Improving Machine Outliner for ThinLTO

(Global Machine Outliner + Frame Code Outliner)

Facebook
Kyungwoo Lee, Nikolai Tillmann
The Machine Outliner Today

• Machine outliner in LLVM significantly reduces code size
  • Works quite well with the whole program mode (LTO).
  • LLVM-TestSuite/CTMark (arm64/-Oz) up to 11% on average

• Under **ThinLTO**, its effectiveness drops significantly
  • Operates within each module scope
  • Misses all cross-module outlining opportunities
  • Identical outlined functions in cross-modules **not deduplicated**

• Frame-layout code tend to not get outlined
  • Generated frame-layout code is irregular
  • Typically optimized for performance
### No Outliner

| a.c: | int f1(int x) { |
|      |   // ...more code... |
|      |   return x * 128 + 77; |
|      | } |
|      | int f2(int x) { |
|      |   // ...more code... |
|      |   return x * 128 + 77; |
|      | } |
|      | int g(int x) { |
|      |   // ...more code... |
|      |   return x * 128 + 77; |
|      | } |

### Machine Outliner

| LTO | int f1(int x) { |
|     |   // ...more code... |
|     |   return __outlined(x); |
|     | } |
|     | int f2(int x) { |
|     |   // ...more code... |
|     |   return __outlined(x); |
|     | } |
|     | int g(int x) { |
|     |   // ...more code... |
|     |   return __outlined(x); |
|     | } |
| ThinLTO | int f1(int x) { |
|        |   // ...more code... |
|        |   return __outlined(x); |
|        | } |
|        | int f2(int x) { |
|        |   // ...more code... |
|        |   return __outlined(x); |
|        | } |
|        | int __outlined(int x) { |
|        |   return x * 128 + 77; |
|        | } |
Typical (Irregular) Frame Code for Speed

• Optimized to reduce # of instructions and micro-operations
  • SP adjustment once for CSR and/or local

• Instructions for handling LR (X30) often comes late in the prologue or early in the epilogue
  • Blocker for outliner

(Prologue)
stp x22, x21, [sp, #48]!
stp x20, x19, [sp, #16]
stp x29, x30, [sp, #32] // Can’t outline
add x29, sp, #32
...

(Epilogue)
ldp x29, x30, [sp, #32] // Can’t outline
ldp x20, x19, [sp, #16]
ldp x22, x11, [sp], #48
ret
Text Size Reduction with Machine Outliner for ThinLTO vs. LTO

- LLVM-TestSuite/CTMark (arm64/-Oz)
- ThinLTO outliners saves 8% code size while LTO does 11% code size.
Proposed Improvements

- Global Outliner in ThinLTO
  - Capture (stable) hashes of outlined functions for all modules
  - Make more outlines (but not folded) if a same hash sequence exists.
  - Realize code-size reduction via linker’s deduplication

- Frame code optimizations
  - Make frame code more homogeneous
  - Custom-outline frame code
Global Outliner in ThinLTO
Recall: ThinLTO

- Frontend compiler .o files in parallel
- After interprocedural analysis, runs in parallel for each module:
  - Opt (HIR)
    - Inlining/Optimizer
  - CodeGen (MIR)
    - RA/Machine Outliner
- Finally, traditional linking combines results
2-round CodeGen!

- Serialize IR just before 1\textsuperscript{st} CG
- Deserialize IR before 2\textsuperscript{nd} CG

1\textsuperscript{st} round:
- Gather MIR hashes of outlined functions

2\textsuperscript{nd} round:
- (Optimistically) outline more candidates that match MIR hashes

Linking:
- Fold outlined functions across modules
Build a Global Prefix Tree in First Round

- Recall: Machine outliner uses a **suffix tree** to find sequences occurring at least 2 times

- For each outlined function (within a module),
  - Hash the machine instruction using a stable hash below
  - Insert the sequence of hashes into a **global prefix tree**

