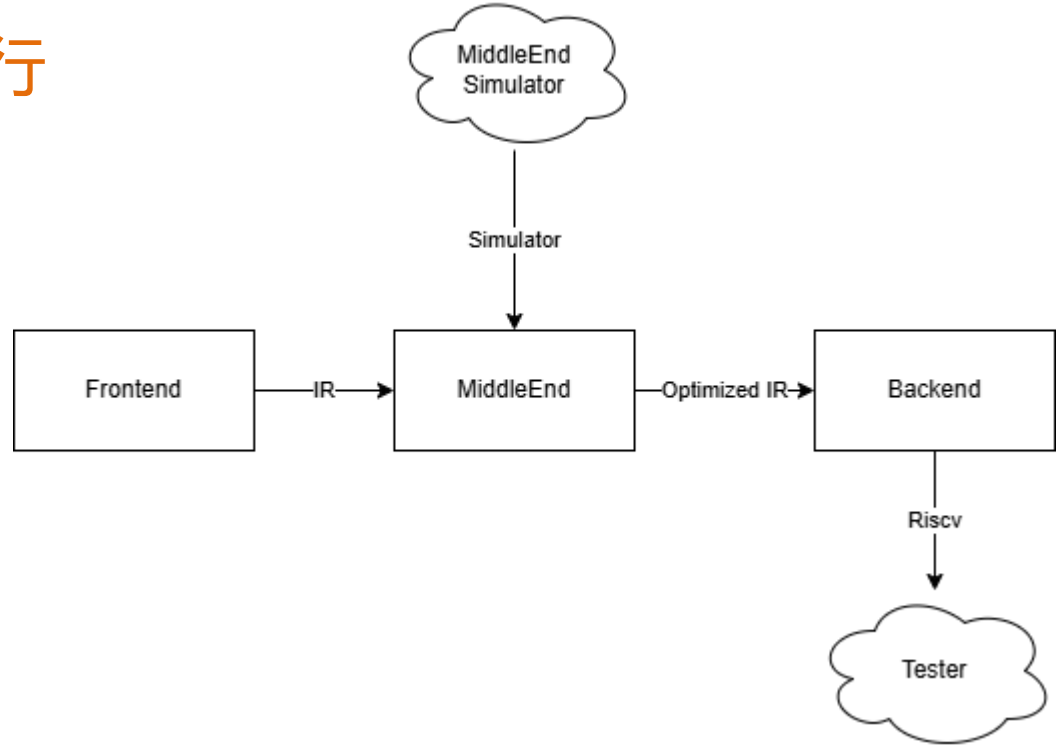
A Brief Analysis of Software Analysis Implementation in Compiler Optimizations

Overview

- 本次分享来源于分享人和队友参加全国计算机系统能力大赛—编译系统设计赛中实现的编译器 SysYc
  - 在测试集上性能超过 gcc O3， 且是比赛中通用优化实现最强的编译器之一
  - 总体采用 Rust 语言实现，共约一万五千行
  - 前端：.sy code -> IR
  - 中端： IR -> optimized IR
  - 后端： optimized IR -> riscv code



FrontEnd

FrameWork

- Pest the elegant parser
  - *pest is a general purpose parser written in Rust with a focus on accessibility, correctness, and performance*
- Using PEG (parsing expression grammars) as input, and transform the sy code to AST form

```
alpha = { 'a'..'z' | 'A'..'Z' }
digit = { '0'..'9' }

ident = { (alpha | digit)+ }

ident_list = _{ !digit ~ ident ~ (" " ~ ident)+ }
          // ^
          // ident_list rule is silent (produces no tokens or error reports)
```

- Parsing Expression Grammar has the following characteristics compared with Context-Free Grammars:
  - ➢ Avoid Ambiguity: A string results in only one effective parse tree or none.
    - Greedy match: every operator consumes as much input as possible and never backtracks (so the expression a*a will always fail, for the first a* consumes all the input ‘a’s)
    - Matching with priority: a rule A <- (expression A)| (expression B) will always match expression A with the higher priority, thus solving ambiguity problems like the dangling else

      (with a syntax like **S** ⟵ 'if' **C** 'then' **S** 'else' **S** / 'if' **C** 'then' **S**)
  - ➢ O(n) Complexity match: by non-recursively matching

