OpenEuler x Compiler SIG

静态分支预测与基本块重排介绍

Compiler SIG

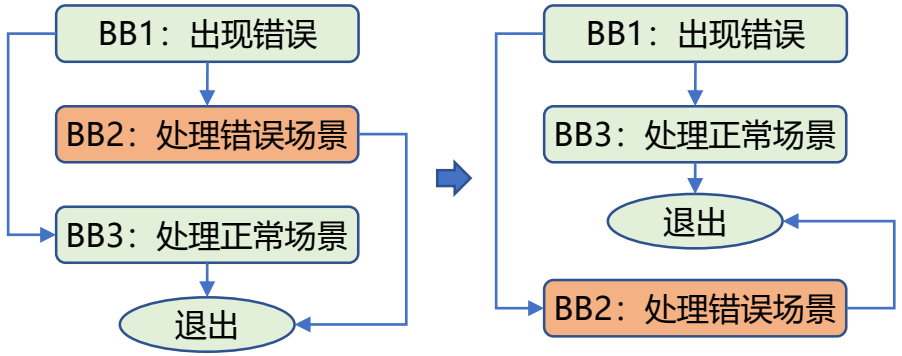
赵川峰

总述

OpenEuler Compiler SIG

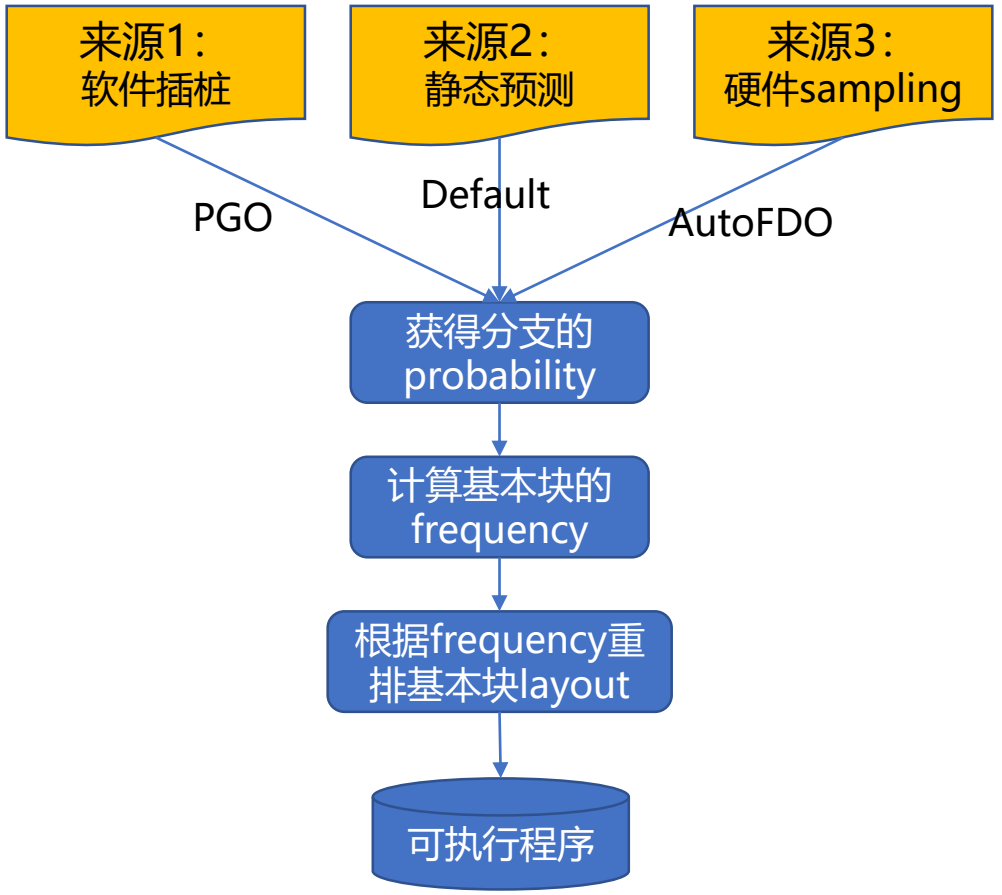
```
if (出现错误) then
   处理错误场景
处理正常场景
```

基本块排布（根据分支概率probability）



执行分支前执行尽量多的顺序代码；

- 顺序代码流减少了I-cache miss；
- 由于预测了正确的分支，减少了流水线的penalty；



软件层面解决方法

目录

基本概念

衡量方式有两种：

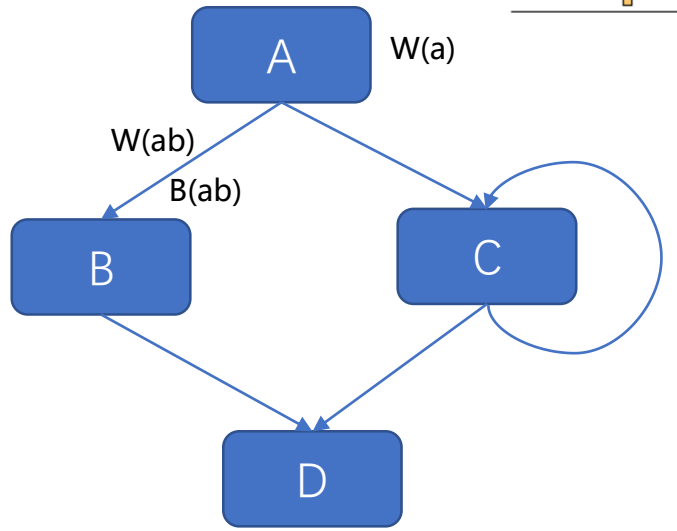
（1）基本块执行的次数；

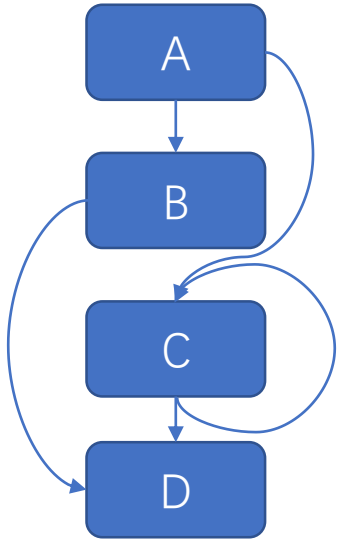
（2）一个基本块转移到另一个基本块的次数；

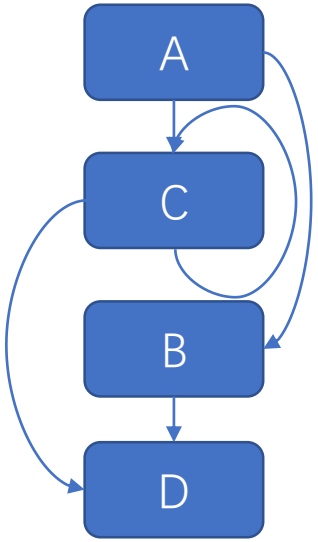
W(ab) —— 等于从基本块A转移到B的总次数；

W(a) —— 等于从基本块A转移到其所有后继的总次数和；

B(ab) —— 分支执行概率，等于W(ab) / W(a)。







静态分支预测介绍

OpenEuler Compiler SIG

启发算法	描述
Loop Branch	If the branch target is back to the head of a loop, predict taken.
Pointer	If a branch compares a pointer with NULL, or if two pointers are compared, predict in the direction that corresponds to the pointer being not NULL, or the two pointers not being equal.
Opcode	If a branch is testing that an integer is less than zero, less than or equal to zero, or equal to a constant, predict in the direction that corresponds to the test evaluating to false.
Guard	If the operand of the branch instruction is a register that gets used before being redefined in the successor block, predict that the branch goes to the successor block.
Loop Exit	If a branch occurs inside a loop, and neither of the targets is the loop head, then predict that the branch does not go to the successor that is the loop exit.
Loop Header	Predict that the successor block of a branch that is a loop header or a loop pre-header is taken.
Call	If a successor block contains a subroutine call, predict that the branch goes to that successor block.
Store	If a successor block contains a store instruction, predict that the branch does not go to that successor block.
Return	If a successor block contains a return from subroutine instruction, predict that the branch does not go to that successor block.

