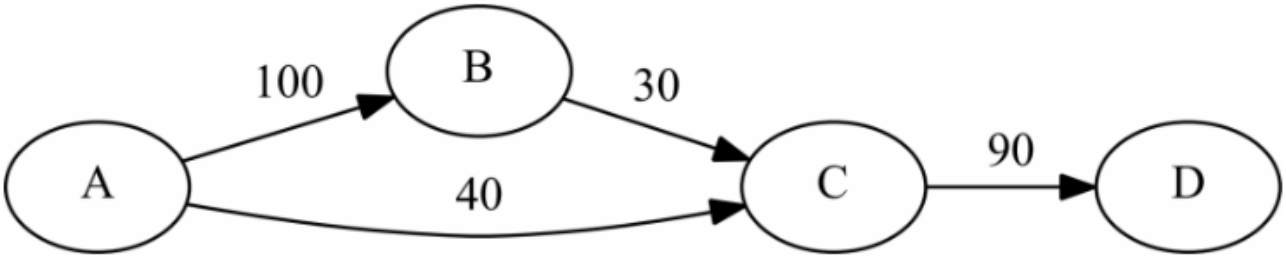
编译器优化那些事儿（14）：函数重排

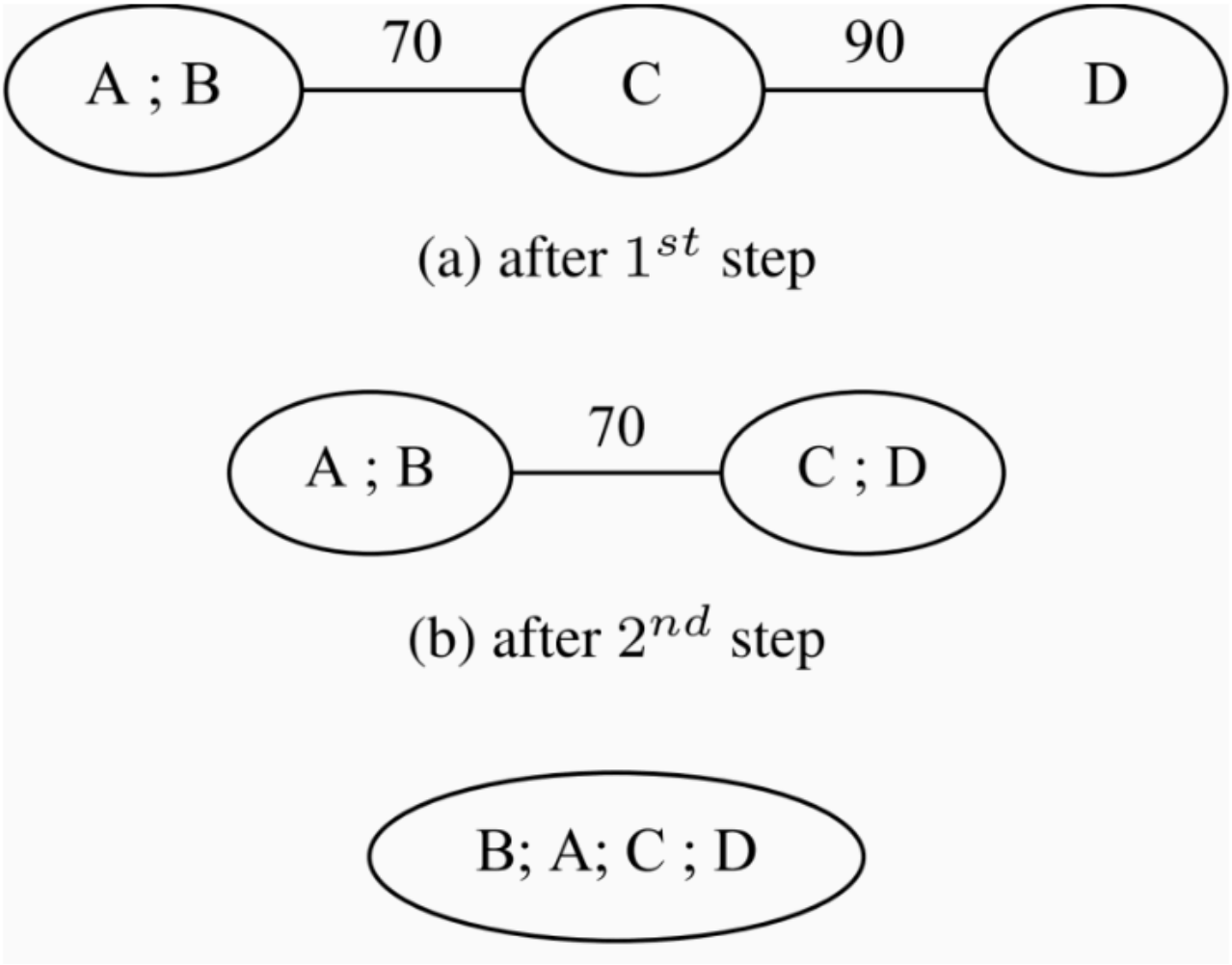
我们知道程序中函数可以通过添加 “-ffunction-sections + 链接脚本”进行代码段函数手动重排。然而函数在没有 Profile 的情况下自动重排并不理想，当前不开启 PGO 时，函数在代码段中的排序是由函数在源文件中定义的决定。当开启 PGO 时，由于编译器在第二次编译时通过 Profile 信息能够得到当前源文件函数的调用关系，能够将函数按照调用顺序进行全局排序，本文介绍PH算法及C3算法在LLVM中的实现。