MiddleEnd

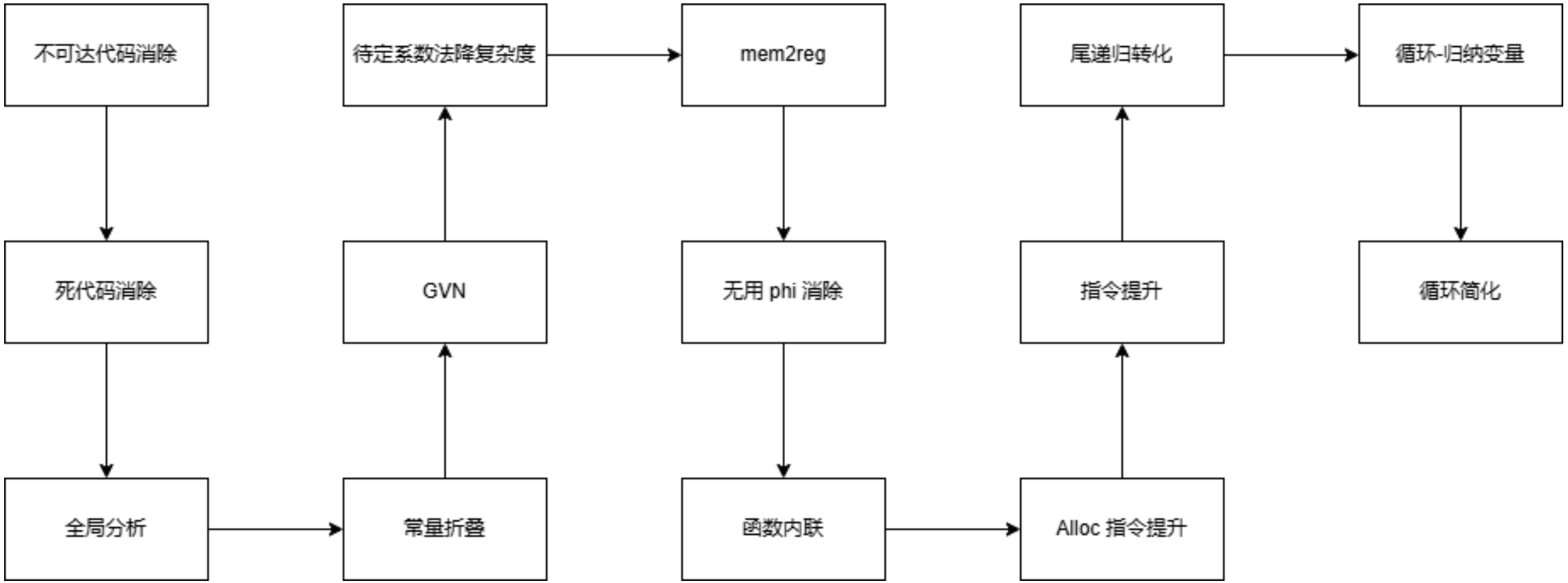
中间代码表示

- IR 较多参考 LLVM IR 形式

```
define i32 @main(){
  entry:
    %13 = call void @_sysy_starttime(i32 21)
    %14 = call i32 @getint()
    %53 = alloca i32*, i32 16
    %54 = call void @func_calc_coef(i32* %53, i32 %14)
    %55 = load f32, i32* %53
    %56 = fmul f32 %55, 1.001
    %57 = getelementptr i32* %53, i32 4
    %58 = load f32, i32* %57
    %59 = fadd f32 %56, %58
    br label %B18
  B18:
    %15 = phi f32 [%59, label %entry]
    %16 = fsub f32 %15, 1.001
    %17 = icmp oeq i32 %16, 0
    br i32 %17, label %B11, label %B12
  B11:
    %18 = call void @putch(i32 112)
    br label %B13
  B12:
    br label %B13
  B13:
```

```rust
#[derive(Clone)]
pub struct ConvertInstr {
    pub var_type: VarType,
    pub target: LlvmTemp,
    pub op: ConvertOp,
    pub lhs: Value,
}
```

Optimizations



Global Analysis-Dominator Tree

清華大學 Tsinghua University

- Domination
  - In a CFG, a node D is said to dominate another node N if every path from the start node to N must go through D
- Dominator Tree
  - The **dominator tree** is a tree structure derived from the CFG, where
    - Each node in the CFG is represented as a node in the dominator tree
    - If a node D dominates a node N and there is no other node P (where P ≠ D and P ≠ N) that also dominates N, then D is the immediate dominator of N.
- 该步之后的无用代码消除，部分冗余消除都是基于 DomTree 的遍历顺序
- 在组里其他同学的工作中， dominator tree 被用来找必经节点节省冗余信息

Optimization-Global Value Numbering

清華大學 Tsinghua University

- 传统的值标号方案：将操作数的值标号和操作符进行 Hash
- 但这样不利于我们发现在代数意义下相等的表达式
- 简单的例子如：Hash(a+b+c) != Hash(a+c+b)
- 即代数意义下相等的表达式经过哈希后得到了不同的值，这不利于我们寻找公共表达式。

