LLVM RFC: LTO and Linker Scripts

Compiler Team

Exported on 10/06/2023

Table of Contents

1 Revisiting Qualcomm and TI Approaches to Communicating Linker Script Info to an LTO Recompile 4
1.1 Motivating Example 4
1.2 What Information Does the LTO Recompile Need? 5
1.3 Qualcomm (Edler Von Koch at al) Approach 5
1.4 TI (Snider et al) Approach 5
1.4.1 Differences From Qualcomm (Edler Von Koch) Approach 6
1.4.2 Rationale 6
1.4.3 Retrospective 6
1.4.4 Takeaways 7
1.5 Recommendations 7
2 Addendum - an exploration of available information in linker scripts 8
2.1 Memory Configuration 8
2.2 Function and Data Object Collection and Placement 8
2.3 What Information from the Linker Script Could the LTO Recompile Utilize? 9
2.3.1 Already Identified 9
2.3.2 Potentially Represented via Symbol Attribute 9
2.3.3 Other Kinds of Information Useful for an LTO Recompile 9

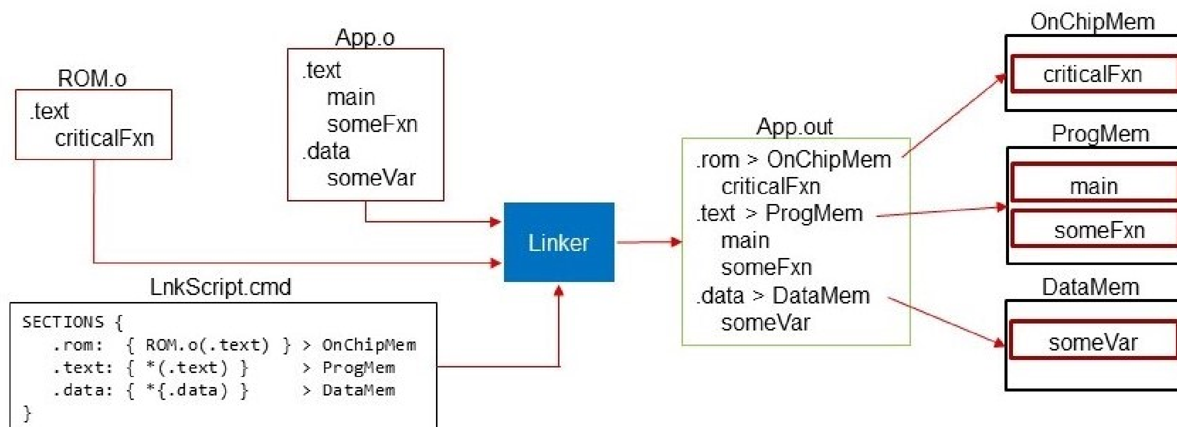
- Revisiting Qualcomm and TI Approaches to Communicating Linker Script Info to an LTO Recompile(see page 4)
  - Motivating Example(see page 4)
  - What Information Does the LTO Recompile Need?(see page 5)
  - Qualcomm (Edler Von Koch at al) Approach(see page 5)
  - TI (Snider et al) Approach(see page 5)
    - Differences From Qualcomm (Edler Von Koch) Approach(see page 6)
    - Rationale(see page 6)
    - Retrospective(see page 6)
    - Takeaways(see page 7)
  - Recommendations(see page 7)
- Addendum - an exploration of available information in linker scripts(see page 8)
  - Memory Configuration(see page 8)
  - Function and Data Object Collection and Placement(see page 8)
  - What Information from the Linker Script Could the LTO Recompile Utilize?(see page 9)
    - Already Identified(see page 9)
    - Potentially Represented via Symbol Attribute(see page 9)
    - Other Kinds of Information Useful for an LTO Recompile(see page 9)

