

参赛经验与技术分享

于剑 清华大学 20220611

目录

- 参赛经验
- 开发经验分享
- 搭建自动化测试
- 自动并行化

参赛经验

怎样搭出来一个能跑的框架

- 前端
 - 用 parser generator 生成一个 parser，由输入的源码生成语法树
 - 遍历语法树，生成 IR
- 中端
 - 在 IR 上做机器无关优化
- 后端
 - 机器相关优化
 - 寄存器分配
 - 代码生成

去年参赛的时间线

- 确定分工，负责后端的去熟悉 arm 指令集
- 考完期末考试动工
- 约定 IR
- 先跑起来再说：
 - 前端要有比较完整的功能
 - 中端后端优化 pass 都可以不要，寄存器分配写个简单版本
 - 后端完成得比较晚，但是 ccz 先写了个 IR 模拟器跑起来了
- 调通正确性
- 加优化 pass，多观察效果不好的测例上生成的汇编
- 加自动并行化 feature

前端

前端

- 一般是写一个语法规则扔给 parser generator（我们用的是 ANTLR），它生成一个 parser
- 在 parser 给出的语法树上遍历，生成 IR
- 比赛的重点不是前端，但前端是我们正确性问题出现最多的地方
- 隐藏的功能测例：看名字猜内容

SysY 的 const 语义

- `const int a = getint();`
- ```
int f(int x) {
 const int y = x + 1;
 return y;
}
```
- SysY 里的 const 并不能这样用
- 从语法规则能看出来，SysY 里的 const 是编译期常量
- const 常量的值可能用在数组长度里，所以要知道它的值才能知道数组的完整类型，才能生成数组访问的 IR
- 处理字面量时小心 -2147483648



# 为短路运算生成代码

- 编译原理课程里可能涉及过
- 两种方法：
  - 向下传递表达式为真/假时的跳转目标，逐层递归进行
  - 向上传递表达式为真/假时被执行的跳转语句列表，递归回溯时逐层把跳转目标填上
- 短路是会改变语义的（因为表达式可能有副作用）

# 变量初始化

- 全局变量初始化
  - 如果全 0，扔进 bss 段里
  - 如果不全为 0，但都是编译期常量，用 data 段
  - 如果有运行期计算的值，生成一段代码来初始化它，程序开始的时候跳过去，初始化之后跳回去执行 main 函数
- 数组的列表初始化
  - 语法比较玄妙

中端

# IR

- SSA：静态单赋值形式
  - 在上面做优化很方便
- 前端如何生成局部变量的访问
  - 生成为 load/store
  - 由中端的 mem2reg pass 转为对 IR 变量的访问
  - 从头到尾 IR 都是满足 SSA 要求的
- 前年参赛队伍的建议：用 LLVM IR，方便与 llvm 对比，用来测试性能和查 bug

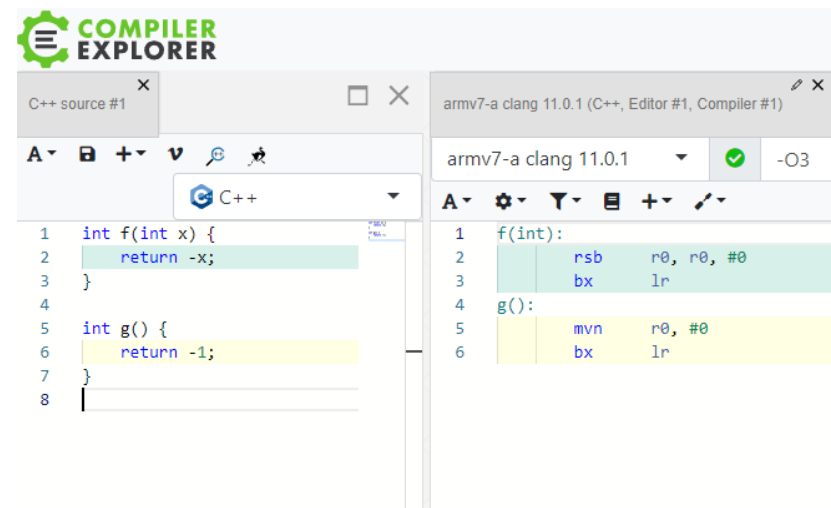
## 中端的优化 pass

- 把优化设计成互相独立的 pass，可用命令行参数开关，方便查错/测试优化效果
- 比较重要的 pass：
  - mem2reg
  - function inline
  - global value numbering
  - global code motion
  - dead code elimination