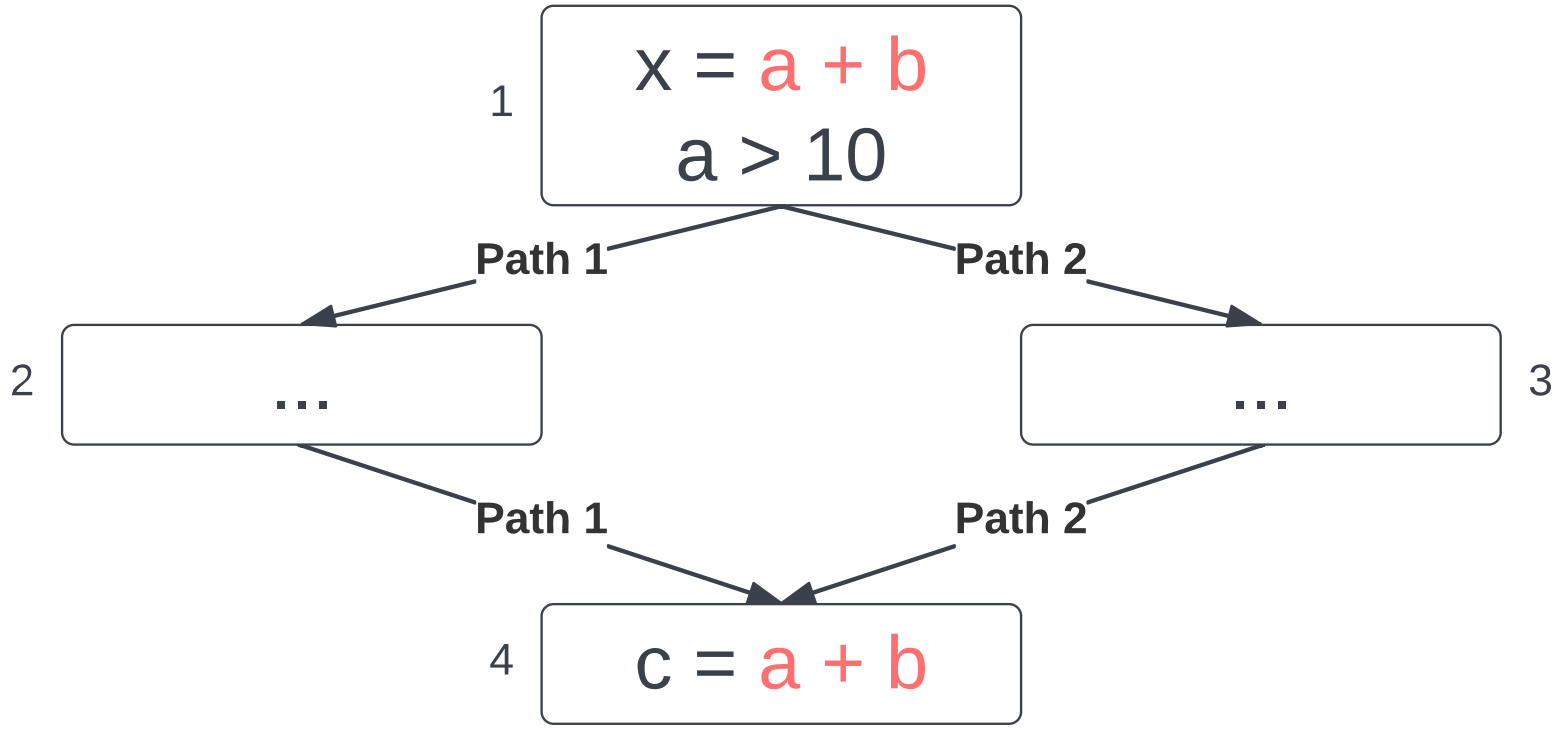
Optimization-Global Value Numbering

- 我们希望构建一个从表达式值空间到值标号空间上的同态映射来作为 hash 方式。
- 我们令值标号空间为Z^{50}，在值标号的计算过程中，我们将表达式的运算符作用于操作数值标号向量的每一个元素上，来得到表达式的值标号。若两个表达式相等，则对应标号必然相等。
- 为了避免标号相等但表达式不相等的情况出现，我们将不可拆分的表达式（如函数参数，I/O 函数返回值）的标号设置为标号空间中的随机数。通过确保随机空间充分大，哪怕在生日悖论的影响下出现碰撞的概率也微乎其微。
- 于是得到了一个能识别出任何相等表达式的值标号算法，而这是大部分值标号算法做不到的。

Optimization-Global Value Numbering

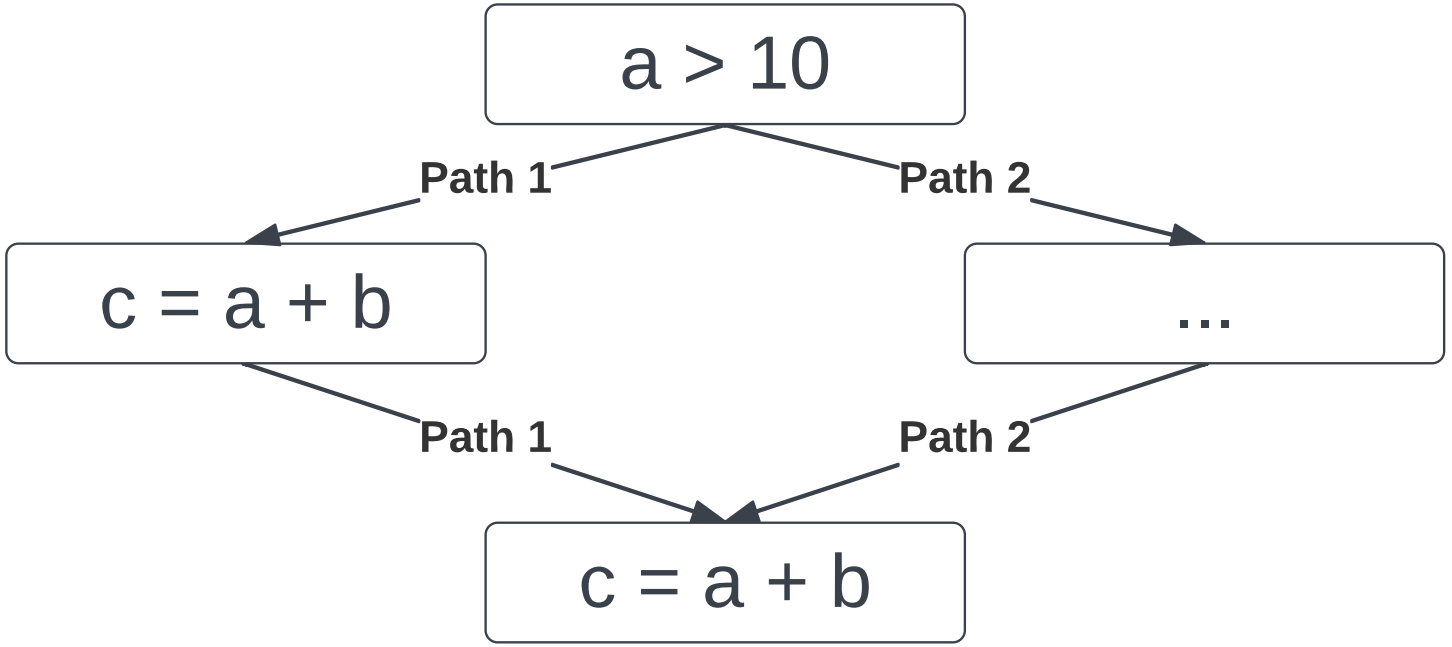
- 特别地，我们将常数 c 的值标号设置为 $c * 1_{50}$
- 通过值标号向量中所有元素是否相等，我们能判断一个表达式的值是否为常数，还可以判断两个表达式是否相差常数。
- 这为寻找归纳变量和计算 store 指令之间的干涉情况提供了便利。