1 Revisiting Qualcomm and TI Approaches to Communicating Linker Script Info to an LTO Recompile

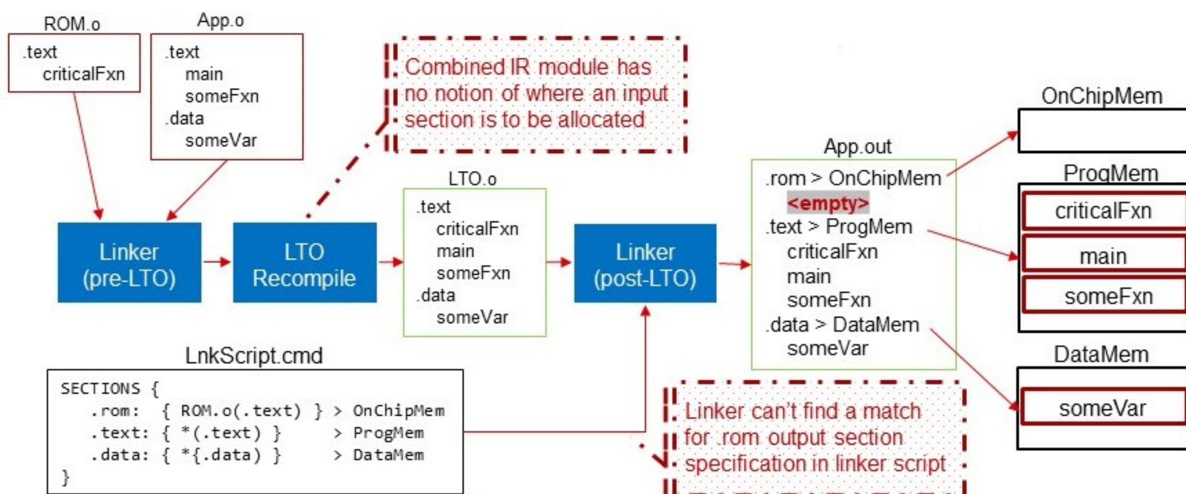
1.1 Motivating Example

Situation: Linker script is used to place a specific function (criticalFxn) in "fast" memory (OnChipMem in this example):

- Without LTO enabled, linker honors placement specified by linker script



- With LTO enabled, info from linker script about placement of function from ROM.o is not available after LTO recompile



- criticalFxn ends up being placed in wrong memory region

1.2 What Information Does the LTO Recompile Need?

- Original Object Module ID associated with a symbol definition must be carried through to the LTO recompile generated object file (LTO.o)
  - Allows the linker to connect a function or data symbol's input section from the LTO.o file to an input section specification in the linker script
- Output Section in which symbol definition's input section is collected must be made known to the LTO recompile
  - Can be utilized in LTO inter-module optimizations to avoid correctness and/or performance issues, for example:
    - Two global constants with the same value that are allocated to separate output sections should not be merged if one of the constants is not guaranteed to be available in target memory at the time of access
    - A function allocated to "fast" memory (e.g. tightly coupled memory) that is inlined into a function allocated to "slow" memory (e.g. off-chip/external) can degrade performance

1.3 Qualcomm (Edler Von Koch at al) Approach

Bringing link-time optimization to the embedded world: (Thin)LTO with Linker Scripts (llvm.org)[1] from Oct 2017 LLVM Dev Conf

- Initial compilation of individual source files
  - compiler adds "linker_input_section" field to a symbol's GlobalObject
  - "linker_input_section" information is recorded in the output IR Symbol Table for a given compilation
- While processing input section to output section mappings as specified in the linker script, the linker
  - identifies the output section associated with a given function or data symbol
  - identifies the "original" object module ID that a symbol is defined in
  - records the output section and original object module ID in (new) fields of the SymbolResolution record for a given symbol
- Output section and module ID information from SymbolResolution records are converted to symbol attributes
  - For RegularLTO, before merging an incoming IR module into the combined IR module
  - For ThinLTO, before and after importing an IR module
- Some inter-module optimizations were modified to become aware of new symbol attributes as needed/ beneficial
  - Constant merging, inlining, function merging, outlining, ...
- Names of sections in LTO recompile generated output file are augmented
  - Specifically, a section that contains the definition of a symbol is augmented with the section's original object module ID
- Post-LTO recompile, the linker decodes augmented section names to assist the linker in finding the correct input section to output section mapping for a given symbol

1.4 TI (Snider et al) Approach

Link-Time Attributes for LTO: Incorporating Linker Knowledge Into the LTO Recompile[2]

---

1 https://llvm.org/devmtg/2017-10/slides/LTOLinkerScriptsEdlerVonKoch.pdf
2 https://llvm.org/devmtg/2022-11/slides/QuickTalk11-Link-TimeAttributesforLTO.pdf

1.4.1 Differences From Qualcomm (Edler Von Koch) Approach

- Currently, the TI linker implementation assumes fat-LTO input object files
- No need to add "linker_input_section" during initial compilation of individual source files
  - Loaded ELF object provides mapping from symbols to input sections
- Like the Qualcomm approach, output sections and original module IDs are identified during processing of input section to output section mappings as indicated in the linker script
- However, instead of recording a symbol's output section and original module ID in a SymbolResolution records, the TI linker
  - Adds the ability to edit IR modules directly (using available LLVM methods)
  - Injects output section and original module ID attributes directly into the current IR module
- Like the Qualcomm approach,
  - Inter-module optimizations are updated to be aware of output section attribute, where appropriate
  - Symbol section names are augmented with a symbo's/section's original module ID
  - Augmented section names are parsed to help the linker find the correct input section to output section mapping

1.4.2 Rationale

- Adding linker capability to inject attributes directly into an IR module provides more flexibility than communicating linker knowledge via SymbolResolution records
- Basic SymbolResolution data structure does not need to be extended
  - Toolchains that use LTO, but whose use cases are not as concerned about honoring linker script instructions are not affected