- **Stable machine instruction hash (valid cross-modules)**
  - 64-bit, using stronger hash function
  - do not hash pointers, but deep meaningful value representations, e.g. names
  - hashes are *quite exact* across modules and (de)serializable.
Global prefix tree: Building (in First Round CG)

a.c:  int __outlined1(int x) {
    return x * 128 + 77;
}

    mov eax, DWORD PTR [rbp-4]
    sal eax, 7
    add eax, 77

int __outlined2(int x) {
    return x * 128 + 33;
}

    mov eax, DWORD PTR [rbp-4]
    sal eax, 7
    add eax, 77
    add eax, 33
Global prefix tree: Hashing (in First Round CG)

a.c: int __outlined1(int x) {
    return x * 128 + 77;
}

mov eax, DWORD PTR [rbp-4] // Y
sal eax, 7
add eax, 77 // U

int __outlined2(int x) {
    return x * 128 + 33;
}

mov eax, DWORD PTR [rbp-4] // Y
sal eax, 7
add eax, 33 // Q

Stable Hashes (actual hashes are 64-bit)
Outlining More in Second Round CG

1) For an outlining candidate (whose sequence occurring at least 2 times)
   • Check if the sequences occur in the global prefix tree.
   • Adjust cost to 0 since it’s been already paid in other module.

2) For sequence occurring only once in a module
   • Iterate instruction sequences to see if there is a match in the tree.
   • If so, optimistically outline such a singleton sequence. (see next slides)
Global prefix tree: Using for matching

b.c:

... mov DWORD PTR [rbp-8], eax  // H
mov eax, DWORD PTR [rbp-4]  // Y
sal eax, 7  // B
add eax, 77  // U
add eax, 33  // R
mov DWORD PTR [rbp-8], eax  // A
...

```
```
Global prefix tree: Using for matching

b.c:

```assembly
... mov DWORD PTR [rbp-8], eax // H mov eax, DWORD PTR [rbp-4] // Y sal eax, 7 // B add eax, 77 // U add eax, 33 // R mov DWORD PTR [rbp-8], eax // A ...
```

![Global prefix tree diagram](image-url)
Global prefix tree: Using for matching

b.c:

...  
mov DWORD PTR [rbp-8], eax // H
**mov eax, DWORD PTR [rbp-4] // Y**
sal eax, 7 // B
add eax, 77 // U
add eax, 33 // R
mov DWORD PTR [rbp-8], eax // A

...
Global prefix tree: Using for matching

b.c:

... 

mov DWORD PTR [rbp-8], eax  // H

mov eax, DWORD PTR [rbp-4]  // Y

sal eax, 7                 // B

add eax, 77                // U

add eax, 33                // R

mov DWORD PTR [rbp-8], eax  // A

...
Global prefix tree: Using for matching

b.c:

...  
mov DWORD PTR [rbp-8], eax // H  
mov eax, DWORD PTR [rbp-4] // Y  
sal eax, 7 // B  
add eax, 77 // U  
add eax, 33 // R  
mov DWORD PTR [rbp-8], eax // A  
...

We found a match... Outline this sequence!
Global prefix tree: Using for matching

b.c:

... 

mov DWORD PTR [rbp-8], eax  // H
mov eax, DWORD PTR [rbp-4]  // Y
sal eax, 7                  // B
add eax, 77                 // U
add eax, 33                 // R
mov DWORD PTR [rbp-8], eax  // A

...
Global prefix tree: Using for matching

b.c:

...  

mov DWORD PTR [rbp-8], eax // H  
mov eax, DWORD PTR [rbp-4]    // Y  
sal eax, 7                   // B  
add eax, 77                  // U  
add eax, 33                  // R  

mov DWORD PTR [rbp-8], eax   // A  

...
Actually...

```
int f1(int x) {
    // ...more code...
    return x * 128 + 77;
}

int f2(int x) {
    // ...more code...
    return x * 128 + 77;
}

int g(int x) {
    // ...more code...
    return x * 128 + 77;
}
```

ThinLTO with 2-round CodeGen

```
int f1(int x) {
    // ...more code...
    return __outlined1(x);
}

int f2(int x) {
    // ...more code...
    return __outlined1(x);
}

int g(int x) {
    // ...more code...
    return __outlined2(x);
}

int __outlined1(int x) {
    return x * 128 + 77;
}

int __outlined2(int x) {
    return x * 128 + 77;
}
```
Outlined Function Deduplication

