使用 Global Machine Outliner 缩减重复代码

美团

宋旭陶/靛青

2021/6/19

开始之前

```
int calc_1(int a, int b) {
    int x = a + b;
    int y = a * x;
    return add(x, y);
}

int calc_2(int a, int b) {
    int x = a - b;
    int y = a * x;
    return add(x, y);
}
```

```
int outlined_function(int a, int x) {
    int y = a * x;
    return add(x, y);
}

int calc_1(int a, int b) {
    int x = a + b;
    return outlined_function(a, x);
}

int calc_2(int a, int b) {
    int x = a - b;
    return outlined_function(a, x);
}
```

Machine Outliner 是什么？

重复代码的合并

```c
int calc_1(int a, int b) {
    int x = a + b;
    int y = a * x;
    return add(x, y);
}

int calc_2(int a, int b) {
    int x = a - b;
    int y = a * x;
    return add(x, y);
}
```

```
<_calc_1>:
  add  w8, w1, w0
  mul  w1, w8, w0
  mov  x0, x8
  b  0x100003f48 <_add>

<_calc_2>:
  sub  w8, w0, w1
  mul  w1, w8, w0
  mov  x0, x8
  b  0x100003f48 <_add>
```

clang -Os cal_12.c -c -o cal_12.o

重复代码的合并

```
<_calc_1>:
  add  w8, w1, w0
  mul  w1, w8, w0
  mov  x0, x8
  b  0x100003f48 <_add>

<_calc_2>:
  sub  w8, w0, w1
  mul  w1, w8, w0
  mov  x0, x8
  b  0x100003f48 <_add>
```

```
<_calc_1>:
  add  w8, w1, w0
  b  0x100003f64 <_OUTLINED_FUNCTION_0>

<_calc_2>:
  sub  w8, w0, w1
  b  0x100003f64 <_OUTLINED_FUNCTION_0>

<_OUTLINED_FUNCTION_0>:
  mul  w1, w8, w0
  mov  x0, x8
  b  0x100003f4c <_add>
```

clang –Os **-moutline** cal_12.c -c -o cal_12.o

Machine Outliner

Outline
与 Inline 相反，Machine Outliner 作为一个 LLVM Pass，**将重复的指令序列替换为一个合并的函数调用**。

现实中有很多重复的指令序列产生，比如引用计数相关调用，我们 Ctrl+C Ctrl+V 的占比很少。

```
<_calc_1>:
  add  w8, w1, w0
  b  0x100003f64 <_OUTLINED_FUNCTION_0>

<_calc_2>:
  sub  w8, w0, w1
  b  0x100003f64 <_OUTLINED_FUNCTION_0>

<_OUTLINED_FUNCTION_0>:
  mul  w1, w8, w0
  mov  x0, x8
  b  0x100003f4c <_add>
```

找到这些重复指令序列

```
<_calc_1>:
  add  w8, w1, w0
  mul  w1, w8, w0
  mov  x0, x8
  b  0x100003f48 <_add>

<_calc_2>:
  sub  w8, w0, w1
  mul  w1, w8, w0
  mov  x0, x8
  b  0x100003f48 <_add>
```