部分冗余消除



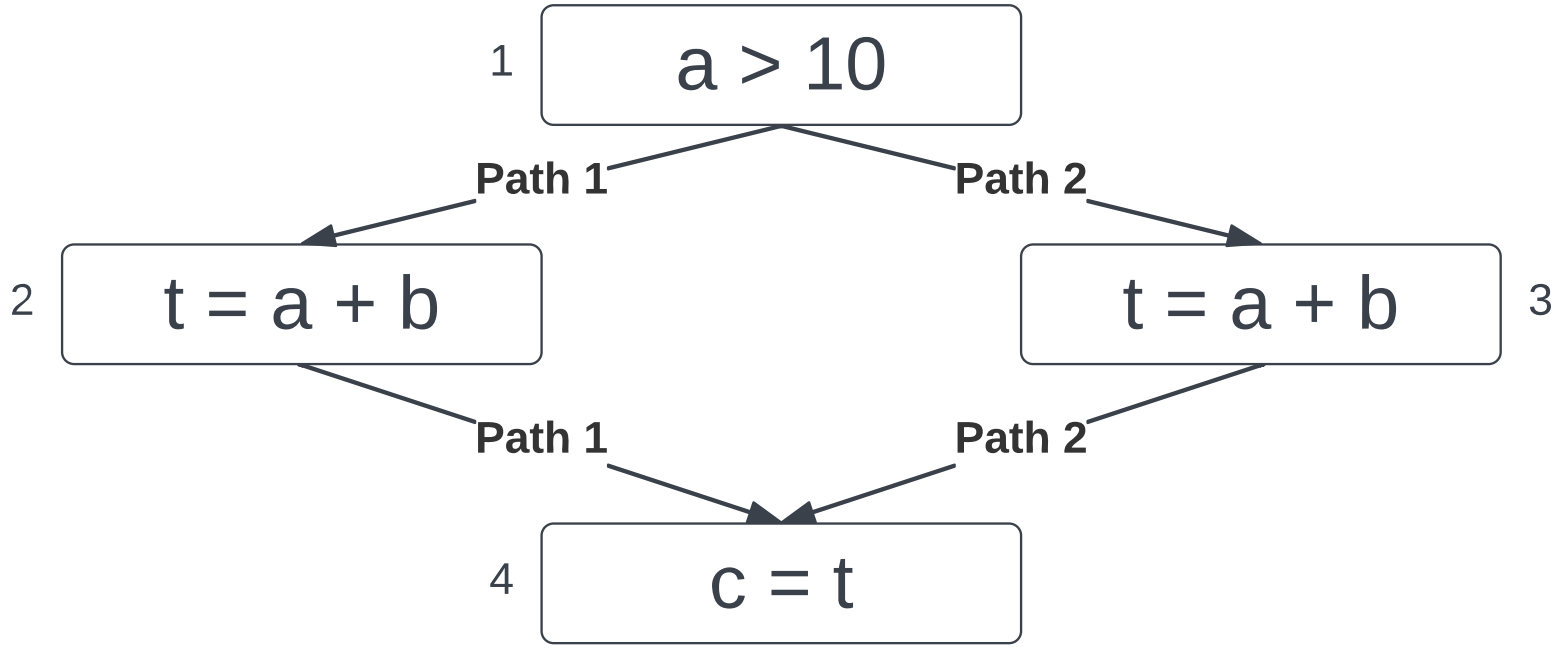
图一：完全冗余

部分冗余消除



图二：部分冗余

部分冗余消除



图三：部分冗余消除

部分冗余消除

- 一般编译器只基于 GVN 实现了完全冗余消除。
- 我们基于 GVN-PRE 算法实现了部分冗余消除
- 特别地，我们实现了对 load 和 store 指令的部分冗余消除，即 GVN-PRE 算法在 ArraySSA 上的扩展。

- 部分冗余消除都是通过插入指令将部分冗余转化为完全冗余，再进行完全冗余消除
  - 对于一个块开头的所有可插入表达式，我们按 use-def 的拓扑序依次考虑每条指令，如果这条指令某个前驱基本块内已经被计算过，那这条指令就是部分冗余的，可插入表达式意为：
    - 该表达式在当前位置可以被计算（所有参数都已经被计算出来）
    - 该表达式在当前位置插入是 down safety 的，即插入后不会有某个原本不能访问该表达式计算结果的块能获得这个表达式的结果。
  - 我们可以通过其他没被计算过的前驱插入指令，以及当前块前面插入 phi 的方式将这条指令变完全冗余。我们将按照支配树的 dfs 顺序进行处理，方便更新子树中变量表示。