参考： "Branch Prediction for Free “ Ball and Larus; PLDI '93.

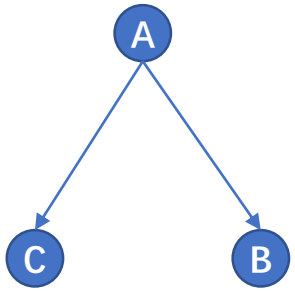
静态分支预测介绍

OpenEuler Compiler SIG

如果一个分支match多个heuristics，怎么办?

--- 方法1：First match

```
If (test == 0) {
   call handle_error()
}
gv = v;
```



Miss Rate	Order						
26.00	Opcode	Call	Return	Store	Point	Loop	Guard
25.52	Call	Opcode	Return	Store	Point	Loop	Guard
25.50	Point	Call	Opcode	Return	Store	Loop	Guard
25.59	Point	Loop	Call	Opcode	Return	Store	Guard
27.12	Opcode	Call	Return	Store	Point	Guard	Loop
25.99	Point	Opcode	Call	Return	Store	Loop	Guard
26.64	Call	Opcode	Return	Store	Point	Guard	Loop
26.04	Loop	Call	Opcode	Return	Store	Point	Guard
25.55	Call	Opcode	Return	Point	Store	Loop	Guard
26.02	Opcode	Call	Return	Point	Store	Loop	Guard

```
1252
1253    // Walk the basic blocks in post-order so that we can build up state about
1254    // the successors of a block iteratively.
1255    for (auto BB : post_order(&F.getEntryBlock())) {
1256      LLVM_DEBUG(dbgs() << "Computing probabilities for " << BB->getName()
1257                        << "\n");
1258      // If there is no at least two successors, no sense to set probability.
1259      if (BB->getTerminator()->getNumSuccessors() < 2)
1260        continue;
1261      if (calcMetadataWeights(BB))
1262        continue;
1263      if (calcEstimatedHeuristics(BB))
1264        continue;
1265      if (calcPointerHeuristics(BB))
1266        continue;
1267      if (calcZeroHeuristics(BB, TLI))
1268        continue;
1269      if (calcFloatingPointHeuristics(BB))
1270        continue;
1271    }
```

静态分支预测介绍

OpenEuler Compiler SIG

--- 方法2：probability合并

首先，设定每一个heuristic的probability

启发算法	Probability of taking branch
Loop Branch	88%
Pointer	60%
Call	78%
Opcode	84%
Loop Exit	80%
Return	72%
Store	55%
Loop Header	75%
Guard	62%

根据Dempster-Shafer证据理论：

$$m_1 \oplus m_2(\{b_i\}) = \frac{\mu\upsilon}{\mu\upsilon+(1-\mu)(1-\upsilon)}$$

$$m_1 \oplus m_2(A-\{b_i\}) = \frac{(1-\mu)(1-\upsilon)}{\mu\upsilon+(1-\mu)(1-\upsilon)}$$

其中：b指某个branch；m1/m2是作用在该b上的两个heuristic；u是第一个heuristic执行的概率，v是第二个执行的概率；

举例：

$$m_1 \oplus m_2(\{b_1\}) = \frac{0.5\times0.7}{0.5\times0.7+0.5\times0.3} = 0.7$$

$$m_1 \oplus m_2(\{b_2\}) = \frac{0.5\times0.3}{0.5\times0.7+0.5\times0.3} = 0.3$$

$$m_1 \oplus m_2 \oplus m_3(\{b_1\}) = \frac{0.7\times0.6}{0.7\times0.6+0.3\times0.4} = 0.778$$

$$m_1 \oplus m_2 \oplus m_3(\{b_2\}) = \frac{0.3\times0.4}{0.7\times0.6+0.3\times0.4} = 0.222$$

参考："Static Branch Frequency and Program Profile Analysis“ Wu and Larus; MICRO-27.

