LAB 2-3

基于LLVM IR的读取优化

公开报告目录

- 项目
  - 动机
  - 工具
  - 核心算法
  - 使用方法
  - 优化示例
  - 优点与不足
  - 展望
- 团队
  - 分工贡献
  - 历史
- 交互
  - 现场演示
  - 询问

项目 – 动机

高中写的 fib.c

```
int main()
{
    int i, fib[100]={0, 1};

    for(i = 2; i < 100; i++)
        fib[i] = fib[i - 1] + fib[i – 2];

    return fib[i - 1];
}
```

项目 – 动机

用 CLANG 编译成 LLVM IR

```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca [100 x i32], align 16
  store i32 0, i32* %1, align 4
  %4 = bitcast [100 x i32]* %3 to i8*
  call void @llvm.memset.p0i8.i64(i8* %4, i8 0, i64 400, i32 16, i1 false)
  %5 = bitcast i8* %4 to [100 x i32]*
  %6 = getelementptr [100 x i32], [100 x i32]* %5, i32 0, i32 1
  store i32 1, i32* %6
  store i32 2, i32* %2, align 4
  br label %7

……
```

项目 – 动机

多条相同的 LOAD 指令

```
COND:
%8 = load i32, i32* %2, align 4
BODY:
 %11 = load i32, i32* %2, align 4
 %16 = load i32, i32* %2, align 4
 %22 = load i32, i32* %2, align 4
INC:
 %26 = load i32, i32* %2, align 4
```

项目 – 动机

迭代变量 i 的值在自增前没有变化

```
int i, fib[100]={0, 1};

for(i = 2; i < 100; i++)
        fib[i] = fib[i - 1] + fib[i – 2];

return fib[i - 1];
```

项目 – 动机

仅需保留一条 LOAD 指令

COND:
%8 = load i32, i32* %2, align 4
BODY:
~~**%11 = load i32, i32* %2, align 4**~~
~~**%16 = load i32, i32* %2, align 4**~~
~~**%22 = load i32, i32* %2, align 4**~~
INC:
~~**%26 = load i32, i32* %2, align 4**~~

项目 – 工具

LLVM PASS

- 官方子系统
- 输入待优化的 **LLVM BC**
- 识别函数、基本块、循环等
- 满足条件则触发执行
- 输出优化后的 **LLVM IR**

项目 – 核心算法

问题和优化基本思想

问题：
源语言中变量每次被取值都会在 **LLVM IR** 中对应一条 **LOAD** 指令

优化基本思想：
源语言中的变量，以它被赋值（作为左值）的语句为分界线，每段代码中无论其被取值（作为右值）的次数，它对应的 **LLVM IR** 虚拟寄存器应当只执行一条 **LOAD** 指令。

项目 – 核心算法

观察 **LLVM IR**

- 逢定义变量生成 **ALLOCA** 语句
- 逢作为左值生成 **STORE** 语句
- 逢作为右值生成 **LOAD** 语句

项目 – 核心算法

读入合并优化基本策略

- 对代表变量 **p** 的虚拟寄存器 **%p**：
  - **%p = ALLOCA TYPE, ALIGN L**：
    - **%p** 指向的值不确定，增加记录为无效记录 **(%p, X)**；
  - **STORE TYPE %q, TYPE* %p, ALIGN L**：
    - **%p** 指向的值改变了，修改记录为无效记录 **(%p, X)**；
  - **%q = LOAD TYPE, TYPE* %p, ALIGN L**：
    - 如果没有键为 **%p** 的有效记录，新建有效记录 **(%p, %q)**；
    - 如果有键为 **%p** 有效记录，删除此语句，后续需要对应更新。
  - 其他语句：
    - 不修改键值对。

