Impact of the current LLVM inlining strategy on complex embedded application memory utilization and performance

Sergei Larin

Senior Staff Engineer,

Harsha Jagasia

Staff Engineer,

Tobias Edler von Koch

Senior Engineer

Qualcomm Innovation Center, Inc.

Feb 04 2017

Talk objective

- Illustrate an off-beat LLVM usage scenario
  - When a small degree of uniform tuning is not enough
- Motivate the importance of this scenario for the average case
  - Why would I care?
- Give a specific example of current LLVM deficiency
  - What is not working, while it probably should…
- Propose a potential fix
  - …that would open new opportunities for everyone, not just the motivational example
- Open a discussion
  - …well, if we manage to convince you…

One LLVM usage scenario

- Target – Qualcomm® Hexagon™ DSP*
  - Embedded VLIW DSP
  - SIMD
  - Sophisticated memory infrastructure
    - Several levels of cache with Tightly Coupled Memory capabilities
  - Battery powered
  - Capable of running SPEC on Linux
- Application
  - A complex wireless networking application
  - Critical performance to code size tradeoffs (real time functionality)
  - Large heterogeneous codebase
  - Extensive use of linker scripts to control specific memory placement and layout
  - Extensive use of post-compile processing
- Build mode
  - LTO and non-LTO build mode mixed in the same build
  - Run time profile is available for most of the code base

* Qualcomm Hexagon is a product of Qualcomm Technologies, Inc.

LLVM usage scenario

- Though relatively unorthodox, this usage scenario has a common theme with many heterogeneous builds
- Embedded targets in general
  - Almost always memory constrained but with real time performance core
    - Some critical function has to be performed in real time fashion
    - A large contingency code section must be present
      - Error handling, uncommon scenario support, debugging etc.
    - A mix of optimization levels and compile options creates a non-uniform optimization space
- Use of linker scripts in projects
  - Not limited to embedded world
    - Linux kernel for instance
  - Has to be used for fine grain control of memory assignment and security features
  - Very often disruptive to implicit assumptions commonly made in LLVM
- LTO offers significant benefits in size and performance

LLVM usage scenario - LTO + Linker script

- Development mode for many large scale application dictates fragmented developer environment
  - System components often developed by different teams and delivered in library format
  - Performance critical and utility libraries developed independently and optimized for different end goals
  - Linker script is used for a variety of purposes most often via custom section rules
    - Memory mapping, post-processing organization, security features
  - -ffunction-sections/-fdata-sections with garbage collection at link time is critical
- LTO and ThinLTO offers a way to level the system and allow compiler unified view of the whole application
  - Except it does not play well with linker scripts

```
section_1:
  foo.o (.tcm.text.*)
section_2
  foo.o (text.*)
```

```
.section_1:
 {
  __foo_start__ = . ;
  *foo1/*:(.text.*)
  *foo2/*:(.text.*)
  __foo_end__ = . ;
  __bar_start__ = . ;
  *bar1/*:(.text.*)
  *bar2/*:(.text.*)
  __bar_end__ = . ;
 }
```

Some LLVM examples

- Global OPT (lib/Transforms/IPO/GlobalOpt.cpp)
  - Some global object optimizations are not safe for globals with explicit section assignment (creation of new objects in a wrong location)
    - OptimizeGlobalAddressOfMalloc
- Constant Merge (lib/Transforms/IPO/ConstantMerge.cpp)
  - Current implementation is safe but pessimistic
    - If GV has explicit section, constant merging is not always safe and is not performed at all, but with known output section it could be done
- Function Merging (aka Outlining) (lib/Transforms/IPO/MergeFunctions.cpp)
  - Opposite of inlining, very effective size optimization, but not safe if two outlining candidates are destined for two different output sections
- Inlining (lib/Analysis/InlineCost.cpp etc.)
  - But most performance-to-size critical of all is inlining
    - Minute adjustment in inline cost calculation produced 0.05% size increase but 2% run time speedup in the driving example

Inlining in LLVM

- Inlining can be seen as two separate tasks
  - Legality of inlining
  - Cost/benefit analysis
- Legality can be a problem in very few corner cases
  - Mostly with LTO
    - Like security – moving function body from a privileged to user space
- Cost calculation is more interesting
  - Even though in general it should be legal to exchange control flow between any .text sections, in practice it is often not advisable
    1) Frozen (zero profile weight) code issue
    2) Frequency vs. latency
    3) Heterogeneous optimization space with LTO
    4) Amount of inlining vs. size of inline scope
  - Not all of those issues are LTO/IPO specific, but concomitant issues are exaggerated by LTO