PH(Pettis-Hansen)算法

PH(Pettis-Hansen)算法是基于加权动态调用图的启发式算法，按权重递减的顺序处理图中的每个边。



在每个步骤中，PH算法将边缘连接的两个节点合并在一起考虑。当两个节点合并时，它们对剩余节点的边缘被合并，其权重被增加。



(1)PH处理最重的边缘 A->B，合并节点A和B，得到图(a)中的图，

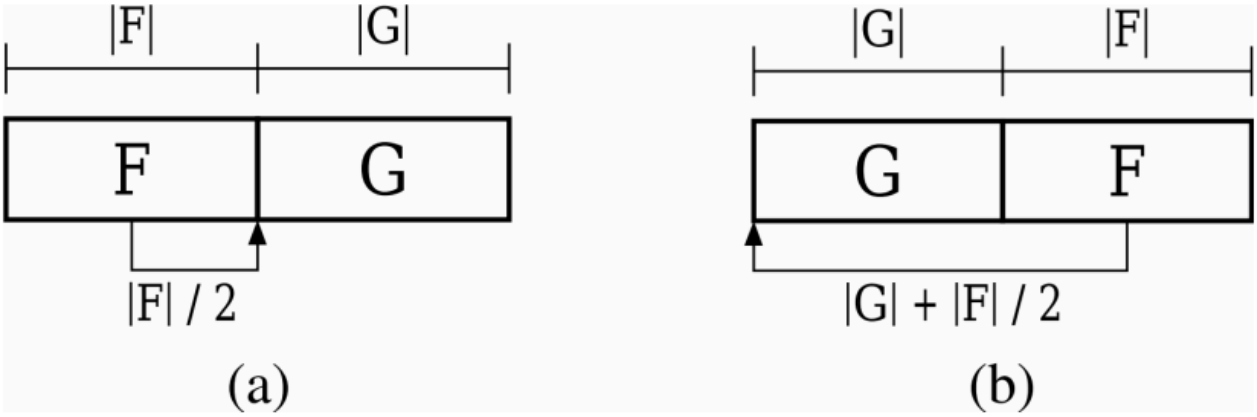
(2)选择图 (a)中连接C和D的最重边。

(3)分析原始图中的边并选择使A和C相邻，因为它们由具有最重的边连接。实现这一决策后，在进行最终合并之前，合并节点 A;B 中的节点被反转。最终顺序如图(c)所示。

C3 (Call Chain Clustering) 算法

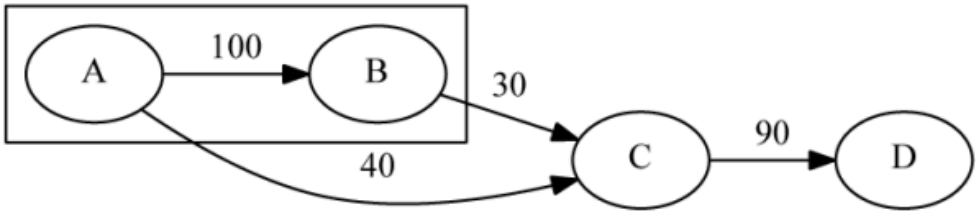
C3(Call Chain Clustering)算法是一种新的基于调用图的启发式调用链聚类算法，使用有向调用图，并且每个点都考虑了函数作为调用者跟被调用者的角色。C3算法的关键点是考虑了调用关系及函数大小。

