代码优化文档

21371133 傅可树

编码前的设计

1. 中间代码优化

1.1 基本块划分

首先要划分基本块。一切的优化都是源于基本块良好的性质(块内没有跳转，都是顺序执行)。由于我之前做的LLVM划分的是伪基本块——一个函数只有一个基本块，每个基本块内都可能有跳转语句。这是极为粗糙的划分。

- 按照跳转语句把基本块划分开。
- 去掉无用的标签。之前生成的时候有很多不必要的标签插入。标签和标签之间甚至没有语句。
- 基本块之间的跳转连边，形成一个有向图。

1.2 Mem2reg

phi的定义：phi函数是一种在控制流图中用来表示多个不同路径上的值的方式。通常，phi函数用于合并不同的值，这些值可能来自于不同的基本块或控制流路径。
在编译器中，phi函数通常用于表示在条件分支语句中不同分支上产生的不同值。
引入phi的作用：
phi函数通常用于在控制流图中处理分支情况，而$Mem2reg$则关注将内存操作优化为对寄存器的操作。在执行$Mem2reg$时，编译器可能会引入phi函数来处理不同分支上的值，因为转换为SSA形式后，需要用phi函数来表示不同路径上的变量赋值。
phi函数的引入和$Mem2reg$的优化一起协同工作，帮助编译器更好地理解和优化程序的控制流和数据流。
但phi插在哪里，又应该如何消除便成了问题
这里就要引入支配的概念。
支配：如果CFG中从起始节点到基本块y的所有路径都经过了基本块x，我们说x支配y
严格支配：显然每个基本块都支配它自己。如果x支配y，且x不等于y，那么x严格支配y
直接支配者：严格支配n，且不严格支配任何严格支配n的节点的节点(直观理解就是所有严格支配n的节点中离n最近的那一个)，我们称其为n的直接支配者
支配边界：节点n的支配边界是CFG中刚好不被n支配到的节点集合.
支配便捷是我们插入phi的核心，假设X中出现了对于变量x的定义def(store)指令，那么对于X可以支配的节点，如果其中没有出现其他定义，那么我们可以在使用x的时候，直接使用def中给x赋的值。

但是，如果进入了X的支配边界，就可能有其他导向边界的路径。而这些路径上也出现了对于x的def，那么我们会根据定义发现这个边界就是这些路径的交汇点。
由此我们得到了插入phi的思路：

- 如果我们发现了对于某个变量的def，找到他所在基本块的支配边界，在那里插入一条phi指令。
- 如果已经插入了，那就增加一条分支。
- 同时，phi指令也属于def需要迭代。

1.3 死代码删除

函数中有一些不会被运行的代码，或者是运行了一大段，但对后面结果毫无作用的代码(产生的变量不参与后续运算)。这些代码就是死代码。
删除死代码可以缩小程序规模，提高程序运行效率。(类比注释，我们其实要做的就是把代码中隐式的注释部分给提取出来)
具体执行步骤:

- **活跃变量分析**: $def-use$链的构建。执行活跃变量分析，确定每个变量在程序中的活跃范围
- **标记未使用的变量**:标记在活跃变量分析中被确定为未使用的变量。
- **控制流图分析**：分析控制流图，找出不可达的代码块
- **标记不可达代码块**：根据上一步的分析结果，对不可达代码块进行标记。
- 合并上面的分析结果，把死代码标记，并进行删除。
- **更新符号表**，因为死代码删除后会对后续的符号表等产生影响。我们要重新做一遍符号表的更新。避免产生bug。

2.目标代码优化

2.1 寄存器分配

寄存器分配是属于目标代码的优化。
$mips$指令集只有32个寄存器。我们在使用的时候需要以一种比较合理的方式去分配。但某些变量有冲突关系。这可以抽象成图着色问题。但这被证明是一个NP完全问题，只有不可接受的阶乘解法。我们尝试用比较优的贪心策略来构造。

- 首先，先类似死代码删除，构建数据流方程。
- 再进行数据流分析。得到冲突图。（冲突图的构建可以是粗粒度的，也可以是细粒度的。但需要保证的是，宁可代码优化程度不高，也不能丢失正确性，这是宗旨）
  下面简单介绍一下图着色算法。
  从冲突图中选择度数（相邻变量的数量）最小的节点，为它着色。然后，将该节点从图中删除，重复此过程，直到所有节点都被着色。这是一种高效的图着色算法，通过节点的合并和分割来降低干涉图的复杂性，进而提高图着色的效率。