项目 – 核心算法

基本块内优化

基于“推导优化基本策略”：

- 设计两个从寄存器到寄存器的映射表 **LOADVAL** 与 **ALIAS**：
  - **LOADVAL** 的一个条目表示键寄存器当前的值已经被值寄存器读取过，若为 **X** 表示目前的值尚未被读取过，初始清空；
  - **ALIAS** 的一个条目表示语句中出现的键寄存器应该被修改成值寄存器，初始清空；

项目 – 核心算法

基本块内优化

- SIMPLIFY(BASICBLOCK B):
  - FOR EACH INSTRUCTION I IN B：
    - %p = ALLOCA TYPE, ALIGN L || STORE TYPE %q, TYPE* %p, ALIGN L
      - LOADVAL[%p]=X;
    - %q = LOAD TYPE, TYPE* %p, ALIGN L
      - IF(LOADVAL.FIND(%p)) ALIAS[%q] = LOADVAL[%p]; DELETE STATEMENT;
      - IF(!LOADVAL.FIND(%p)) LOADVAL[%p] = %q;
    - OTHERWISE
      - FOR EACH OPERAND IN STATMENT
        - SUBSTITUTE OPERAND WITH ALIAS[OPERAND];

项目 – 核心算法

跨基本块优化

基于“基本块内优化策略”：

- 思考
  - 利用某个块的所有前驱基本块执行 **Simplify** 后的 **LOADVAL** 的信息，可能可以把前驱基本块中的信息应用到该基本块上，作为下一次迭代的“前提”。
  - 一个迭代回合会将信息传递一步，所以某条 **LOAD** 信息理论上传遍全 **CFG** 所耗费的回合数至多是该 **CFG** 最长链的长度 **D** 再减去 **1**，所以如果最后 **D – 1** 个迭代回合都没有执行任何删除，那么意味着无可优化，并且所有的虚拟寄存器改写已经完成，迭代终止。
  - 修改 **LOADVAL** 与 **ALIAS**：
  - 每个基本块独享一个 **LOADVAL**，初始清空，也就是没有前提；
  - 每个函数的所有基本块共享一个 **ALIAS**，初始清空；

项目 – 核心算法

跨基本块优化

- 前提更新策略：
  - 清空前提；
  - 如果所有前驱基本块的 **LOADVAL** 中都有一致的键值对 **(%p, %q)**（允许有一些前驱结点没有关于 %p 的键值对），那么前提新增键值对 **(%p, %q)**；
  - 如果一些前驱基本块的 **LOADVAL** 中关于 **%p** 的值不一致，那么前提新增键值对 **(%p, X)**。

项目 – 核心算法

跨基本块优化

- **UPDATE(BASICBLOCK B):**
  - **B.PREC = Φ;**
  - **FOR EACH PREDECESSOR P OF B：**
    - **FOR EACH ITEM T IN P.LOADVAL**
      - **IF(B.PREC [T.FIRST] DOESN’T EXIST) B.PREC[T.FIRST] = T.SECOND;**
      - **IF(B.PREC[T.FIRST] EXISTS)**
        - **IF(B.PREC[T.FIRST] != T.SECOND) B.PREC[T.FIRST] = X;**

项目 – 核心算法

函数内优化

- **OPTIMIZE(FUNCTION F):**
  - **ALIAS = Φ;**
  - **COUNTDOWN = D;**
  - **FOR EACH BASICBLOCK B IN F, B.PREC = Φ;**
  - **DO**
    - **COUNTDOWN--;**
    - **FOR EACH BASICBLOCK B IN F, SIMPLIFY(B);**
    - **IF(ANY STATEMENT DELETED) COUNTDOWN = D;**
    - **FOR EACH BASICBLOCK B IN F, UPDATE(B);**
  - **WHILE COUNTDOWN > 0;**

项目 – 核心算法

跨函数优化

- 存在跨函数优化的可能性，例如函数使用了非局部变量。
- 调用函数的结构与基本块跳转不同，相对更复杂。
- 暂时不做跨函数优化，或者说目前认为需要优化的情形较为罕见。

