编译原理实验申优文档

19373794 范泽恒

2021 年 12 月 22 日

1 前言

关于设计架构已经在设计文档中说明。所以这篇文章只会重点讲述优化那些事儿。关于优化，理论课中我学到了很多很多方法，例如以数据流分析为核心的死代码删除，以 DAG 图算法为核心的公共子表达式消除。这些算法对于编译器性能的提升都有比较大的作用。但是我在实际操作的时候，还是觉得以数据流分析为中心的优化，具有一个很明显的缺点，那就是每一个优化都十分割裂，一次数据流分析需要花很大的功夫，但是只适用于少数几种优化。所以我更加青睐于一种更为强大的代码优化手段，即以 SSA 为基础的优化。

2 SSA 那些事儿

2.1 SSA 是什么

SSA，即为静态单赋值的中间代码形式，什么是静态单赋值？通俗的说，就是一个变量最多只会被赋值一次。一个变量在被使用前一定被唯一的确定了一次赋值。乍一看，这玩意有啥意义？这就不得不提一下咱们理论课学习到一个著名数据流分析，”def -use” 分析，可以想一想”def -use” 分析本质上是在做什么？答案是算出每一个基本块中的活跃变量。众所周知，要做”def-use” 分析，首先需要计算每一个基本块的”use” 和”def” 信息，从最后的基本块开始向上递推，每次将基本块的数据流信息合并到它所在流图的父节点处。重复做直到不变为止。这里的重复性其实就表现了普通数据流分析的繁琐性。为了解决这种繁琐性，SSA 提出了一个比较新奇的视角。那就是直接将数据流信息融入中间代码。变量一次定义多次使用，实际上方便了很多的处理，比如常量传播，复写传播，死代码删除等。举一个比较直观的例子。由于一个变量只会被赋值一次，所以一旦一个变量被赋值，那么之后只要出现这个变量，就可以唯一确定那个变量。而不用考虑变量值已经改变的问题。所以，实际上数据流分析已经不再需要。但是 SSA 也确实存在许多的副作用，比如引入了许多许多原来并没有出现过的变量。给后续的简化产生了许多的副作用

2.2 how to build a SSA ?

2.2.1 基本块划分

无论做什么样的优化，基本块的划分永远是第一步，所以，首先需要做的就是划分基本块。算法比较简单，模仿书上的算法做即可:

- 在每一个跳转指令后插入一个 tag
- 在每一个跳转指令能调到的起始点插一个 tag，重复上述操作直到不能操作为止。

实际上，做中间代码翻译时，很多需要被插的 tag 实际上已经被插好了，所以这里只要做按照”tag” 分组即可。

2.2.2 关于 phi 函数

为了实现上述所说的" 一个变量只会被赋值一次"，我们为了解决那些多赋值的问题，最直接的方法就是添加更多的变量，每一个值在二次被赋值的时候重新给它一个名字。这个动作在基本块内都十分容易实现。但是跨基本块呢？解决难度一下次就提升了上来。现在，我们引入 SSA 的神器。那就是 phi 函数，我们定义 phi 函数的形式如下:

$$phi(A) = phi(A_1, A_2, A_3, \ldots, A_n)$$

这个形式乍一看很容易让人迷惑，实际上，这个函数的意思十分简单，就是以一种十分抽象的形式体现出一种 "选择" 的动作。即从 $A_1, A_2, A_3, \ldots, A_n$ 中选择一个赋值给 A 。我们将这个函数与每一个基本块中需要用到的变量做一个一一映射，并将他们放在每一个基本块的首部，假设基本块 A 对于变量 a 是先使用的关系，那么我们添加一个关于 a 的 phi 函数。将所有到达这个基本块的基本块中对变量 a 的赋值而产生的新变量添加在函数右侧。这样就解决了跨基本块赋值的问题。

2.2.3 支配树和支配边界

这里需要引入一个图论中的概念，"支配树"。首先，我们需要引入 "支配点" 的概念，A 支配 B 即为，在有向图上，任意从起点到 B 的路径上都会经过 A。如果 $A \neq B$ 则，称 A 严格支配 B。按照这个定义，可以确定图上所有点之间的支配关系，可以证明这样的支配关系最终形成了一颗树。

那么，这个支配树有什么用呢？按照上面的描述，我们只需要将前面基本块的变量全部放在各自的后继基本块，然后通过 phi 函数就可以解决问题。但是实际上，这样做效率并不高，因为会产生大量的新变量，大幅度扩大建 SSA 的副作用。所以，对于 phi 函数的插入，我们需要足够的谨慎。