目录

基本块重排—经典算法

OpenEuler Compiler SIG

Pettis与Hansen在其论文中提到这两种算法。

1、Bottom-up

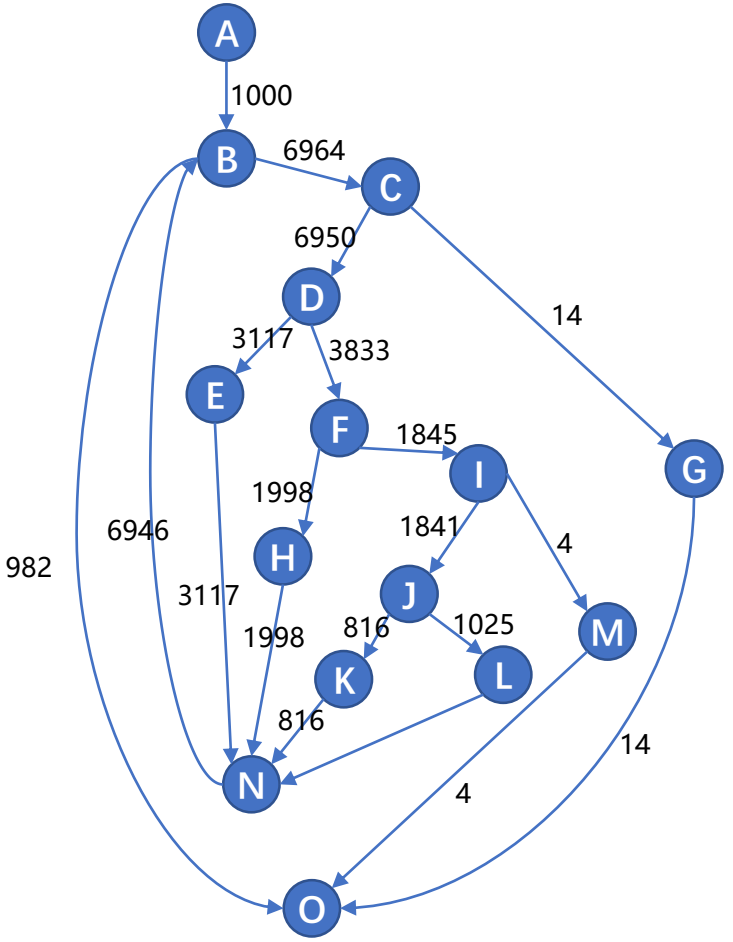
思想：让执行频率更高的分支成为fall-through边。

- 首先，把每一个BB看做一个chain的头尾；
- 从执行frequency（weight）最高的BB开始把BB连成新的chain；
- 两个chain合并。一个边的源是一个chain的tail，而目的是另一个chain的head，则可以合并，否则不可以。
- 合并时选择哪个分支边的依据是branch probability。