项目 – 核心算法

模块优化

- **OPTIMIZE(MODULE M):**
  - **SKIP METADATA PART;**
  - **FOR EACH FUNCTION F IN M, OPTIMIZE(F);**
  - **SKIP METADATA PART;**

项目 – 使用方法

方法一

- 确保 **LoadOpt.so** 和 **run.sh** 在同一目录下。
- 在该目录下执行 **./run.sh /path-to/src**
  - **src** 的扩展名可以是 **.c**，**.ll** 或者 **.bc**；
  - 优化的 **LLVM IR** 放在和 **src** 同目录下的 **src-opt.ll** 中；

项目 – 使用方法

方法二

- 将 **LoadOpt** 文件夹放置到 **/path-to-llvm-src/lib/Transforms** 下。
- 修改 **/path-to-llvm-src/lib/Transforms/CMakeLists.txt**
  - 新增 **add_subdirectory(LoadOpt)**
- 在 **/path-to-llvm-build/** 下使用 **make** 重新编译 **LLVM**。
- 执行以下命令进行优化。
  - **opt -load /path-to-llvm-build/lib/LoadOpt.so -labopt -disable-output /path-to/src 2> /path-to/dest**
  - **src** 必须是 **BITCODE**;
- 优化的 **LLVM IR** 放在 **/path-to/dest** 中。

项目 – 优化示例

CLANG 直接编译的 fib.ll

```
; METADATA

define i32 @main() #0 {
        …
        ret i32 %33
}

; METADATA
```

项目 – 优化示例

LABOPT 优化后的 fib.ll

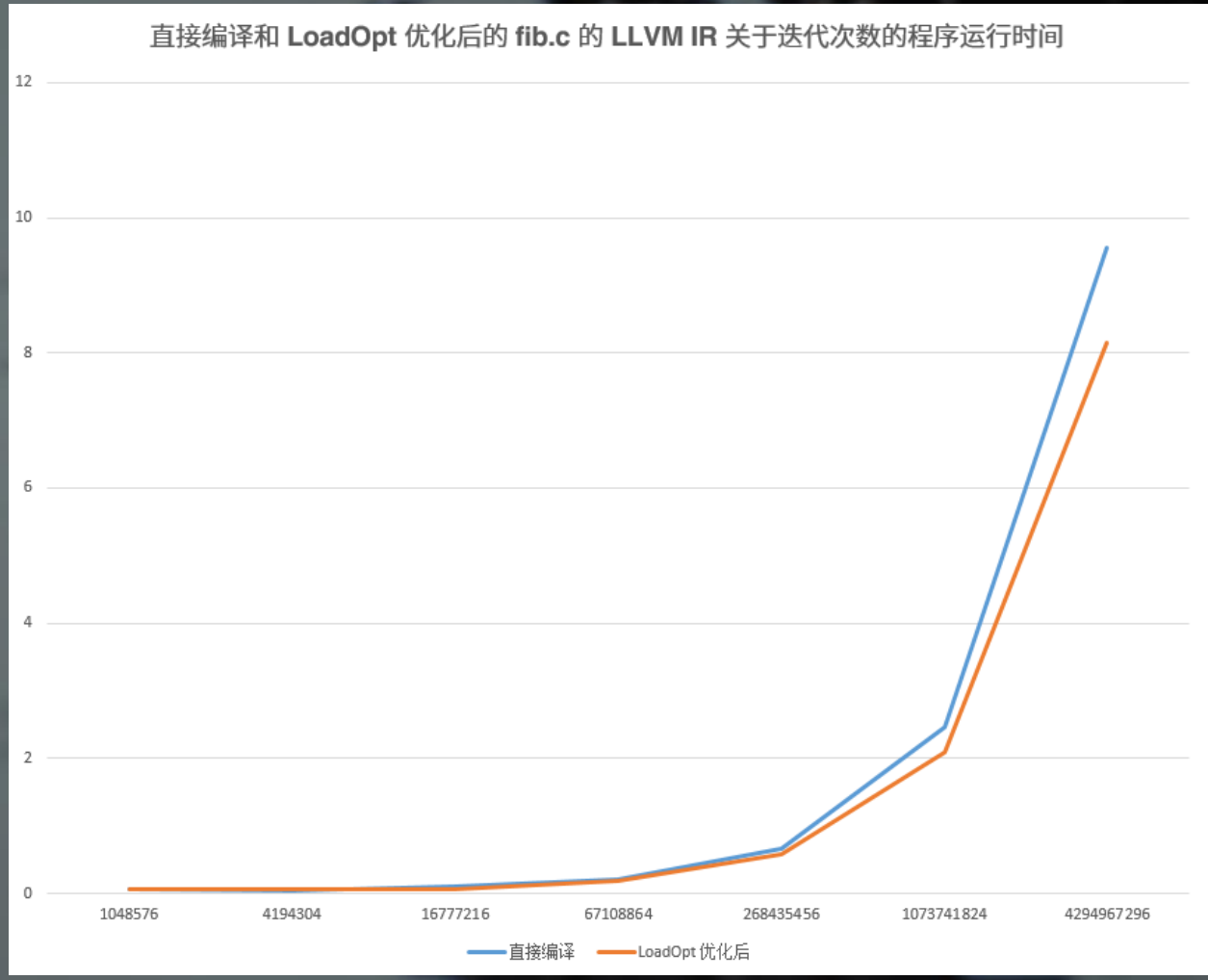
```
; METADATA

define i32 @main() #0 {
        …
        ret i32 %28             ; 6 LOADS OF i, 5 OPTIMIZED
}

; METADATA
```