Optimization-循环并行

清華大學 Tsinghua University

- 自动发现可以并行的循环，并且进行并行
- 平衡并行的收益和 syscall 调用开销
- 在 matmul 测例有显著提升
  - 并行度 33.224/8.663= 3.83

```
user@starfive:~/para$ ./run_on_board_in.sh ~/tmp0820/mat0618.s ~/tmp0814/matmul1.in
Timer@0023-0092: 0H-0M-8S-289569us
TOTAL: 0H-0M-8S-289569us
341954021
real    0m8.663s
user    0m33.224s
sys     0m0.000s
Process exited with code: 0
```

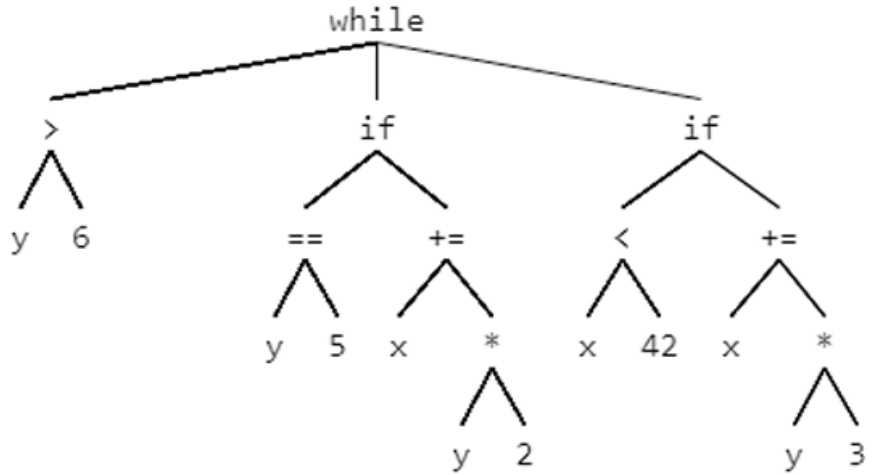
Optimization-记忆化递归

- 实现了 python 中的 cache 装饰器
  - 实现缓存机制，用于存储函数的计算结果或数据，以便在将来快速访问，而无需再次进行计算
- Hash表实现，从而实现线性复杂度的计算

```
int fib(int n){
    if (n <= 0){
        return 0;
    }
    if (n == 1){
        return 1;
    }
    return (fib(n -1) + fib(n-2)) % 1024;
}
```

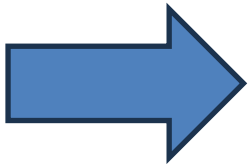
Optimization-代数化简

- 构建表达式树，实现代数化简




```
a+b-(a-c)

(a+b+a+b+.....+a+b)/1000
```



```
b+c

a+b
```

Optimization-待定系数法降复杂度

- Motivating example
  - 对于一个纯函数
  - 多次递归调用，指数复杂度 though 良好性质
  - 总复杂度为指数

$$func(data, num) = [f1, f3](num)^T \cdot [data, 1] = f1(num) \cdot data + g(num)$$

  - 构造专门函数，求解 f1(num),g(num)
  - 进而改写为一次递归，复杂度直接降到 O(n)！

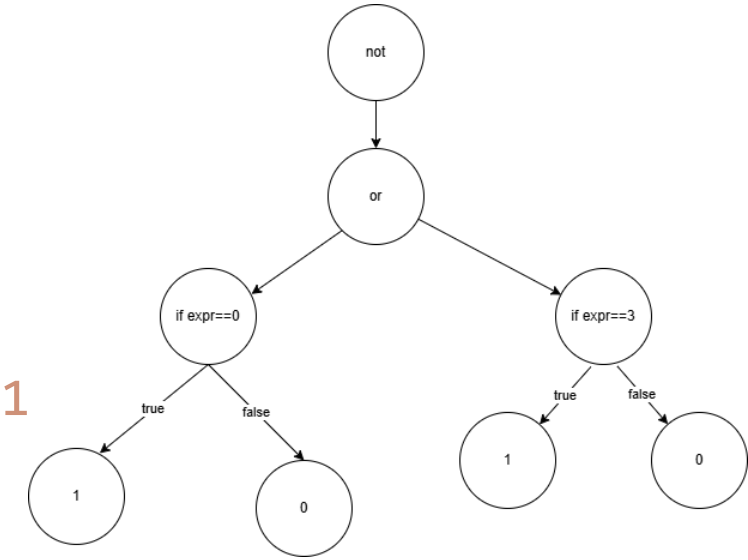
```
float func(float data, int num) {
    if (num < 0) {
        return 0;
    }
    num=num-1;
    data = data + func(data, num);
    data = data - func(data, num);
    return data;
}
```

Why this Expression Tree

- 在上面两个 step 中的表达式树，和 angr 符号化变量的表示形式基本一致, so why this form?
  - 方便表示很复杂的形式，且即使是表达算数类型也不会直接造成误差，如浮点数精度损失
    - 在上述待定系数法求解过程中尝试过将一个变量表示为 [constant1,constant2][data,1]^T 的形式，但是因为浮点数加乘不符合结合律和交换律，会产生误差
  - 方便从底向上/从顶向下进行约束求解或者代数化简
    - 从底向上：如将 (a + b)- (a - c) 化简，可以用 [constant1,constant2, constant3][a,b,c]^T 的形式表示树中每个节点