什么时候不需要 phi 函数？显然，如果一个块 B 被 A 支配，那么 A 中定义的所有变量在 B 中都并不需要 phi 函数，直观理解就是在程序开始运行到运行到 B 的过程中，B 别无选择，只能接受 A 的变量。所以，插 phi 函数显得很浪费。所以，有支配关系的两个点是不需要插入任何的 phi 函数的。下面引入另一个概念" 支配边界"。

一个点 A 的支配边界 C 定义为:A 不严格支配 C，但是存在 C 的一个前继 B，A 严格支配 B 直观理解就是 A 一路支配到 B 这里，但是突然支配不下去了，开始有除了 A 以外的点，能到达这儿了。那么这时候，A 就遇到了它的支配边界。理论可以证明，phi 函数的最少插法，就是对于每一个基本块，都将此基本块中定义的变量的 phi 函数插在它的所有支配边界中。

那么，剩下的只剩一个问题，怎么求支配树和支配边界呢？在本次作业中我并没有追求较高的效率，所以我采取了一种十分暴力的方法。求 A 的支配点可以直接从定义入手，我们从起点开始进行深度优先搜索 (dfs)，将 A 点首先打上 vis 使得其不会被访问。深搜结束后，所有没有达到的点，都是 A 的支配点。这样对于每一个点都操作一遍，复杂度是 $O(n^2)$。支配边界可以直接枚举所有基本块的支配点，然后枚举支配点的后继节点，只要找到一个不被 A 支配的点，那么就找到了 A 的一个支配边界。

最后，终于轮到了我们本部分的主角了，那就是" 支配树"，支配树满足，书上任意一个节点都支配它的所有子节点。如何建立这样的层次关系呢？我采用了一种基于拓扑排序的算法。

- 首先按照支配关系建立出支配图，显然，这是一个 DAG 图。
- 我们统计支配图上每一个点的入度，每次删掉入度为 0 的点，并将所有和它相连的点的入度减一。
- 如果产生新的入度为 0 的点，那么，将这两个点之间连一条边，新产生入度为 0 的点是上一个点的直接儿子。

这样支配树就得到了。

2.2.4 插 phi 与重命名

关于插入 phi 函数，前面多多少少都提过一点了，每一个点只需要在它们的所有支配边界上插入它们的块中定义的变量即可。这里有一个需要注意的地方。那就是我们当前只是插了一个 phi 函数进去，并没有给它的赋值区间做定义，也就是说，当前 phi 函数只有左部，并没有右部。为所有 phi 函数找到它的右部就是下一步应该干的事了。

重命名，是真正将中间代码转换成 SSA 的一步。这一步主要完成对于多重赋值变量的重命名以及各个 phi 函数右部的插入工作。首先，我们对于待编译代码中每一个定义的变量都准备一个栈和一个计数器。初始将 0 插入栈中。之后算法如下:

- 对于一个基本块 A，我们遍历它的每一个语句（注意当前所有 phi 函数都被插到了基本块的首部，并且看做赋值语句）。
  - 如果当前语句需要使用变量 X ，则取出 X 对应的栈的栈顶，并将变量名和取出的数字组成一个新的变量并替换掉原来的变量
  - 如果当前语句需要定义变量 X，则将变量对应的计数器加一，并将变量名和计数器对应的数字组成一个新的变量替换原有的变量。并将数字压入 X 的栈中。
- 之后遍历当前块在流图上的后继节点 Y ，对于后继点 Y，遍历 Y 的所有 phi 函数，对于 phi 函数的左部名 X ，将 X 和当前 X 对应的栈顶数字的组合插入到 phi 函数的右部。
- 遍历当前点在支配树上的后继节点，递归运行这个函数
- 回溯到这个节点时，将这个节点定义的所有新变量释放掉，也就是从变量对应的栈中移除。

至此为止。SSA 其实已经建好了。

2.2.5 phi 函数的翻译

翻译 phi 函数实际上并不应该在建 SSA 的时候做。这是在大部分优化已经做完的情况下做的事情。因为实际上，翻译 phi 函数会直接破坏这个 SSA。下面我们谈一下如何翻译 phi 函数。

很显然 phi 函数右部的每一个变量名都来自于一个唯一的基本块。所以，实际上只需要在提供 phi 函数的基本块运行结束的时候进行一些基本的变量赋值便可完成 phi 函数的翻译。但是存在一个比较讨厌的情况，对于条件跳转指令，我们并不知道它会让这个块转移到哪里，所以需要为了翻译这个 phi 函数另外准备一个块来做一个转移。将 phi 函数放在这个块中翻译，一旦程序运行到这个块，就说明了条件跳转选择了这个分支，也就不存在因矛盾而产生的错误了。

