

# Machine Block Placement

code layout optimizations.

龙英池

2022 年 7 月 29 日

- 1 代码布局 (Code Layout)
- 2 代码块放置 (Block Placement)
- 3 LLVM 中如何实现这个算法
- 4 参考文献 (References)

## 代码布局 (Code Layout)

# 什么是代码布局

指令连续地、有顺序地储存在内存中。LLVM 中，这些优化过程抽象成控制流图中的基本块 (Basic Block, BB) 的排列方式。

```
C Codes:
if (a < b)
    c = 1;

; a is in a0
; b is in a1
; c is in a2
bge a0 a1 .LBB0_2
j .LBB0_1
li a2 1
.LBB0_2:
```

# 机器代码布局优化

基于机器相关的，代码布局的优化主要包含，基本块放置 (Basic block placement)、基本块对齐 (Basic block alignment)、冷热代码分离 (Hot-Cold Splitting)[1].

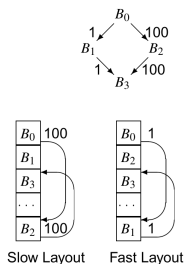


图: 两种不同布局的优劣 [2]

# 基本块放置 - Basic Block Placement

C Codes:

```
// hot path  
if (cond)  
    coldFunc();  
// hot path again
```

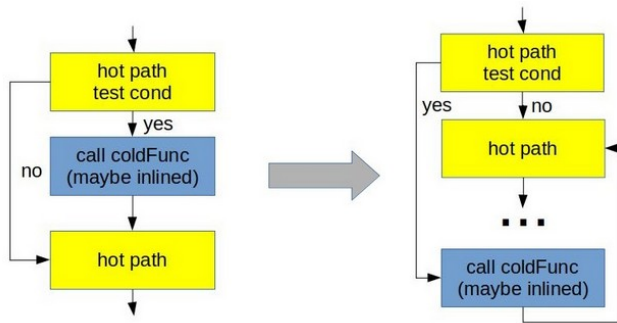
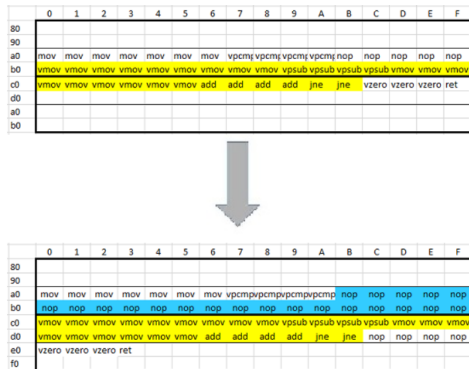


图: 布局优化示例

- 不进行分支跳转往往比跳转代价更低 (maintain fall through)
- 更好地利用 Cache ( $\mu$ op-cache) (局部性)

# 基本块对齐 - Basic Block Alignment

A brief explanation of this manner:



图：基本块对齐

## 冷热代码分离 - 分离前

冷热代码分离的核心思想是，将冷代码（如错误处理）分成一个独立的函数调用，从原过程（procedure）中分离，然后用一个函数调用语句代替冷代码，从而提高热代码的连续性、局部性。

```
void foo(bool cond1, bool cond2) {  
    // hot path  
    if (cond1)  
        // cold code 1  
    //hot code  
    if (cond2)  
        // cold code 2  
}
```

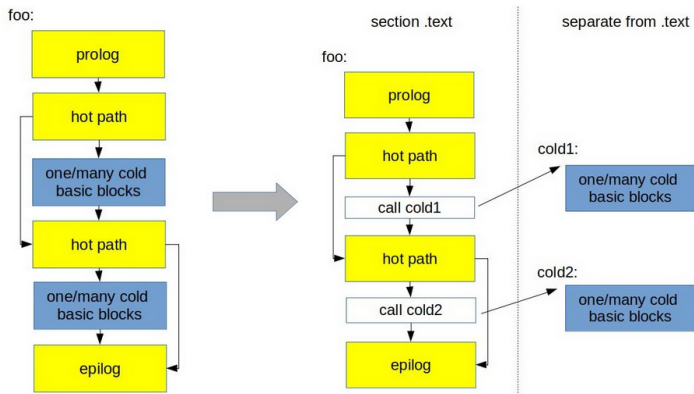


## 冷热代码分离 - 分离后的代码

```
void foo(bool cond1, bool cond2) {  
    // hot path  
    if (cond1)  
        cold1();  
    //hot code  
    if (cond2)  
        cold2();  
}  
  
void cold1() __attribute__((noinline)) { // cold code 1 }  
void cold2() __attribute__((noinline)) { // cold code 2 }
```

