编译优化文档

总述

就最终呈现的结果来看，我的编译器完成的优化总体种类不多，比较主要的有：中间代码的代数运算、基本块合并、无用代码删除，目标代码生成阶段的寄存器分配、乘除法优化、基本块合并与排序输出、窥孔优化等。

中间代码优化

事先代数运算与Const常量传播（完成于生成过程中）

这一部分的优化思路是以编译过程中的计算取代编译生成的程序中的运算。

事先计算时，基于赋值式中的常量，尽可能提前运算出结果。基于语法分析形成的表达式树，可以以类似属性计算的方式，在`PrimaryExp → '(' Exp ')' | LVal | Number`的分析子程序中取出Number的值，将常量的值作为属性在`Exp`或`Cond`的分析过程中传递，直到遇到非常量或者整个表达式计算完毕为止。

常量传播方面，我的编译器主要实现了const常量传播。因为const修饰的常量值是不能改变的，因而在`Lval`分析子程序中，可以在匹配到常量时，直接将其替换为相应的值，从而纳入到事先计算之中。

对于包含常数四则运算等的代码，这一方法能取得较为明显的效果。即使不能直接得到结果，常量的代入也可以为后续优化创造条件（如乘除法的优化要求乘除数为常量）。此外，基于到达定义分析的全局常量传播等，可以进一步减少代码中事实上的变量（无法事先判断当时该变量的值或者判断过程过于复杂）的数量，有助于更加彻底地完成事先计算过程、提高性能。

基本块合并（完成于优化方法）

在中间代码生成的过程中，由于每个if-else语句、循环语句配套的基本块数量是一定的，因此可能会产生不少碎片基本块，例如，在我的编译器中，一个for循环后紧跟一个for循环块，就会出现前一个for循环块的出口块中只有一条语句，用于跳转至下一个for循环块的初始设定块。这些碎片基本块的出现增加了不少不必要的跳转，导致程序性能的下降。基本块合并等操作就是尽可能地减少碎片基本块、减少不必要的跳转语句，从而提升性能。此外，基本块合并能够为基本块内的优化（窥孔优化等）提供更多空间，增强这些优化方式的效果。

在我的编译器中，中端基本块合并主要进行以下操作：检查每个基本块可以跳转进入的基本块（除函数调用外），明确各个基本块的前后继关系（这一关系在之后活跃变量分析等也会用到）。随后检查是否存在两个基本块a/b，a只有一个后继基本块为b，b只有一个前继基本块为a，若有则将a、b合并。

在中间代码上，br条件跳转是将条件真、条件假的跳转操作合并为一条语句，一定程度上限制了基本块的合并操作。在mips汇编代码上，它会被拆分成条件真跳转和条件假时无条件跳转两个单label跳转语句，此时可以进一步合并基本块，见后端优化中的基本块合并部分。

无用代码的删除（完成于优化方法）

在源程序中，可能会引入一些没有实际意义的代码。这些无用代码，或是进行了运算，但没有实际意义，如声明一个变量a，却从未将a应用到其他变量的计算或输出之中；或是在运行过程中不可能到达，如条件恒真时的else块、return语句后的语句等。前者会导致实际运算量增加、代码运行效率下降，相应的，删除这些无用代码可以提高程序性能；后者虽然不造成运算量增加，但增大了生成代码的代码量，对这一部分无用代码的优化是降低程序占用空间的优化。总之，适当删除无用代码是必要的。

这些无用代码，一部分可以在中间代码生成过程中直接发现并删除，如在逻辑与表达式中检测到0，则不再往下分析该表达式（逻辑与表达式遇到0不会再往下计算）；同理，在逻辑或表达式中检测到1，也不再往下分析该表达式。除此之外，还有相当一部分的无用代码需要在中间代码生成之后，在优化方法中发现并删除。