2、Top-down

思想：让执行频率低的分支成为forward分支（避免backward开销）

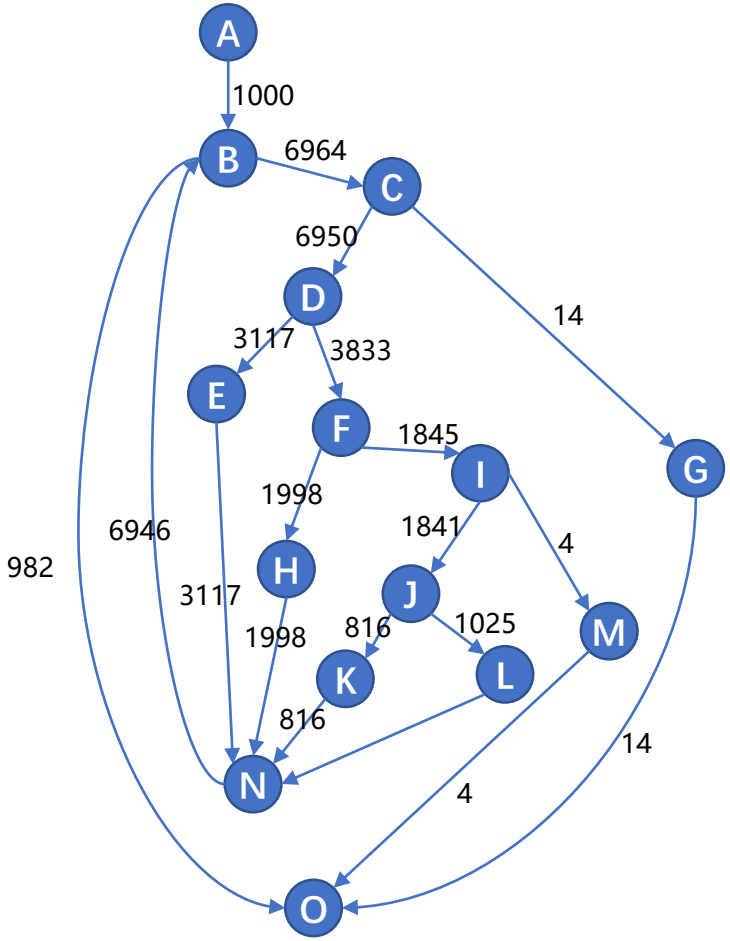
- 从函数entry BB开始；
- 选择weight高的BB加入chain，如果后继BB都在chain里，则重新生成另一个chain；
- 按top续合并chains。



注：参考自（3）

基本块重排—经典算法（bottom-up）

OpenEuler Compiler SIG



根据BB的weight从大到小获取如下chains：

E (6) I (8) A (11) G (14) K (13) M (15)

N (4) J (9) O (12)

B (1) L (10)

C (2)

D (3)

F (5)

H (7)

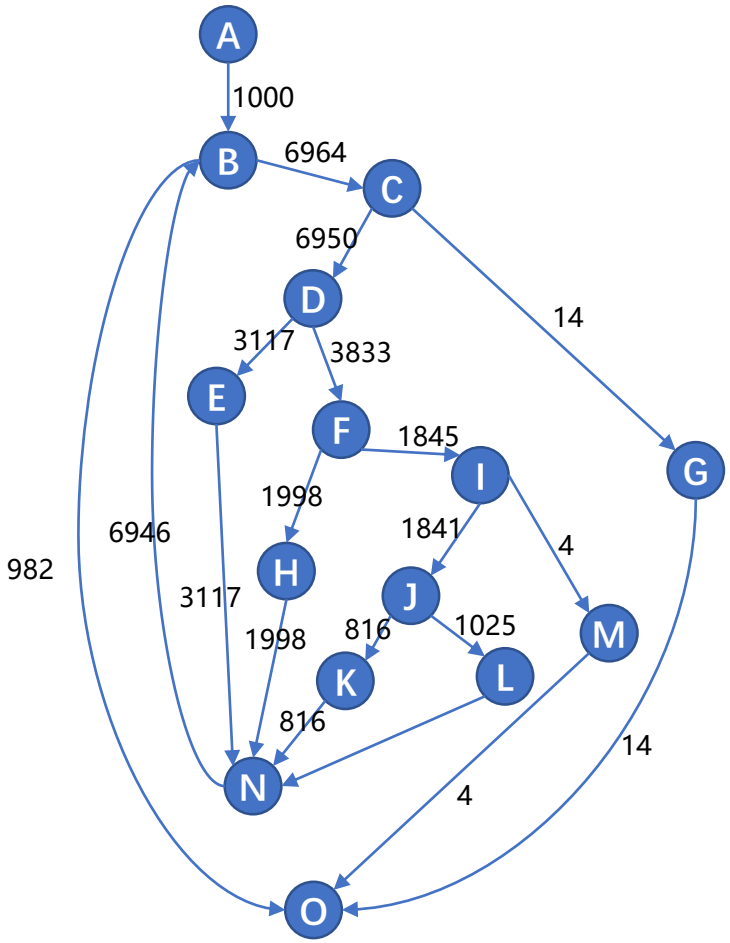
合并chains，原则是taken的分支尽量fall-through（根据6个条件跳转决定顺序）

