SysY Trivial Compiler 项目分享

“华为毕昇杯” 编译系统设计赛

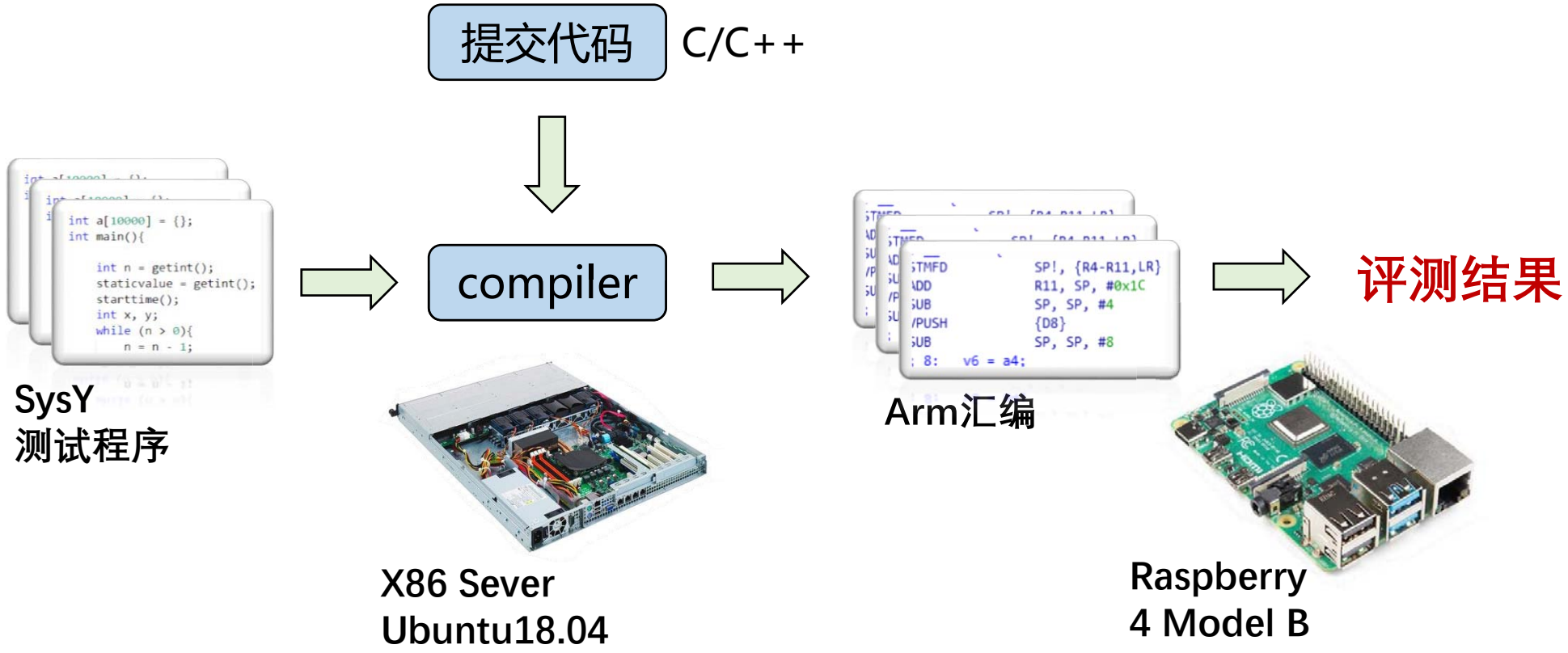
燃烧我的编译器

中国科学技术大学
陈清源、黄奕桐、章耀辉、曾明亮

目录

目录

比赛规则概览


提交代码
C/C++
compiler
int a[10000] = {};
int main(){
int n = getint();
staticvalue = getint();
starttime();
int x, y;
while (n > 0){
n = n - 1;
SysY
测试程序
X86 Sever
Ubuntu18.04
Arm汇编
评测结果
Raspberry
4 Model B