如下图所示，用|F|表示函数F的大小，F中任何指令从F的条目到地址空间的平均距离为 |F|/2 。

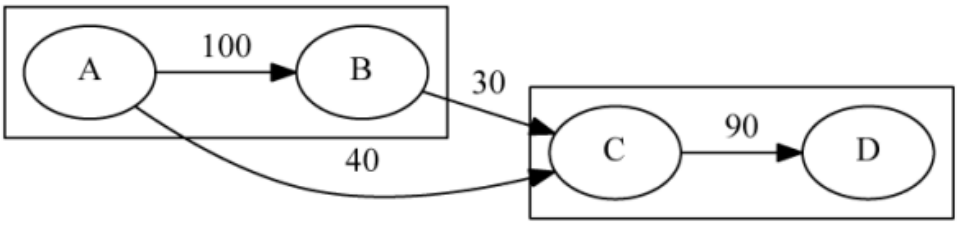


如上图(a)从F的输入到对G的调用在F内的地址空间的平均调用距离为|F|/2。距离越大，局部性越差；越过缓存线或者页面边界的可能性越大。

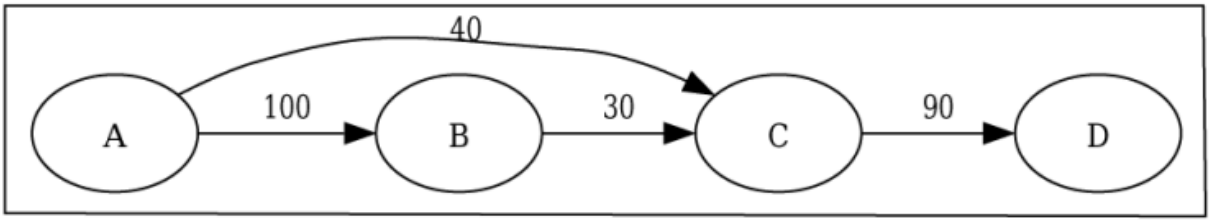
而上图(b)中从函数F到G的调用距离需要加上G的体积，故平均调用距离为|G| + |F|/2。



(a) after 1^{st} cluster merge



(b) after 2^{nd} cluster merge



(c) after 3^{rd} and final cluster merge

C3算法主要步骤为：

(1)首先将每个函数都单独放在一个 Cluster 中。

(2)然后，当处理每个函数时，它的 Cluster 会附加到包含调用图中最可能的前一个函数的 Cluster 中。目的是希望将函数尽可能靠近其最常用的调用方，并且我们按照程序中最热函数到最冷函数的优先级来执行。