3 具体的优化

现在，我们有了一个 SSA 了，于是我们就可以开始为所欲为啦。

3.1 死代码删除

死代码，通俗的讲，就是那些永远不会运行到的代码，这些代码会严重的拖慢程序执行的效率。所以删除死代码是最重要的几个优化之一。

我采取了如下算法进行了死代码删除:

- 首先找到所有的有用变量。我们定义有用的变量为，跳转，输出，函数返回值，函数压栈用到的变量，这些变量对于后续执行十分重要，所以需要将它们留下来。

- 假设上述变量组成了一个集合 S ，我们扫描每一个基本块的中间代码，每次将给这些变量赋值的变量塞入 S ，一直进行如上操作直到 S 集合不再继续扩大为止。
- 至此我们就找到了所有的有用变量，接下来我们把代码中所有对没有用的变量的赋值语句删除。

这个操作实际效果在一般的优化手段中可能不会产生很大的影响，但是在 SSA 结构中，这个优化至关重要。因为 SSA 产生了很多无用变量，这个优化可以将大量无用的 phi 函数产生的赋值去除。

3.2 复制传播

所谓复制传播，就是出现 A 赋值给 B ， B 赋值给 C 的时候，可以将 A, B, C 都替换成 A 以减少赋值指令。

前面说过，在 SSA 的代码结构下。当一个变量出现，那么我们就可以唯一确定这个变量的值。所以，SSA 下的复制传播十分简单。我采取了并查集算法优化的常量传播。当 A 将值赋给 B ，则我们就在 A B 之间连一条边。最后在一个联通集合的变量，显然都可以被赋值成一个值。之后我们遍历每一条指令，将所有变量名替换成次连通块中得一个变量名。最后遍历所有的指令，对于赋值语句，将两边变量名相同的指令删除。至此，就完成了赋值传播。这样可以节省出大量的赋值指令，而且也大幅方便了下面的常量传播。

3.3 常量传播

其实我这个优化和上一个优化是一起做的，但是为了保证逻辑的清晰性，这里再次提一下。复制传播没有做的事就是，一些与常数有关的赋值指令，我们显然可以让变量与常数也连一条边，然后把所有的相关变量都替换成这个常数。这样就可以方便将一些耗时巨大的指令优化掉。

从这个想法出发，可以将上述复制传播的算法优化成如下。

- 首先遍历所有的中间代码找出所有的赋值语句，将有赋值关系的两个变量（包括常数）之间连接一条边。
- 对于 phi 函数定义的变量，之后 phi 函数右部的变量都为相同的变量时，才能将左部的变量名和右部的变量名连一条边。但是实际上，如果出现这种情况，phi 函数显然是并不需要的，可以直接将其去掉。
- 之后对于每一个快所有代码和 phi 函数，把其中出现的所有变量都替换成它在连通块中找到的祖先节点。
- 遍历所有变量，对于两边都是常数的计算指令和两边都是常数的条件跳转指令，直接计算出结果，计算指令替换为赋值一个常数，条件跳转直接将不可能的跳转分支去除。
- 回到第一步。按此步骤重复 2000 次左右。

这样就可以消除大量的无用赋值，大幅提升代码优化效果。

3.4 基本块合并

在 SSA 结构中，跨基本块的操作都需要消耗大量的代价，因为存在 phi 函数的翻译问题。这点在寄存器分配中也是很重要的，因为基本块结束后就要将所有的寄存器放回内存，由此产生大量的访存指令。

在做完上面的三个操作后，可能会产生一些无用的块，这时候需要将这些块合并，以避免产生无意义的跳转基本快的浪费。合并基本块本身并不难。对于每一个基本块，它可以和另一个基本块合并当且仅当当前基本块只能从那个基本块到达，且另一个基本块与它相连只存在它一个后继节点。新产生的基本块会

继承两个块的所有 phi 函数。因为合并两个块的特殊关系，所以实际上只需要将后面的节点的后继点和前面的节点的前继点作为新产生的点的后继点和前继点。

实际上，由于限制过于严格，这个优化能发挥的作用有限。也许有更好的合并基本块的方式留给我以后有机会的时候再探索。

3.5 公共子表达式合并

这个优化大部分情况下其实可以利用复制传播实现。但是对于运算指令的情况下，还是需要这个优化来消除多余的计算指令。SSA 和复制传播做完的情况下，公共子表达式可以合并，当且仅当两个表达式完全相同。所以实现这个优化的算法十分简单。

