

# TrivialCompiler 技术报告

MashPlant 李晨昊

2020 年 12 月 22 日



## ① 简介

## ② 前端

## ③ IR 设计

## ④ IR 上的优化

## ⑤ 机器码生成及优化

## ① 简介

## ② 前端

## ③ IR 设计

## ④ IR 上的优化

## ⑤ 机器码生成及优化

# 简介

- 2020 年全国大学生系统能力大赛编译系统设计赛参赛作品，排名第二，获得一等奖
- 我与陈晟祺，陈嘉杰组队完成
- 将类似 C 语言的源语言 SysY 语言编译到 ARMv7-A，在树莓派上运行

[提交](#) [最新提交](#) [排名](#) [我的提交](#)[编译系统设计赛-决赛 / 决赛](#)

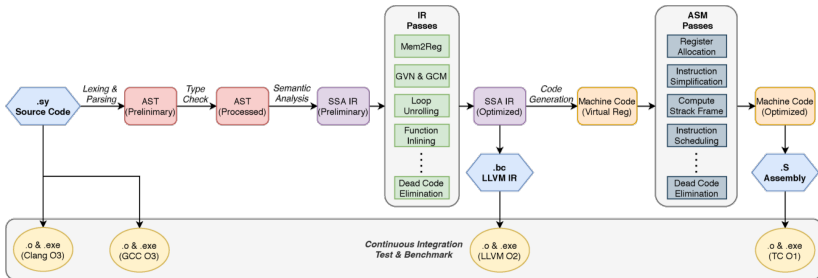
## 排行榜

[我的排行榜明细 >](#)

#	账号	Team	提交次数	最后提交时间	功能得分	性能得分	总分
1	chen16614	燃烧我的编译器 / 中国科学技术大学	44	2020-08-20 15:56:46	100	93.26	94.75
2	HarryChen	编程是一件很危险的事情 / 清华大学	7	2020-08-20 11:14:30	100	69.13	75.99

# 架构

- C++17, 基于 CMake 构建, 代码量约 6000 行
- 从源文件经过 Lexing Parsing 构建 AST, 类型检查, 语义分析, 然后转换成 SSA 形式的 IR, 在 IR 上进行一系列的优化, 再转化为机器码
- 我主要负责前端和 IR 上的优化, 另外两人主要负责后端机器码生成



## ① 简介

## ② 前端

## ③ IR 设计

## ④ IR 上的优化

## ⑤ 机器码生成及优化

# 前端

- Parser 采用我编写的 Parser Generator lalr1<sup>1</sup> 自动生成
- lalr1 内部调用 re2dfa<sup>2</sup> 生成 lexer
- lalr1 采用龙书<sup>3</sup>中算法 4.63 描述的基于 LR(0) FSM 生成 LALR(1) 解析表的高效算法，在代码生成速度和生成的代码的执行速度上都超过了 Yacc/Bison<sup>4</sup>

---

<sup>1</sup><https://github.com/MashPlant/lalr1>

<sup>2</sup><https://github.com/MashPlant/re2dfa>

<sup>3</sup>Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA.

<sup>4</sup>基于 C99 文法 (<http://www.quut.com/c/ANSI-C-grammar-y-1999.html>) 测试，测试输入选自 <https://github.com/c-testsuite/c-testsuite>

# 前端

- 今年的 Rust 版编译实验也采用 lalr1，多数其他同学使用基于 LL(\*) 文法的 ANTLR
- ANTLR 提供了更丰富的功能以简化文法的编写，但本质上都是编写产生式和语法动作
- ANTLR 生成的代码需要配合 Runtime 使用，LL(\*) 文法的解析速度也较低；lalr1/Yacc/Bison 都生成无依赖的代码