calc_1:
A
B
C
D
calc_2:
E
B
C
D

1. 每个指令（包含操作数）当作一个字符
2. 一个函数就是个指令序列 -> 字符串

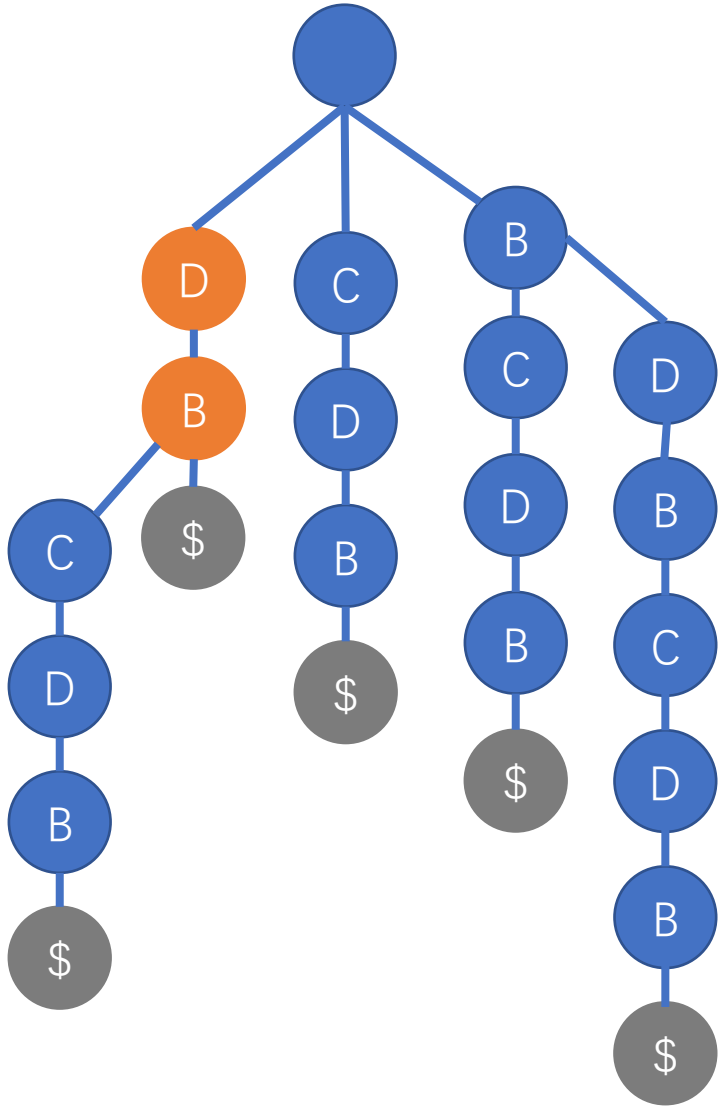
💡 寻找重复子串

Suffix Tree 后缀树

查找 BDBCDB

BDBCDB$
DBCDB$
BCDB$
CDB$
DB$
~~B$~~

每个字符都会出现在元素的首位，
有重复字串出现，一定在多个字串的公共前缀中



屏蔽多个函数相连

calc_1:
A
B
C
D
calc_2:
E
B
C
D

calc_x:
F
B
C
D
E
C

A B C D '$0' E B C D '$1' F B C D E C '$2'

每个函数尾创建全局唯一 “字符”

-moutline 和 -Oz

- -Oz 为二进制大小优化考虑，自带了 –moutline pass 的传递

🤔

那么直接在 Xcode 上打开 –Oz 优化或者在 clang 上增加 –moutline 参数？

话题结束？

编译多个文件

```c
// cal_12.c

int calc_1(int a, int b) {
    int x = a + b;
    int y = a * x;
    return add(x, y);
}

int calc_2(int a, int b) {
    int x = a - b;
    int y = a * x;
    return add(x, y);
}
```

```c
// cal_34.c

int calc_3(int a, int b) {
    int x = a / b;
    int y = a * x;
    return add(x, y);
}

int calc_4(int a, int b) {
    int x = a * b;
    int y = a * x;
    return add(x, y);
}
```

```
0000000100003f28 <_calc_1>:
100003f28:       add  w8, w1, w0
100003f2c:       b  0x100003f38 <_OUTLINED_FUNCTION_0>

0000000100003f30 <_calc_2>:
100003f30:       sub  w8, w0, w1
100003f34:       b  0x100003f38 <_OUTLINED_FUNCTION_0>

0000000100003f38 <_OUTLINED_FUNCTION_0>:
100003f38:       mul  w1, w8, w0
100003f3c:       mov  x0, x8
100003f40:       b  0x100003f20 <_add>

0000000100003f44 <_calc_3>:
100003f44:       sdiv  w8, w0, w1
100003f48:       b  0x100003f54 <_OUTLINED_FUNCTION_0>

0000000100003f4c <_calc_4>:
100003f4c:       mul  w8, w1, w0
100003f50:       b  0x100003f54 <_OUTLINED_FUNCTION_0>

0000000100003f54 <_OUTLINED_FUNCTION_0>:
100003f54:       mul  w1, w8, w0
100003f58:       mov  x0, x8
100003f5c:       b  0x100003f20 <_add>
```

