编译器优化那些事儿（9）： Machine Outliner

Machine Outliner简介

在嵌入式领域，代码体积(code size)优化能够减少内存的使用，对产品的竞争力至关重要。对当代产品而言， code size优化可以为产品放入更多特性，增强竞争力；对下一代产品而言，code size优化能够带来产品功耗和成本的竞争力提升[1]。LLVM Machine Outliner是一种编译器优化技术，可以识别重复代码片段，将其提取出来并转换成一个独立的函数，从而降低程序code size。

示例

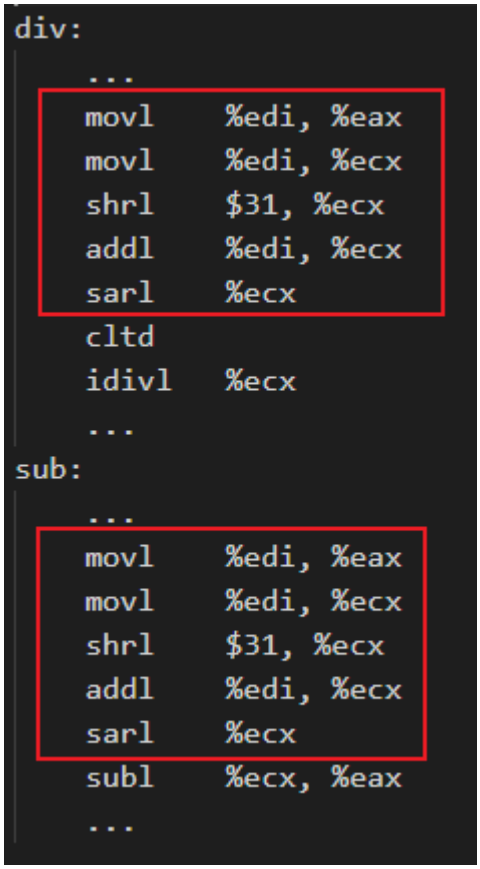
通过以下代码示例，描述是否开启Machine Outliner优化的效果：

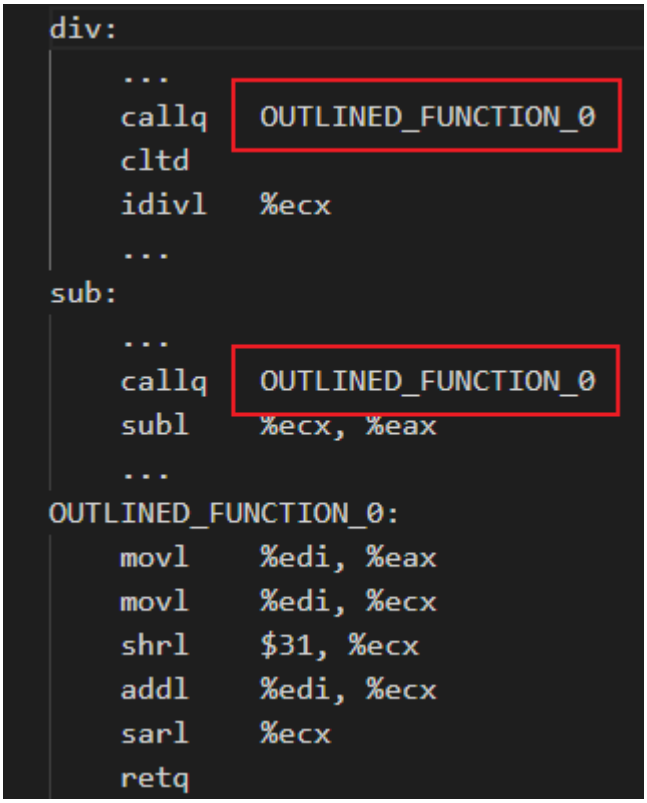
```
1. int div(int x) {
2.     int a = x / 2;
3.     int b = x;
4.     return b / a;
5. }
6.
7. int sub(int x) {
8.     int a = x / 2;
9.     int b = x;
10.    return b - a;
11. }
```

以上代码片段由div和sub两个函数组成，通过函数参数x，对变量a和b赋值，然后分别返回b和a相除，以及b和a相减的结果。其中，关于变量a、b赋值部分为重复代码片段。