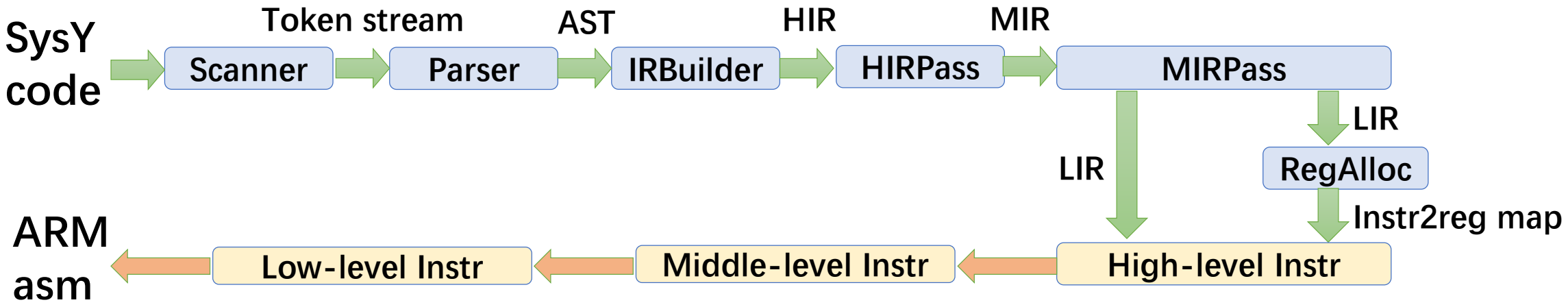
目录

- **SysY Trivial Compiler 项目分享**
  - 比赛规则概览
  - 编译器的架构设计

SySY Compiler 架构设计

三个层次的 IR

- HIR　保留代码 if 与 while 等结构，方便结构级变换
- MIR　简洁优雅的语义，承载大量通用优化 Pass
- LIR　贴近硬件架构，便于指令融合、调度与选择



三个层次的虚拟指令

- HInstr　智能翻译成多条 MInstr，不可使用临时寄存器
- MInstr　智能翻译成多条 LInstr，可用临时寄存器 R12 R14
- LInstr　智能翻译成多条 ARM Instr，可用临时寄存器 R11

目录

- **SysY Trivial Compiler 项目分享**
  - 比赛规则概览
  - 编译器的架构设计
  - 永无止境地追求性能
    - 中间代码的极致化简

中间代码的极致化简

HIR Pass：

循环合并，
分支合并，
结构级变换

LIR Pass：

乘加指令，
左移子指令，
等指令融合

HIR

- LoopMerge
- AccumulatePattern
- HighIRsimplyCFG
- MergeCond
- …

LIR

- InstructionSchedule
- InstructionFuse
  - SpliGEP
  - SplictRem
  - removeUnuseOP
  - convertMulDivToShift
  - …

MIR

Analysis

- IRCheck
- LoopFinder
- Dominator
- ActiveVar
- ReachDefinition

Transform

- Mem2reg
- BBConstPropagation
- DeadCodeEliminate
- FunctionInline
- SimplifyCFG
- LoopInvariant
- Mutithreading
- Vectorizaiton
- PowerArray
- RefactorPartins
- CondSimplify
- …

MIR Pass：

常量传播，
函数内联，
循环不变式外提等
大量通用优化pass

中间代码的极致化简


```
; Module name: 'SysY code'
[ 4 x i32]* @global  = global [ 4 x i32] [ i32 0 i32 0 i32 0 i32 1  ]
define i32@ififElse()
<label>entry        ;preds:
    i32* %0  = Alloca
    Store i32 5 i32* %0
    i32* %1  = Alloca
    Store i32 10 i32* %1
    Br <label> %3
<label>3        ;preds: %entry
    i32 %4  = Load i32* %0
    i1 %5  = CmpEQ i32 %4 i32 5
    Br i1 %5 <label> %7 <label> %14
<label>7        ;preds: %3
    i32 %8  = Load i32* %1
    i1 %9  = CmpEQ i32 %8 i32 10
    Br i1 %9 <label> %10 <label> %11
<label>10        ;preds: %7
    Store i32 2 i32* %0
    Br <label> %14
<label>11        ;preds: %7
    i32 %12  = Load i32* %0
    i32 %13  = Sub i32 %12 i32 2
    Store i32 %13 i32* %0
    Br <label> %14
<label>14        ;preds: %11    %10    %3
    i32 %15  = Load i32* %0
    ret i32 %15
define i32@main()
<label>entry        ;preds:
    i32 %0  = Call ififElse
    i32* %1  = GEP [ 4 x i32]* @global i32 %0
    i32 %2  = Load i32* %1
    ret i32 %2
```