💢

相同的 OUTLINED_FUNCTION_0 没有合并！

clang -Os -moutline main.c cal_12.c cal_34.c -o main

Uber 分享做了什么？

https://eng.uber.com/how-uber-deals-with-large-ios-app-size/

编译到可执行文件的流程

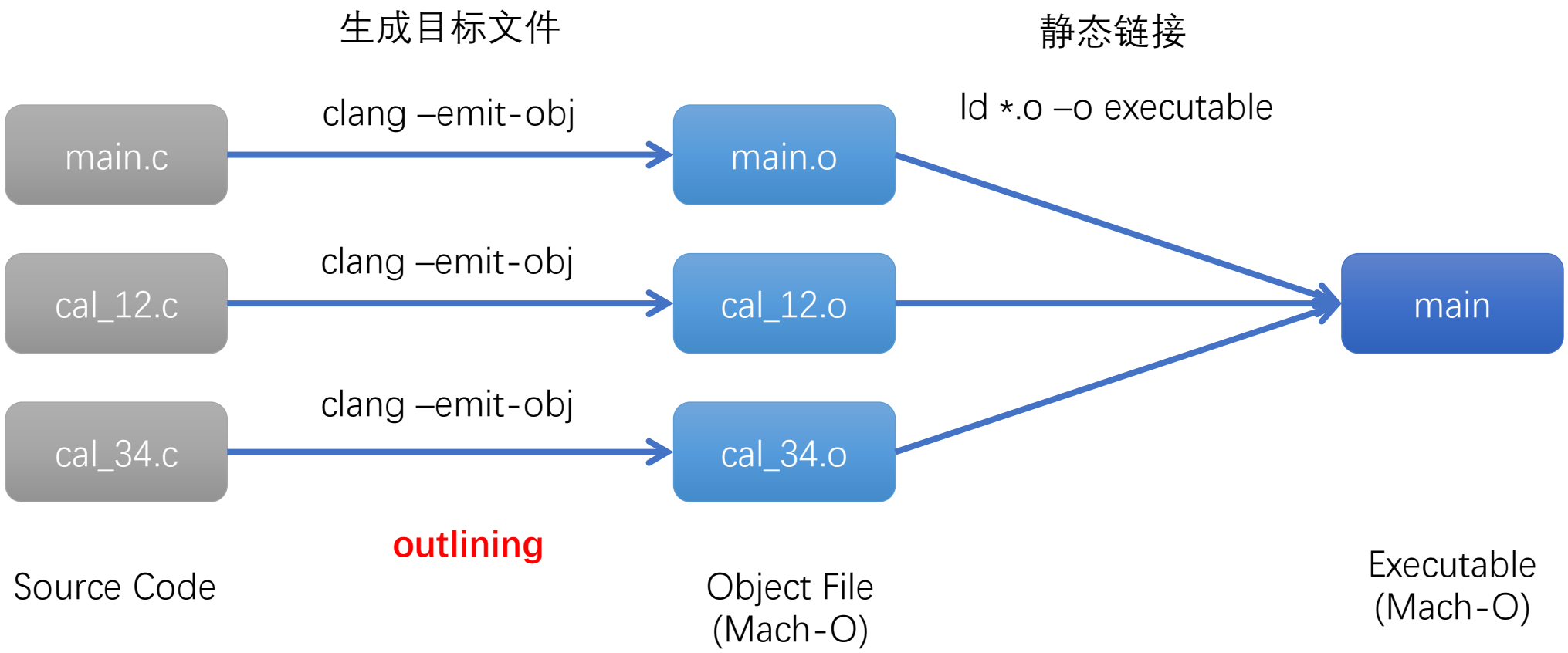
clang -Os -moutline main.c cal_12.c cal_34.c -o main **-###**

-### Print (but do not run) the commands to run for this compilation

1. clang –cc1 –emit-obj –o main.o main.c
2. clang –cc1 –emit-obj –o cal_12.o cal_12.c
3. clang –cc1 -emit-obj –o cal_34.o cal_34.c
4. ld main.o cal_12.o cal_34.o –o main

编译到可执行文件的流程

clang -Os -moutline main.c cal_12.c cal_34.c -o main **-###**



生成目标文件流程

LLVM intermediate representation（“**LLVM IR**”）有两种表达形式：
- .bc LLVM Bitcode
- .ll human-readable LLVM assembly language

```
clang -Os cal_12.c -c -emit-llvm -o cal_12.bc
clang -Os cal_12.c -S -emit-llvm -o cal_12.ll
llvm-as cal_12.ll -o cal_12.bc
llvm-dis cal_12.bc -o cal_12.ll
```