在优化方法中，我们认定以下代码为无用代码：

- 声明(alloca/全局定义)了但从未被使用的变量。分为两种情况：
  - 严格未使用，即声明后从未进行过load/getelementptr/getint赋值操作的变量，则与这个变量相关的所有语句(alloca/store)均为死代码；
  - 仅出现getint赋值操作、无load/getelementptr操作的变量，为了保证程序的IO行为与优化前的一致，则保留声明语句及getint赋值语句，其他相关语句(store等)均为死代码。
- 该条语句所得临时寄存器值从未被使用的计算语句（函数调用除外，因为函数可能通过内部printf输出、操作全局变量等方式造成实际影响）；
- 某个基本块中，第一条br/ret语句之后的所有语句，因为程序运行到br/ret之后必然跳转离开该基本块；
- 从函数入口（函数的第一个基本块）到函数出口（ret语句）过程中不可到达的基本块，整个基本块都是无用的；
- 在整个程序中从未被调用过的函数。

某条死代码的删除包括两个部分：

- 从代码集合中删除这条代码；
- 删除这条代码对应的所有Use，也即删除所有User对这条代码的Use。

删除一部分死代码之后，部分`Instruction`的Use列表会发生改变，产生新的未被使用过的变量/临时寄存器值，形成新的优化机会。因此可以考虑多次进行无用代码删除，直至所有代码在一遍无用代码扫描删除后不再发生改变为止。

更进一步地，为了删除重复定义覆盖等造成的无用代码，可以引入到达定义分析等工具，将所赋值从未被使用过的赋值语句等删除。我的编译器暂未对这一方面的无用代码进行进一步删除。

后端优化

寄存器分配（完成于生成过程中）

寄存器分配的效果相当明显，由于访问寄存器的速度远快于访问内存的速度，因而寄存器分配可以通过减少存取内存的方式大大降低运行开销。

对于跨块变量的全局寄存器分配，我采用的是图着色方案。其基本实现为：

- 通过活跃变量分析，确定各基本块的use和def集合，迭代求出in集合；
- 基于in集合确定跨块情况和冲突图：同时在多个in集合中出现的变量属于本次分配中考虑的跨块变量；处在同一个in集合的变量互相冲突。
- 采用启发式图着色算法，得到着色序列，再根据冲突情况依次分配寄存器（着色），完成全局的寄存器分配。

对于不涉及跨块的变量，由于一个基本块中可能有大量的赋值语句，若将所有临时变量全部放入内存，仍然会带来相当大的访存开销。因而，我的编译器以基本块为单位，再进行一次寄存器分配。其基本实现为：

- 对基本块所在的函数检查全局寄存器的分配情况，基本块寄存器分配不占用跨块活跃变量的寄存器；
- 检查基本块内是否存在已经被分配过寄存器的变量，若已经被分配过则不再分配，保证一个变量对应一个寄存器；

- 对于剩余变量，采取**线性扫描**的方法，确定各个寄存器活跃的起始时间和结束时间；
- 确定一个活跃变量列表，当到达时间t时，检查：
    - 若某变量从时间t开始活跃，则检查各个寄存器，查看是否存在活跃变量列表内的变量（即冲突变量），不存在则将该寄存器分配给该变量。若没有这样的寄存器，则分配内存。同时，将该变量移入活跃变量列表中；
    - 若某变量在时间t停止活跃，则将其从活跃变量列表中移出。

在代码生成时，后端首先寻找对应的操作数是否被分配了寄存器，是则直接从寄存器中取值，不是则先从内存中取值到指定的临时寄存器中，再从临时寄存器取值。同理，若结果值被分配了寄存器则直接存入寄存器中，否则则先存入指定的临时寄存器，再存入内存。

