Linker Code Size Optimization for Native Mobile Applications

Gai Liu

in collaborations with Umar Farooq, Chengyan Zhao, Xia Liu and Nian Sun
Impact of Code Size on Mobile Applications

More features → larger apps

Larger apps decrease user engagement

Install conversion rate increase per 10MB decrease in APK size by market

Today’s topic: transformations to reduce code size for mobile apps
State of the Art Approaches

- Key is to enable **global** size optimization
  - Commonly through link-time optimization (LTO) + machine outlining
  - Monolithic LTO based approach [1]

\[\text{~3x slowdown vs. default pipeline!}\]

[1] Chabbi, Lin, and Barik, CGO’21
State of the Art Approaches

- **Key is to enable global size optimization**
  - Commonly through link-time optimization (LTO) + machine outlining
  - Monolithic LTO based approach [1]
  - Size optimization based on thin LTO with two codegen rounds [2]

[1] Chabbi, Lin, and Barik, CGO’21

~40% slowdown
Some Additional Challenges...

<table>
<thead>
<tr>
<th>Compatible with existing build pipeline</th>
<th>Small build time penalty</th>
<th>Optimizes binary form libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic LTO</td>
<td>![Happy Face]</td>
<td>![Sad Face]</td>
</tr>
<tr>
<td>Size optimizing thin LTO</td>
<td>![Sad Face]</td>
<td>![Sad Face]</td>
</tr>
</tbody>
</table>

New approach?

~50% code is from binary in our builds
Proposal: Build an **Optimizing** Linker

- Existing linkers are good at symbol manipulation
  - Dead symbol stripping, function deduplication, etc.
  - But no sophisticated **code transformations inside functions**

- **Optimizing code size in linker:**
  - Compatible with existing build pipeline
  - Small build time penalty
  - Optimizes binary form libraries

- `-fuse-ld=myld`
- Lightweight & specialized linker passes

---

5
vs. Post Linking Optimization

Snippet of a post linking optimization pipeline:

**Linker Opt.**
- One implementation per linker
- Easy integration into build pipeline

**Post Linking Opt.**
- Independent of specific linker
- Additional step in the build pipeline
Rest of the Talk

• An infrastructure for linker code size optimization
  • Implemented on top of Apple’s ld64 linker
    • Looking into llc porting
    • Targeting ARM64 architecture
• >10% binary size reduction in our iOS apps
• <10% end-to-end build time overhead
Key Components

Utilities & Analyses
- Instruction decoder
- Sequence hashing

Code transformations
- Sequence outlining
- Safe ICF

Non-code section updates
- Exception table
- Debug info
- etc.
Prepare for Outlining

• Decode the instruction stream

```assembly
f45c: sub sp, sp, #48
f460: stp x20, x19, [sp, #16]
f464: stp x29, x30, [sp, #32]
f468: add x29, sp, #32
f46c: adrp x8, 44223
f470: ldr x8, [x8, #3288]
f474: stp x0, x8, [sp]
f478: adrp x8, 55266
f47c: ldr x1, [x8, #376]
f480: mov x0, sp
f484: bl _objc_msgSendSuper2
f488: mov x19, x0
f48c: cbz x0, 0xf4b4
f490: adrp x8, 44163
f494: ldr x0, [x8, #256]
f498: adrp x8, 55295
f49c: ldr x1, [x8, #2976]
f4a0: bl _objc_msgSend
f4a4: ldr x8, [x19, #8]
f4a8: str x0, [x19, #8]
f4ac: mov x0, x8
f4b0: bl _objc_release
f4b4: mov x0, x19
f4b8: ldp x29, x30, [sp, #32]
f4bc: ldp x20, x19, [sp, #16]
f4c0: add sp, sp, #48
f4c4: ret
```
Prepare for Outlining

• Mark “pivot” instructions
  • Instructions that are not suitable for outlining
  • Targets of branch instructions
  • Exception table range endpoints
Find Outlining Candidates