- FunctionInline
- Mem2Reg
- GlobalLocal

测例：24_if_test3.sy

```
; Module name: 'SysY code'
define i32@main()
<label>entry        ;preds:
    [ 4 x i32]* %19  = Alloca Init 0
    i32* %20  = GEP [ 4 x i32]* %19 i32 3
    Store i32 1 i32* %20
    i1 %7  = CmpEQ i32 5 i32 5
    Br i1 %7 <label> %8 <label> %15
<label>8        ;preds: %entry
    i1 %10  = CmpEQ i32 10 i32 10
    Br i1 %10 <label> %11 <label> %12
<label>11        ;preds: %8
    Br <label> %15
<label>12        ;preds: %8
    i32 %14  = Sub i32 5 i32 2
    Br <label> %15
<label>15        ;preds: %11    %12    %entry
    i32 %21  = PHI i32 5 <label> %entry i32 2 <label> %11 i32 %14 <label> %12
    i32* %1  = GEP [ 4 x i32]* %19 i32 %21
    i32 %2  = Load i32* %1
    ret i32 %2
```

中间代码的极致化简

ConstPropagation

```
; Module name: 'SysY code'
define i32@main()
<label>entry          ;preds:
    [ 4 x i32]* %19  = Alloca Init 0
    i32* %20  = GEP [ 4 x i32]* %19 i32 3
    Store i32 1 i32* %20
    Br <label> %8
<label>8          ;preds: %entry
    Br <label> %11
<label>11         ;preds: %8
    Br <label> %15
<label>12         ;preds:
    Br <label> %15
<label>15         ;preds: %11    %12
    i32 %21  = PHI i32 2 <label> %11 i32 3 <label> %12
    i32* %1  = GEP [ 4 x i32]* %19 i32 %21
    i32 %2  = Load i32* %1
    ret i32 %2
```

SimplyCFG

```
; Module name: 'SysY code'
define i32@main()
<label>entry          ;preds:
    [ 4 x i32]* %19  = Alloca Init 0
    i32* %20  = GEP [ 4 x i32]* %19 i32 3
    Store i32 1 i32* %20
    i32* %1  = GEP [ 4 x i32]* %19 i32 2
    i32 %2  = Load i32* %1
    ret i32 %2
```

LIR Passes

```
; Module name: 'SysY code'
define i32@main()
<label>entry          ;preds:
    [ 4 x i32]* %19  = Alloca Init 0
    Store i32 1 [ 4 x i32]* %19 i32 12
    i32 %2  = Load [ 4 x i32]* %19 i32 8
    ret i32 %2
```

目录

- **SysY Trivial Compiler 项目分享**
  - 比赛规则概览
  - 编译器的架构设计
  - 永无止境地追求性能
    - 中间代码的极致化简
    - 硬件资源的充分利用

充分利用硬件的资源

- **指令级并行：指令的软流水**
  - **InstructionSchedule** IR Pass　　测例表现：由于代码规模，调度效果不明显
  - 依据指令使用资源，调度减少流水线执行的 stall

充分利用硬件的资源

- **指令级并行：指令的软流水**
- **数据级并行：指令的向量化**
  - **Vectorization** IR Pass
  - 手工构造规则，以避免循环间数据依赖，来实现向量化的过程

测例表现：mm，mv 有 2倍的加速比提升

```
for (int k = 0; k < n; k++) {
    x = x + B[k] * C[k] ;
}
```

→

```
for (int k = 0; k < n; k=k+8) {
    x = x + B[k] * C[k] ;
                    ⋮
            + B[k+7] * C[k+7] ;

    //  %x = VV 8 B, C
}
```

充分利用硬件的资源

- 指令级并行：指令的软流水
- 数据级并行：指令的向量化
- 线程级并行：
  - 如何并行？

```
for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                A[i][j] = A[i][j] + B[i][k] * C[k][j];
            }
        }
    }
```

线程级并行

- **前端支持：**
  - 循环间依赖分析，写冲突分析
  - Z3分析循环模式 → 解释器模拟执行 (规则限制下的 Trivial 方法)