chain 1(B) before chain 4(O)
chain 1(C) before chain 4(G)
chain 1(F) before chain 2(I)
chain 2(I) before chain 6(M)
chain 2(J) before chain 5(K)

D有点特殊，会出现backward分支

A E N B C D F H I J L G O K M

基本块重排—经典算法（Top-down）



从函数的entry开始选取BB组成chains：

A (1)　E (8)　I (9)　O (12)　K (13)　G (14)　M (15)

B (2)　J (10)

C (3)　L (11)

D (4)

F (5)

H (6)

N (7)

合并chains，原则是top排序，无依赖顺序时按weight从大到小排序：

A B C D F H N　E　I J L　O　K G M

基本块重排—改进算法

OpenEuler Compiler SIG

LLVM中使用了bottom-up算法的改进版，主要有如下改进：

（1）从内层循环的BB开始构建chain；

（2）合并chain的启发性算法改进

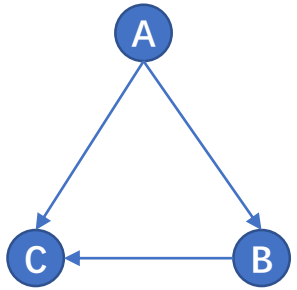
1）selectBestSuccessor：在当前chain最后一个BB的后继们中选取Best BB。选取方法比较相对复杂。

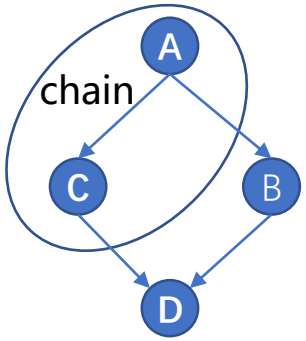
2）selectBestCandidateBlock：根据候选BB的weight选择一个Best BB

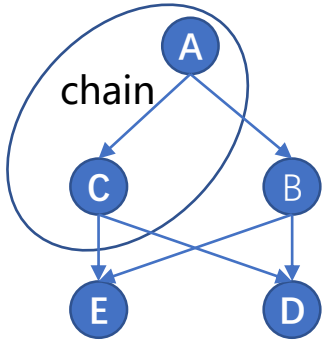
3）getFirstUnplacedBlock：如果上面方法没有找到Best BB，则选取一个没有被放置的BB。