- 扫描所有的中间代码，遇到三元计算指令，就将计算指令的右部操作数通过 map 的数据结构保存并将其映射成计算指令的结果
- 之后所有具有相同计算数的计算指令合并，具体做法，就是在 map 中查找计算指令的结果，并将计算转换为赋值。

之后会多出一些新的赋值变量，接下来可以再一次做复制传播和常量传播来优化。

3.6 寄存器分配

至此进入生成 mips 优化部分。分配寄存器对于机器码执行效率很重要，它所占的影响我个人认为甚至要高于上述优化。

我的寄存器分配策略如下：采用一个贪心的算法，对于每一个基本块，每次遇到一个对一个变量的使用，都无条件的将一个寄存器分配给它。当寄存器不够的时候，我优先将下次使用距离最远的变量踢出分配寄存器的集合，并视情况将寄存器中的值存回去。下次只使用距离定义为，从当前语句出发，下次能被使用到的位置距离当前的距离。每当跨越基本块，进入函数调用前，我都会把寄存器尺清空。

经过一些实践，我发现寄存器基本上不会有被全部使用完的情况，影响寄存器分配效率的关键因素其实是将寄存器放回去的策略。如果每次基本块结束都将所有寄存器都放回，会导致程序运行效率大大降低。所以寄存器的放回策略，其实要比分配策略更加重要。

当我们跨越基本块的时候，真的有必要将所有的寄存器都放回去码？答案是没有必要。在 SSA 形式下，没一个变量一旦被用到那么它一定已经有一个确定的值。所以，当我们离开一个基本块时，没有必要将一个非当前基本块定义的变量放回去。因为它们在离开定义它们的基本块时就已经被赋值了，且存进内存了。还有一种情况，如果这个变量不会在后续节点中使用，同样也不需要将这个变量放回寄存器。

上面两个有关寄存器放回的优化看似微不足道，实际上它发挥的作用要远远大于我上面所说的所有优化。效果十分惊人。可能也是在大优化的铺垫下，小优化才会发挥出它的功能吧。

3.7 窥孔优化

剩下的只有一些比较简单容易实现的优化了。

- 对于”jump” 后直接出现 jump 目标”label” 的情况，可以很容易将这个 jump 去除。
- 可以将类似于 sne 和 beq 串在一起的指令进行合并，条件设置为 1 和为 0 跳转可以合并成一个条件跳转指令，这样可以产生一些比较微小的优化
- 还有一些运算强度削弱操作，比如遇到 mod 指令时，如果 $a \bmod b$ 下有 $a < b$ 可以直接不执行这条指令，这样可以产生以一条条件指令换一个除法指令这样稳赚不赔的优化。再比如遇到乘法指令，如果是对常数的乘法，且常数为 2 的次幂，可以直接优化成左移操作，可以节省 2 的代价。

3.8 代码内联

在 c 语言环境下，inline 字符代表着可以将这个函数和代码块内联在一起。所谓代码内联就是将函数代码直接放在应用这个函数的代码区。这样可以节省大量的压栈操作。还可以方便后续的赋值传播常量传播等优化。

我的内联策略是，除了递归函数，其余函数全部采取内联操作。将函数的中间代码直接嵌入调用它的代码中，并将参数变量重新命名填入符号表。这使我在处理一些常量传参函数时有很大的优势。

4 遇到的困难以及感想

在本次作业中，我总计完成了大概 4000 行代码，全部都由 *cpp* 完成。我遇到的主要的困难也都与 debug 和对于 cpp 这个语言的不熟悉。比如一开始错误处理时我想使用多文件编写，但是却连 Clion 的编译文件都不会写。之后我通过大量的上网查阅相关资料和询问同学，解决了对于 cpp 的使用问题。在有一定代码累计后，我发现我对于 cpp 的使用变得熟练，debug 速度也更加的快了。所以我感觉通过编译实验这门课我最大收货不仅仅是自己独立写出了一个编译器，更是通过自己克服各种困难用一种新的语言写出了一个比较大型的工程。我觉得这是一个十分令人自豪的事情。

编译实验至此就结束了，经过漫长的码代码和痛苦的 debug 后，我得到了一个我之前从来没有想过能够通过我自己实现的东西。虽然最后的竞速排名可能有高有低，但是通过自己的奋斗和竞争争取到的东西永远是最美妙的。