- **后端支持：**
  - 轻量级多线程框架

轻量级多线程框架

- **经典的多线程实现：每个线程有独立的栈空间**
- **我们的多线程实现：每个线程共享部分栈空间**
  - 更高的性能
  - 更方便变换
  - 更易于实现

轻量级多线程框架

更高的性能

```
int my_func(.....) {
    ..... some context on stack .....
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                A[i][j] = A[i][j] + B[i][k] * C[k][j];
            }
        }
    }
    ......
}
```

Call pthread_create()

```
int matrix_mul(int t, ......(context).....) {
    for (int i = starti(t); i < endi(t); i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                A[i][j] = A[i][j] + B[i][k] * C[k][j];
            }
        }
    }
}
```

经典的多线程实现（包装成函数）

```
int t = MTSTART();
for (int i = starti(t) ; i < endi(t) ; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            A[i][j] = A[i][j] + B[i][k] * C[k][j];
        }
    }
}
MTEND(t);
```

独创的多线程实现（就地完成）

轻量级多线程框架

更方便变换

```
int F(……) {

    ……

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                A[i][j] = A[i][j] + B[i][k] * C[k][j];
            }
        }
    }

    ……

}
```

```
int F(……) {

    ……

    int thread_id = __mtstart();
    int L = (thread_id + 0) * n / num_threads;
    int R = (thread_id + 1) * n / num_threads;
    for (int i = L; i < R; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                A[i][j] = A[i][j] + B[i][k] * C[k][j];
            }
        }
    }
    __mtend(thread_id);

    ……

}
```

注：实际变换是在 IR 层面上完成的

轻量级多线程框架

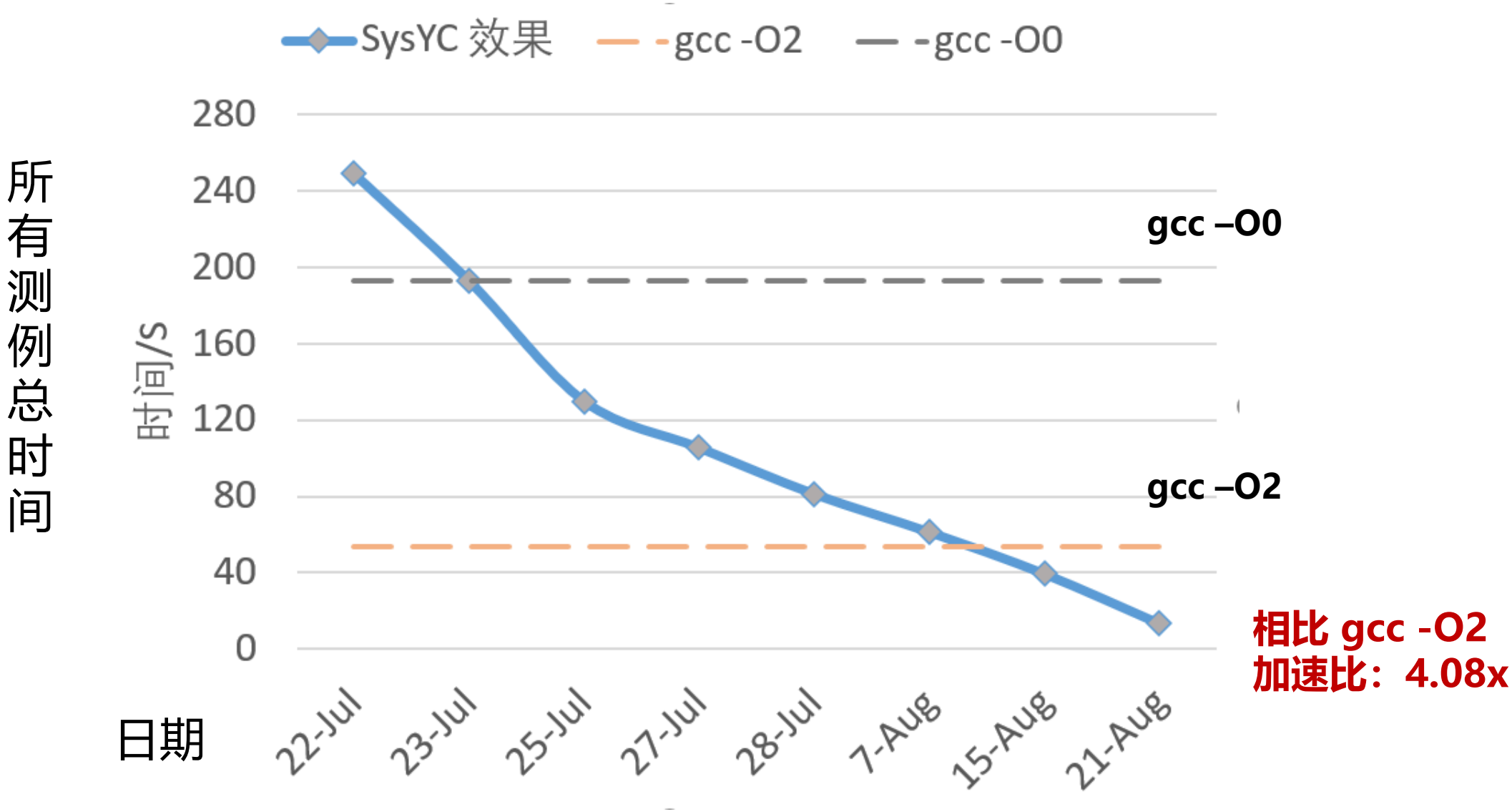
- **更易于实现**
  - Size of libpthread.so: **130KB**
  - Size of our thread library: **49 lines of asm code**