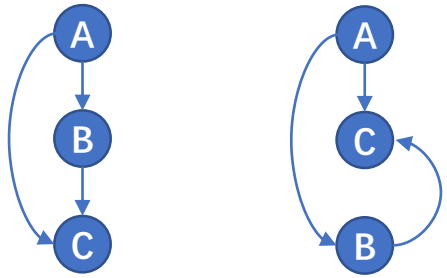
（2）必要时复制BB，获得更好的layout。

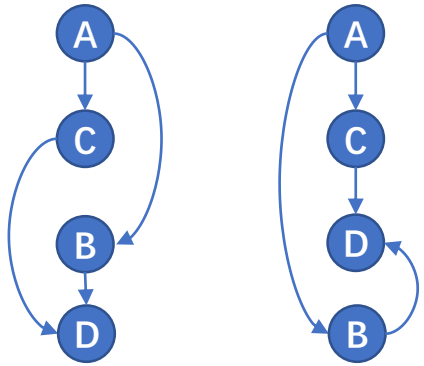
基本块重排—改进算法

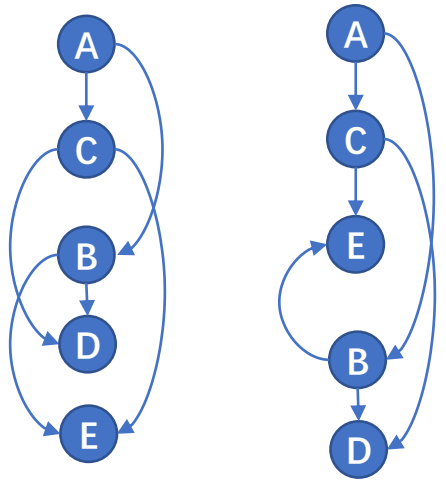




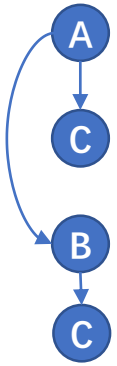








复制C，得到更好的layout

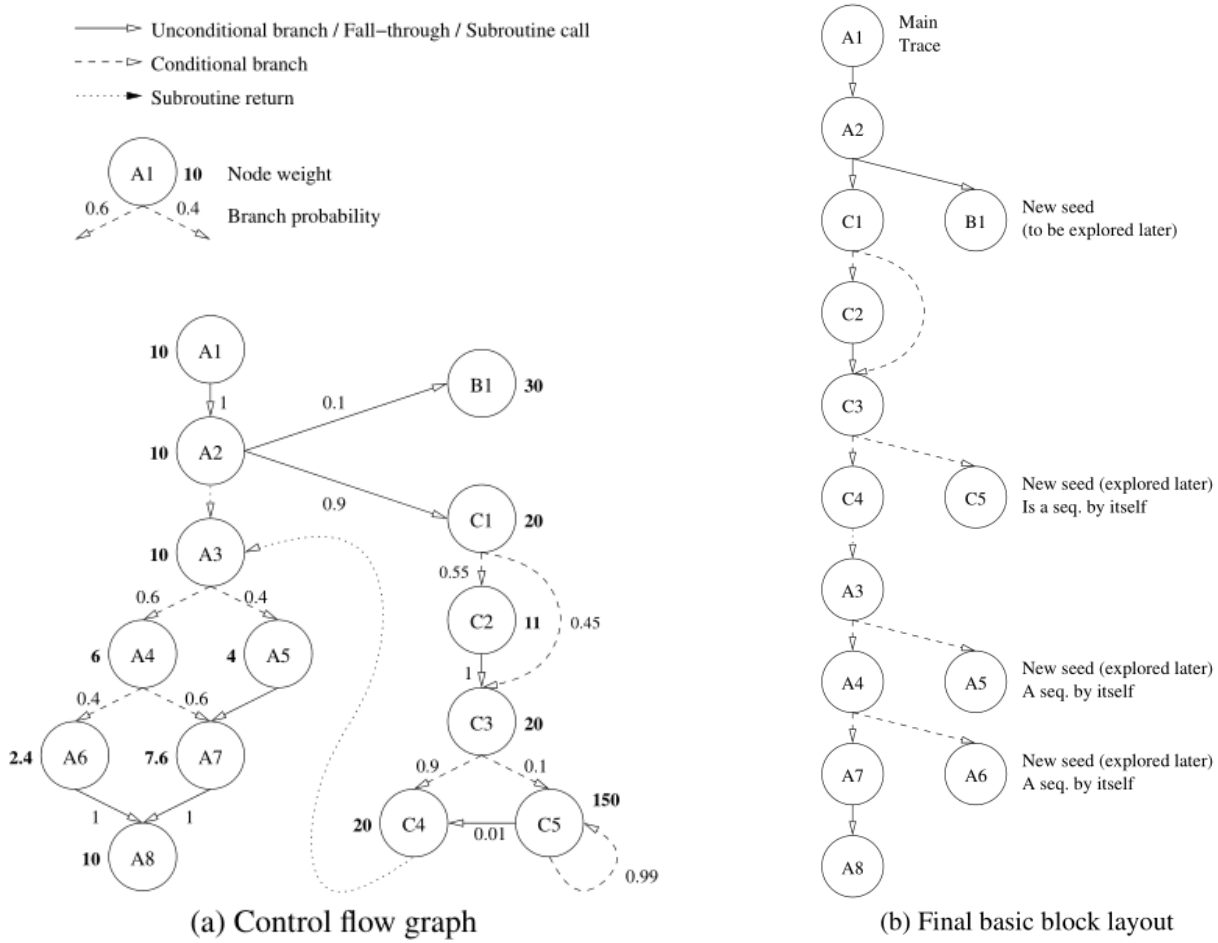


基本块重排—改进算法

OpenEuler Compiler SIG

STC（Software Trace Cache）目前应用在GCC的基本块重排优化遍上。我认为其实是top-down算法的改进版。

- 选取seed。一般情况下选取函数入口作为seed，所有的子例程入口作为seed list，按执行weight排序。
- 多轮构建trace。以第一个seed为起点，基于贪心算法构建相应trace，当所有基本块都包括在基本块序列时开始下一个seed。
- Branch Threshold和Exec Threshold。如果一个边的probability小于Branch Threshold或一个BB执行次数小于Exec Threshold，后继BB将作为下一轮的seed。每一轮这两个阈值都会减少，直到0。
- 合并trace。把trace合并在一起，必要时可以复制trace。



GCC和LLVM上的实现

OpenEuler Compiler SIG

- GCC相关代码
  - 分支预测
    - profile-count.c // profile_probability类的定义
    - profile-count.h // profile_probability类的声明
    - Predict.def // 不同predictor的定义
    - predict.c // Branch Prediction, Static Branch Frequency and Program Profile Analysis
  - 基本块重排
    - bb-reorder.c // 基本块重排算法
- LLVM相关代码
  - 分支预测