## 冷热代码分离 - 图

这样做的好处是，热指令都会存在同一个 cache line 里，可以提高 CPU 前端数据结构的利用效率，例如 I-cache 和 DSB-Cache



图：将冷代码过程转换为函数调用

## 代码块放置 (Block Placement)

## 从 CFG 到代码布局 (Code Layout) - Extended-TSP

控制流图 (CFG) 由若干基本块 (Basic Blocks) 组成, 代码块放置的目标是让经常连在一起执行的基本块, 在最终的代码布局 (Code Layout) 中放在一起, 这样最直接的好处就是更好地利用 l-cache。

事实上这个问题直到 2021 年才被形式化, 背后的数学问题是 Ext-TSP [3]. 给定有向图  $G = (V, E)$ , 带有正边权  $w: E \rightarrow R^+$ , 一个不递增的单调代价函数  $f(\cdot)$ , 满足  $f(1) = 1$ ,  $f(i) = 0, i > k$ ,  $k$  是一个问题相关的常量。

构建  $V$  的一个序列  $d$ , 最优化

$$\sum_{(u,v) \in E} f(|d_u - d_v|) \cdot w(u, v) \quad (1)$$

$d_u \in \{1, \dots, |V|\}$ , 是  $u$  在序列中的位置。

## 控制流图链剖分 - Pettis-Hansen

首先对 CFG 建立链剖分，建立的原则是，经常被执行的边被放在一起，形成一条链 (Chain)。

一条链包含一个或多个基本块 (BB)，并且每个路径都有一个优先级 (priority) 决定了他们在最终的代码布局中的情况。

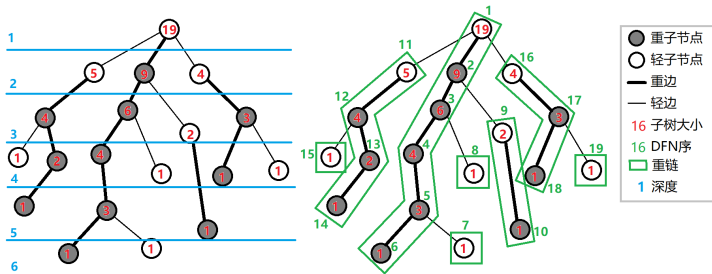


图: [4] 轻重链剖分，树上链剖分的一种

## Kruskal ? Prim ? - Pettis-Hansen

CFG 没有树这么好的性质，CFG 是一个有向图。构建链的方法是每次从边集合中选一条没有被选中的边（按照边的权重顺序），然后形成链状结构。

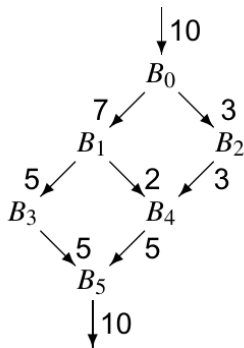


图: CFG 的一个例子 [2]

## CFG 上的贪心启发式链剖分 (Greedy heuristic chain div on CFGs)

```
 $E \leftarrow | \text{edges} |$   
for each block  $b$  do  
    make a degenerate chain,  $d$ , for  $b$   
     $\text{priority}(d) \leftarrow E$   
end for  
 $P \leftarrow 0$   
for each CFG edge  $\langle x, y \rangle, x \neq y$ , in decreasing freq order do  
     $t \leftarrow \text{priority}(a)$   
    append  $b$  onto  $a$   
     $\text{priority}(a) \leftarrow \min(t, \text{priority}(b), P++)$   
end for
```

# CFG 上的贪心启发式链剖分 (Greedy heuristic chain div on CFGs)

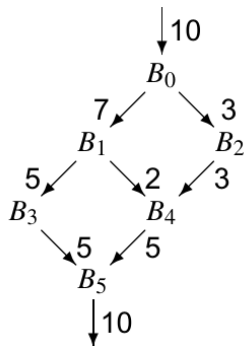


图: CFG 的一个例子 [2]