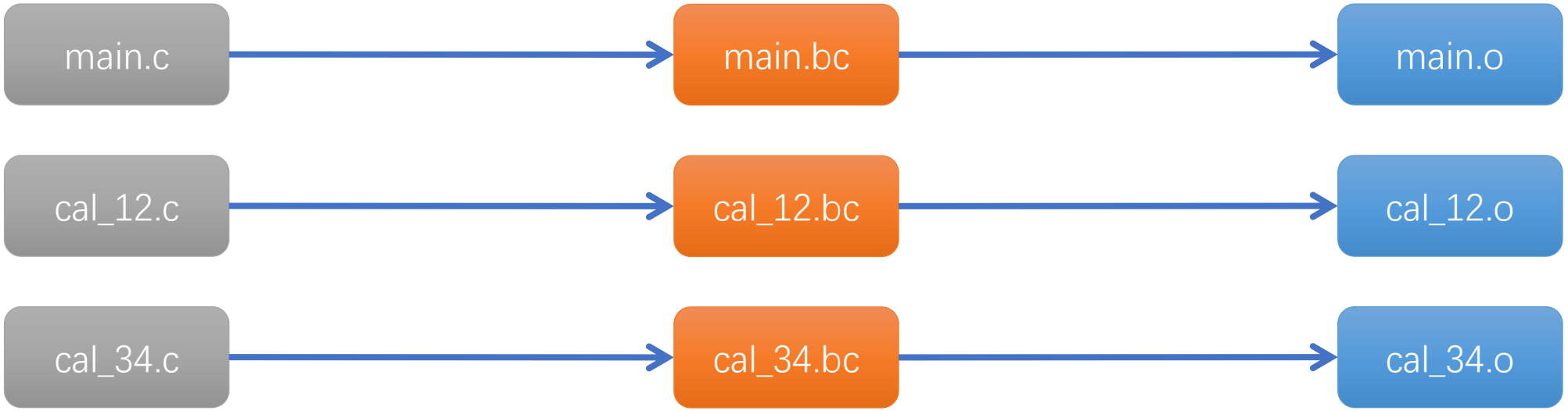
```
; ModuleID = 'cal_12.c'
source_filename = "cal_12.c"
target datalayout = "e-m:o-i64:64-i12
target triple = "arm64-apple-macosx1

; Function Attrs: nounwind optsize ssp
define i32 @calc_1(i32 %0, i32 %1) local
  %3 = add nsw i32 %1, %0
  %4 = mul nsw i32 %3, %0
  %5 = tail call i32 @add(i32 %3, i32 %4)
  ret i32 %5
}
...
```