# 编程技巧<sup>1</sup>

- AST 节点和 IR 节点都涉及大量多态。C++ 原生的面向对象机制性能开销较大，在对象中保存一个整数值编码以判断实际类型，逻辑上与 `dynamic_cast` 一样
- 访问节点的时候依据类型分别处理。这就是 OOP 引入 Visitor 模式时的反面写法，但我认为这样逻辑清晰，易于组合。这种写法与 Visitor 模式也不冲突，各有适用之处

```
if (auto x = dyn_cast<Assign>(s)) {  
    // ...  
} else if (auto x = dyn_cast<ExprStmt>(s)) {  
    // ...  
} else if (auto x = dyn_cast<DeclStmt>(s)) {  
    // ...  
} else if (auto x = dyn_cast<Block>(s)) {  
    // ...  
} else if (auto x = dyn_cast<If>(s)) {  
    // ...  
} else if (auto x = dyn_cast<While>(s)) {  
    // ...  
} else if (isa<Break>(s)) {  
    // ...  
} else if (isa<Continue>(s)) {  
    // ...  
} else if (auto x = dyn_cast<Return>(s)) {  
    // ...  
}
```

<sup>1</sup><https://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html>

① 简介

② 前端

③ IR 设计

④ IR 上的优化

⑤ 机器码生成及优化

# 各种 IR

- IR(Intermediate Representation): 介于 AST 和机器码之间的一层, 便于进行优化, 分析, 解释执行等工作
- IR 有很多形式: 基于栈的 IR(Stack IR), 基于虚拟寄存器的 IR(Reg IR), SSA(Static Single Assignment) IR
  - Stack IR: 运算指令没有运算数, 而是取出栈顶元素进行运算, 将结果放回栈顶
  - Reg IR: 类似 (RISC) 汇编, 有明确的操作数寄存器和目标寄存器, 寄存器没有数量限制, 一般一个变量对应一个寄存器
  - SSA IR: 类似 Reg IR, 但一个寄存器只能有一个赋值点, 借助  $\phi$  函数实现多次对一个变量赋值的语义

# 各种 IR

```
int s = 0;
for (int i = 0; i < 100; i = i + 1) s = s + i;
```

## Reg IR

```
s = 0
i = 0
L0:
  cond = i < 100
  br cond, L1, L2
L1:
  s = s + i
  i = i + 1
  br L0
L2:
```

## SSA IR

```
L0:
  br L1
L1:
  s0 =  $\phi$  [0, L0], [s1, L2]
  i0 =  $\phi$  [0, L0], [i1, L2]
  cond = i0 < 100
  br cond, L2, L3
L2:
  s1 = s0 + i0
  i1 = i0 + 1
  br L1
L3:
```

## Stack IR

```
const 0
store s
const 0
store i
L0:
  load i
  const 100
  binary_lt
  br L1, L2
L1:
  load s
  load i
  binary_add
  store s
  load i
  const 1
  binary_add
  store i
  br L0
L2:
```

# 各种 IR

	Stack IR	Reg IR	SSA IR
优化	困难	中等	容易
生成	容易	中等	困难
传输	容易 <sup>1</sup>	中等	中等
阅读	困难	中等	困难

- 因为各自的特点，不同的 IR 有不同的使用场景
  - Stack IR 多用于网络传输，如 JVM Bytecode，WebAssembly
  - Reg IR 多用于高层分析验证，如 Rust MIR，和教学级编译器
  - SSA IR 多用于工业级编译器，如 GCC，LLVM
- TrivialCompiler 使用 SSA IR

<sup>1</sup>虽然 Stack IR 的指令条数多，但指令中不用编码操作数，每条指令所占空间少

## SSA IR

- $\phi$  函数必须位于基本块的开头
- $\phi[v_1, L_1] \dots [v_n, L_n]$  中,  $L_1, \dots, L_n$  必须恰好是本基本块的所有前驱基本块 (因而实现时无需在  $\phi$  函数中保存基本块)
- 若运行时从  $L_i$  基本块进入本基本块, 则  $\phi$  函数的值是  $v_i$
- **一个基本块开头所有  $\phi$  函数同时计算**
  - 例如某基本块开头  $a = \phi[b, L_1]$ ;  $b = \phi[a, L_1]$ , 语义是交换  $a$  和  $b$  的值, 而不是都赋成  $b$

```
L0:
  br L1
L1:
  s0 =  $\phi$  [0, L0], [s1, L2]
  i0 =  $\phi$  [0, L0], [i1, L2]
  cond = i0 < 100
  br cond, L2, L3
L2:
  s1 = s0 + i0
  i1 = i0 + 1
  br L1
L3:
```