System Call: Clone()

```
__mtstart:
    movw r11, #0
    movt r11, #256
    sub sp, sp, r11
    push {r0, r1, r2, r3}
    mov r3, r7
    mov r2, #4
.__mtstart_1:
    sub r2, r2, #1
    cmp r2, #0
    beq .__mtstart_2+0
    mov r7, #120
    mov r0, #273
    mov r1, sp
    swi #0
    cmp r0, #0
    bne .__mtstart_1+0
.__mtstart_2:
    mov r10, r2
    mov r7, r3
    pop {r0, r1, r2, r3}
    movw r11, #0
    movt r11, #256
    add sp, sp, r11
    mov pc, lr
```

```
__mtend:
    cmp r10, #0
    beq .__mtend_2+0
.__mtend_1:
    mov r7, #1
    swi #0
.__mtend_2:
    push {r0, r1, r2, r3}
    mov r1, #4
.__mtend_3:
    sub r1, r1, #1
    cmp r1, #0
    beq .__mtend_4+0
    push {r1, lr}
    sub sp, sp, #4
    mov r0, sp
    bl wait
    add sp, sp, #4
    pop {r1, lr}
    b .__mtend_3+0
.__mtend_4:
    pop {r0, r1, r2, r3}
    mov r10, #0
    mov pc, lr
```

