Qualcomm

Customizing link using plugins

Parth Arora, Rishabh Bali and Shankar Kalpathi Easwaran

Qualcomm India Private Limited

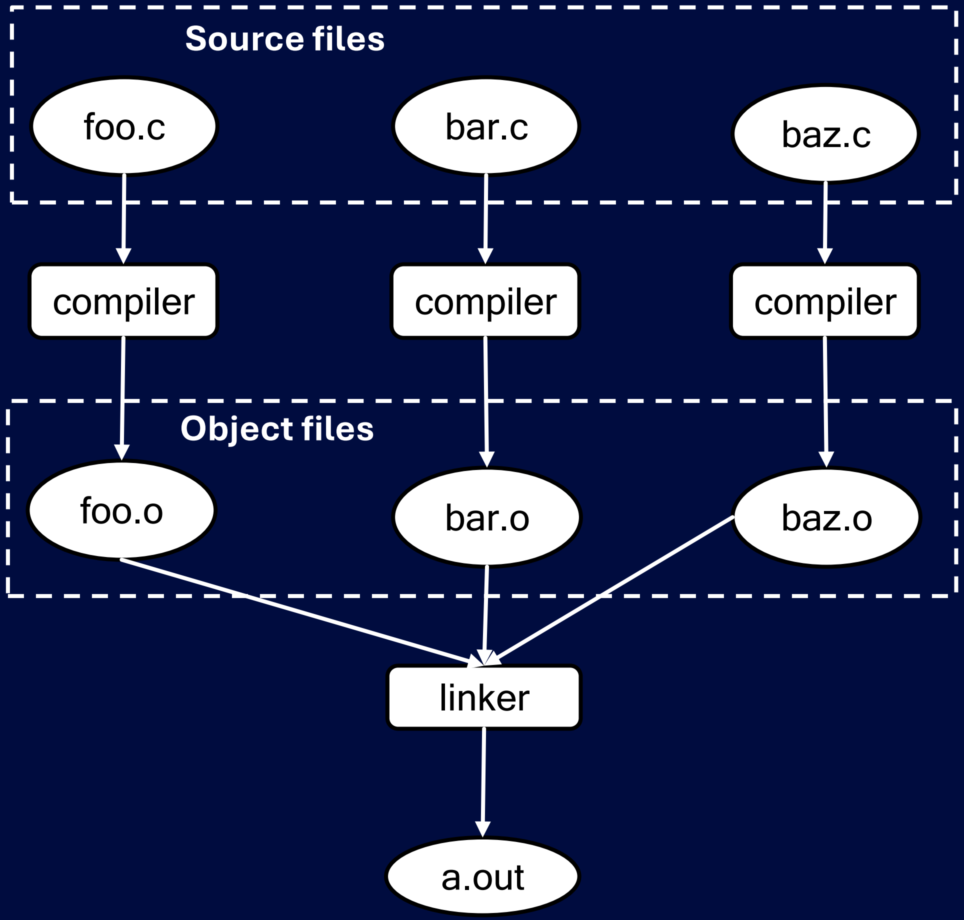
rbali@qti.qualcomm.com

Agenda

1. eld plugin Framework
2. Plugin Use Cases
   - Symbol Checking
   - Layout Optimization