后端

# 如何熟悉 arm 指令集

- 读文档
  - 在读巨大的文档之前，先了解整体的结构
    - 整数寄存器
    - 调用约定
    - 控制流转移
- 写一点汇编代码来确认自己的理解，可以调用 libc 里的函数
- 多用 godbolt.org
  - 不知道怎么做一个算术运算？
  - 不知道怎么载入一个地址/常量？



The screenshot shows the Compiler Explorer interface. On the left, the C++ source code is displayed in a file named 'C++ source #1':

```
1 int f(int x) {
2 return -x;
3 }
4
5 int g() {
6 return -1;
7 }
8
```

On the right, the assembly output for 'armv7-a clang 11.0.1' is shown, with optimization level '-O3'. The assembly code is:

```
1 f(int):
2 rsb r0, r0, #0
3 bx lr
4 g():
5 mvn r0, #0
6 bx lr
```

# 后端

- 首先遍历 IR，生成机器指令组成的控制流图
  - 一条 IR 语句可能对应不止一条机器指令
  - SSA destruction
  - 此时的每条指令直接对应着一条汇编指令，但其中的变量还是伪寄存器
- 寄存器分配前的优化 pass
- 寄存器分配
- 寄存器分配后的优化 pass



## 寄存器分配前的优化 pass

- 寄存器分配之前有接近 SSA 的性质
  - 一个伪寄存器要么被若干个分布于不同基本块的 move 指令定值，要么由一个基本块里连续的几条指令定值（载入复杂常量时的 movw+movt）
- 方便做指令合并、常量传播等操作

```
void optimize_before_reg_alloc(Program *prog) {
 for (auto &f:prog->funcs) more_constant_info(f.get());
 for (auto &f:prog->funcs) inline_constant(f.get());
 for (auto &f:prog->funcs) merge_shift_binary_op(f.get());
 for (auto &f:prog->funcs) merge_add_ldr_str(f.get());
 for (auto &f:prog->funcs) remove_unused(f.get());
}
```

# 寄存器分配

- 我们实现的是 George, L., & Appel, A.W. (1996). Iterated register coalescing. 中的图着色寄存器分配算法
  - coalesce: 如果一个 move 指令两端的伪寄存器被分配到同一个机器寄存器上, 就可以消除掉这个 move
  - 迭代式消除尽量多的 move
  - 文章里还介绍了怎么处理调用约定
- spill
  - 尽量减小 spill 代价
  - 为了选择被 spill 的伪寄存器, 对 spill 代价进行估价
    - 配合中端对基本块执行频率的估计
    - 一条指令加载的常量的 load 代价为 1, 两条指令加载的常量的 load 代价为 2, 不是常量的伪寄存器要 spill 到栈上, load/store 代价设为 5

```
register allocation for function main
reg_n = 168
using ColoringAllocator
Register allocation:
spill: 2
move instructions eliminated: 10
Callee-save registers used: 9
```

## 寄存器分配后的优化 pass

- 去除能消除掉的 move
- 用 arm 的条件码消除小的分支
- 其他杂七杂八的优化

```
void optimize_after_reg_alloc(Func *func) {
 remove_unused(func);
 remove_identical_move(func);
 remove_no_effect(func);
 direct_jump(func);
 eliminate_branch(func);
}
```

# 自动化测试

# 为什么要做自动化测试

- 好处
  - 及时确认改动的正确性（至少在公开测例上）
  - 方便对比不同版本的优化性能，观察优化效果
  - 与 gcc, llvm 等基线做性能对比
- 不夸张地说，lwpie 搭好测试之后我的开发流程很快就依赖于它了

## feature

- 触发评测
  - 特定分支发生更新时
  - 手动触发 web hook, 指定 commit id 和命令行参数
- 结果展示: 一堆 json, 开个 file server 展示
- 写了个脚本, 指定两个版本, requests 拉下来对应的 json 对比性能测例上的运行时间