• Soundness in the presence of hash collision
  • Hashes only used to determine which outlined functions to create in module
  • Introduce unique names for outlined functions across modules by attaching
    • Module Id
    • Hash of machine instructions of outlined function
  • Enable link-once ODR to let the linker deduplicate functions

• Support for further outlining of outlined functions
  • Relevant when running machine outliner multiple times (in each CodeGen)
  • When hashing call, use hash of outlined functions only (not full unique name)
  • This enables more matching in global prefix tree!
Frame Code Optimizations

with examples for AArch64/iOS
Homogeneous Frame Code for Size

• Prologue
  • Start with FP/LR save
  • SP pre-decrement by 16 byte in order while saving CSR
  • Explicit FP(X29) setting
  • Local allocation

• Epilogue
  • Local deallocation
  • SP post-increment by 16 byte in order while restoring CSR
  • End with FP/LR restore

(Prologue)
  stp x29, x30, [sp, #−16]!
  stp x20, x19, [sp, #−16]!
  stp x22, x21, [sp, #−16]!
  add x29, sp, #32
  ...

(Epilogue)
  ldp x22, x21, [sp], #16
  ldp x20, x19, [sp], #16
  ldp x29, x30, [sp], #16
  ret
Custom-Outlined Frame Code Helpers

• Synthesized helpers by compiler
  • Eagerly populate possible helpers in each module pass
  • Unique naming with LinkOnce-ODR to deduplicate helpers by linker
• Unwind code is still in place at each prologue site.

(Prologue)
stp x29, x30, [sp, #16]!
bl _PROLOG_INTEGER_19202122
add x29, sp, #32
...

(Epilogue)
bl _EPILOG_INTEGER_21221920
ldp x29, x30, [sp], #16
ret
Optimizing Epilogue – Outlining FP/LR Restore

• Touching LR is tricky in outliner
• Use a scratch register, X16 to stash/restore LR value to the context of epilogue.
• Useful for a tail-call epilogue that a direct branch follows.

(Epilogue)
bl _EPILOG_INTEGER_21221920LRFP
ret

(Helpers)
_EPILOG_INTEGER_21221920LRFP:
mov x16, x30 // Save LR of epilogue to X16
ldp x22, x21, [sp], #16
ldp x20, x19, [sp], #16
ldp x29, x30, [sp], #16 // Restore LR (of caller)
br x16 // Jump on X16 back to epilogue
Optimizing Epilogue - Tail-Call Helper

• Function *return* is folded into the helper

• Branch (B) instead of Call (BL) at epilogue

• Return to the original caller from the helper

• Ideally, helpers can be merged at different offsets for further saving

(Epilogue)

b _EPILOG_INTEGER_21221920LRFP_TAIL

(Helper)

_EPILOG_INTEGER_21221920LRFP_TAIL:

ldp x22, x21, [sp], #16
ldp x20, x19, [sp], #16
ldp x29, x30, [sp], #16
ret
Evaluation
Global/FrameOpt Outliners with ThinLTO

- Global outliner saves 11%, which is already on par with LTO
- Global outliner + FrameOpt saves up to 15% on average.
LinkTime (ThinLTO + Linking)

- Link time slowdown is 1.5X on average.
- Caused by the repeated code gen and more deduplications
- Still, a fraction of LTO build time.
Evaluation with Some Large Applications

• Number of outlined instruction sequences almost doubles for large internal benchmark

• Total build time (compilation + link) is within ~5% overall wall-time overhead for large internal benchmark

• Even measured performance improved due to page faults reductions.
Future work

• Alternatives to running CodeGen twice
  • Persist hashes, re-use in later builds
  • Trading effectiveness for improved build times

• Build global suffix tree
  • Capture still missed opportunities that are not beneficial in any single module

• Make MIR fully (de)serializable
  • Save the time running the first part of codegen twice

• Avoid generating identical outlined functions
  • That then need to get folded by linker