outlining

compiler

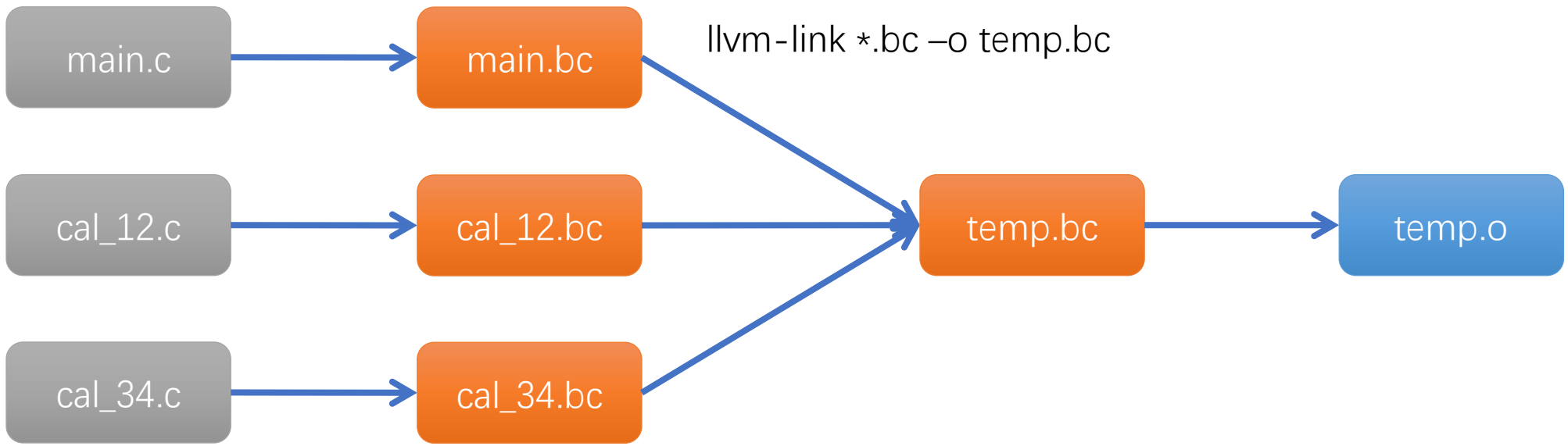
assember

合并 IR


main.c
main.bc
main.o
cal_12.c
cal_12.bc
cal_12.o
cal_34.c
cal_34.bc
cal_34.o

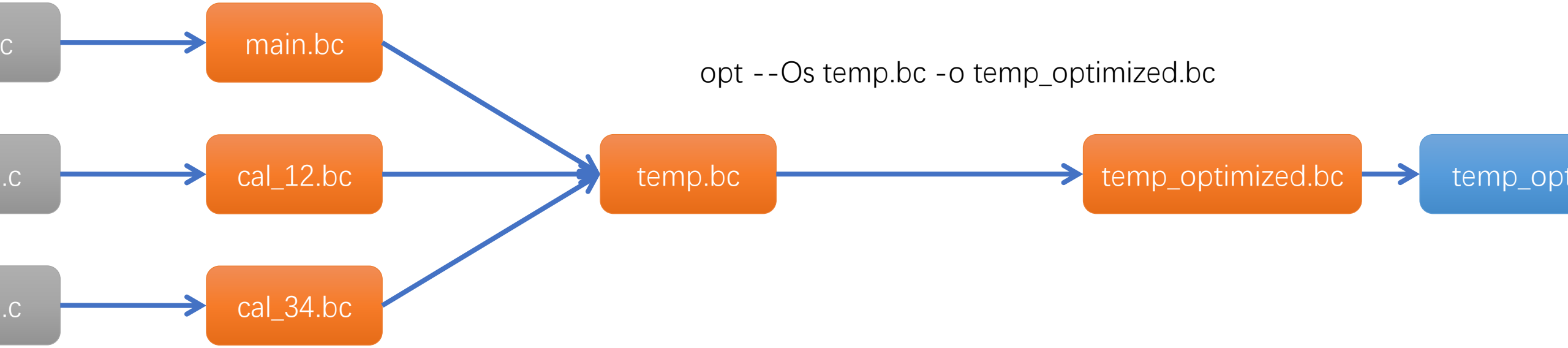
合并 IR

llvm-link - LLVM bitcode linker

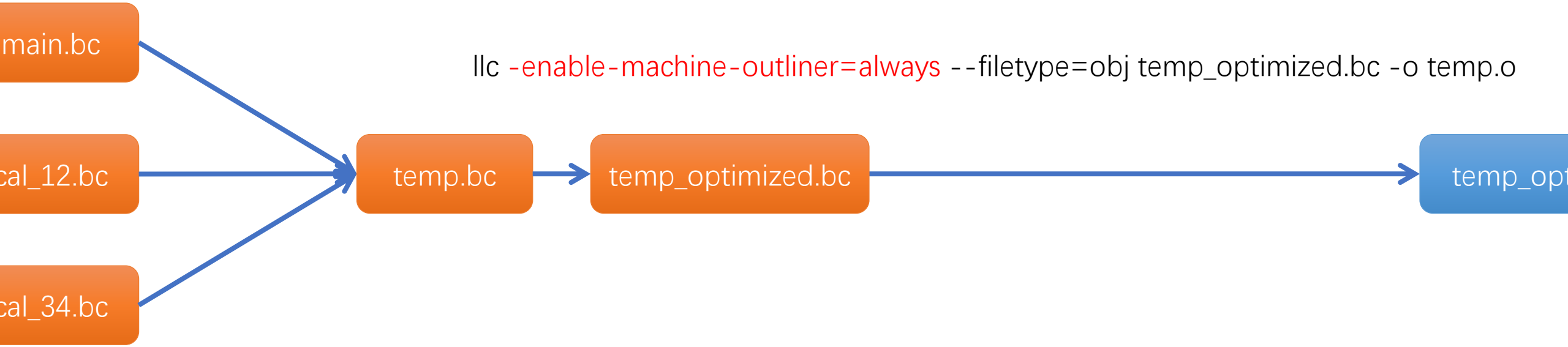


clang *.c -c -emit-llvm -o *.bc

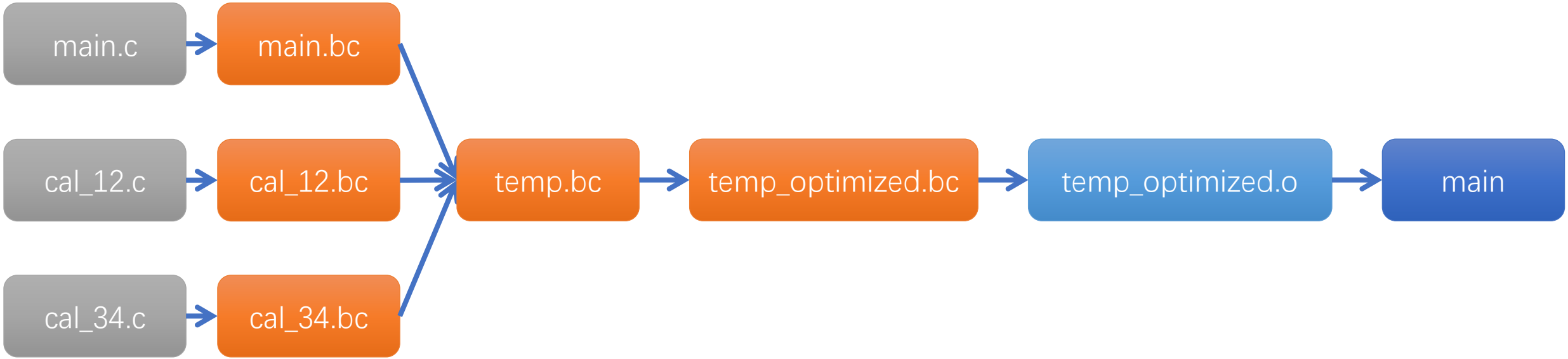
opt - LLVM optimizer



llc - LLVM static compiler



完整构建流程

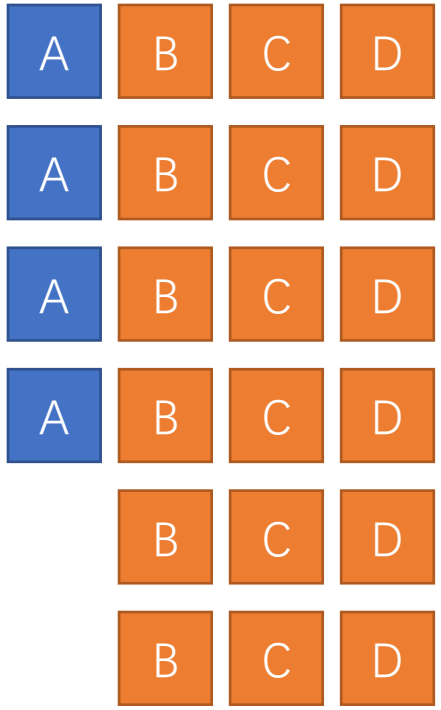

main.c
main.bc
cal_12.c
cal_12.bc
cal_34.c
cal_34.bc
temp.bc
temp_optimized.bc
temp_optimized.o
main
compiler ir
llvm-link
opt
llc
ld

完整命令

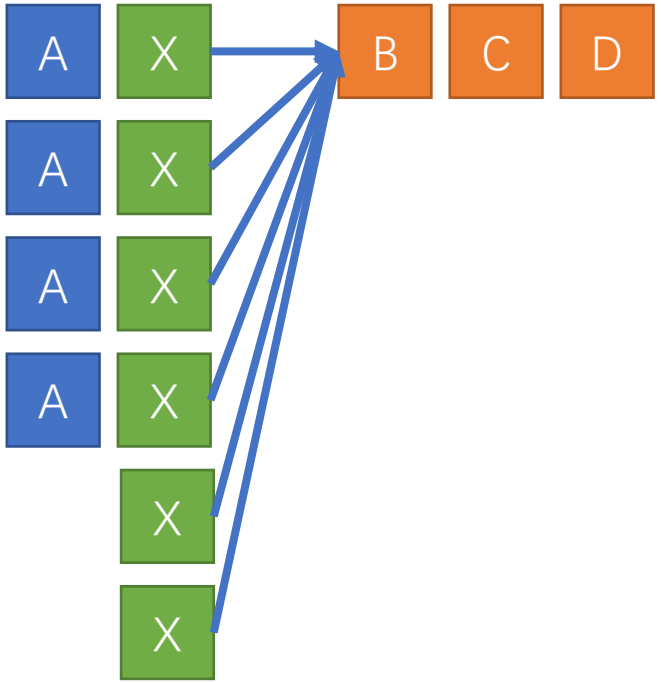
```
clang -Xclang -disable-llvm-passes -Os \
main.c -c -emit-llvm -o main.bc
clang -Xclang -disable-llvm-passes -Os \
cal_12.c -c -emit-llvm -o cal_12.bc
clang -Xclang -disable-llvm-passes -Os \
cal_34.c -c -emit-llvm -o cal_34.bc

llvm-link main.bc cal_12.bc cal_34.bc -o temp.bc

opt --Os -objc-arc-contract temp.bc -o temp_optimized.bc

llc -enable-machine-outliner=always --filetype=obj temp_optimized.bc -o temp_optimized.o

ld -arch arm64 -platform_version macos 11.0.0 11.3 –lSystem temp_optimized.o -o main
```

```
0000000100003f3c <_calc_1>:
100003f3c:      add w8, w1, w0
100003f40:      b   0x100003f5c <_OUTLINED_FUNCTION_0>

0000000100003f44 <_calc_2>:
100003f44:      sub w8, w0, w1
100003f48:      b   0x100003f5c <_OUTLINED_FUNCTION_0>

0000000100003f4c <_calc_3>:
100003f4c:      sdiv    w8, w0, w1
100003f50:      b   0x100003f5c <_OUTLINED_FUNCTION_0>

0000000100003f54 <_calc_4>:
100003f54:      mul w8, w1, w0
100003f58:      b   0x100003f5c <_OUTLINED_FUNCTION_0>

0000000100003f5c <_OUTLINED_FUNCTION_0>:
100003f5c:      mul w1, w8, w0
100003f60:      mov x0, x8
100003f64:      b   0x100003f34 <_add>
```