Edge	Set of Chains	P
—	$(B_0)_E, (B_1)_E, (B_2)_E, (B_3)_E, (B_4)_E, (B_5)_E$	0
$(B_0, B_1)$	$(B_0, B_1)_0, (B_2)_E, (B_3)_E, (B_4)_E, (B_5)_E$	1
$(B_3, B_5)$	$(B_0, B_1)_0, (B_2)_E, (B_3, B_5)_1, (B_4)_E$	2
$(B_4, B_5)$	$(B_0, B_1)_0, (B_2)_E, (B_3, B_5)_1, (B_4)_E$	2
$(B_1, B_3)$	$(B_0, B_1, B_3, B_5)_0, (B_2)_E, (B_4)_E$	3
$(B_0, B_2)$	$(B_0, B, B_3, B_5)_0, (B_2)_E, (B_4)_E$	3
$(B_2, B_4)$	$(B_0, B_1, B_3, B_5)_0, (B_2, B_4)_3$	4
$(B_1, B_4)$	$(B_0, B_1, B_3, B_5)_0, (B_2, B_4)_3$	4

图: 算法每个过程选择的边, 和对应的边集



## 基于 WorkList 组合最终的代码布局 (Code Layout)

```
 $t \leftarrow$  chain headed by the CFG entry node  
 $WorkList \leftarrow \{(t, priority(t))\} \leftarrow \text{Heap}[5]$   
while  $WorkList \neq \emptyset$  do  
    remove a chain  $c$  of lowest priority from  $WorkList$   
    for each block  $x$  in  $c$  in chain do  
        place  $x$  and the end of assembly codes  
    end for  
    for each block  $x$  in  $c$  do  
        for each edge  $\langle x, y \rangle$  where  $y$  is unplaced do  
             $t \leftarrow$  chain containing  $\langle x, y \rangle$   
            if  $(t, priority(t)) \notin WorkList$  then  $WorkList \leftarrow WorkList \cup \{(t, priority(t))\}$   
            end if  
        end for  
    end for  
end while
```

## Example

Edge	Set of Chains	P
—	$(B_0)_E, (B_1)_E, (B_2)_E, (B_3)_E, (B_4)_E, (B_5)_E$	0
$(B_0, B_1)$	$(B_0, B_1)_0, (B_2)_E, (B_3)_E, (B_4)_E, (B_5)_E$	1
$(B_3, B_5)$	$(B_0, B_1)_0, (B_2)_E, (B_3, B_5)_1, (B_4)_E$	2
$(B_4, B_5)$	$(B_0, B_1)_0, (B_2)_E, (B_3, B_5)_1, (B_4)_E$	2
$(B_1, B_3)$	$(B_0, B_1, B_3, B_5)_0, (B_2)_E, (B_4)_E$	3
$(B_0, B_2)$	$(B_0, B, B_3, B_5)_0, (B_2)_E, (B_4)_E$	3
$(B_2, B_4)$	$(B_0, B_1, B_3, B_5)_0, (B_2, B_4)_3$	4
$(B_1, B_4)$	$(B_0, B_1, B_3, B_5)_0, (B_2, B_4)_3$	4

图：上一个过程形成的链

Step	WorkList	Code Layout
—	$(B_0, B_1, B_3, B_5)_0$	
1	$(B_2, B_4)_3$	$B_0, B_1, B_3, B_5$
2	$\emptyset$	$B_0, B_1, B_3, B_5, B_2, B_4$

图：生成最终的代码布局

## LLVM 中如何实现这个算法

算法主体主要实现在 `lib/CodeGen/MachineBlockPlacement.cpp`

其中分支概率等信息来源于 `Block{Frequency, ProbabilityInfo}`

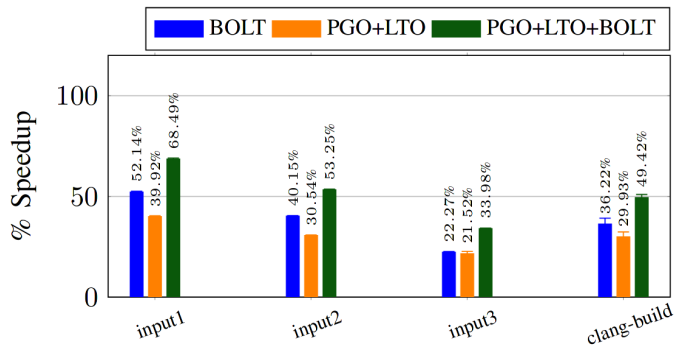
各个 Target 需要实现 { `analyze`, `insert`, `remove` } `Branch` 等虚函数, RISC-V 在 5 年前, D40808[6] 实现; 两年前, D84833 包含了一个间接分支实现 [7]