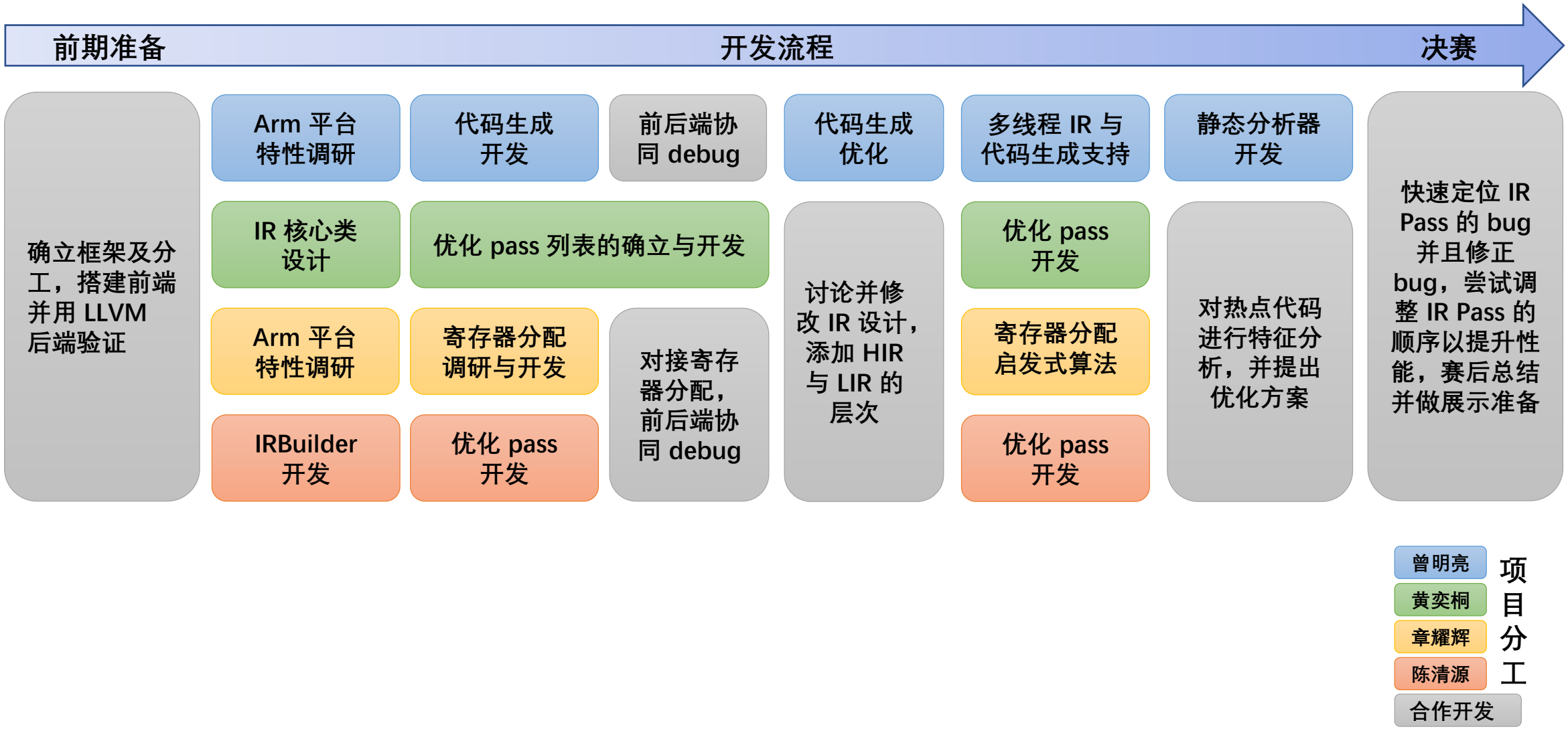
优化历程



相比 gcc -O2
加速比：4.08x

目录

科学的分工与合作


前期准备
开发流程
决赛
确立框架及分工，搭建前端并用 LLVM 后端验证
Arm 平台特性调研
代码生成开发
前后端协同 debug
代码生成优化
多线程 IR 与代码生成支持
静态分析器开发
快速定位 IR Pass 的 bug 并且修正 bug，尝试调整 IR Pass 的顺序以提升性能，赛后总结并做展示准备
IR 核心类设计
优化 pass 列表的确立与开发
讨论并修改 IR 设计，添加 HIR 与 LIR 的层次
优化 pass 开发
对热点代码进行特征分析，并提出优化方案
Arm 平台特性调研
寄存器分配调研与开发
对接寄存器分配，前后端协同 debug
寄存器分配启发式算法
IRBuilder 开发
优化 pass 开发
优化 pass 开发
曾明亮
黄奕桐
章耀辉
陈清源
合作开发
项目分工

目录

编译器开发中的“双份的快乐”

- “写编译器既要调试代码，又要调试代码产生的代码——双份的快乐.jpg”

- “每次全部测例AC后，加一个pass又挂掉一大半测例”，“就像希腊神话里的西西里弗斯将巨石推到山顶后巨石又滚回山下”

目录

对比赛的建议与反馈

赛后调查问卷结果

编译课程实验情况统计

选项	小计	比例
没有实验	0	0%
需要完成词法分析器	32	91.43%
需要完成语法分析器	30	85.71%
需要生成 AST	18	51.43%
需要生成 IR	15	42.86%
需要将 IR 变为 SSA 形式	0	0%
需要做简单中间代码优化（如死代码删除）	10	28.57%
需要做复杂中间代码优化（如过程间优化）	1	2.86%
需要生成汇编代码	13	37.14%
本题有效填写人次	35	