Linker overview

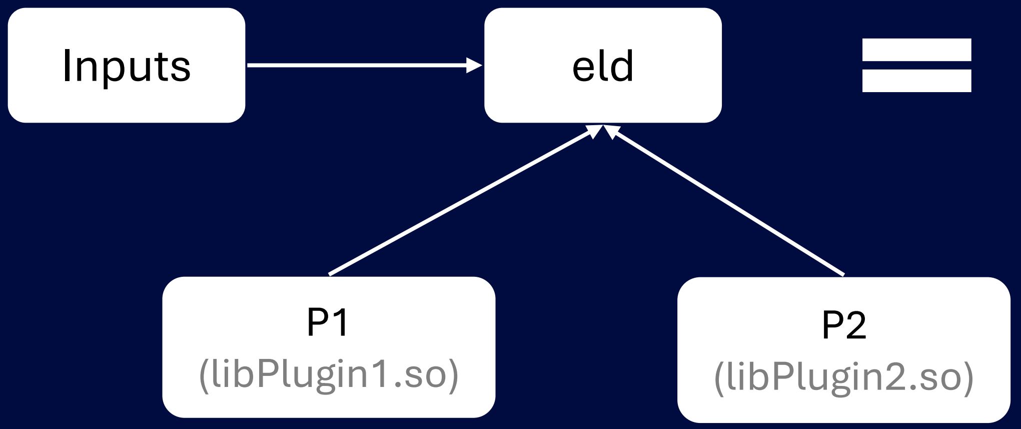
- Linker plays a key role in a toolchain
  - "Transforms all the ingredients into a meal that is ready to eat."
- Ingredients are the individual object files.
- Meal: binary that can run on a simulator/device
- Linker operation:
  - Read Input files
  - Resolve symbols
  - Match input sections (ELF) to output sections using linker script
  - Layout image
  - Resolve relocations
  - Output image
- Widely available linkers for building applications:
  - GNU Linker, gold linker (gold), LLVM linker (lld), mclinker, Mold linker (mold), Wild linker, ELF toolchain linker



Plugin framework

- eld plugin framework allows users to customize the link behavior without any modifications to the linker.

- The plugin framework provides finer control over the image layout than what is possible using traditional linker scripts.



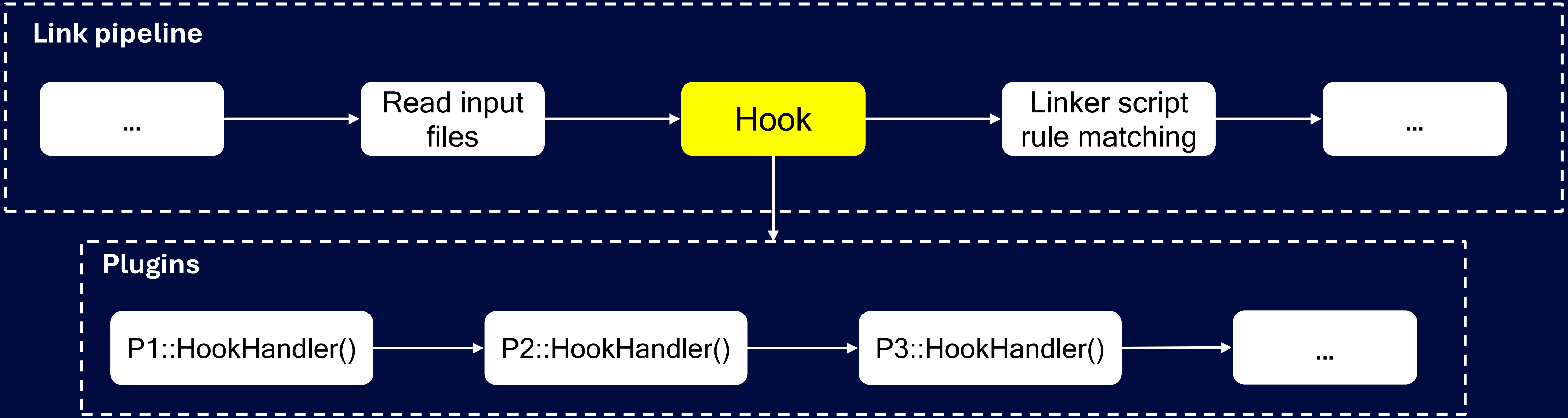
Plugins customized link behavior

P1 and P2 are plugins

Plugin framework

- Plugins can inspect and modify any of the key linker features and data structures such as: symbols, sections, relocations, input files, rule-matching, diagnostics, LTO, garbage-collection, and more.

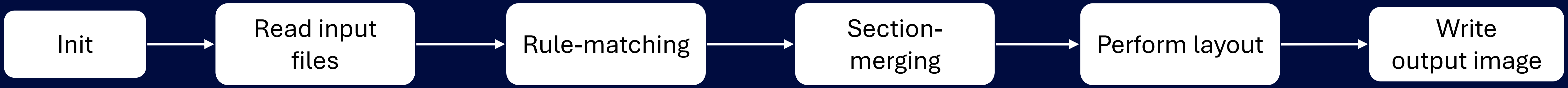
- The plugin framework provides hooks at key points of the link pipeline and plugins implement hook handlers to customize the link behavior.