Inlining cost in LLVM - Frozen code issue

- lib/Analysis/InlineCost.cpp does use PGO and provides user specifiable thresholds

```
// We introduce this threshold to help performance of instrumentation based
// PGO before we actually hook up inliner with analysis passes such as BPI and BFI.
static cl::opt<int> ColdThreshold(    "inlinecold-threshold", cl::Hidden, cl::init(225),
                                      cl::desc("Threshold for inlining functions with cold attribute"));
static cl::opt<int>    HotCallSiteThreshold("hot-callsite-threshold", cl::Hidden, cl::init(3000),
                                            cl::ZeroOrMore,
                                            cl::desc("Threshold for hot callsites "));
```

- But it also has this code:

```
if (OnlyOneCallAndLocalLinkage)
  Cost -= InlineConstants::LastCallToStaticBonus;
```

- Which virtually guarantees single use local function inlining with the default cost
  - Reasoning behind it is generally sound - it should always be beneficial to inline a single callee

Inlining cost in LLVM - Impact on cache locality

- Experimental methodology
  - Two builds differ only by LTO scope – one is larger than the other
  - Collect hardware traces for both
    - Allows reconstruction of actual execution stream
  - Compare the two traces

- Observations
  - Overall code size decrease by about 0.05%
  - Worsening of L1 instruction cache line efficiency by 2%
  - Hottest function with respect to L1 instruction cache misses exhibits:
    - 1.97x increase in number of executed packets (VLIW instructions)
    - 4.5x increase in size
    - 15x increase in dead bytes (L1 resident code that is never executed)
  - Measurable increase in dead bytes observed in more than100 functions
  - Observe section utilization of some sections drop > 10%

- Upon inspection all frozen functions were inlined as a single local callee object overriding low profile count

Inlining cost in LLVM – Frequency vs. latency

- Function foo() calls bar()
  - foo() is a performance critical function and is placed to a specialized memory (locked cache or custom memory, extremely size constraint)
  - If bar() is a proverbial "error handling" code, and destined to a stashed-away-compressed-never-intended-to-use area of memory…
    - Introducing bar()'s code into foo()'s high cost memory is highly wasteful and might simply overflow the precious storage
- But how do I really know that foo() is highly valuable and bar() is not?
  - Profile information might be a good clue
    - High use frequency == importance
  - But what if bar() is a latency critical function – it is not used often, but it _MUST_ be readily available for execution
  - Only code developer can really tell….
    - …and developers often communicate such decision through section assignment
      - It could be explicit section set on bar()
      - Or path of bar()'s file is via a linker script assigned to a specific memory range
      - Finally it can be used in post-processing
- How do I know relative *importance* of foo() and bar() while I compute inline cost?

Inlining cost in LLVM - Heterogeneous optimization space

- This is especially critical for LTO
- Inlining needs to be adjustable to various use scenario and in general has to be exposed to the user

```
static cl::opt<int> InlineThreshold(    "inline-threshold", cl::Hidden, cl::init(225), cl::ZeroOrMore,
                                        cl::desc("Control the amount of inlining to perform (default = 225)"));
static cl::opt<int> HintThreshold(    "inlinehint-threshold", cl::Hidden, cl::init(325),
                                        cl::desc("Threshold for inlining functions with inline hint"));
```

- But these use scenarios vary greatly even in the same application
- During LTO code intended for different performance and placement is presented to the inliner at the same time
  - Function attributes can provide some hint, but are generally not detailed enough
    - There is no “per function” inline threshold for instance
  - LTO also changes linkage of a function
    - Globals are internalized for LTO
    - …and linkage is used for inlining cost computations

Inlining cost in LLVM - Amount of inlining vs. size of inline scope

- Relative amount of inlining should not be dependent on _size_ of inlining scope (number of functions visible during inlining), but it is…
  - When inlining scope is small, the few callee candidates have higher likelihood to be inlined
  - When full scope resolution is available more inlining candidates are available
    - This naturally produces different inline decisions and inliner seems to behave more greedily
    - Callees being inlined are often enlarged by previous inline decisions
- We have observed correlation between inline associated code growth and scope size (normalized to number of functions)
  - It is not critical, in single % points, but dependency is not linear
- The inlining scope is not used at all in cost computation, and we would have to adjust overall inlining thresholds to limit the code growth when LTO is used
  - This yet again worsens the heterogeneous optimization space dilemma

Improving Inlining in LLVM

- Short of major (slowly up-coming) changes in pass manager pipeline, and addressing multiple passes inlining in LLVM can still be improved by following:
  - Differentiate PGO use during inlining decision
    - More scrutiny to “frozen” code, and better detection of such code
  - Expose output section information from linker to compiler
  - Per-function options proliferation on module merging (LTO and ThinLTO)
    - Closely related to per-function optimization settings
  - Per-section optimization options

clang -flto -Os –section-options=“.tcm.text.*; -Oz –inline-threshold=10” foo.o bar.ll

Thank you

Follow us on:

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