编译比赛期间自学情况统计

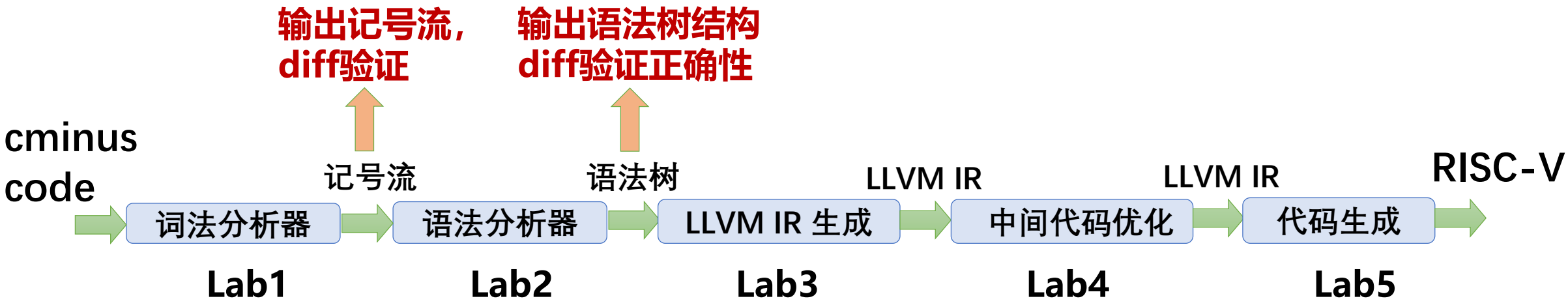
选项	小计	比例
比赛前就都会了	1	1.85%
词法分析器	22	40.74%
语法分析器	22	40.74%
AST 生成	19	35.19%
IR 设计	27	50%
IR 生成	25	46.3%
将 IR 转化为 SSA 形式	19	35.19%
过程内优化 IR Pass	24	44.44%
过程间优化 IR Pass	16	29.63%
寄存器分配	36	66.67%
指令选择	25	46.3%
指令调度	22	40.74%
产生汇编代码	33	61.11%
本题有效填写人次	54	

对比赛的建议与反馈

- **增加Tutorial（>50%）**
  - 缺少学习方向，没有达到比赛期望的教学效果
- **测例（>50%）**
  - 性能测试用例增加覆盖面，尽量符合现实，避免手动构造
  - 功能测试用例覆盖面，尽量覆盖性能测试设计的语法点
- **规则限制**
  - 开放更多的技术栈，以及工具的使用

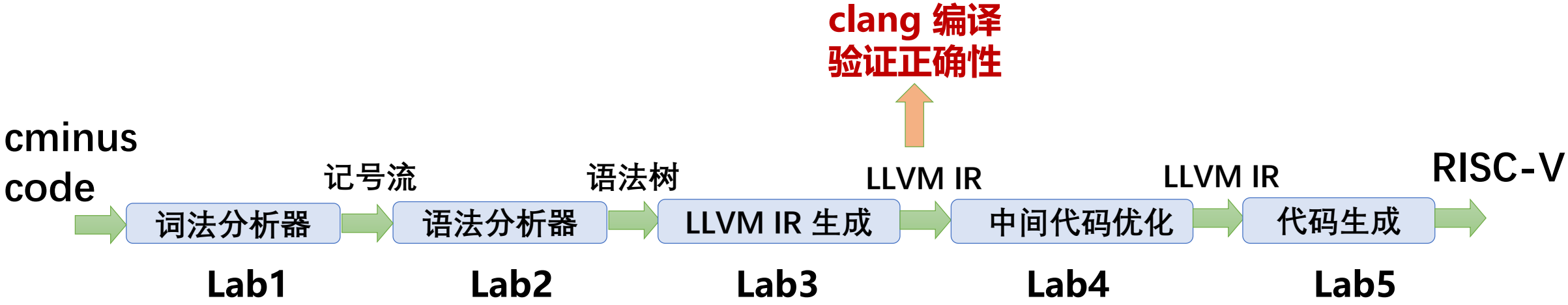
目录

实验的设计架构


输出记号流，
diff验证
输出语法树结构
diff验证正确性
cminus
code
记号流
语法树
LLVM IR
LLVM IR
RISC-V
词法分析器
语法分析器
LLVM IR 生成
中间代码优化
代码生成
Lab1
Lab2
Lab3
Lab4
Lab5