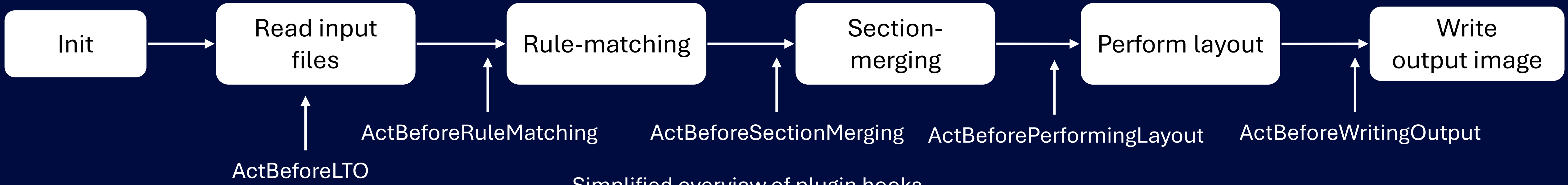
Plugin framework: Plugin Hooks

- Hooks names indicate where the hook is placed within the link pipeline. There are two kinds of hooks:
  - **Visit<Component>**: These hooks are called just after creating the component. For example: VisitSection and VisitSymbol.
  - **ActBefore<LinkState>**: These hooks are called just before the linker enters a particular link state. For example: ActBeforeRuleMatching and ActBeforeSectionMerging.

  - Simplified link pipeline:



Plugin framework: Plugin Hooks



Init → Read input files → Rule-matching → Section-merging → Perform layout → Write output image

ActBeforeLTO

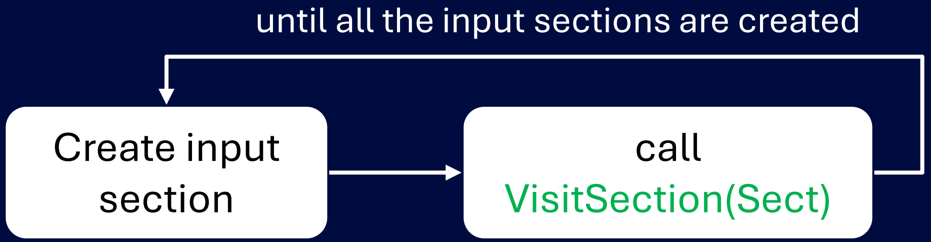
ActBeforeRuleMatching

ActBeforeSectionMerging

ActBeforePerformingLayout

ActBeforeWritingOutput

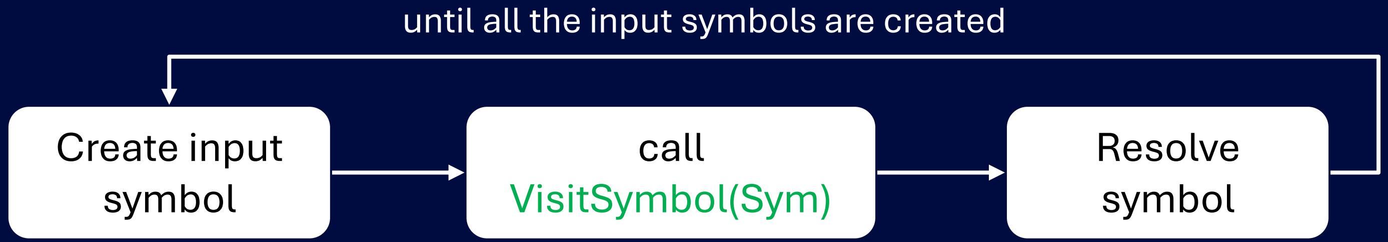
Simplified overview of plugin hooks



until all the input sections are created

Create input section → call <span>VisitSection(Sect)</span>

During reading section headers of an input file.
(Reading input files state)



until all the input symbols are created

Create input symbol → call VisitSymbol(Sym) → Resolve symbol

During reading symbols of an input file.
(Reading input files state)

Plugin Examples

Weak and Common Symbols

- Weak symbols are a way to define symbols (functions or variables) that **can be overridden** by other definitions during linking or loading

Here the definition from 1.c will be considered for symbol foo and 3.c will be ignored.