在构建上述分配机制时，我遇到并处理了一些细节问题：

- 在未经mem2reg优化的条件下，存在实际出现在源程序中的变量（表现为有alloca内存和store/load过程）和临时变量（某行计算过程中出现、使用）的区别。为充分识别跨块变量、提升寄存器利用效率，我引入了等价机制。具体内容为：对于alloca的变量%1，一般认为`store i32 %2, i32* %1`中的%2、`%3 = load i32, i32* %1`中的%3与%1等价，识别use/def集合、块内线性扫描时当作%1处理。实现这一机制的基础为：对于非数组变量，一般程序中使用的是变量的值，而不需要关心变量的存储地址（因此，即使存在load/store关系，我没有将getelementptr的地址结果与某个临时变量等价，因为数组地址在传参等场合会被使用，不能被数组元素值覆盖）。
- 对于某些变量，会出现事实上跨块但没有跨块活跃（在多个in集合出现）的现象，如某个变量在b1基本块被定义（def），在后继的b2基本块被使用（use），则只会在b2的in集合内找到该变量。寄存器分配仍然会正常进行，但由于其活跃过程出现了中断（在b1基本块中不再活跃，但尚未进入b2基本块），因此可能会出现寄存器被重复使用、值被覆盖的情况。为此，引入特判机制：若在分析某一基本块时发现某个变量已经在分析其他基本块时分给了寄存器，则说明该变量事实上发生了跨块，则分给一块内存，在需要时将其值存入内存中。
- 在基本块内、基本块跨块等场合，会出现寄存器内某个临时变量覆盖另一个临时变量的现象。有时被覆盖的值是后续程序仍然需要的（如全局变量），有时另一个临时变量的值要从内存中取得，则需要在代码生成、寻找是否被分配了寄存器时，检查寄存器中的值是否确实为需要的变量，并加入对应的存取内存操作以保证值的正确性。同时，在基本块跳转等操作前，也需要将寄存器中的变量值存入内存。

基于寄存器分配与使用，还可以衍生出一些优化议题：

- 两个阶段允许使用的寄存器数量。全局变量图着色赋值过程允许使用的寄存器越多，无变量可取的情况越少，图着色法进入寄存器的变量数量越多。但相应的，被占用的寄存器数量也会增多，局部寄存器分配阶段无寄存器可分配的情况也会增加。因此，需要权衡不同阶段允许使用的寄存器数量。我的编译器中，图着色过程允许使用8个寄存器，在局部的变量寄存器分配阶段再开放所有18个寄存器的分配。
- 函数跳转需要保存的寄存器。函数跳转时，即使这个函数自身用到了某个寄存器，由于目标函数运行过程中不一定会覆盖该寄存器的值，这一寄存器的值也不一定需要保存。通过检查函数自身使用的寄存器、分析不同函数之间相互调用的关系，可以得到不同函数最终会使用到的寄存器，在调用函数时只需要将被调用函数用到的寄存器的值保存进栈即可。对于涉及大量函数调用（如循环调用或递归情形）的程序而言，这种优化可以一定程度降低访存开销。

乘除法优化（运算强度削弱）（完成于生成过程中）

无论是实际CPU还是竞速排序的设定下，乘除法的开销都远大于加减法的开销。然而，部分情况下的乘除法可以转化为其他开销显著降低的形式：

- 乘数为0：由于结果恒为0，可将乘0改为对目标寄存器加法赋值为0；

- 乘除数为1：由于结果恒为自身，可将乘/除以1改为对目标寄存器加法赋值为数据源寄存器的值；
- 乘除数为-1：由于结果恒为相反数，可将乘/除以1改为对目标寄存器减法赋值为0减去数据源寄存器的值；
- 乘除数为其他2的幂次：乘除法可以转化为位运算的形式。

对于乘以2的n次方，位运算转化较为简单，直接将其左移n位即可。

对于除以2的n次方，位运算转化存在不确定性。若被除数为正数，则右移n位不存在误差；若被除数为负数，即使通过算术右移能解决符号问题，在不能整除的情况下也会存在误差。如`-9 / 4 ( = -2)`会被转化为`(ffff fff7) >> 2 = (ffff fffd) = -3`。幸运的是，这一误差是可以在无条件判断的情况下修正的。具体而言，实现不区分正负的`div $1, $2, 2^n`，可以借用两个寄存器`$3, $4`，将其转化为：

```
    srl $3, $2, 31      #   逻辑右移判断正负，0表示正，1表示负
    sll $4, $3, n
    subu $4, $4, $3     #   若$2为负，则$3为1，$4为2^n - 1；反之，$3为0，$4为0
    addu $4, $4, $2     #   若$2为负，被除数加2^n - 1校准误差；若$2为正，被除数不变，
不产生新误差
    sra $1, $4, n
```