2.2 乘除法优化

- 通过算术运算将除常数的运算，用乘法和二进制移位运算代替。
- 将乘法运算变为移位运算。

2.3 窥孔优化

- 一个寄存器被存入之后又被读取。此时连续的两条sw, lw指令可以消解
- 加法中加数为0，直接使用li
- 减法中减数为0,被减数直接赋值。被减数为0,减数取相反数后赋值。
- 乘除法中有0，直接赋0

编码完成之后的修改

1.支配树的实现

1.1 支配树的定义

对于一张有向图，我们确定一个起点S。
对于一个点k，称x为其支配点当且仅当，删去点x后，S无法到达k，即S到达k的所有路径都必须通过x。
容易发现，一个点的支配点不止一个。
同时，也很容易发现，如果我们将一个点k与其最近的支配点连边，那么会形成一个树形结构，这个树形结构即支配树。
接下来我们所成的支配点一般指其在支配树上的父亲。求解支配树可以用$Lengauer-Tarjan$算法，做到$O(nlogn)$的时间复杂度内求解。

1.2 支配树的构造

首先，显然一个点k的支配点是所有能够直接到达k的点在支配树上的LCA。那么就很简单了，我们按照拓扑序依次做下去，建立反图找到所有能够直接到达当前点的点，倍增求解LCA即可。
$Lengauer-Tarjan$算法分为三步

- 求出dfs树，得到每个点k的dfs序d_k
- 求半支配点
- 求支配点

1.3 半支配点

一个点k的半支配点是指一个dfs序最小的点x，使得x可以通过一条路径$x, x_1, x_2, ..., x_c, k$到达点k，求满足对于任意$1 \leq i \leq c$，有$d_{x_i} > d_k$.
只要能求出半支配点，就有办法求解支配点，所以我们先来求解半支配点。
对于一个可以直接到达点k的点x，有两种可能的情况：

1.若$d_x < d_k$,则x可能是k的班支配点。
2.若$d_x > d_k$，则考虑其所有祖先u满足$d_u > d_k$，u的半支配点可能是点k的半支配点。
这两种情况是充分的，易证。

考虑按照dfs序倒序考虑所有点k。枚举所有能够直接到达点k的点x，第一种情况可以快速维护，但第二种情况需要考虑其所有祖先，我们不能暴力枚举。
考虑维护一个带权并查集，每次处理完一个点k，我们就将k往其 dfs 树上的父亲合并，同时维护一个点并查集中除根以外的所有祖先中，半支配点dfs序最小的那个点，就可以在一次$O(logn)$的时间复杂度内完成第二种情况的查询。
至此，我们已经在$O(nlogn)$的时间复杂度内完成了半支配点的求解。