1.c

1.C

```
//foo definition
int foo() {return 12;}
```

2.c

```
extern int foo();
int main() {return foo();}
```

3.c

```
__attribute__((weak)) int foo()
{ return 2;}
```

- An uninitialized global variable that is not marked extern, and is not given a strong definition (i.e., no initialization). It is placed in the "common" section by the compiler and resolved by the linker.

1.c

```
int x; // uninitialized global
var
```

2.c

```
int x; // uninitialized global
var
```

3.c

```
extern int x;
print(x)
```

SymbolChecker

- Weak and common symbols can be error-prone. How to modify the build process to emit a warning / error on detecting a weak or a common symbol from certain input files?

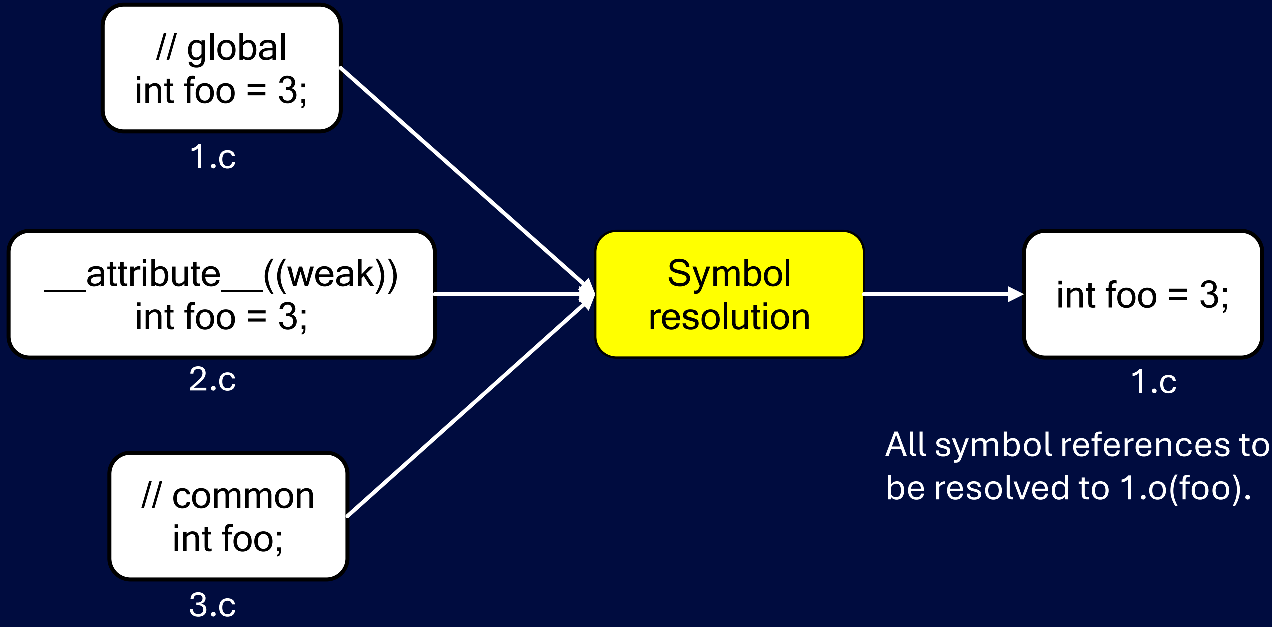
- Linker symbol resolution sees all the symbols. How to modify linker symbol resolution to report warning / error on seeing *bad* symbols?

- Modify the core linker for each custom feature / behavior is not a scalable solution. So how to add custom behavior to linker symbol resolution?

- Linker plugins!!!

SymbolChecker: What is symbol resolution?

- A linker plugin can tweak symbol resolution behavior. But what is symbol resolution?
- Symbol resolution is the process of selecting symbols. The selected symbols form the output image symbol table and are used to resolve symbol references.