Why this Expression Tree

- 在上面两个 step 中的表达式树，和 angr 符号化变量的表示形式基本一致, so why this form?
  - 方便从顶向下进行约束求解或者代数化简
    - 如在 angr 中会有以下表现形式： `<Bool ~((if *<BV64 Type3InputBuffer_73_64>_103_4096[134:133] == 0 then 1 else 0) | (if *<BV64 Type3InputBuffer_73_64>_103_4096[134:133] == 3 then 1 else 0)) == 0>`
    - z3 solver 难以直接求解如上含有条件关系的式子，而 expression tree 方便按照树中路径拆分成 condition + value 形式进行后续求解



Range Analysis（Unimplemented）

Tsinghua University

- Given the value ranges of some variables in the IR, we can use range analysis to propagate and calculate the value ranges of others variables.

```
if (a < 100 && b < 100)
  {
    c = a + b;
    if (c < 200)
      foo (c);
  }
```

- Also uses a form similar to the expression tree.

- Be useful in scenes like deadcode elimination, branch reducing and such.

Optimization-循环

- Prerequisite: 循环树
  - 在现有的语言定义下，若某一个块支配了自己的前驱，则这个块就是一个循环的入口。
  - 而如果某一个入口支配了另一入口，则它是另一入口的父循环。
  - 据此可以将每个函数的循环构建成一个森林，再将整个函数看作一个仅执行一次的循环，成为森林的虚点，则构建出了循环树。
  - 这为后续的循环优化打下了基础

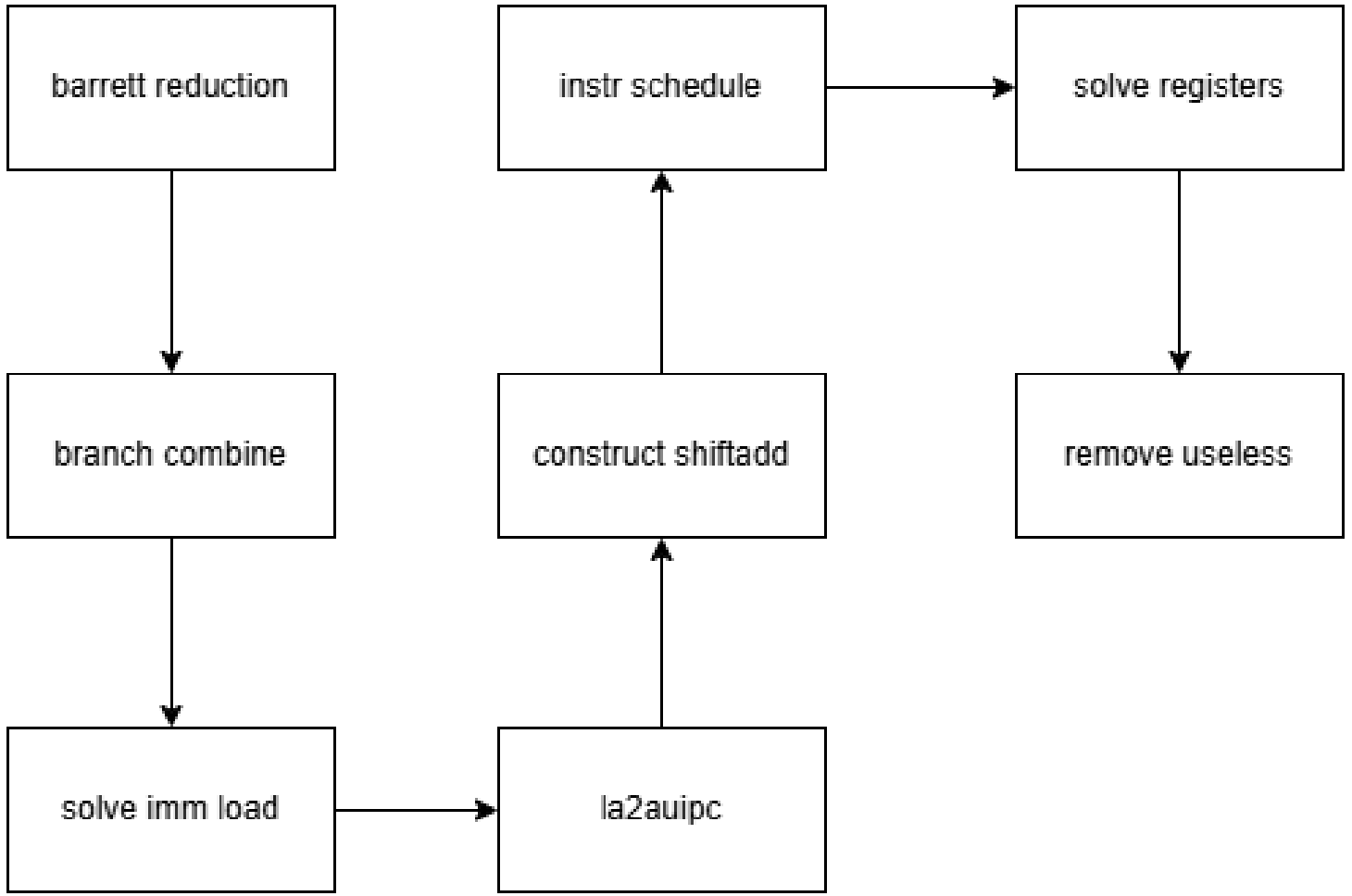
Optimization-循环

清華大學 Tsinghua University

- 归纳变量外推
  - 在一个能确定循环次数的循环中，若某归纳变量未在循环内使用，则在编译期直接计算出结束循环时它的取值，取消在循环内对它的计算
- 强度削弱
  - 对于需要在循环内使用的归纳变量，则尝试对它的计算过程进行简化。
  - 常见的简化有：乘法变成加法，连续的加法合并
- 循环展开
  - 对于循环次数是常数的循环，可以将其循环体复制常数份，顺序执行

