优化文档

中端优化

> 在中端优化的全过程都要保证SSA，并维护所有llvm value的use-def关系

SSA

基本原理

SSA即静态单赋值，每个变量（在LLVM中表现为虚拟寄存器）都仅在赋值时被定义，且仅会被赋值一次，这样的中间代码约束可以有效地降低后续各类数据流分析的复杂度

为实现SSA，在目前的LLVM ir中，使用的是alloca/load/store的内存形式，而访问内存的效率比访问寄存器要慢很多，因此需要将局部变量改装为真正的SSA

经过改造后，所有的局部变量都可以表示为虚拟寄存器的形式，这样通过后续的寄存器分配可以减少Mips代码中对内存的操作以提高效率

之所以需要借助内存来实现SSA，主要的原因在于对某个虚拟寄存器内变量的不同分支赋值会违反SSA规则，因此SSA主要关注于"交汇块"，采用在交汇块头部插入**Phi指令**的方式

phi指令形如 `%value = phi [%v1,%b1],[%v2,%b2],...` ，意为根据跳转至当前块的来源块来确定本value的获得值，从而解决了分支问题

流图-支配分析

这部分做到的是，确定每个变量可能的来源基本块

引起"来源基本块"的歧义的就是数据流的分支，其余的地方不需要phi；因而只需要考虑分支处的数据来源，即基本块"汇聚"的地方

这些"汇聚"的块就是某个块的**支配边界**，为找到支配边界，需要依次求出流图和支配树

流图分析可以得到一个基本块的前驱和后继，只需要根据块内遇到的第一个跳转的目标块即可确定（这个跳转之后的指令都是不可达的，可以直接删去），根据跳转的目标可以构建一个控制流图

支配分析可以分为三步：

- 对于一个块B，求出其支配它的块D和它支配的块d，D是其流图中所有的前驱的支配者的交集加上其自身，根据这一原则循环分析直到D不再变化
- 将支配集转为支配树，即求出每个块的直接支配者，就是d集合减去d中所有块的严格支配块；由于每个块仅有一个直接支配者，因此是一个树形结构
- 计算支配边界，对流图中的每条边(a,b)开始，将b加入a的支配边界集合，并将a转为其直接支配者，直到a严格支配b

LLVM中SSA实现

根据伪代码，遍历每个变量，考虑其每个定义所在的块，在其支配边界中插入空的Phi指令（即没有块来源等数据的Phi）

之后，对变量进行重命名，实际上，是对所有非数组的局部变量的原始的alloca-load-store方式的内存操作全部代替为Phi提供的虚拟寄存器，并补全Phi的数据

为此需要对每个函数在支配树中进行深度优先遍历，在每个块中：

- 对局部变量的Alloca，记录该地址，移除指令
- 对Store，将该地址中的值更新，移除指令
- 对Load，用该地址存储的值代替原本Load的值，移除指令
- 对Phi，用该value将其所代替的原本局部变量地址中的值更新

之后，更新其流图中的所有后继块首部的Phi的分支数据，将来自本块的情况加入对应的Phi指令

此外，在移除指令以及后续的更新中，需要维护use-def链以及流图、支配等分析数据

GVN

目的是尽量实现更多的**指令复用**，不要每次使用都重新再计算一遍

GVN可以采用简单的HashMap代替法，即将每个指令对应的Value加入到HashMap里，如果遇到HashCode相同的指令可以直接从Map中代替

对于一些具有交换律或逆交换律的指令如`add`可以在放入HashMap时同时放入两种情况的HashCode

GVN可以直接代替，是因为：

- SSA形式的中间代码保证了不会出现重复赋值的行为
- 后续的GCM可以根据指令间的依赖关系保证修改后指令的正确性

GVN需要和GCM进行绑定，以保证正确性，这是因为GVN可能用不同分支块中的指令来代替，即GVN中，某些指令使用的Operand可能并没有定义

GCM

GCM的作用有两个：

1. 类似于循环不变式外提，尽量降低循环内部的复杂度
2. 弥补GVN的失误，保证每个指令的Operand，一定要在使用前定义，即定义点所在的块一定要支配使用点所在的块

它通过挪动指令的位置来实现，具体如下：

- 固定(Pin)某些指令，包括Phi，跳转，对有副作用函数的Call；此外，由于Load和Store对于def-use的判定比较复杂，这里也固定了
- 尽量将每个指令向前移动，直到其Operand的定义点；具体计算时，需要首先移动一个指令所有的Operand，再移动该指令
- 尽量将每个指令向后移动，直到其所有use的最小公共祖先处；需要首先移动所有每个指令所有的User，再移动该指令，这里特判一下：如果User是Phi，那该指令需要在phi对应的来源块就定义而非phi所在的块
- 在以上求出的指令的活动范围下，为每个指令选择一个循环深度最小的块，同时尽量使其后移（即支配树深度更大）
- 在块中，同样根据其Operand和User的限制，确定其在块内的位置