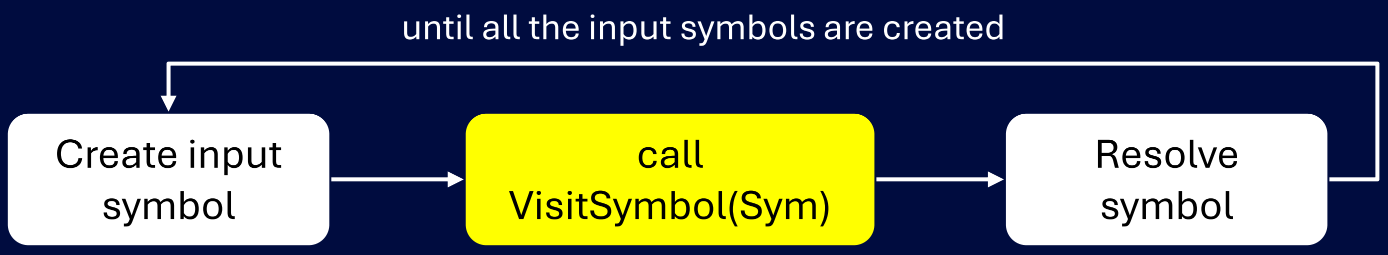


All symbol references to foo will be resolved to 1.o(foo).

SymbolChecker: How to tweak symbol resolution?

- A linker plugin customize link behavior by overriding plugin hooks.

- Which hook to override to tweak symbol resolution?



During reading symbols of an input file.
(Reading input files state)

- `virtual void VisitSymbol(plugin::InputSymbol S) {}`

SymbolChecker: How to tweak symbol resolution?

-
  ```
  void VisitSymbol(plugin::InputSymbol S) override {
     if (S.isCommon() || S.isWeak()) {
       auto diagID = getLinker()->getErrorDiagID("Bad symbol: %0");
       getLinker()->reportDiag(diagID, S.getName());
     }
  }
  ```

- If an error is reported from VisitSymbol, then the link ends with the error.

- How to only check for certain input files?

  ```
  auto I = S.getInputFile();
  bool b = shouldCheckSymbols(I.getFileName());
  if (b && (S.isCommon() || S.isWeak())) {
    // ..
  }
  ```

Layout Optimization

- Linker script is the go-to method for defining complex image layouts.
- Linker plugins provide more control and flexibility than the linker scripts.
- But why does the image layout matter?
  - Cache-locality depends upon the image layout. Better cache-locality means better performance.
  - Fast and slow memory (TCM and non-TCM). What to place where?
  - Hardware-specific constraints.
  - A good layout can help to reduce memory footprint.

LayoutOptimizer

- Functions are tagged with priorities and should be placed in the priority order for better cache-locality and performance.

1.c

```
#define HOT_FUNCTION(priority)
__attribute__((section(".text.hot." #priority)))

HOT_FUNCTION(1)
int foo() { ... }

HOT_FUNCTION(14)
int baz() { ... }

...

HOT_FUNCTION(2)
int bar() { ... }

...
```

script.t

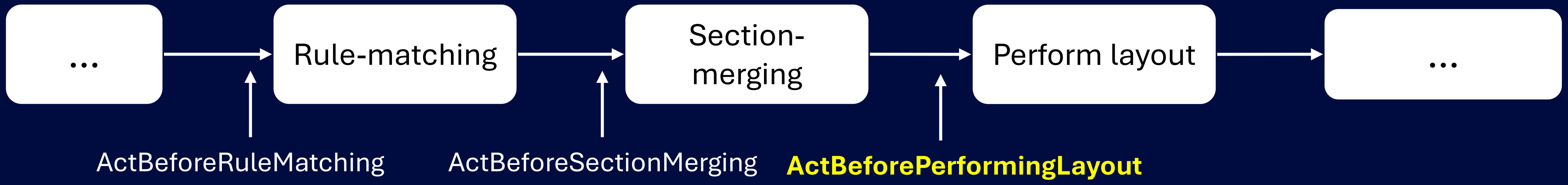
```
SECTIONS {
  A : { *(.text.hot.*) }
}
```

How to place .text.hot.* sections in the priority order?

LayoutOptimizer: How to reorder sections?

- A linker plugin can modify the layout by moving sections around.