是否开启Machine Outliner优化，在汇编层面的区别如下表所示：

未开启Machine Outliner	**开启Machine Outliner**





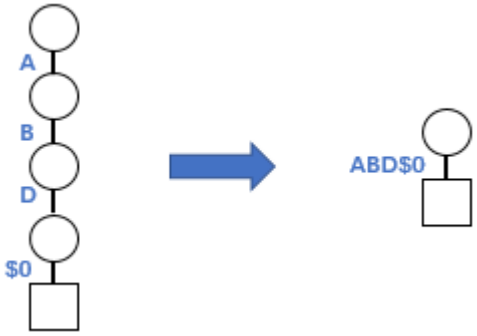
如果不开启Machine Outliner优化，则会分别在标签div和sub下生成相关的重复汇编指令。开启Machine Outliner优化，则会将重复指令提取为单独的函数，并且在重复指令初始位置添加函数调用，从而降低程序code size。在编译阶段，可以通过使用 -mllvm -enable-machine-outliner=always 选项开启Machine Outliner优化，提取出的函数统一以“OUTLINED_FUNCTION_n”的形式命名。

后缀树

Machine Outliner优化依靠后缀树(Suffix Tree)的形式进行重复指令序列的识别，后缀树的构造和重复字符串查询操作均可在线性时间复杂度内完成，从而实现了Machine Outliner优化的效率提升。

后缀树是一种将字符串所有后缀存储在一棵树中的数据结构，目的是用来支持快速搜索和大量字符串匹配和查询，能高效解决很多关于字符串的问题[2]。字符串S对应的后缀树，也就是由该字符串所有后缀所共同构成的压缩字典树。

字典树(Trie)是一种树形数据结构，其中每条边用来表示一个字符，且每个节点出边对应的字符都不相同，将根节点到某一节点路径上所经过的字符拼接起来，即为该节点所表示的字符串。压缩字典树(Compressed Trie)由字典树演变而来，将字典树中的单节点链条压缩为一个节点，即将相邻的具有相同前缀的节点合并，可得到对应的压缩字典树。字符串“ABD$0”对应的字典树和压缩字典树如下所示：

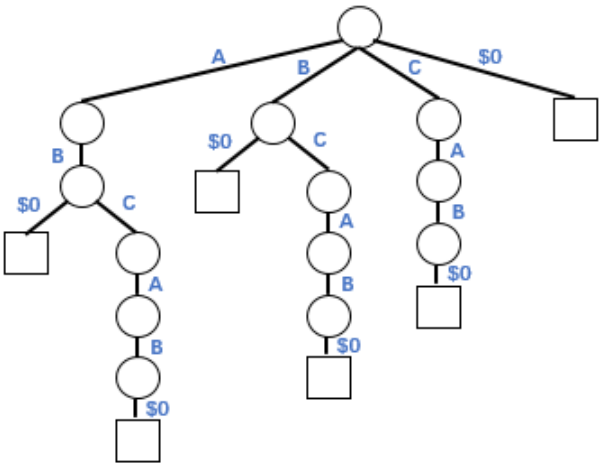
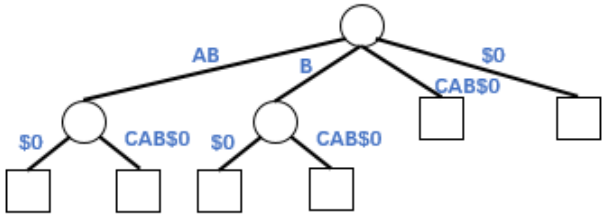


后缀树的构建需要经过字符串后缀生成和压缩字典树构建两步：

(1)生成字符串S的所有后缀，以“ABCAB$0”(“$0”是结束字符)为例，该字符串的所有后缀为：

```
1. ABCAB$0
2. BCAB$0
3. CAB$0
4. AB$0
5. B$0
6. $0
```

(2)为以上所有后缀生成字典树，并且合并节点生成相应的压缩字典树：

字典树	压缩字典树
	

令字符串S的长度为n，通过构建字符串S所对应的后缀树，即可在O(n)时间复杂度内，完成字符串重复次数，以及重复字符串长度的检索[3]。

重复次数搜索：假设字符串T在字符串S中重复次数为m，则字符串S应有m个后缀以字符串T为前缀，即字符串T所对应节点的叶节点数量为其重复次数。

重复字符串长度搜索：由于重复字符串出现次数大于1，所以字符串T在后缀树中一定以非叶节点的形式表示，字符串T的长度为该非叶节点到根节点所经过的字符总数。

编译器实现

LLVM对于Machine Outliner的实现在寄存器分配之后，主要集中在MachineOutliner.cpp中，基于机器指令表示(MIR)进行函数的分析和提取，以实现程序code size优化。