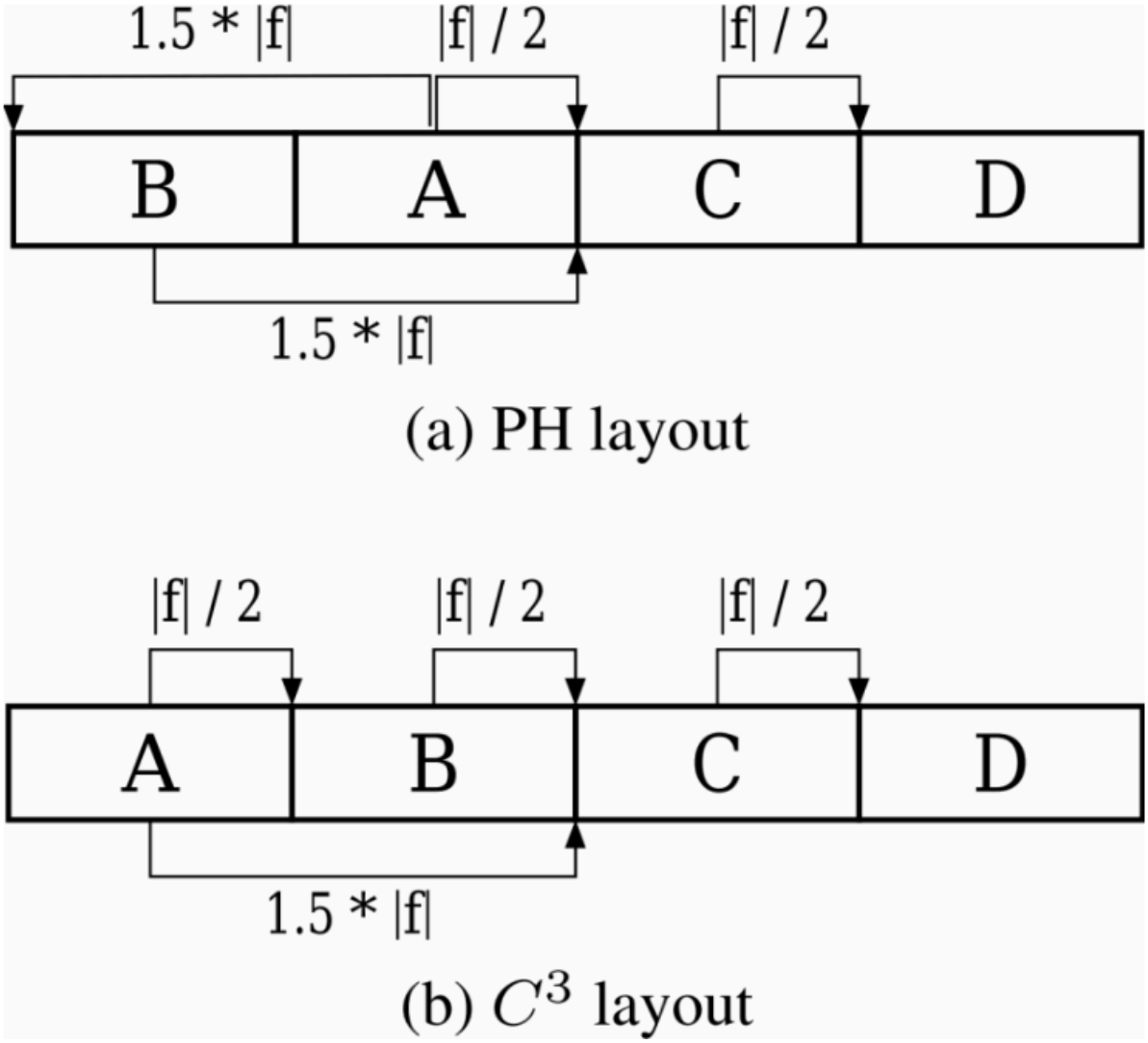
(3)通过遵循这个顺序，C3有效地将较热的函数进行优先级排序，从而可以将其放在其首选的前一个函数旁边。

(4)当其中一个 Cluster 大于合并阈值时则阻止这两个 Cluster 合并。C3使用的合并阈值是页大小，因为超出此限制，进一步增加 Cluster 的大小将不会带来好处：过大的话，无法容纳在同一个 cacheline 或内存页中。

(5)C3的最后一步是排序最终的 Cluster 。按照密度度量递减顺序对 Cluster 进行排序。此度量是执行 Cluster 中所有函数所花费的总时间量(从 Cluster 中每个函数 inline_summaries 中time的总和)除以 Cluster 中所有函数的总大小(从 Cluster 中每个函数 inline_summaries 中size的总和)：

$$density(c) = \frac{time(c)}{size(c)}$$

虽然我们通过设置合并阈值来限制 Cluster 的大小，但是依然放在相邻的地址空间中。尽量将大部分执行时间打包在尽可能少的代码页中，以进一步提高局部性。与同样热但较小的函数相比，较大的热函数对缓存层次结构施加更大的压力。因此，优选后者将最小化覆盖大多数程序执行时间所需的高速缓存行或TLB页数。



PH算法与C3算法开销对比：

$$
\begin{aligned}
&\mathrm{cost}(\mathrm{PH}) = 100 * 1.5 * |f| + 40 * 0.5 * |f| + 30 * 1.5 * |f| + 90 * 0.5 * |f| \\
&\therefore \mathrm{cost}(\mathrm{PH}) = (150 + 20 + 45 + 45) * |f| = 260 * |f| \\
&\mathrm{cost}(C^3) = 100 * 0.5 * |f| + 40 * 1.5 * |f| + 30 * 0.5 * |f| + 90 * 0.5 * |f| \\
&\therefore \mathrm{cost}(C^3) = (50 + 60 + 15 + 45) * |f| = 170 * |f|
\end{aligned}
$$