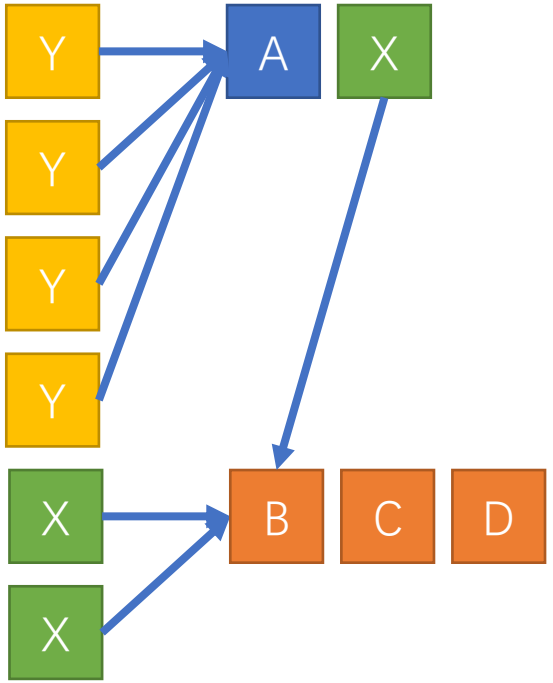
一次效果不够好再来一次 Outline？



21 个指令



13 个指令



11 个指令

我们该如何应用？

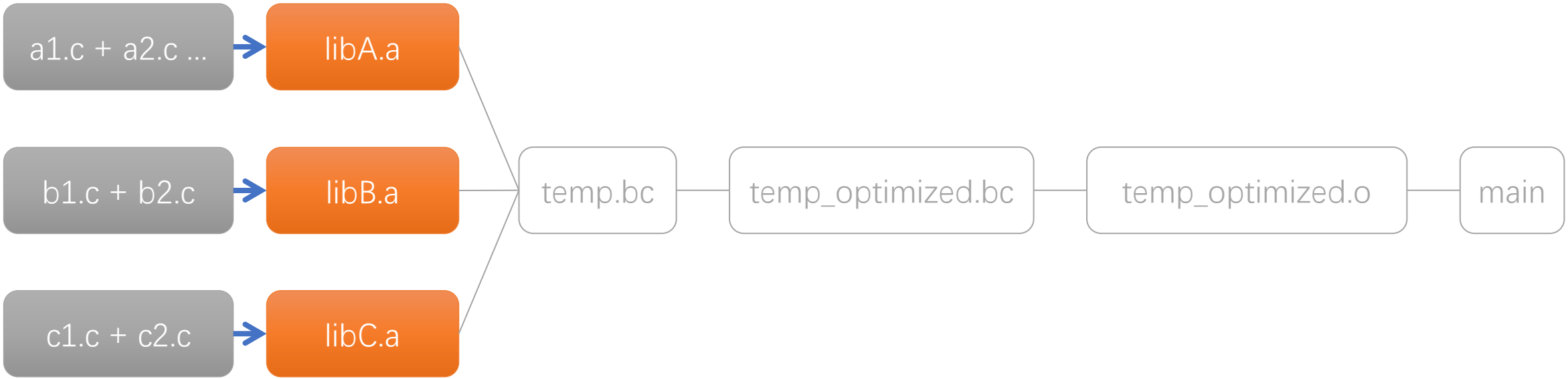
基于 Xcode 改造工程？

Build Settings

CC – 修改编译 C/Objective-C 使用的命令行工具
CCPLUSPLUS – 修改编译 C++/Objective-C++ 使用的命令行工具
LD/LDPLUSPLUS – 修改静态链接使用的命令行工具
LIBTOOL – 修改创建 .a 使用的命令行工具
OTHER_CFLAGS – 编译 C/Objective-C 额外添加的参数，传递给 CC