项目 – 优化示例

以上口说无凭，我们会在现场演示阶段证实。

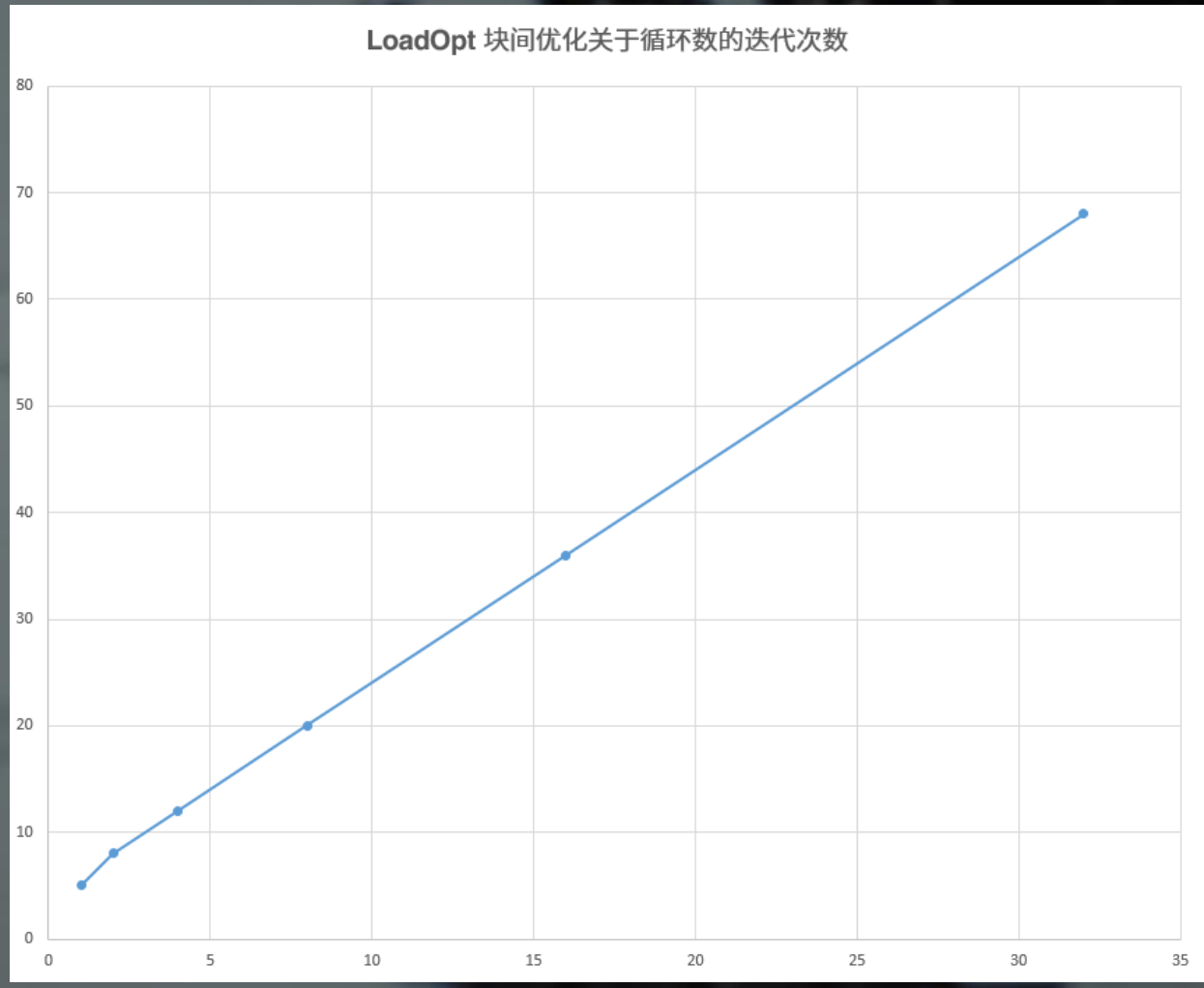
项目 – 优点与不足

规模充分大时程序耗时优化 **5% ~ 15%**



项目 – 优点与不足

优化算法的渐进时间复杂度为线性



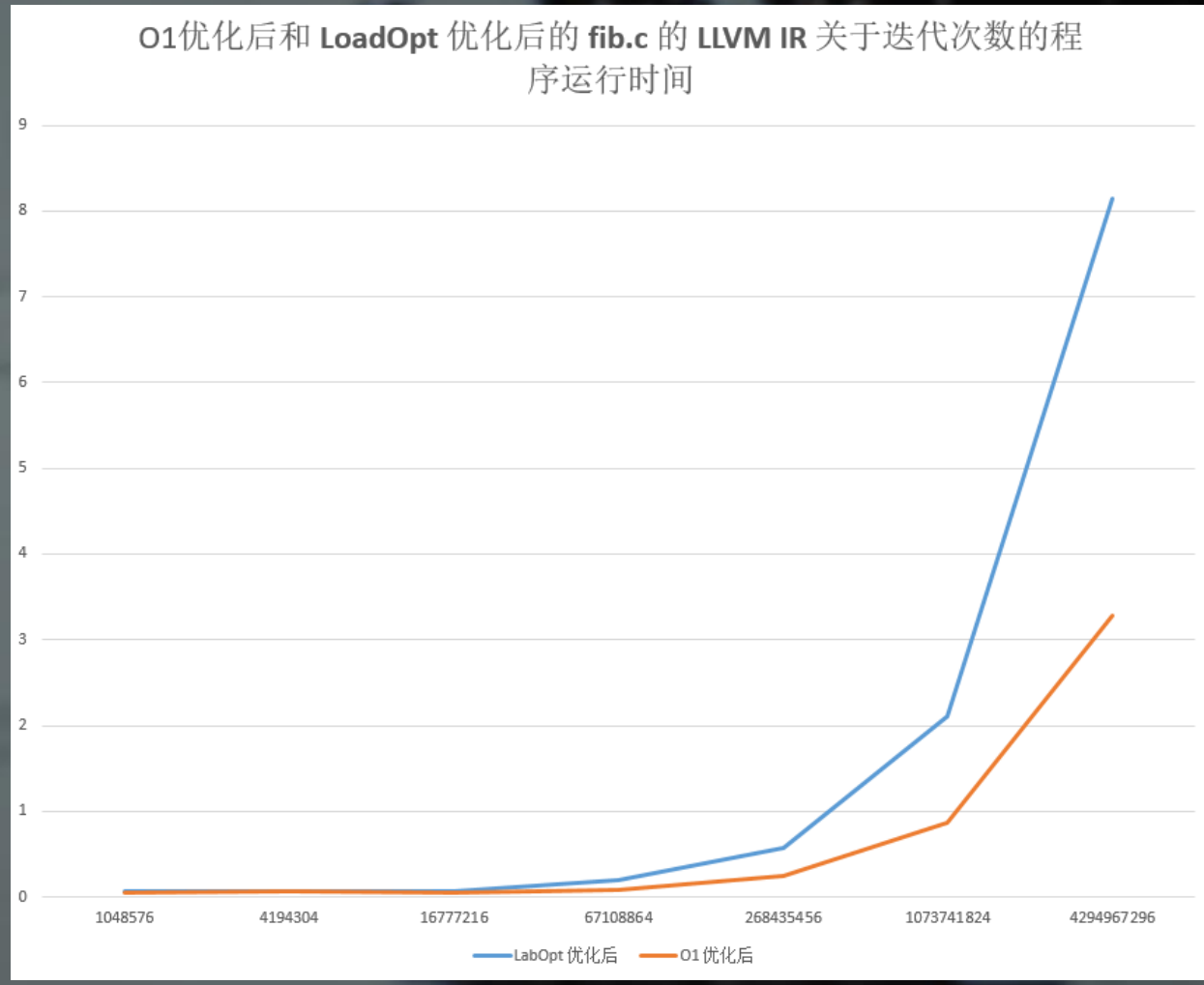
项目 – 优点与不足

```
OPTIMIZE(FUNCTION F):
    ALIAS = Φ;
    FOR B : F, B.PREC = Φ;
    DO
        FOR B : F, B.INITLV = B.PREC;
        FOR B : F, SIMPLIFY(B);
        FOR B : F, UPDATE(B);
    WHILE EXISTS B IN F, B.INITLV !=
    B.PREC;
```

算法主体 **SIMPLIFY** 和 **UPDATE** 可并行

项目 – 优点与不足

优化性能仅为 **O1** 的 **40%** 附近



项目 – 优点与不足

```
OPTIMIZE(MODULE M):
    SKIP METADATA PART;
    FOR F : M, OPTIMIZE(F);
    SKIP METADATA PART;
```

未实现跨函数优化

项目 – 优点与不足

```
FOR B : F
        // WE HAVE 3 LOADVAL MAPS
        // B.INITLV
        // B.LOADVAL
        // B.PREC
```

当前实现空间开销较大

项目 – 展望

- **读取合并**的跨函数优化
- **读取合并**的跨文件优化
- 增加**读取提升**优化

- 常量替换优化?
- 闭循环优化?
- 无效分支优化?