编译器侧的实现过程可划分为指令映射、后缀树构建和函数提取三个阶段：

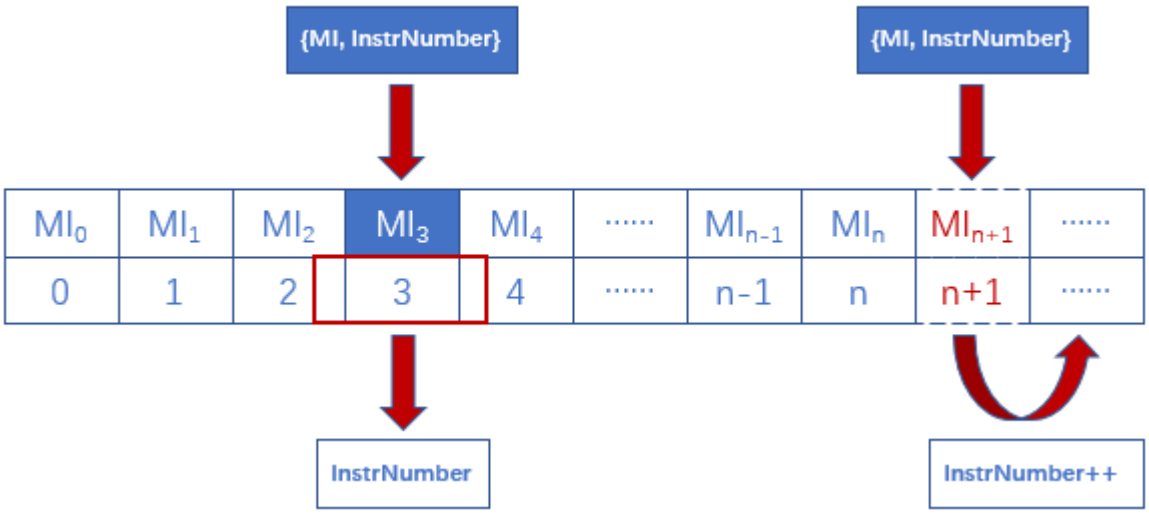
(1)将指令映射成特定的无符号数，并以指令-无符号数对的形式存储在Map中；

(2)以无符号数组为输入构建指令序列对应的后缀树，并且找出所有长度大于2的重复指令序列；

(3)遍历后缀树并进行收益计算，从而得到包含正收益序列的候选列表，对于具备收益的候选项进行函数外提。

1. 指令映射

首先需要遍历源文件对应的所有指令，将所有合法指令映射为无符号数(InstrNumber)，并以指令-无符号数对的形式存储在Map中，如果两条指令的操作码和操作数均相同，则认为两条指令相同。对于所访问的每条指令，首先应该在Map中查询是否已经存储了相同的指令，如果是，则返回对应的InstrNumber；否则，将该指令插入到Map中，InstrNumber自加。至此，所有指令均以无符号数的形式进行唯一标识，以作为构建后缀树的输入。而对于读写栈指针、PC相关，以及其他与call、ret指令有数据依赖的指令，将被判定为非法指令，为保证程序运行的正确性，这些指令不参与上述映射过程。



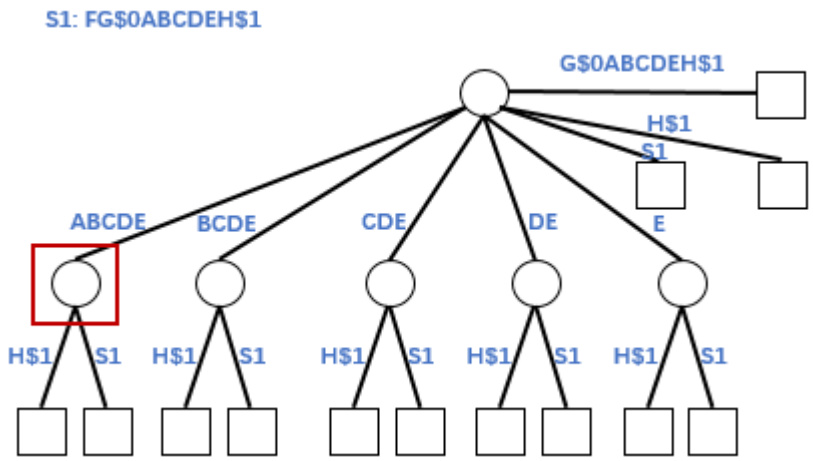
2. 后缀树构建

假定将无符号数以字符形式表示，以指令映射输出的无符号数组为输入，通过添加终结符和构建后缀树，即可在线性时间复杂度内，完成字符串S的所有重复子字符串长度、重复次数和起始下标的计算[4]，这些重复字符串将以候选列表的形式输出。

(1)以第2节所示汇编指令为例，经过指令映射和添加终结符后可得到字符串S：ABCDEFG$0ABCDEH$1，其中添加终结符可避免跨函数指令序列提取。

```
div:
    ...
    movl    %edi, %eax
    movl    %edi, %ecx
    shrl    $31, %ecx
    addl    %edi, %ecx      ---> ABCDEFG$0
    sarl    %ecx
    cltd
    idivl   %ecx
    ...
sub:
    ...
    movl    %edi, %eax
    movl    %edi, %ecx
    shrl    $31, %ecx
    addl    %edi, %ecx      ---> ABCDEH$1
    sarl    %ecx
    subl    %ecx, %eax
    ...
```