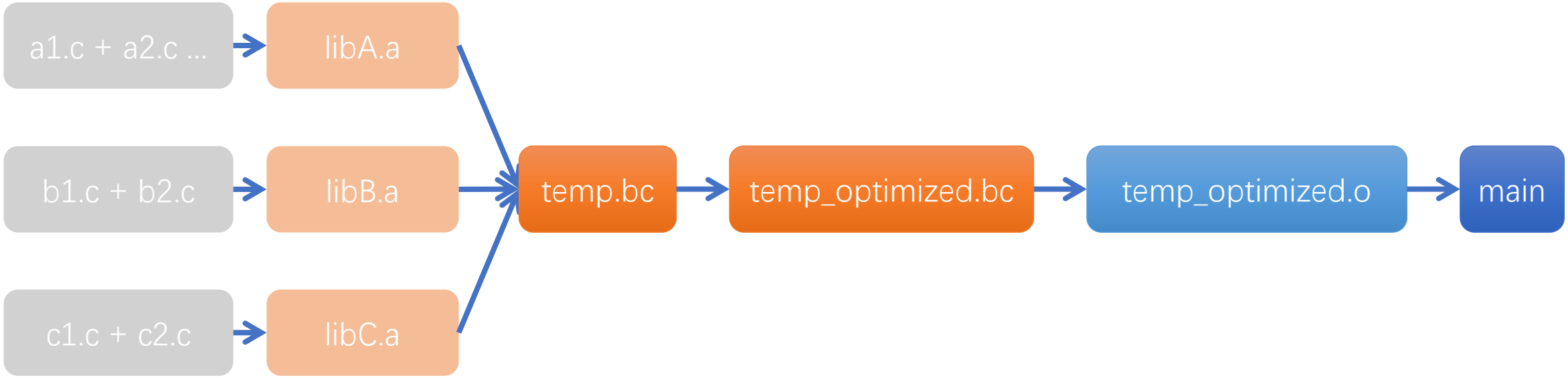
基于 Xcode 改造工程？

使用 CC/OTHER_C_FLAGS 添加 -emit-llvm 等参数，生成 Object File 改为 LLVM IR
使用 LIBTOOL 创建自定义生成 .a 的逻辑：llvm-link 合并组件内的 IR



基于 Xcode 改造工程？

使用 LD 配置，修改静态链接流程：
1. 使用 llvm-link 合并所有的 lib*.a
2. 使用 opt 优化 IR
3. 使用 llc 完成 outline 并生成 Object File
4. 回到原本的链接流程



Link Time Optimization - LTO

链接时优化（LTO）：

借助静态链接可以获取程序全局信息的机会，做一些全局优化，这样优化可以提高运行时的性能，并进一步减少二进制的大小。

静态链接在 LTO 的流程

1. **判断所有输入的文件（.o）的类型，找到所有 LLVM Bitcode 文件**
2. 解决符号引用关系
3. **合并所有的 Bitcode 为一个 Bitcode，使用 LLVM Pass 优化 Bitcode 并生成 Object File**
4. 消除无用代码等优化

😈 LLVM Bitcode 是 LLVM IR 的一种表示形式
😈 只要输入的文件有 Bitcode，就会有 LTO 流程！
😈 开启 LTO 的真正操作是编译 Object File 变成 IR Bitcode
😈 打开 LLVM_LTO 选项将为 clang 添加 –flto 参数

借助 LTO 完成 Global Machine Outliner

- 打开 LTO：
  - C 系语言 – 设置 **LLVM_LTO** 为 true
  - Swift – 在 OTHER_SWIFT_FLAGS 添加 **-Xfrontend -lto=llvm-full -Xfrontend -emit-bc**
- 在 ld 添加 outline 流程
  - **-Wl,-mllvm,--enable-machine-outliner=always,-mllvm,--machine-outliner-reruns=N**

💡 N: 重复 outlining 的次数
💡 -Wl: 表示 clang 传递给 ld 的参数

总结

- Machine Outliner 通过合并重复指令序列减少指令个数
- 通过 --machine-outliner-reruns 参数指定重复 Outline 的次数可以进一步减少指令个数
- Machine Outliner 在 IR 转换为机器码的过程中执行，为了实现全局 Outline 需要合并所有 IR 为一个大 IR
- LTO 完美匹配了全局 Global Machine Outliner 需要的条件

PS

- Uber 为我们带来了重复 outline 的能力，目前 Xcode 12.5 已经集成了全部功能
- 需要 outline 的组件需要使用源码编译并开启 LTO 选项
- 构建时间真的很长
- 由于增加了调用，实际应用可能存在性能影响
- 不建议用在小型 App 上

参考

- How Uber Deals with Large iOS App Size
- 2016 LLVM Developers' Meeting: J. Paquette "Reducing Code Size Using Outlining"
- 2019 LLVM Developers' Meeting: J. Lin "Link Time Optimization For Swift"
- Reducing code size with LLVM Machine Outliner on 32-bit Arm targets
- LLVM Link Time Optimization: Design and Implementation

上手试试？

https://github.com/DianQK/LTOGlobalMachineOutliner

Thanks