若乘数/除数为负数，则拆分为求乘/除值，再通过减0取得相反数。

这一优化的效果较为明显。一方面，数组取址（乘4转化为左移2位）等在正常程序中相当常见，可优化项相当之多；另一方面，其优化程度也相当大（考虑扩展指令转化为基本指令，乘常数成本由5降低为1-2，除常数成本由27降低为5-6）。

此外，还可以进行进一步的转化，如将乘法转为多次位移求和等。不过进一步优化的效果值得商榷。以乘法转化为例，由于乘法的消耗尚且不大，若拆分项较多（例如，43 = 2^5 + 2^3 + 2^1 + 2^0 或 2^6 - 2^4 - 2^2 - 2^0），取得的效果还不如直接相乘。我的编译器没有对其他常数的情况进行进一步优化。

基本块合并与排序输出（完成于优化方法）

与中端优化中的基本块合并相似，这类优化的本质同样是减少跳转开销、增加窥孔优化机会。

中间代码中的`br i1 %x1, label %_b1, label %_b2`的跳转形式，在mips中拆分成了`bne $1, $0, _b1 j _b2`两条指令，其中`j`指令类似于单label的`br`指令，这为进一步合并各个基本块提供了可能。取每个基本块最后一天指令的无条件跳转`j`指向的后继为直接后继，若对于基本块组a/b，a直接后继到b，b只有a一个前继基本块，则a可以与b合并。

除此之外，对于某些情况下可能出现的内容仅为一条无条件跳转的基本块（记为a，a中跳转语句跳转到b），则可以删去a，对a的所有前继基本块，将跳转到a的语句改为跳转到b，由此可以进一步减少不必要跳转。

此外，mips的代码执行逻辑为：若不存在跳转，则直接按代码顺序向下跨基本块运行。若基本块b1、b2相邻，无论b1块末尾有无跳转到b2的指令，接下来都会运行b2基本块的代码。因此，若能合理安排各基本块的顺序，即可进一步减少跳转开销。这里在输出代码的过程中，采用深度优先搜索的思想遍历基本块相互直接调用（代码块末尾的`j`指令指向）形成的图，从而确定输出顺序、输出mips代码。

对于计算开销较小、开销以条件语句及循环等跳转为主的小程序而言，这一优化的收益尚且可观；对于开销主要在计算过程的大型程序而言，跳转优化的收益可能并不是那么明显。此外，相比仅从跳转过程入手的跳转优化，循环展开等专门优化方法的收益显然更高。

杂项窥孔优化（完成于优化方法）

这一阶段通过消除一部分冗余代码（可能来源于源程序，也可能来源于代码生成程序为保证程序正确性而添加的冗余），来尽可能地减少不必要的寄存器赋值、内存读写等操作。

在我的编译器中，目前已经实现的窥孔优化种类为：

- 若有存数语句`sw $1, label($2)`，当`$1`、`$2`值均未发生改变，而出现`lw $1, label($2)`时，后续的lw是多余的。同理也可对`lw`后的多余`sw`进行操作。当调用函数（`jal`）时需要特判，防止窥孔时删除从栈帧中取值恢复现场的`lw`指令。这一优化多应用于基本块合并之后，因为我的编译器会在每个基本块结束之前，将所有占用寄存器而在内存中有空间的变量存入内存以刷新数值，在下一个基本块中再重新取出。
- 若有设置常数的`add $1, $0, num`，当`$1`值未发生改变时，新的`add $1, $0, num`赋值指令是多余的。这一优化多应用于输出阶段，减少对`$v0`（系统调用的操作数）、`$a0`（输出的字符/常数值）的重复赋值。
- 若有移动寄存器值的`add $1, $0, $2`，当`$1`、`$2`值均未发生改变，则新的`add $2, $0, $1`指令是多余的。

窥孔优化对于降低不必要的开销可以取得一定的效果。在实际应用中，窥孔情况考虑越多、不必要代码删除越彻底，窥孔优化的效果越明显。当然，因为考虑欠全而出错的概率也会越高。新增窥孔删代码情形前，需要慎重考虑待删代码是否在所有情形下都不影响程序运行。