- Hash sequences across all functions
- Skip pivot instructions

Example: len = 5

```
<table>
<thead>
<tr>
<th>hash</th>
<th>freq</th>
<th>len</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
```

pivot instructions

- 0xf4b4
- ...
- ....

```
f45c: sub sp, sp, #48
f460: stp x20, x19, [sp, #16]
f464: stp x29, x30, [sp, #32]
f468: add x29, sp, #32
f46c: adrp x8, 44223
f470: ldr x8, [x8, #3288]
f474: stp x0, x8, [sp]
f478: adrp x8, 55266
f47c: ldr x1, [x8, #376]
f480: mov x0, sp
f484: bl _objc_msgSendSuper2
f488: mov x19, x0
f48c: cbz x0, 0xf4b4
f490: adrp x8, 44163
f494: ldr x0, [x8, #256]
f498: adrp x8, 55295
f49c: ldr x1, [x8, #2976]
f4a0: bl _objc_msgSend
f4a4: ldr x8, [x19, #8]
f4a8: str x0, [x19, #8]
f4ac: mov x0, x8
f4b0: bl _objc_release
f4b4: mov x0, x19
f4b8: ldp x29, x30, [sp, #32]
f4bc: ldp x20, x19, [sp, #16]
f4c0: add sp, sp, #48
f4c4: ret
```
Find Outlining Candidates

- Hash sequences across all functions
- Skip pivot instructions

Example: len = 5

<table>
<thead>
<tr>
<th>hash</th>
<th>freq</th>
<th>len</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>h2</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

pivot instructions

<table>
<thead>
<tr>
<th>0xf4b4</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>....</td>
</tr>
</tbody>
</table>

Hash sequences across all functions

- f45c: sub sp, sp, #48
- f460: stp x20, x19, [sp, #16]
- f464: stp x29, x30, [sp, #32]
- f468: add x29, sp, #32
- f46c: adrp x8, 44223
- f470: ldr x8, [x8, #3288]
- f474: stp x0, x8, [sp]
- f478: adrp x8, 55266
- f47c: ldr x1, [x8, #376]
- f480: mov x0, sp
- f484: bl _objc_msgSendSuper2
- f488: mov x19, x0
- f48c: cbz x0, 0xf4b4
- f490: adrp x8, 44163
- f494: ldr x0, [x8, #256]
- f498: adrp x8, 55295
- f49c: ldr x1, [x8, #2976]
- f4a0: bl _objc_msgSend
- f4a4: ldr x8, [x19, #8]
- f4a8: str x0, [x19, #8]
- f4ac: mov x0, x8
- f4b0: bl _objc_release
- f4b4: mov x0, x19
- f4b8: ldp x29, x30, [sp, #32]
- f4bc: ldp x20, x19, [sp, #16]
- f4c0: add sp, sp, #48
- f4c4: ret
Find Outlining Candidates

- Hash sequences across all functions
- Skip pivot instructions

**Example:** len = 5

<table>
<thead>
<tr>
<th>hash</th>
<th>freq</th>
<th>len</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>h2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>h3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Linear scan-based algorithm:
- Easy to implement & debug
- Linear time complexity (under a max sequence length)
- Exhaustive for a given len

```
0xf4b4
...  
....
```

pivot instructions

```
f45c: sub sp, sp, #48
f460: stp x20, x19, [sp, #16]
f464: stp x29, x30, [sp, #32]
f468: add x29, sp, #32
f46c: adrp x8, 44223
f470: ldr x8, [x8, #3288]
f474: stp x0, x8, [sp]
f478: adrp x8, 55266
f47c: ldr x1, [x8, #376]
f480: mov x0, sp
f484: bl _objc_msgSendSuper2
f488: mov x19, x0
f48c: cbz x0, 0xf4b4
f490: adrp x8, 44163
f494: ldr x0, [x8, #256]
f498: adrp x8, 55295
f49c: ldr x1, [x8, #2976]
f4a0: bl _objc_msgSend
f4a4: ldr x8, [x19, #8]
f4a8: str x0, [x19, #8]
f4ac: mov x0, x8
f4b0: bl _objc_release
f4b4: mov x0, x19
f4b8: ldp x29, x30, [sp, #32]
f4bc: ldp x20, x19, [sp, #16]
f4c0: add sp, sp, #48
f4c4: ret
```
Find Outlining Candidates