    - llvm/lib/Analysis/BranchProbabilityInfo.cpp
    - llvm/lib/Analysis/LazyBranchProbabilityInfo.cpp
    - llvm/lib/CodeGen/MachineBranchProbabilityInfo.cpp
  - 基本块重排
    - llvm/lib/CodeGen/MachineBlockPlacement.cpp

目录

毕昇编译器上的增强

OpenEuler Compiler SIG

在毕昇编译器做如下增强：

（1）应用Ball and Larus描述的heuristics；

（2）调整和增强部分基本块重排的部分算法；

应用效果：

Spec2017int整体提升1%，其中perfbench提升10%；

参考资料

OpenEuler Compiler SIG

1、"Branch Prediction for Free“ Ball and Larus; PLDI '93. https://www.classes.cs.uchicago.edu/archive/2017/fall/32001-1/papers/ball-larus-branch-predict.pdf

2、 "Static Branch Frequency and Program Profile Analysis“ Wu and Larus; MICRO-27. https://dl.acm.org/doi/pdf/10.1145/192724.192725

3、Profile Guided Code Positioning，https://dl.acm.org/doi/pdf/10.1145/989393.989433

4、Software Trace Cache，https://www.tesisenred.net/bitstream/handle/10803/5969/05CHAPTER4.pdf?sequence=5&isAllowed=y

OpenEuler x Compiler SIG

Thank You.

Compiler SIG 专注于编译器领域技术交流探讨和分享，包括 GCC/LLVM/OpenJDK 以及其他的程序优化技术，聚集编译技术领域的学者、专家、学术等同行，共同推进编译相关技术的发展。

毕昇编译公众号

Compiler 交流群小助手