还有两个细节：

一是如何求循环深度，由于错误处理那里一般会保存有深度，因此可以直接使用，但之后可能频繁修改流图，需要频繁维护对应的循环深度

也可以进行专门的循环分析，如果基本块a支配b并且存在b到a的回边，那么就存在一个循环

二是为什么指令块选择时需要使其尽量后移，因为GCM是可能把本不会执行的分支中的代码提前的，这样就可能导致不必要的性能消耗，而且还无法通过分析删除，因此要尽量将其后移使得其尽量晚出现

死代码删除

将一些无用的指令删去

删去的原则是：寻找“有用”的指令的闭包，剩下的指令就都可以删去

这个闭包的初始集合是指所有有副作用的指令，包括：

- 输入输出指令
- call有副作用的函数
- store指令，并且其存储的地址是全局变量以及之后会被load或被作为参数传递的局部数组
- 跳转指令

之后，其操作数也都是"有用"的，重复直到构建出完整的有用指令集

函数内联

为了降低函数调用时保存和恢复现场、跳转、参数传递的消耗

> 补充：函数内联的最大的作用是为其他优化遍暴露更多的机会，而非简单地减少函数调用的几个指令

采用的方法是直接将函数的内容植入原本的call指令的位置，这里只对非递归，且调用函数也非递归函数进行了内联

首先，要将调用函数中的指令逐个复制，当然要求是深克隆

使用一个HashMap存储原始函数中的Value与内联到本函数中的Value的对应关系，在深克隆时用到的Value都应该从HashMap中取得

这个HashMap应该具有一些初始值：

- 函数参数：函数的形参对应于原本要传入的实参
- 函数的块：函数的每个块都新建一个作为对应，供跳转指令使用
- 全局变量：对应的值就是其本身，因此也可以不将其实际放入，而是用一个get方法判断，同样的还有ConstNumber

此外，为确保函数在克隆时使用到的所有变量都已经进行了克隆，因此需要遍历函数的支配树来对每个块逐指令地进行克隆，或者从入口块开始BFS遍历

需要特别考虑Ret指令，将其变为对原函数后继块的跳转指令，并将对应的跳转情况作为加入后继块的相应Phi指令中

如果遍历支配树，需要特别考虑原函数中的Phi指令，Phi中的情况不一定已经克隆过，因此需要等到所有块都遍历完成后，填充Phi指令的所有情况以及来源

这样，就得到了对应函数的深克隆块序列，下面需要将其插入原本块中：

- 将call指令所在的块根据Call指令分成两个，前驱块插入一个到块序列入口的跳转，后继块由对应序列跳转到
- 将原本Call指令的值代替为后继块中相应的由序列ret指令得到的Phi指令的值
- 重新进行函数调用、流图、支配等各类分析

> 需要注意，如果启用了内联，那么就需要采用较好的寄存器分配算法，否则大体积的函数可能导致寄存器被错误占用，反而降低效率

常量折叠和传播

对于一些可以在编译期就能求出值来的指令，将其替换为常数

一类是操作数为常数的，如 `%t0 = add 3,4` ，则可以将 `%t0` 替换为常数7

另一类是结果为常数或可以简化的，例如 `mult %v,0` ， `div 0,%v` 可以替换为0； `%t0 = add %v, 10 ; %t1 = add %t0, 10` 可以替换为 `%t2 = add %v, 20` （前提是 `%t1` 不会被其他指令所使用）

此外，对于一个有且仅有一个返回值，返回值为常数且没有副作用（不存储全局变量值，不调用有副作用的函数，传入参数没有指针类型）的函数，其调用也可以用其常数返回值代替

后端优化

> 后端优化是指，"机器相关的优化"，但并不代表一定要在后端实现；由于中端有SSA与丰富的分析流数据，同时每种优化可能还会带来更多的优化机会，因此个人认为优化遍应该尽可能在中端进行（例如乘除法优化）