- ……
- 向 O1 看齐。

团队 – 分工贡献

庞茂林
- 调研 **LLVM PASS**
- 编写核心代码
- 执行性能测试

李文睿
- 编写报告
- 对外交流
- 执行组间测试

金孜达
- 设计算法逻辑
- 撰写 **READ.ME**
- 负责班级展示

团队 – 历史

最初的设想是**静态变量替换**

- 简单来说，就是尽可能把变量换成常量

团队 – 历史

我们甚至都写好了语法制导

- %i = alloca TYPE * k, align t {%i.p = 1; %i.s = new bool[k]; %i.sval = new TYPE[k];}
- %i = getelementptr %j, TYPE * k {%i.p = %j,p; %i.s = %j.s + sizeof(TYPE) * k; %i.sval = %j.sval + sizeof(TYPE) * k;}
- %i = load TYPE, TYPE* %j, align t {%i.p = 0; %i.s = *%j.s; %i.sval = *%j.sval;}
- store TYPE %j, TYPE* %i, align t {%i.p = 1; %i.s = &%j.s; %i.sval = &%j.sval;}
- %i = UnaOp FLAG TYPE %j {%i.p = 0; %i.s = %j.s; %i.sval = UnaOp(%j.sval);}
- %i = BinOp FLAG TYPE %j, %k {%i.p = 0; %i.s = %j.s & %k.s; %i.sval = BinOp(%j.sval, %k.sval);}
- %i = call RETTYPE @FUNC(PARA0TYPE %j0, PARA1TYPE %j1, ...) {%i.p = RETTYPE.p; %i.s = FUNC.s; %i.sval = FUNC.sval;}

团队 – 历史

甚至都写好了 **MARKDOWN**

过程

1. 初始情况下，让所有基本块 $Bi.in$ 和 $Bi.out$ 包含的所有虚拟寄存器的属性都带有初值 $p = 0, s = 1, sval = 0$;
2. 反复执行如下循环直到再无改变为止：
    1. 对所有Bi，执行 $Bi.out = Deduce(Bi, Bi.in)$;
    2. 对所有Bi 和 $\%j$，如果对所有 $Bk \in pred(Bi)$，都有 $Bk.out.\%j.s == 1 \wedge Bk.out.\%j.sval == m$ ，那么令 $Bi.in.\%j = Bk.out.\%j$，否则令 $Bi.in.\%j = (Bk.out.\%j.p, 0, 0)$;