(2)以字符串S为输入，构建后缀树：



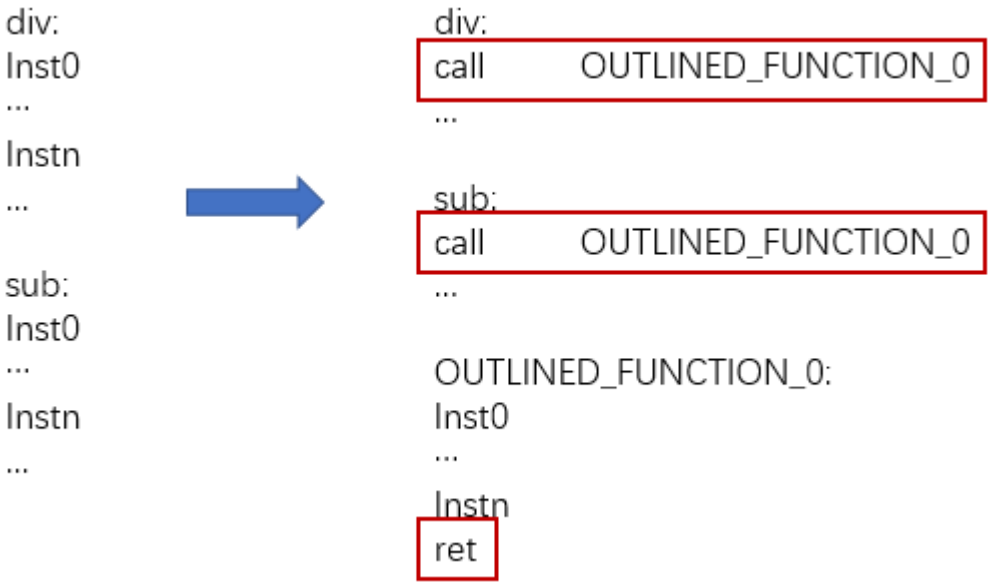
令ABCDE所指向的节点为P，单个字符所表示的距离为1，则节点P到根节点的距离为5，大于其他非叶节点到根节点的距离，因此ABCDE为字符串S中的最长重复子字符串T。节点P有两个子节点，因此字符串T的重复次数为2，且T在S中的起始下标分别为[0,4]，[8,12]。

3. 函数提取

完成后缀树构建和重复字符串解析后，还需要对提取该重复字符串对应的指令序列进行code size收益计算，计算公式如下：

```
1. codesize_benefit = codesize_before - codesize_after
2. codesize_before = 指令序列重复次数 * 指令序列codesize
3. codesize_after = 插入call指令的codesize + 指令序列codesize + 插入ret指令的codesize
```

如果收益大于1，则提取同一重复字符串对应的所有指令序列，以构造outline函数，并在函数末尾额外添加ret指令。而对于重复字符串指向的下标位置，需要删除初始指令序列，并且通过call指令增加对outline函数的调用。



总结

本文对Machine Outliner的基本概念和实现方法进行了简单介绍，通过将所有指令映射成为无符号数，并且以后缀树的形式对重复指令序列进行高效识别，最后提取具有正收益的指令序列作为outline函数，实现程序code size优化，从而提高代码的可读性并且减少程序的内存占用。

在源码中大量使用宏、模板，以及循环展开的场景下，开启Machine Outliner优化将会获得明显的code size收益；而对于程序本身code size很小、结构化设计良好，或者包含大量违反外提约束的情况，Machine Outliner对code size的优化效果不显著。此外，在LLVM14及更高版本上，完成了多次outline的实现[5]，相比于本文所述的单次outline，能够进一步实现code size提升。

参考文献

1. Chabbi M, Lin J, Barik R. An experience with code-size optimization for production iOS mobile applications[C]//2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2021: 363-377.

2. Bieganski P, Riedl J, Carlis J V, et al. Generalized suffix trees for biological sequence data: applications and implementation[C]//HICSS (5). 1994: 35-44.

3. Schneider J G, Mandile P, Versteeg S. Generalized suffix tree based multiple sequence alignment for service virtualization[C]//2015 24th Australasian Software Engineering Conference. IEEE, 2015: 48-57.

4. https://www.llvm.org/devmtg/2016-11/Slides/Paquette-Outliner.pdf

5. Support repeated machine outlining, December 2019, [online] Available: https://reviews.llvm.org/D71027.