功能	实现	源代码位置
边权	<code>BlockFrequency</code>	[8]
块放置	<code>MachineBlockPlacement</code>	<code>buildCFGChains()[9]</code>
块对齐	<code>MachineBlockPlacement</code>	<code>alignBlocks()[9]</code>
分支分析	<code>RISCVInstrInfo</code>	<code>analyzeBranch()[10][6]</code>

表: 各个功能的实现情况和所在的位置

# BOLT - Binary optimization and layout tool

Facebook 在 2018 年开源了他们的二进制优化工具 – BOLT[11][12]。这个仓库现在已经合并到 LLVM，可以在对编译后的二进制进行基本的重排，冷热代码分离等工作。



图：优化后的 Clang

## LLVM 对 Pettis-Hansen 算法的改进

2021 年 12 月, MachineBlockPlacement 的基本功能有了新的改进。D113424[13] 引入一个方案, 来优化已经完成链剖分之后的, 生成最终的代码布局的过程。

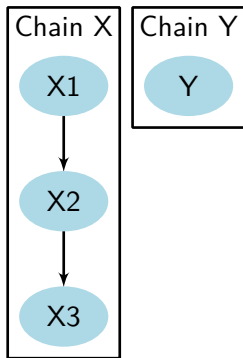


图: 需要合并的两个 Chain 示意

## 参考文献 (References)

# References I

- [1] D. Bakhvalov, “Machine code layout optimizations. | Easyperf.” <https://easyperf.net/blog/2019/03/27/Machine-code-layout-optimizatoins>, 03 2019.
- [2] K. D. Cooper and L. Torczon, *Engineering a compiler*. Elsevier, 2011.
- [3] J. Mestre, S. Pupyrev, and S. W. Umboh, “On the extended tsp problem,” *arXiv preprint arXiv:2107.07815*, 2021.
- [4] I, “树链剖分 - OI Wiki.” <https://oi-wiki.org/graph/hld/>, 03 2022.
- [5] G. E. Forsythe, “Algorithms,” *Communications of the ACM*, vol. 7, no. 6, pp. 347–349, 1964.
- [6] LLVM Project, “[RISCV] Implement branch analysis - Phabricator.” <https://reviews.llvm.org/D40808>, 07 2017.



## References II

- [7] LLVM Project, “[RISCV] Implement indirect branch generation in position independent code for the RISC-V target - Phabricator.” <https://reviews.llvm.org/D84833>, 07 2020.
- [8] LLVM Project, “lib/Analysis/BlockFrequencyInfoImpl.cpp - Doxygen.” [https://llvm.org/doxygen/BlockFrequencyInfoImpl\\_8cpp\\_source.html](https://llvm.org/doxygen/BlockFrequencyInfoImpl_8cpp_source.html), 07 2022.
- [9] LLVM Project, “lib/CodeGen/MachineBlockPlacement.cpp - Doxygen.” [https://llvm.org/doxygen/MachineBlockPlacement\\_8cpp\\_source.html](https://llvm.org/doxygen/MachineBlockPlacement_8cpp_source.html), 07 2022.
- [10] LLVM Project, “lib/Target/RISCV/RISCVInstrInfo.cpp - Doxygen.” [https://llvm.org/doxygen/RISCVInstrInfo\\_8cpp\\_source.html](https://llvm.org/doxygen/RISCVInstrInfo_8cpp_source.html), 07 2022.
- [11] Facebook, “BOLT - Binary Optimization and Layout Tool.” <https://github.com/llvm/llvm-project/tree/main/bolt>, 04 2022.

- [12] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, “Bolt: a practical binary optimizer for data centers and beyond,” in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, IEEE, 2019.
- [13] LLVM Project, “ext-tsp basic block layout - Phabricator.”  
<https://reviews.llvm.org/D113424>, 12 2021.