消除Phi

对于一个简单的phi语句 `%v1 = phi [%v2,%b2], [%v3,%b3]`

消除phi即需要在汇集处的前驱块的末尾加上move指令来在不同情况下修改寄存器

这时需要考虑b1的前驱块（例如b2)的后继块的数目：

- 若仅有一个后继，直接放在末尾
- 若有更多后继，则需要考虑到指令对于其他后继的影响，应该在b2到b1中创建一个中间块midb1，在其中放置move指令

还需要考虑的是由于所有phi指令是并行赋值的，各个move应该互不干扰，因此可以使用一个中间寄存器进行赋值，之后将冗余的虚拟中间寄存器删去

"冗余"的中间寄存器是指其对应的原始move指令的目标寄存器并没有作为值赋给其他寄存器，或者其赋值给了其他寄存器但是是在它被赋值之前

寄存器分配

采用的是图着色寄存器分配方法，与教程上的略有不同：

首先要进行活跃变量分析

遍历每个块，计算其中先定义后使用的Value，以及先使用后定义的Value，得到每个基本块的def和use集

之后，需要在流图中从后向前分析，使用以下公式计算直到达到不动点：

$$in[B] = use[B] \cup (out[B] - def[B])$$

$$out[B] = \cup_{\text{后继块}P}\ in[P]$$

最后，在块内进行指令集活跃分析，同样要从后向前分析，在定义点从活跃变量集合中去掉，在使用点将其加上，最终得到每个指令处的活跃变量集合

在每个指令点，这些活跃变量待分配寄存器，因此是相互冲突的，需要将其放入冲突图中，它们之间相互连接构成完全子图

经过上述活跃变量分析，得到了冲突图，之后，按照以下顺序操作冲突图：

- 简化：去掉度数小于K（待分配寄存器的数目）的节点（不包括Move指令的涉及的节点，即非冻结节点），将其入栈
- 合并：在确保度数大于等于K的节点数目不增加的情况下，将move指令的目标以及源值对应的节点合并，之后分配寄存器时它们分配同一个寄存器，继续简化
- 冻结：重复以上步骤直到不再变化，这时需要将一个Move指令冻结，放弃合并它的机会，之后可以继续进行简化

- 溢出：以上步骤也无法进行，则可以选择一个合适的节点（这个节点可以随机选，也可以根据函数来计算，如下）加入栈中，这时可能暴露更多简化的机会，重复简化，直到冲突图为空
- 分配：将栈中的节点依次加回图中，若邻居占用了全部的寄存器，则不分配；否则随机选择一个寄存器来使用

在"溢出"一步中，选择的溢出节点的代价按照以下函数计算：

$$f(v) = \sum_{use \in B} \alpha^{loopDepth(B)} + \sum_{def \in B} \alpha^{loopDepth(B)}$$

$$cost(v) = (\sum_{u \in conflictMap[v]} f(u)) - f(v)$$

其中α是一个待定的常数，这里选择了10

> 由于时间原因，并没有做restart，但这个阶段非常重要，不过在竞速样例强度较低的情况下可能不明显

指令选择

Mars会提供一些指令，它们在执行时会被翻译为多条指令，而这种翻译可能并非是最优的；又或者某些指令可以用更简便的来代替：

- 与常数进行比较的各类跳转指令如 `sgt r0,r1,1 bne r0,zero,label` 可以合并为 `bgt` ，这样做的理由是在Mars中， `sgt` 类指令如果同常数作比较的话一律会视为32位立即数用4条指令翻译，而 `bgt` 指令在常数可以用16位立即数翻译的情况下，可以用较少的指令进行比较并跳转
- 教程中提到的 `subi` 指令

在实现上，第2种可以在后端实现，而第1种则是在中端实现的，因为后端没有进行专门的数据流分析，并不能确定 `sgt` 指令是否在之后还会被使用，因此不能贸然进行合并，而在中端有着use-def数据，则可以对仅有一个user的 `sgt` 指令进行合并

基本块合并与重排序

在llvm中间代码中，每个基本块都以一个跳转指令结束，而在Mips中，指令是按照顺序执行的，并没有基本块的约束，因此如果跳转指令指向的就是下一个块，那就可以去掉块尾的跳转指令

更进一步，可以通过对块的排序尽量增加这样的去掉跳转的机会

可以对所有mips指令以J跳转分割为若干个块，若末尾有J跳转，则将跳转的目标块移动到其后一个；若没有J跳转，则重新选择一个未排序的块

乘除法优化

乘除法的代价较高，若其一个操作数为常数，则可以对指令进行优化

乘法的优化为：

$$r * b = (r << m) + (r << n)$$

其中常数$b = (1 << m) + (1 << n)$

对于这样的常数，即其二进制表示中有至多两个1的数，将其转为加法-移位是可以节省时间的，而多于两个，则代价会更大，就没有必要了

除法的优化为：

$$\frac{dividend}{divisor} = (dividene \times multiplier) >> shift$$

根据论文中的算法，所需要的multiplier（即m）和shift（即l）应满足：

Theorem 4.2 *Suppose m, d, ℓ are nonnegative integers such that $d \neq 0$ and*

$$2^{N+\ell} \le m * d \le 2^{N+\ell} + 2^{\ell}. \tag{4.3}$$

Then $\lfloor n/d \rfloor = \lfloor m * n/2^{N+\ell} \rfloor$ for every integer n with $0 \le n < 2^N$.

这样得到的m是高于32位寄存器所能存储的范围的，因为希望dividene*multiplier得到的结果不要在HI和LO中分开，而总是在一个之中，即在HI寄存器中

这样它和dividend的乘法就不能简单地使用mult了，而应该采用分配律：

$$(dividene << 32 + dividene * (m - 2^{32})) >> shift$$

使得它们能够存放在32位寄存器中运算，这里加号的两侧都是取高32位运算的

这样仍然会产生问题，加法是可能溢出的；但是由于在加法之后会进行算术右移操作，因此可以对两侧的操作数进行预先的移位以及进位处理，降低其大小，来防止加法时出现的溢出

此外，语义要求的除法取整是向0取整而非向下取整，因此在dividend为负数时需要加一来补充，可以通过加上逻辑左移31位的结果来实现

窥孔优化

对于一些简单的指令组合，在一个小范围内进行优化：

- move r, r
- move r0, r1 ; move r0, r2 这里要判断r2≠r0
- sw r0, addr(sp) ; lw r1, addr(sp)