团队 – 历史

结果时候复核发现

- 我们思考的静态替换的层次太浅、没法用在循环里。
- 如果说读入合并优化的效果算是初见成效；
- 那么没法用在循环里的优化效果简直就是微乎其微。

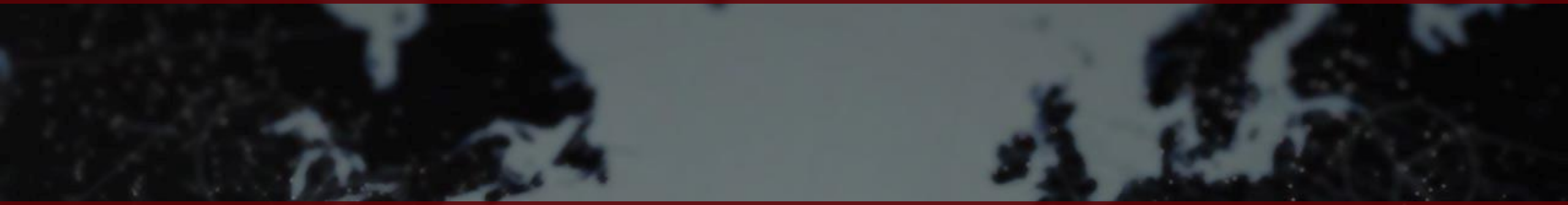
遂压栈。

团队 – 历史

关于如何解析 **LLVM IR** 我们也有争论

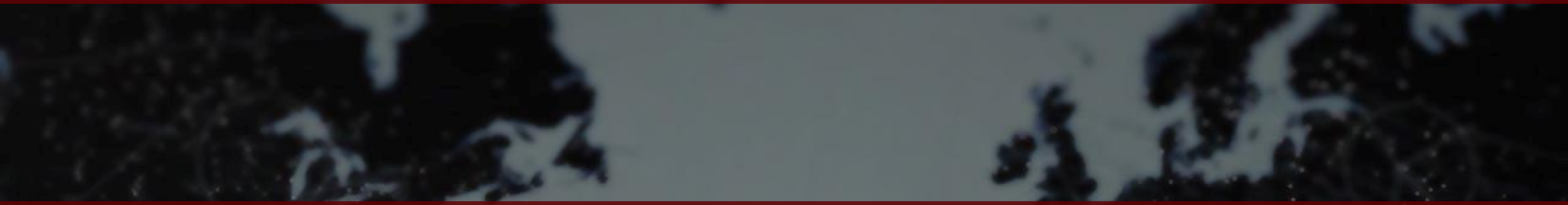
- 直接手动解析费时费力……
- 在 **LAB 2-2** 的基础上优化？那只能适配 **C1** 语言……
- 再套用一趟 **ANTLR**？……
  - **LAB 2-3 += 1-1 + 1-2 + 1-3 + 2-1 + 2-2 ?**
  - 在 **GITHUB** 上真的找到了 **LLVM IR** 的 **.G4** 文件