1.4.3 Retrospective

- Implementation is straightforward, but adds code weight
  - LTOLLVMContext object, bitcode reader and writer added for parsing and editing IR modules.
  - Output section and original object module ID identification is already present in pre-existing linker
  - Symbol attributes are injected into the incoming IR module, updated IR module (in a temp file) is passed to LTO::add()
  - Is adding IR module editing capability justified if we're just going to use it for two pieces of linker script information?
- Is there more information from the linker script or elsewhere that can be useful to the LTO recompile?
  - To understand whether there is more information from the linker script that can be passed along to the LTO recompile, I enumerated the bits of information that one can glean from a linker script (see "Addendum" section below).
  - Besides the original object module ID and output section, there does not appear to be any obvious pieces of information that could be attached to a symbol in the form of an attribute
  - However, the linker script does provide information about the target memory configuration that is to be assumed at link time
    - Such information could be useful for the LTO recompile
    - For example, given information about available memory region types (relative access speeds), it seems intuitive that the hot/cold splitting optimization could express to the linker a preference for hot code to be placed in faster memory to realize some performance improvement
    - Symbol attributes are not the right vehicle for communicating this information to the LTO recompile

- Passing memory configuration information from the linker script to the LTO recompile will need its own API

1.4.4 Takeaways

- The flexibility afforded by having the linker generate attributes directly into a given IR module is not as advantageous as originally anticipated
- In practice, there is not much beyond a symbol's original module ID and output section that would be useful to represent as symbol attributes in an incoming IR module to the LTO recompile
- There is information about the memory configuration defined in a linker script that could be useful to LTO optimizations

1.5 Recommendations

- When considering which approach to pursue in terms of upstreaming, the LLVM Embedded Toolchain community should consider the trade-off between adding capability to the linker to directly inject symbol attributes into an incoming IR module vs. extending the basic SymbolResolution data structure to carry linker placement info to the LTO recompile
    - Recall that one of the rationales for the TI approach was to avoid extending the basic SymbolResolution data structure in order to avoid affecting toolchains that are not concerned about honoring linker script instructions
    - There may be alternative ways of extending the SymbolResolution data structure that are more amenable for upstreaming (e.g. perhaps a variable length array of symbol attribute descriptors?)
    - I suspect that adding a "linker_input_section" attribute during the compilation of individual source files, as prescribed in the Qualcomm approach, is not necessary
- TI intends to further investigate
    - How to communicate memory configuration information from the linker script to the LTO recompile
    - How memory configuration information from the linker script can be utilized during the LTO recompile

2 Addendum - an exploration of available information in linker scripts

This is effectively a brainstorm of what information is contained in a linker script, the idea being to make a reasonable attempt to uncover any information that could be potentially useful to an LTO recompile.

2.1 Memory Configuration

- Memory Regions
    - Address Range
    - Page
    - Sharing - accessible to all processors in a multi-processor application
    - Attributes
        - Executable
        - Read Only
        - Read-Write
    - Type
        - Very Fast Access - for example, Tightly Coupled Memory; very limited resource
        - Fast Access - for example, Local On-Chip Memory; limited resource
        - Slow Access - for example, External Memory; relatively unlimited resource

2.2 Function and Data Object Collection and Placement

- Output Sections
    - Mapping of input sections to output sections (i.e. collection)
        - Assumes: each function and data object is defined in its own input section
        - Allows for a single symbol to be associated with each input section
    - Alignment
    - Size
    - Placement Instructions
        - Mapping of each output section to a
            - specific address,
            - single memory region,
            - one of a list of memory regions, or
            - split among a list of memory regions
        - Placement mapping can pertain to
            - both the load and run placement
            - a load placement, and a separate mapping for run placement
                - Assumes application is responsible for copying contents from load location to run location at run-time before referencing any symbols defined in the output section
        - HIGH operator can be used to instruct the linker to allocate an output section to highest address available in a specified memory region
    - Operators
        - Alignment of output section's start location
        - Padding and alignment of output section
            - Aligns output section's start location, and
            - Pads length of output section to a multiple of specified alignment
        - Define linker symbol

            - Symbol value set to start location of output section
            - Symbol value set to end location of output section
            - Symbol value set to size of output section
            - Symbol on LHS of assignment operator
                - RHS is a well-defined expression (absolute value known at link time)
        - Dot ('.') operator used to define padding within an output section
    - Sharing
        - Contains Shared Code
        - Contains Globally Shared RO Data
        - Contains Globally Shared RW Data
        - Thread Local Storage

2.3 What Information from the Linker Script Could the LTO Recompile Utilize?

2.3.1 Already Identified

- Original Module ID - name of object file containing function or data object input section
- Output Section - name of output section in which an input section is collected; needed for correctness in some use cases

2.3.2 Potentially Represented via Symbol Attribute

- Sharing - identify function or data object symbol as being defined in a shared object file

LTO optimizations may need to bear in mind that a shared function or data object is accessible from other applications besides the one that is currently being built.

2.3.3 Other Kinds of Information Useful for an LTO Recompile

- List of Memory Regions - available for allocating to
    - Attribute of Memory Region - is it read/write, read-only, executable?
    - Type of Memory in Region - is it very fast access, fast access, slow access?

An LTO optimization like hot/cold splitting, for example, might want to express a preference for a function or data object to be allocated to very fast, fast, or slow memory to boost performance.