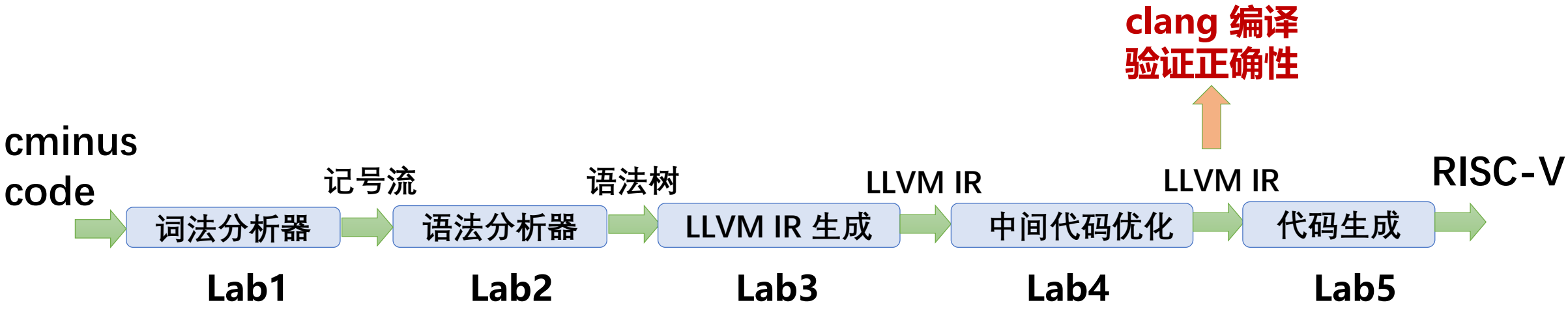

flex+bison工具实现，
同时介绍其他工具作
为拓展

实验的设计架构


clang 编译
验证正确性
cminus code
记号流
语法树
LLVM IR
LLVM IR
RISC-V
词法分析器
语法分析器
LLVM IR 生成
中间代码优化
代码生成
Lab1
Lab2
Lab3
Lab4
Lab5


Before:
调用 LLVM 核心类实现
After:
调用助教简略化设计的
核心类实现

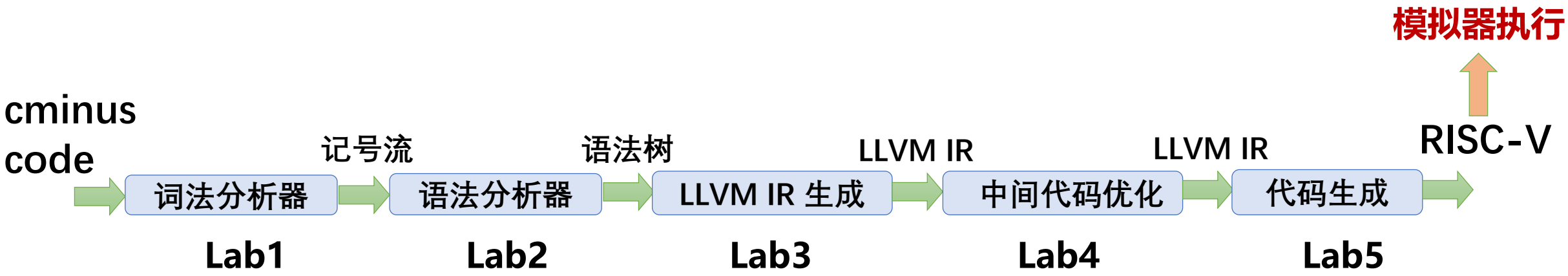
实验的设计架构



讲练结合（新增实验）

- 阅读Mem2reg pass的流程
- 实现块内优化pass及分析pass
- 选做块间优化pass

实验的设计架构



（设计中）
提供代码生成框架，学生实现寄存器分配算法与RISC-V汇编生成

实验设计的思考

- **减负与重视实践权衡：**
  - 详尽的文档与助教提供的代码框架，讲练结合，让学生仅关注核心实现逻辑
  - 团队合作开发，git管理
  - 分级实验与选做设置，学生选择面更广。
- **引进前沿技术栈**
  - LLVM，RISC-V
- **未来展望**
  - 课程实验后端实验可以联合体系结构实验（FPGA上实现RISCV-32I的五级流水线CPU）做出综合性的成果

谢谢大家！

谢谢筹划比赛的老师
谢谢一直陪伴我们的指导老师