BackEnd

Optimization-Overview

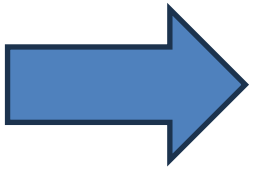


Optimization-peephole

- 除常数优化 (Barrett Reduction)
  - 一条 div 有 9~64 个时钟周期，并且相应硬件单元有冷却时间，而加减乘指令只有 1~3 个时钟周期
  - 利用数论上的等价变换，用多条加，减，位运算指令替代除法指令

- Branch Combine

```
xor s1, s1, s3          # xor %3, %1, %2
seqz s1, s1             # seqz %4, %3
bne s1, x0, L_12        # bne %4, x0, L_12
```



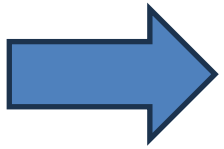
```
bne s3,s1,L_12
```

Optimization-peephole

- ShiftAdd
  - 利用板子自带的 shiftadd 指令，进行 shift + add 指令合并

- La2auipc
  - La 为伪指令，可以转化为 auipc + add
  - 改变寻址方式，每次节省一条指令！

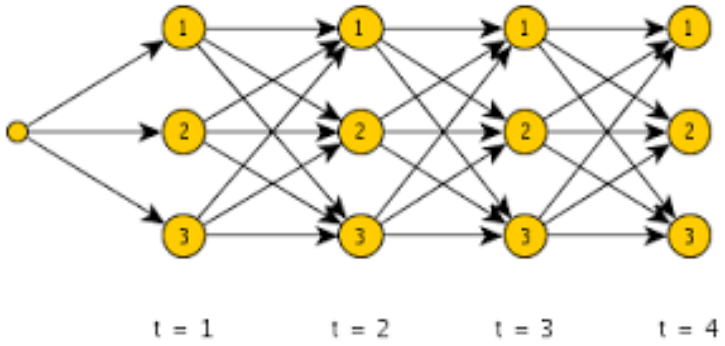
```
la %1, sym
lw %2, 0(%1)
// calculations with %2....
sw %32, 0(%1)
```

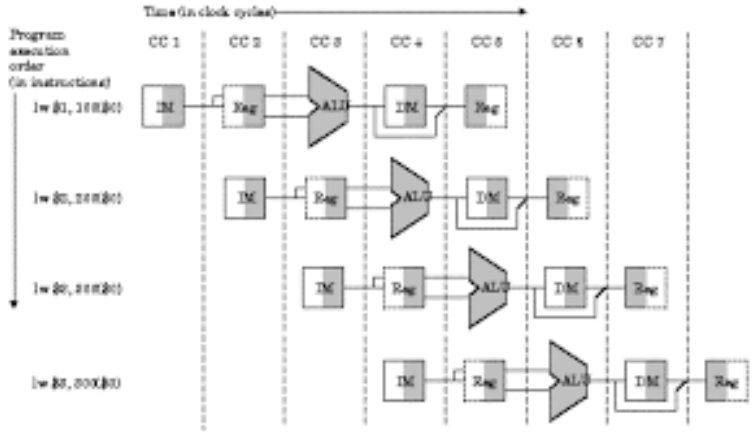


```
lui %1, %hi(sym)
lw %2, %lo(sym)(%1)
// calculations with %2
sw %3, %lo(sym)(%1)
```

Optimization-InstrSchedule

- 指令调度
  - 首先将指令调度问题转换成了一个优化问题，并且设置代价函数来取得最优
  - 该估价函数求出发射一个指令的收益，并期望最大化收益，该指令与当前已发射指令和可用寄存器有关
  - 采用神经网络中使用的 **Viterbi** 算法进行搜索，求解一个最优的指令调度序列





Optimization-InstrSchedule for Reg

Tsinghua University

- 降低寄存器压力的指令调度
  - 考虑到寄存器溢出带来的访存负担，我们优先发射最能终结寄存器生命周期的指令
  - Do liveliness analysis for each block, and find out the instruction where each register is last used.
  - Construct instruction dag for each block based on the def-use dependency and inherent order of certain instructions
  - If an instruction ends the lifetime of a variable directly or tends to end the lifetime, we give it a higher value in the judgement function