• Sort candidates by profitability

<table>
<thead>
<tr>
<th>hash</th>
<th>freq</th>
<th>len</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>h2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>h3</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h_n</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sort by profitability

<table>
<thead>
<tr>
<th>hash</th>
<th>freq</th>
<th>len</th>
</tr>
</thead>
<tbody>
<tr>
<td>h15432</td>
<td>38</td>
<td>5</td>
</tr>
<tr>
<td>h80</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>h912</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Profitable

Not profitable
Outline Transformation

Relocations are handled automatically by downstream linker step

Update cbz’s relative offset
Update Non-Code Sections

• Fix content of sections that rely on instruction addresses
  • __debug_info
  • __debug_line
  • __gcc_except_table

-DWARF debugging information entry

```
-foo:
100340c8c:  fd 7b 3f a9 stp x29, x30, [sp, #-16]
100340c90:  b0 f0 ff 97 bl OUTLINED_PROLOG_87
100340c94:  ff 03 01 d1 sub sp, sp, #64
100340c98:  48 5c 04 d0 adrp x8, 35722
100340c9c:  08 91 40 f9 ldr x8, [x8, #288]
100340ca0:  e8 07 00 f9 str x8, [sp, #8]
100340ca4:  c8 47 04 b0 adrp x8, 35065
100340ca8:  00 f5 43 fd ldr d0, [x8, #2024]
100340cac:  08 00 00 90 adrp x8, 0
100340cb0:  08 01 33 91 add x8, x8, #3264
100340cb4:  b3 b0 fd 97 bl LD_OUTLINED_6406
100340cb8:  ff 03 01 91 add sp, sp, #64
100340cbc:  9d eb ff 17 b OUTLINED_EPILOG_314
```

... size saving

```
100340ca0: ...
```

(DW_TAG_subprogram)

```
DW_AT_low_pc (0x0000000100340c8c)
DW_AT_high_pc (0x0000000100340ca0)
... (0x0000000100340c8c)
```
Safe Identical Code Folding

• Outlining reduces redundancies at instruction sequence level
• ICF removes redundancies at function level
  • Mature pass in llD
• We enhanced the ICF pass in ld64
  • More aggressive size opt
  • Improved compile time
  • Made it safe under pointer comparisons
    • Use redirection instead of direct replacement
• More details offline
Implementation & Experimental Setup

- ~3500 lines of C++ code as ld64 passes
- Benchmarks
  - Three commercial iOS apps
    1. News recommendation
    2. Short video hosting and sharing
    3. Enterprise collaboration client
  - mixture of Objective-C, Swift, C++, Rust
    - 1 to 2 million functions per app
- Build machine
  - 2.6 GHz, 6-core CPU, 64 GB RAM

ld64 pass pipeline

<table>
<thead>
<tr>
<th>Passes/Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective-C optimizations</td>
</tr>
<tr>
<td>Stub/GOT/TLV generation</td>
</tr>
<tr>
<td>Linker outlining</td>
</tr>
<tr>
<td>Order atoms</td>
</tr>
<tr>
<td>Safe identical code folding</td>
</tr>
<tr>
<td>Branch island/shim generation</td>
</tr>
<tr>
<td>DTrace probe processing</td>
</tr>
<tr>
<td>Compact unwind encoding</td>
</tr>
</tbody>
</table>
Size Reduction

No noticeable performance degradations from production data
Build Time Comparisons

• Linker outline + ICF overhead
  • Doubles the link time
  • <10% overhead overall

<table>
<thead>
<tr>
<th>Wall Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>icf_safe</em> pass</td>
</tr>
<tr>
<td><em>outline</em> pass</td>
</tr>
<tr>
<td>Total link time</td>
</tr>
<tr>
<td>Total build time</td>
</tr>
</tbody>
</table>

Build time breakdown

• Comparison
  • Monolithic LTO increases link time by > 2 hours for our apps
Summary

• We presented a framework for native code size optimization within the linker
  • Minimal change to build pipeline
  • Small build time overhead
  • Optimizes binary libraries

• arXiv preprint
  • arxiv.org/abs/2210.07311v1