1.4 利用半支配点求解支配点

考虑点k到其半支配点x的这条链(不包括x)上半支配点dfs序最小的点u及其半支配点v，则k在支配点有两种情况：
1.若$x = v$，则k的支配点就是x.
2.若$d_x > d_v$，则k的支配点是u的支配点
可以证明这两种情况是充分必要的。
我们再次考虑按照dfs序倒序处理。
对于一个点k，我们考虑其在半支配点x，现在需要完成在dfs树上查询k到x这条链（不包含x）中半支配点dfs序最小的点。
我们发现这个查询就在半支配点的求解中出现过。
考虑在求解半支配点同时求解支配点。当我们求出k的半支配点后，将 k的半支配点与k连边，维护半支配树。同时令x为k在 dfs树上的父亲，枚举x在半支配树上的儿子y，容易发现当前并查集中维护的正好就是y到x(不包含x)中半支配点dfs序最小的点。
不过一个问题是，如果查询出来的结果是第二种情况，此时我们并不知道得到点的支配点，所以把得到点先存下来，最后再按照dfs 序正序处理一遍就好了。
同时，如果我们每一次都枚举k的父亲x在半支配树中的所有儿子，会导致多次枚举了某些点，所以每次枚举完之后需要将这些儿子清空才能保证复杂度正确。
至此我们完成了整个$Lengauer-Tarjan$算法。复杂度为$O(nlogn)$

```
void add(int x,int y,int tp){
	qxx[++qxx_cnt]=(edge){y,h[x][tp]},h[x][tp]=qxx_cnt;
	return;
}
int dfn[maxn],fa[maxn],alt,back[maxn];
void dfs(int x,int f){
	dfn[x]=++alt,fa[x]=f,back[alt]=x;
	for(int i=h[x][0];i;i=qxx[i].next){
		int v=qxx[i].to;
		if(dfn[v])continue;
		dfs(v,x);
	}
	return;
}
int idom[maxn],sdom[maxn],mi[maxn],dsu[maxn];
int top(int x){
	if(dsu[x]==x)return x;
	int gt=top(dsu[x]);
	if(dfn[sdom[mi[x]]]>dfn[sdom[mi[dsu[x]]]])mi[x]=mi[dsu[x]];
	return dsu[x]=gt;
}
int n,m;
void lengauer_tarjan(){
	for(int i=1;i<=n;i++)sdom[i]=mi[i]=dsu[i]=i;
	dfs(1,0);
	for(int p=alt;p>=2;p--){
		int x=back[p];
		for(int i=h[x][1];i;i=qxx[i].next){
			int v=qxx[i].to;
			if(!dfn[v])continue;
			top(v);
			if(dfn[sdom[mi[v]]]<dfn[sdom[x]])sdom[x]=sdom[mi[v]];
		}
		dsu[x]=fa[x],add(sdom[x],x,2),x=fa[x];
		for(int i=h[x][2];i;i=qxx[i].next){
			int v=qxx[i].to;
			top(v);
			idom[v]=(sdom[mi[v]]==x?x:mi[v]);
		}
		h[x][2]=0;
	}
	for(int i=2;i<=alt;i++){
		int x=back[i];
```

```
                if(idom[x]!=sdom[x])idom[x]=idom[idom[x]];
        }
        return;
}
void pre(){
        for(int i=1;i<=m;i++){
                int x=e[i].x,y=e[i].y;
                add(x,y,0),add(y,x,1);
        }
        lengauer_tarjan();
}
```

2.$def-use$链的构建

死代码删除中最重要的就是$def-use$链的构建。$def-use$链的构建。

- 比较重要的点就是多个分支的$Def-use$合并。$Union$中就处理的是合并的情况
- $def-use$链需要迭代处理。直到没有变化为止。
- add_chain中写的是进入某个基本块之后，对于块内元素$def-use$链的处理。

```
void travel(int rt,int vector<string,vector<int>> vec){
    add_chain(rt,*vec);
    for (int i=0;i<a[u].son.size();i++){
        int v=a[u].son[i];
        travel(v,vec);
    }
}
void Union(int u){
    for (int i=0;i<In[u].size();i++){
        id=In[u][i];
        for (int j=0;j<vec[id].size();j++){
            vec[u].push_back(vec[id]);
        }
    }
}
void add_chain(int rt,vector<string,vector<int>> *defuse){
    if (def==NULL) return;
    if (a[rt].type==VAR){
        defuse.push_back(a[rt].name);
    }else{
        if (a[rt].type==ASSIGN){
            defuse.push_back(a[rt].name);
            add_chain(a[rt].son[0],defuse);
        }else{
            if (a[rt].type==Exp){
                add_chian(a[rt].son[0],defuse);
                add_chain(a[rt].son[1],defuse);
            }
        }
    }
}
```

3.图着色寄存器分配

图着色寄存器分配本质就是用启发式的算法实现分配。

```
vector<int> spill;
set<int> g[N];
inline bool check(int u,int c){
    for (auto v:g[u]){
        if (color[v]==c){
            return 0;
        }
    }
    return 1;
}
void colorGraph(int cnt){
    for (int i=0;i<g[u].size();i++){
        S.insert(i);
    }
    while (!vec.empty()){
        int u=*S.begin();
        S.erase(u);
        if (g[u].size()>=cnt){
            spill.push_back(u);
        }else{
            int col=0;
            while (!check(u,col)){
                col++;
            }
            color[u]=col;
            for (auto v:g[u]){
                g[v].erase(u);
            }
        }
    }
}
```