## SSA IR

- 理论上可以用和 Reg IR 一样的数据结构表示，只是加上  $\phi$  函数，但这样无法利用一个寄存器只有一个赋值点的性质。怎样快速知道一个寄存器在哪里定义，被谁使用了
  - 可以用整数表示寄存器，再维护一个数组记录每个寄存器的定义位置和使用位置
  - LLVM 使用更高效且更复杂的表示方法，感兴趣的同学可以参考<https://www.zhihu.com/question/41999500>
  - TrivialCompiler 中使用类似 LLVM 早期版本的表示方法

- 1 简介
- 2 前端
- 3 IR 设计
- 4 IR 上的优化**
- 5 机器码生成及优化



# Mem2Reg

- 存在从 AST 直接生成 SSA IR 的方法，但我没有研究过，常规方法是生成“伪”SSA，再用 Mem2Reg 转化成真正的 SSA
- SSA 要求一个寄存器只能有一个赋值点，但可以向一片内存写入任意多次，“伪”SSA 中把变量都对应到栈上的内存
- Mem2Reg 将这样的内存访问转化为寄存器访问，它是之后一切基于 SSA IR 的优化的基础

```
L0:
  s = alloca int
  i = alloca int
  br L1
L1:
  tmp0 = *i
  cond = tmp0 < 100
  br cond, L2, L3
L2:
  tmp1 = *s
  tmp2 = *i
  *s = tmp1 + tmp2
  tmp3 = *i
  *i = tmp3 + 1
  br L1
L3:
```

⇒

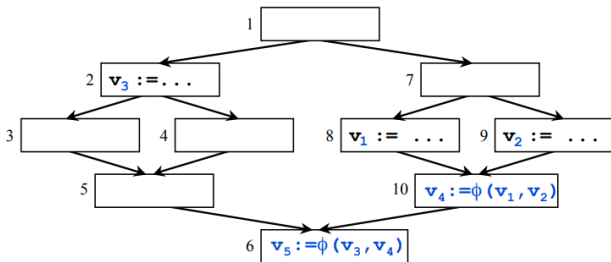
```
L0:
  br L1
L1:
  s0 =  $\phi$  [0, L0], [s1, L2]
  i0 =  $\phi$  [0, L0], [i1, L2]
  cond = i0 < 100
  br cond, L2, L3
L2:
  s1 = s0 + i0
  i1 = i0 + 1
  br L1
L3:
```

# Mem2Reg

- 定义基本块的支配边界  $DF$  (Dominance Frontier):  
 $DF(n) = \{x | n \text{ 支配 } x \text{ 的某个前驱, 且 } (n = x \text{ 或 } n \text{ 不支配 } x)\}$
- 集合的支配边界:  $DF(S) = \bigcup_{n \in S} DF(n)$
- 集合的迭代支配边界:  
 $DF_1(S) = DF(S); DF_{i+1}(S) = DF(S \cup DF_i(S))$ , 迭代至收敛得到  $DF^+(S)$
- 支配边界的高效计算用到后面提到的支配树, 这里略过

# Mem2Reg

- 一个支配边界的例子<sup>1</sup>:



DF(8) = {10}

DF(9) = {10}

DF(2) = {6}

DF({8,9}) = {10}

DF(10) = {6}

DF({2,8,9,10}) = {6,10}

$DF(d) = \{n \mid \exists p \in \text{pred}(n), d \text{ dom } p \text{ and } d \nmid \text{dom } n\}$

<sup>1</sup><https://www.cs.colostate.edu/~mstrout/CS553Fall06/slides/lecture17-SSA.pdf>

# Mem2Reg

- 第一步：收集对一个变量的所有写入的基本块集合  $S$ ，为  $DF^+(S)$  中的每个基本块插入  $\phi$  函数
- 直观理解：在支配边界上这个变量从不同前驱来值可能不同，所以需要插入  $\phi$  函数。因为  $\phi$  函数本身也是一次定值，需要把插入的地方也考虑进去再计算  $DF$ ，这就是  $DF^+$  的意义
- 实现时只计算每个基本块的  $DF$ ，不计算  $DF^+$ ，迭代插入  $\phi$  函数即可