  - 建议小组改名为 **Матрёшка**。
- 特此感谢 **G2** 的曾明亮为我们指点 **LLVM PASS** 明路。

交互 – 现场演示



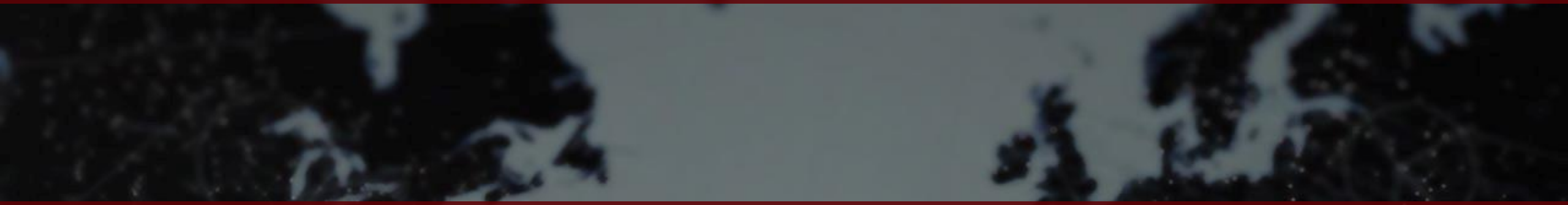
1. 展示使用脚本执行一次对 **fib.c** 的 **LabOpt** 优化；

交互 – 现场演示



2. 对 **fib.c** 比较优化前后的 **LLVM IR** 文件；

交互 – 现场演示

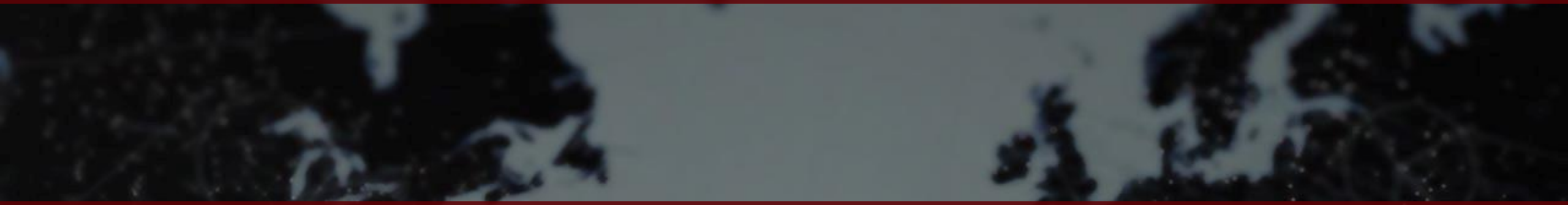


3. 改为采用重新编译生成 **LabOpt.so** 并优化；

交互 – 现场演示

4.对 **fib.c** 进行 **LabOpt** 优化前后的性能比较；

交互 – 询问




RET 0
感 谢 观 看 本 实 验 报 告