- Which hook to override for moving sections around?



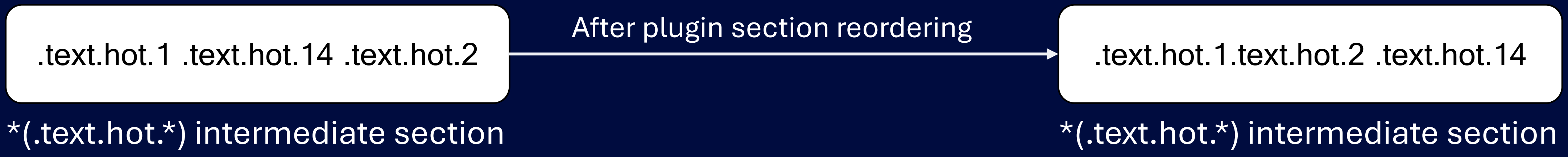
- `virtual void ActBeforePerformingLayout() {}`

LayoutOptimizer: How to reorder sections?

- Section merging link step joins together all the input sections that matches a linker-script rule into an intermediate section. There is one intermediate section for each linker script rule.

- Output sections are formed of these intermediate sections. Modifying these intermediate sections directly affect the output section.

- Plugin can reorder contents from one intermediate section to another using the LinkerWrapper APIs: AddChunk, RemoveChunk, and UpdateChunk.


.text.hot.1 .text.hot.14 .text.hot.2
After plugin section reordering
.text.hot.1.text.hot.2 .text.hot.14
(.text.hot.) intermediate section
(.text.hot.) intermediate section

Layout Optimization: Further improve layout: Section Budgeting

- What if all the hot sections total size is greater than the available size for output section A?

1.c

```
#define HOT_FUNCTION(priority)
__attribute__((section(".text.hot." #priority)))

HOT_FUNCTION(1)
int foo() { ... }

HOT_FUNCTION(14)
int baz() { ... }

...

HOT_FUNCTION(2)
int bar() { ... }

...
```

Max size: 0x200

script.t

```
SECTIONS {
  A : { *(.text.hot.*) }
  B : { *(.text.warm.*) }
}
```

LayoutOptimizer: Further improve layout: Section Budgeting

- What if all the hot sections total size is greater than the available size for output section A?

- A config file can be passed to the plugin defining the overflow policy:

```
A:
  max-size: 0x200
  overflow: B
```

- Plugin can read the config file and move chunks from A to B if A overflows.

Layout Optimization: Further improve layout: Section reordering without changing section name.

- What if changing section names to '.text.hot.$PRIORITY' is not feasible / favorable?
  - 'hot.$PRIORITY' pollutes the section names which can make debugging difficult.
  - $PRIORITY may not be known at compile time, making it unfeasible for embedding $PRIORITY in section names.

- A config file can be passed to a plugin that contains mapping of original section names to custom section names. Plugin can then instruct linker to use custom section names for section rule-matching.

```
.text.foo: .text.hot.1
.text.bar: .text.hot.14
.text.baz: .text.hot.2
 ...
```

SectionConfig.yaml

Thank you

Nothing in these materials is an offer to sell any of the components or devices referenced herein.

References in this presentation to “Qualcomm” may mean Qualcomm Incorporated, Qualcomm Technologies, Inc., and/or other subsidiaries or business units within the Qualcomm corporate structure, as applicable. Qualcomm Incorporated includes our licensing business, QTL, and the vast majority of our patent portfolio. Qualcomm Technologies, Inc., a subsidiary of Qualcomm Incorporated, operates, along with its subsidiaries, substantially all of our engineering, research and development functions, and substantially all of our products and services businesses, including our QCT semiconductor business.

Snapdragon and Qualcomm branded products are products of Qualcomm Technologies, Inc. and/or its subsidiaries. Qualcomm patented technologies are licensed by Qualcomm Incorporated.

Follow us on: in 𝕏 ◎ ▶ f

For more information, visit us at qualcomm.com & qualcomm.com/blog

References

- qualcomm/eld: Embedded Linker
- Linker Plugins — ELD documentation
- eld/Plugins at main · qualcomm/eld
- ELD User Guide — ELD documentation