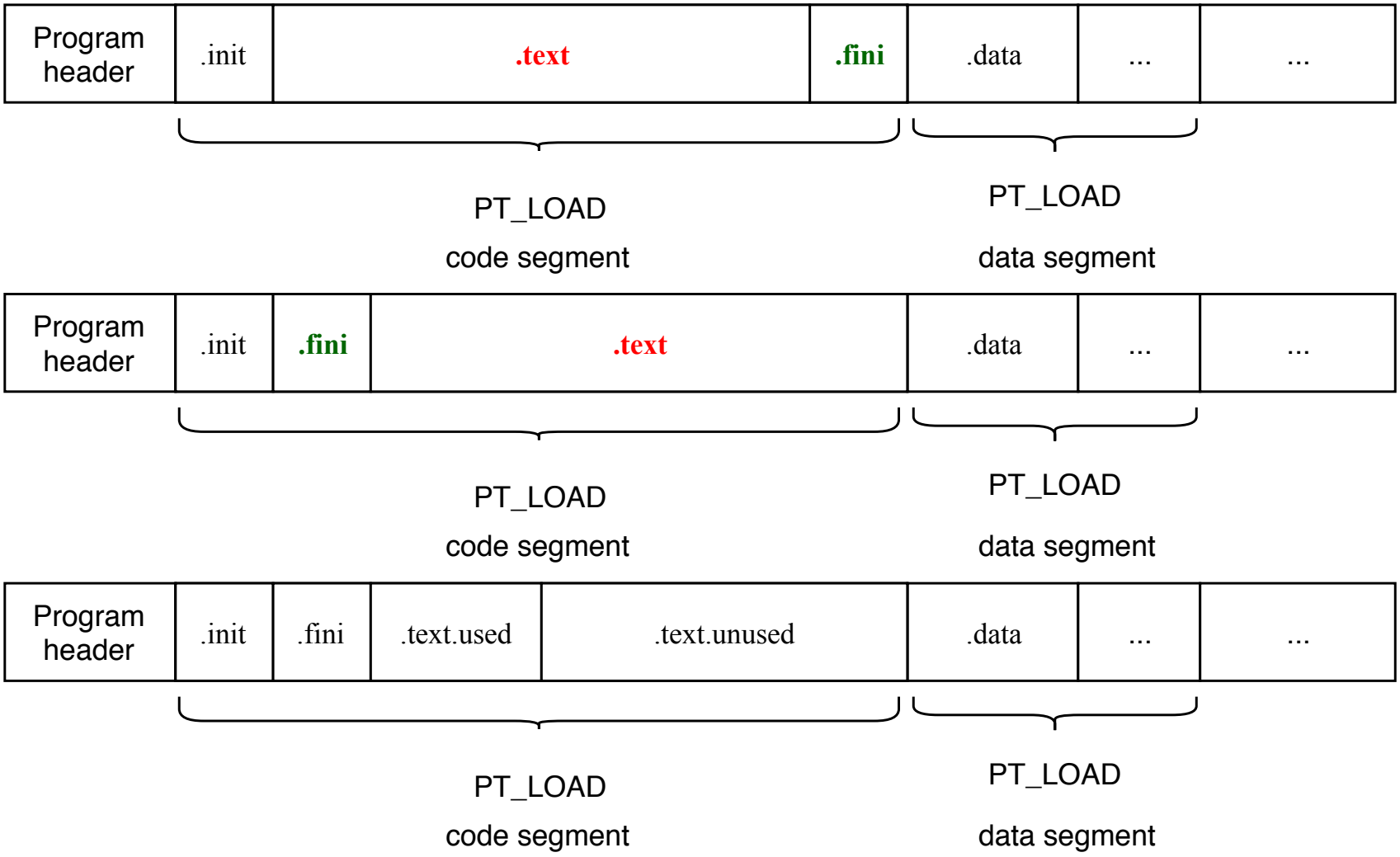
But it also must be the last (and only the last)! Because otherwise it will affect extraction of `.a` members.

Eliminating Unused Symbols, Take 3 (segshrink)

Idea:

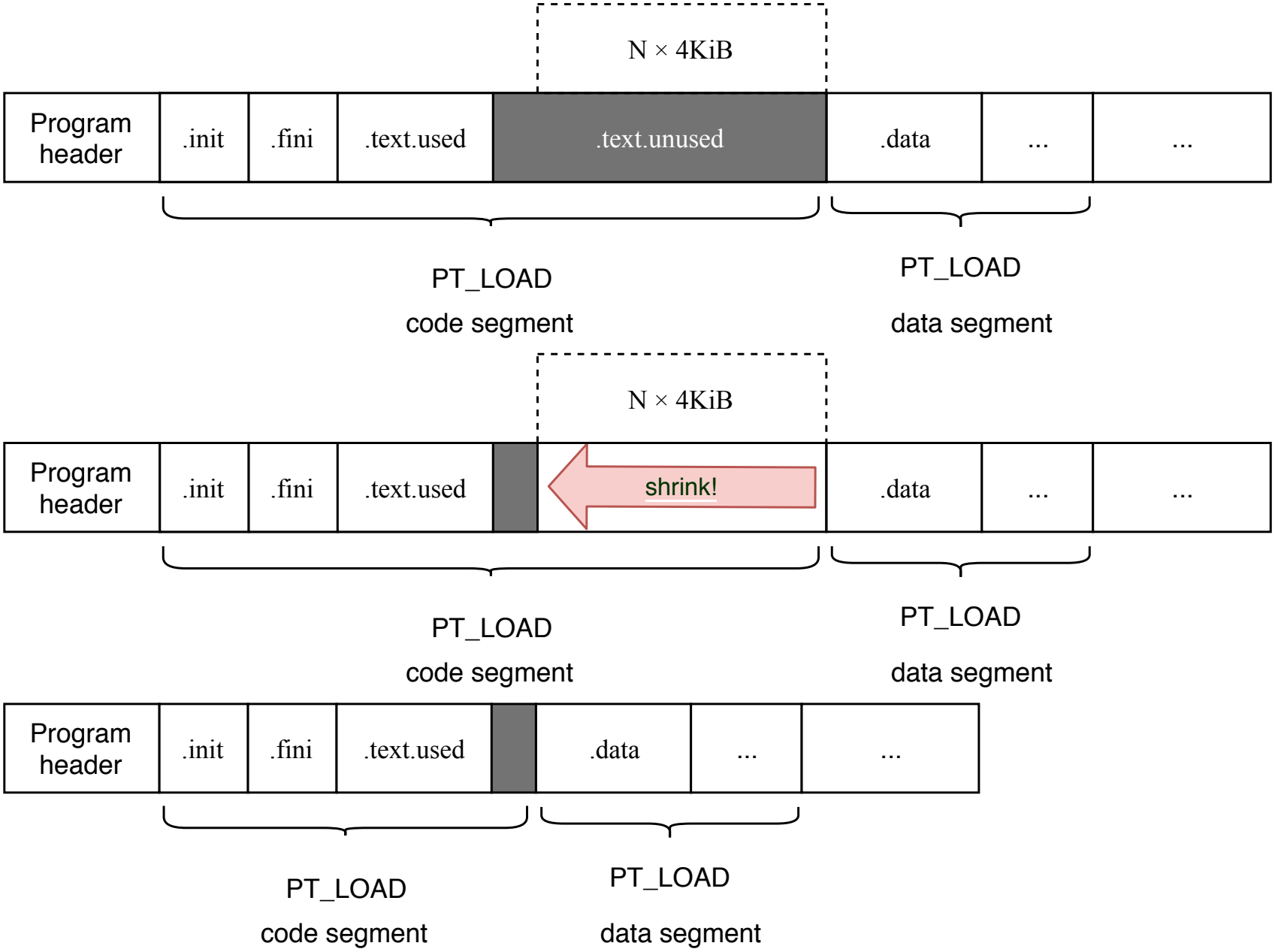
- binary post-processing
- divide loadable segments into used/unused, chop off the tails
- (this requires link-time section reordering, implemented as a plugin)

Eliminating Unused Symbols, Take 3 (segshrink)



(A thinko: `s/Program header/ELF header/`)

Eliminating Unused Symbols, Take 3 (segshrink)



Eliminating Unused Symbols, Take 3 (segshrink)

Pros:

- better reproducibility: configure tests at step 3 will probe unmodified (modulo reordering) binaries, same as at step 1
- potential to eliminate more: no need to consider `mains` of configure tests as roots for reachability analysis
- doesn't suffer from any linker bugs (related to `--gc-sections`, versioned symbols, or plugin API implementation)

Cons/limitations:

- requires `LDPT_UPDATE_SECTION_ORDER` plugin interface which is only implemented in Gold
- and a small patch for Gold (move `ORDER_FINI`, `ORDER_EHFRAME` above `ORDER_TEXT`)
- hard to regenerate and shrink `.dynstr`, `.dynsym` (and references to it), and hash tables (not done in our PoC implementation)
- 4K alignment overhead (missed optimization) per segment

Bibliography


V. Ivanishin, E. Kudryashov, A. Monakov, D. Melnik, and J. Lee.
System-wide elimination of unreferenced code and data in dynamically linked programs.
In *2017 Ivannikov ISPRAS Open Conference (ISPRAS)*, pages 1–5, Nov 2017.


System V application binary interface.
`http://www.sco.com/developers/gabi/latest/ch4.symtab.html`.


Ian Lance Taylor.
Linkers.
`https://lwn.net/Articles/276782/`.


WHOPR driver design.
`https://gcc.gnu.org/wiki/whopr/driver`.