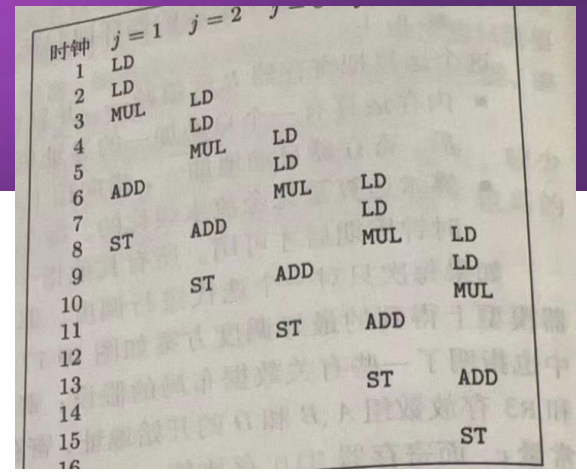
$$\mathrm{LIVE}_{in}[s] = \mathrm{GEN}[s] \cup (\mathrm{LIVE}_{out}[s] - \mathrm{KILL}[s])$$
$$\mathrm{LIVE}_{out}[final] = \emptyset$$
$$\mathrm{LIVE}_{out}[s] = \bigcup_{p \in succ[s]} \mathrm{LIVE}_{in}[p]$$

$$\mathrm{GEN}[d : y \leftarrow f(x_1, \cdots, x_n)] = \{x_1, \ldots, x_n\}$$
$$\mathrm{KILL}[d : y \leftarrow f(x_1, \cdots, x_n)] = \{y\}$$

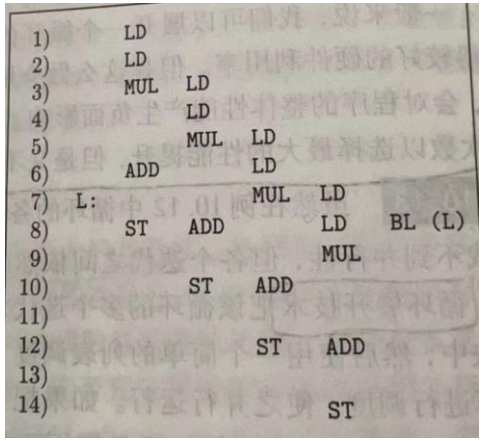
Optimization-InstrSchedule based on hardware pipelining

- 硬件流水线指令调度
  - 软件流水线采用的估值函数是该指令到 DAG 中没有后继节点的指令的最短用时，是很多主流算法采取的实现
  - 而硬件流水线则根据硬件环境（开发板型号）模拟了一个可以同时执行两条算数指令，一条跳转指令，一条浮点指令，一条访存指令的流水线
  - 对于每一条指令，估值函数为把该指令放到流水线上带来的时间增量
  - 直接效果是可以增加访存指令之间的距离，带来较为显著的性能提升

Optimization-龙书软流水（unimplemented）



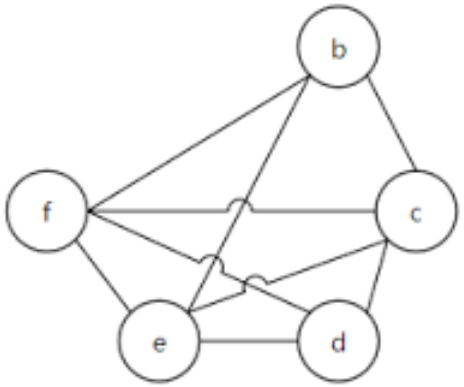
- 基于循环软流水的指令调度（龙书算法）
  - 多次迭代的时候，调整不同循环中的起始间隔和一个循环中的指令间的间隔，来使得硬件流水上总体执行的时间最少
  - 在求解循环最小间隔时，用到了类似 taint analysis 的方法，具体来说，首先我们确定单次循环中哪些变量和归纳变量有关，使用了 taint analysis
  - 此外我们展开前 1..n 次循环，对于第 n 个循环的写入值，它可能时和很多变量有关且经过了较为复杂的计算，而我们分析它最早依赖于哪次循环的变量时，也是用了 taint analysis 去 taint 它依赖的各个变量，然后分析他们依赖的变量最早哪次循环中被写入
  - 但是龙书算法会增加寄存器负担，所以未被合并入最终版本




```
for (int i = 2; i < 10; i++)
{
    d[i] = d[i - 1] + d[i - 2];
}
```

Optimization-图染色寄存器分配

- 一般编译器会实现朴素的图染色寄存器分配算法，每次只溢出一个寄存器，然后需要重建干涉图来溢出下一个寄存器，后续使用被溢出的值时，需要使用 load 语句恢复。
- 我们观察到，确定一个寄存器需要溢出后，我们可以虚拟地把它从干涉图上删去，得到一个子图。**在子图上需要溢出的寄存器一定在原图上需要溢出。**
- 因此，我们实现的图染色寄存器分配一次遍历可以确定多个需要溢出的寄存器



Optimization-图染色寄存器分配

清華大學 Tsinghua University

- 经实验，在公开测例上，最多执行三遍图染色，可以识别出所有需要溢出的寄存器
- 此外，我们还有两大提升
- 1. 在寄存器溢出后，我们可以在线性时间内对控制流信息（livein, liveout）进行增量更新。
- 2. 若溢出值重算代价不高，则使用重算替代 load 语句来恢复寄存器的值（例如 li 加载立即数和 la 加载全局地址）

Optimization-BasicBlock Reordering

清華大學 Tsinghua University

- Pettis-Hansen Heuristic
  - 根据每个 basic block 的入边类型预测出它的调用频率，进而将调用频率较高的块放在程序较前的位置，实现较好的缓存利用
  - 部分 Windows 驱动中实现了全局的 block reordering,具体表现形式是，同一个函数的基本块之间不是连续的，可能同个函数的基本块之间还有其他函数的块（本来这里应该有个示意图 但是简单翻了一下没找到 有机会再补吧）
  - linux kernel 中的 FG-KASLR 选项可能会降低运行性能可能也是打乱了 basic block 的顺序从而导致缓存利用不够

Q&A

清華大學
Tsinghua University

Thank you for listening!