# Mem2Reg

- 第二步：确定  $\phi$  的参数和 load 的结果。从起始节点开始 DFS：
  - 跳过基本块开头的  $\phi$  函数，对之后的每条语句：
    - 对变量的 store：删除它，记录变量最新的值是被 store 的值
    - 对变量的 load：删除它，对它的使用替换成变量最新的值
  - 访问完语句后，依次访问基本块的所有后继。对一个后继先访问它开头的  $\phi$  函数，将属于这个变量的  $\phi$  对应位置替换成变量最新的值，然后记录变量最新的值是这个  $\phi$ ，然后访问这个基本块
  - 全部后继访问完后，恢复进入这个基本块时变量的值（可以用栈实现）

GVN & GCM <sup>1</sup>

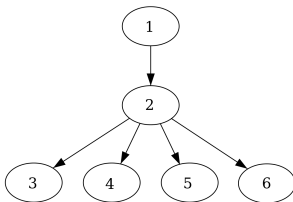
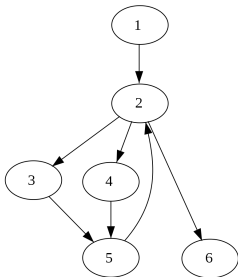
- GVN & GCM 可以消除冗余计算，提取循环不变式，减轻寄存器压力，代数简化，常量折叠...
- GVN: 按照某个顺序访问所有表达式，如果  $e_2$  和之前遇到的  $e_1$  运算类型相同且参数的 value numbering 相同，那么  $e_2$  和的  $e_1$  的 value number 相同，可以用  $e_1$  替换  $e_2$ 
  - 内存访问也可以处理，将内存操作依赖的内存操作视作参数，LLVM 中有 MemorySSA 模块，TrivialCompiler 也初步实现了
  - 在这一步顺便进行代数简化，常量折叠
- 这样生成的程序可能是不合法的，一个表达式不一定在使用它前被计算，通过 GCM 使之合法

---

<sup>1</sup>Cliff Click. 1995. Global code motion/global value numbering. SIGPLAN Not. 30, 6 (June 1995), 246–257.

# GVN & GCM

- 定义基本块直接支配关系  $x \text{ idom } y$ :  $x$  支配  $y$ , 且不支配任何支配  $y$  的其他节点
- 除开始节点外每个节点恰被一个节点直接支配, 这构成一棵树, 称作支配树
- 节点在支配树中的深度称作支配深度。直观理解支配深度更大, 就被更多层 if 语句包裹, 更有可能不执行



# GVN & GCM

- GCM: 先移动到到尽量早的位置, 再移动到到尽量晚的位置
- `schedule_early`: 依次用 `schedule_early` 调度所有参数, 如果一个参数所在基本块支配深度小于自身, 就将自身移动到参数的基本块。这样可以保证自身被参数支配
- `schedule_late`: 先用 `schedule_late` 调度所有使用者, 从支配树上所有使用者的 LCA 到原始位置的一条链上都是可行的位置, 选择循环嵌套层次最浅, 支配深度尽量深的
- GCM 不关心基本块内指令的顺序, 有专门的局部调度算法



- 1 简介
- 2 前端
- 3 IR 设计
- 4 IR 上的优化
- 5 机器码生成及优化

# 指令选择

- 通常编译器后端先进行指令选择，再进行优化和寄存器分配
- 指令选择将 IR 指令翻译到带有虚拟寄存器和机器寄存器的机器指令，机器寄存器用于满足调用约定等需求
- SSA IR 中运算，访存，函数调用等指令的翻译与 Reg IR 类似，但指令集中一般没有  $\phi$  函数，需要翻译成赋值语句
  - 对于  $x = \phi[v_1, L_1] \dots [v_n, L_n]$ ，在  $L_i$  末尾生成  $x = v_i$
  - 如果有多个  $\phi$  函数  $\phi_1, \dots, \phi_m$ ，赋值  $x_1 = v_{i1}; \dots; x_m = v_{im}$  必须是并行执行的，一种实现方式是先把  $v_{i1}, \dots, v_{im}$  赋给  $t_1, \dots, t_m$ ，再把  $t_1, \dots, t_m$  赋给  $x_1, \dots, x_m$