# feature

```
$ python3 judge.py cmp 56-6617e50 59-6617e50

56-6617e50 functional test is passed
 flag = --set-num-thread=4
59-6617e50 functional test is passed
 flag = --no-loop-parallel
performance_test2021/01_mm1 : 6.331360 | 6.302190
performance_test2021/01_mm2 : 5.927452 | 5.586143
performance_test2021/01_mm3 : 4.677644 | 4.114599
performance_test2021/02_mv1 : 1.158941 | 4.164607
performance_test2021/02_mv2 : 0.572614 | 1.740453
performance_test2021/02_mv3 : 0.825150 | 2.765581
...
avg: 1.257412
```

```
// 20210819001919
// results/76-775ff68/performance.json

{
 "performance_test2021-public/00_bitset1": 0.429163,
 "performance_test2021-public/00_bitset2": 0.856378,
 "performance_test2021-public/00_bitset3": 1.287312,
 "performance_test2021-public/01_mm1": 6.262323,
 "performance_test2021-public/01_mm2": 5.753694,
 "performance_test2021-public/01_mm3": 4.601857,
 "performance_test2021-public/02_mv1": 1.093833,
 "performance_test2021-public/02_mv2": 0.537267,
 "performance_test2021-public/02_mv3": 0.729663,
 "performance_test2021-public/03_sort1": 0.623242,
 "performance_test2021-public/03_sort2": 10.098014,
 "performance_test2021-public/03_sort3": 1.761576,
 "performance_test2021-public/04_spmv1": 2.993274,
 "performance_test2021-public/04_spmv2": 2.041131,
 "performance_test2021-public/04_spmv3": 1.287658,
 "performance_test2021-public/conv0": 4.009294,
 "performance_test2021-public/conv1": 8.463102,
 "performance_test2021-public/conv2": 7.476332,
 "performance_test2021-public/fft0": 3.599862,
 "performance_test2021-public/fft1": 7.674059,
 "performance_test2021-public/fft2": 7.431724,
 ...
}
```

# 如何搭建

- 工作流程
  - 触发评测时拉取对应版本的仓库，编译出编译器，并把它在测例上跑一遍，生成汇编代码
  - 树莓派链接生成的汇编代码，在测例输入上跑
  - 一个 web server 展示测试结果
- 考虑到树莓派性能，不建议把第一步和第三步放在树莓派上
- 跨机器通信时注意安全问题
  - 尤其是允许指定命令行参数时，容易导致远程代码执行



# 自动并行化

## 并行效果

- 把循环静态地等距分割成 num\_threads 段，每段一个线程
- 特殊处理归纳变量和累加变量
- 效果等价于

```
#pragma omp parallel for reduction(+:s) schedule(static)
for (i=L; i<R; ++i) s+=F(i);
```

- 实际上不允许循环里有函数调用（由于比较激进的 inline 策略，并没有造成多少影响）
- 这允许了一个比较 tricky 的实现：各个线程共用栈
- 过于 tricky，不建议模仿

# 线程创建与回收

- 使用的 linux syscall: clone, waited, exit
- 用汇编实现了 \_\_create\_threads 和 \_\_join\_threads 两个函数，它们的语义：

```
int __create_threads(int n) {
 --n;
 if (n <= 0) {
 return 0;
 }
 for (int i = 0; i < n; ++i) {
 int pid = clone(CLONE_VM | SIGCHLD,
 sp, 0, 0, 0);

 if (pid != 0) {
 return i;
 }
 }
 return n;
}
```

```
void __join_threads(int i, int n) {
 --n;
 if (i != n) {
 waitid(P_ALL, 0, NULL, WEXITED);
 }
 if (i != 0) {
 _exit(0);
 }
}
```

# 线程创建与回收

- 为什么要用汇编实现？
- 由于各个线程共用栈，直接用 C 写的话，进入和退出函数时会如常保存和恢复寄存器，栈上的 load/store 会有 data race。
- 因为同样的原因，不能调用 glibc 里对 syscall 的包装函数，而是做 inline syscall。
- 需要手动安排比较奇怪的栈布局
- 见 <https://github.com/kobayashi-compiler/kobayashi-compiler/blob/main/runtime/armv7/thread.S>

## 我们的仓库

- <https://github.com/kobayashi-compiler/kobayashi-compiler>
- 自动化测试: <https://github.com/kobayashi-compiler/kobayashi-test>
- 出于教学目的, 比赛后我给它加了个 RISC-V32 后端, 但是自动并行化由于缺少测试环境 (在 qemu-riscv32 的用户态模拟上得到了奇怪的结果) 还不支持, 欢迎来帮助我们。