传统的函数排序PH算法平均提高了应用程序2.6%的性能。实验评估表明，在PH算法的基础上，C3算法进一步提高了应用程序的性能，平均提高了2.9%。

通过以下代码示例，描述开启 PGO 后函数重排效果：

```
#include<stdio.h>
void __attribute__((noinline)) foo()
{
    printf("this is foo!\n");
}

void __attribute__((noinline)) bar()
{
    printf("this is bar!\n");
}
```

```
11.
12. int main()
13. {
14.     int a;
15.     scanf("%d", &a);
16.     if (a >= 0) {
17.         foo();
18.     } else {
19.         bar();
20.     }
21.     return 0;
22. }
```

通过使用反馈编译优化后，CGProfile 模块计算每个BB块中Call指令的profile，将父函数与子函数之间添加一条边并以当前BB块的profile计数作为权重，扫描所有函数之后 LLVM 添加 llvm.module.flags 如下：

```
1. !llvm.module.flags = !{!0}
2. !0 = !{i32 5, !"CG Profile", !1}
3. !1 = distinct !{!2, !3, !4, !5, !6}
4. !2 = !{ptr @foo, ptr @puts, i64 2}
5. !3 = !{ptr @bar, ptr @puts, i64 3}
6. !4 = !{ptr @main, ptr @__isoc99_scanf, i64 5}
7. !5 = !{ptr @main, ptr @foo, i64 2}
8. !6 = !{ptr @main, ptr @bar, i64 3}
```

在 CodeGen 时，将会向ELF文件发射一对符号索引及权重，汇编形式如下：

```
1. .cg_profile foo, puts, 2
2. .cg_profile bar, puts, 3
3. .cg_profile main, __isoc99_scanf, 5
4. .cg_profile main, foo, 2
5. .cg_profile main, bar, 3
```

在写入ELF文件时，这些文件将作为 (From, To, Weight) 元组放入 SHT_LLVM_CALL_GRAPH_PROFILE 类型的段，这里段名为 .llvm.call-graph-profile ，该段可以使用 llvm-readobj 解析:

```
1. $ llvm-readobj --cg-profile demo.o
2. File: demo.o
3. Format: elf64-x86-64
4. Arch: x86_64
5. AddressSize: 64bit
6. LoadName: <Not found>
7. CGProfile [
8.   CGProfileEntry {
9.     From: foo (6)
10.    To: puts (7)
11.    Weight: 2
12.  }
13.  CGProfileEntry {
14.    From: bar (8)
15.    To: puts (7)
16.    Weight: 3
17.  }
18.  CGProfileEntry {
19.    From: main (9)
20.    To: __isoc99_scanf (10)
```

```
    Weight: 5
  }
  CGProfileEntry {
    From: main (9)
    To: foo (6)
    Weight: 2
  }
  CGProfileEntry {
    From: main (9)
    To: bar (8)
    Weight: 3
  }
]
```

当 lld 链接器通过 readCallGraphsFromObjectFiles 函数读到 CGProfile 数据后，结合函数Size信息建立聚类，假如函数输出在同一段中则按照上述C3算法将函数重新排序。同时还提供选项 --call-graph-ordering-file 可以将函数调用链数据传入链接器进行排序。

总结

本文对LLVM中函数重排的算法和实现方案进行了简单介绍，通过反馈编译将函数运行时保存在IR上，在 CodeGen 阶段输出到 .cg_profile 中，然后将所有函数的 CallGraph 保存在obj文件中；链接器读取相关段后进行解析，结合函数Size信息进行C3算法函数排序，从而降低程序运行的时 iCache-miss 与 iTLB-miss 。

对于超大程序且整体运行流程较为固定的程序，函数重排优化将会获得明显的收益；而对于程序本身较小，或者外部调用较多的动态库，可能优化效果并不明显。此外函数重排算法还可以结合运行时 Time Profile 维度排序聚族、冷热BB块拆分后聚族等算法，进一步实现性能的提升。

参考

(1)K. Pettis and R. C. Hansen, "Profile guided code positioning", Proceedings of the ACM Conference on Programming Language Design and Implementation, pp. 16-27, 1990 : https://dl.acm.org/doi/pdf/10.1145/93548.93550

(2)G. Ottoni and B. Maher, “Optimizing function placement for largescale data-center applications” in 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2017, pp. 233–244 : https://dl.acm.org/doi/pdf/10.5555/3049832.3049858

(3)Profile Guided Function Layout in LLVM and LLD_simon: https://llvm.org/devmtg/2018-10/slides/Spencer-Profile%20Guided%20Function%20Layout%20in%20LLVM%20and%20LLD.pdf

(4)社区patch: https://reviews.llvm.org/D36351