# 窥孔优化

- TrivialCompiler 中利用 ARMv7-A 指令集进行了很多窥孔优化，这里介绍除常数优化
- 很多处理器上除法都是非常耗时的操作，除法的延迟可达加法的几十倍，而乘法的延迟和加法在一个数量级。除常数是一种常见的操作，希望能够避免用除法计算它
- `int i, i / 2` 可以优化成 `i >> 1` 吗?

# 窥孔优化

- TrivialCompiler 中利用 ARMv7-A 指令集进行了很多窥孔优化，这里介绍除常数优化
- 很多处理器上除法都是非常耗时的操作，除法的延迟可达加法的几十倍，而乘法的延迟和加法在一个数量级。除常数是一种常见的操作，希望能够避免用除法计算它
- `int i, i / 2` 可以优化成 `i >> 1` 吗?
- 不能! 整数除法向 0 取整, `-1 / 2 == 0`, `-1 sra 1 == -1`
- `i / 2021` 又该怎么优化呢?

# 窥孔优化

- TrivialCompiler 采用论文中<sup>1</sup>描述的除常数优化算法
- $i / 2 == (i + (i < 0)) \gg 1$
- $i / 2021 == ((i + ((\text{int64}(-2118793861) * i) \gg 32)) \gg 10) + (i < 0)$

---

<sup>1</sup>Torbjörn Granlund and Peter L. Montgomery. 1994. Division by invariant integers using multiplication. SIGPLAN Not. 29, 6 (June 1994), 61–72.

DOI:<https://doi.org/10.1145/773473.178249>

# 窥孔优化

- TrivialCompiler 采用论文中<sup>1</sup>描述的除常数优化算法
- $i / 2 == (i + (i < 0)) \gg 1$
- $i / 2021 == ((i + ((\text{int64}(-2118793861) * i) \gg 32)) \gg 10) + (i < 0)$
- 很多人习惯使用带符号整数，但一些场景下这限制了编译器的优化，无符号整数除常数生成的代码更精简

---

<sup>1</sup>Torbjörn Granlund and Peter L. Montgomery. 1994. Division by invariant integers using multiplication. SIGPLAN Not. 29, 6 (June 1994), 61–72.

DOI:<https://doi.org/10.1145/773473.178249>

# 寄存器分配

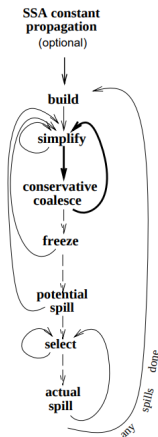
- TrivialCompiler 采用论文<sup>1</sup>中描述的图着色寄存器分配算法
- 指令选择生成的代码中存在大量赋值语句，通过寄存器分配解决这个问题：如果赋值左右操作数被分配到同一个寄存器，这次赋值就可以去掉
- 论文中有更详细的描述和伪代码，这里只大致介绍重要思想

---

<sup>1</sup>Lal George and Andrew W. Appel. 1996. Iterated register coalescing. ACM Trans. Program. Lang. Syst. 18, 3 (May 1996), 300–324.  
DOI:<https://doi.org/10.1145/229542.229546>

# 寄存器分配

- 构建干涉图：与课中描述的类似，但特殊处理赋值：即使右操作数在赋值语句后存活，也不用在左右操作数间连边
- 对机器寄存器预着色，预着色的节点间互不相同色，着色过程中也不再分配颜色
- 着色过程：选择度数小于机器寄存器数目的节点删除之，若不存在则按照启发式算法选择一个。同时尝试进行节点合并，freeze 等操作。重复直至删除完，逆序添加回去，为每个节点选择邻居都没有的颜色，若没有可用的颜色证明它需要 spill，分配完成后依据需要 spill 的节点重写程序，再分配一次





*Thanks!*