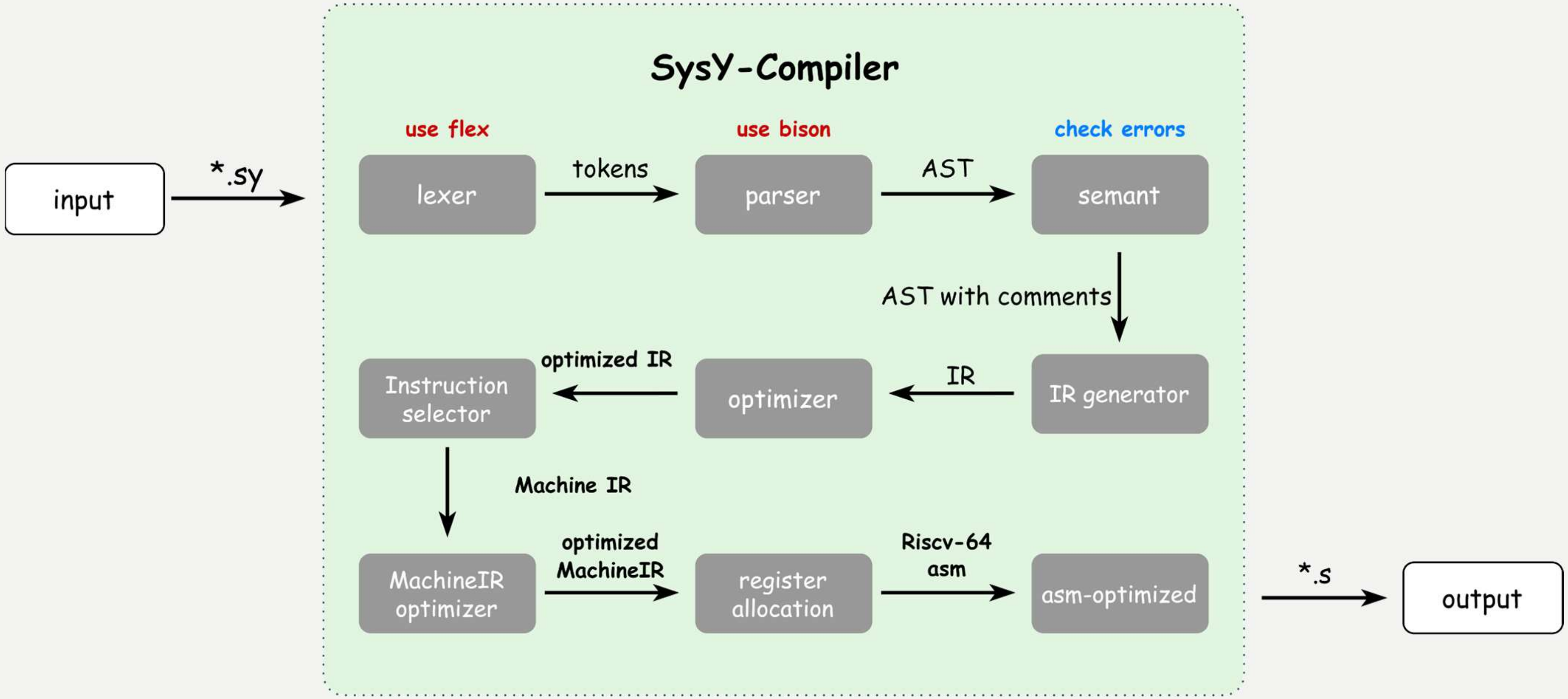
编译器架构


input
*.sy
SysY-Compiler
use flex
lexer
tokens
use bison
parser
AST
check errors
semant
AST with comments
IR generator
IR
optimizer
optimized IR
Instruction selector
Machine IR
MachineIR optimizer
optimized MachineIR
register allocation
Riscv-64 asm
asm-optimized
*.s
output

词法分析

```
1  int a = 5;
2  int main()
3  {
4      putf("hello world\n");
5      return 0;
6  }
```

使用flex辅助生成tokens

1	Token	Lexeme	Property	Line	Column
2	INT	int		1	0
3	IDENT	a	a	1	4
4	ASSIGN	=		1	6
5	INT_CONST	5	5	1	8
6	SEMICOLON	;		1	9
7	INT	int		2	0
8	IDENT	main	main	2	4
9	LPAREN	(		2	8
10	RPAREN	)		2	9
11	LBRACE	{		3	0
12	IDENT	putf	putf	4	4
13	LPAREN	(		4	8
14	STR_CONST	"	hello world\n	4	23
15	RPAREN	)		4	24
16	SEMICOLON	;		4	25
17	RETURN	return		5	4
18	IDENT	a	a	5	11
19	SEMICOLON	;		5	12
20	RBRACE	}		6	0

语法分析

```c
int a = 5;
int main()
{
    putf("hello world\n");
    return 0;
}
```

使用Bison辅助生成语法树

```
Program
  VarDecls    Type: Int
    VarDef    name:a    scope:-1
      init:
        VarInitVal_exp
          Intconst    val:5    Type: Void
  FuncDef    Name:main    ReturnType: Int
    Block    Size:2
      ExpressionStmt:    Type: Void
        FuncCall    name:putf    Type: Void
          FuncRParams:
            StringConst    type:string    val:hello world\n
      ReturnStmt:
        Lval    Type: Void        name:a    scope:-1
```

语义分析

```
Program
  VarDecls    Type: Int
    VarDef    name:a    scope:-1
      init:
        VarInitVal_exp
          Intconst    val:5    Type: Void
  FuncDef    Name:main    ReturnType: Int
    Block    Size:2
      ExpressionStmt:    Type: Void
        FuncCall    name:putf    Type: Void
          FuncRParams:
            StringConst    type:string    val:hello worl
      ReturnStmt:
        Lval    Type: Void        name:a    scope:-1
```

```c
int a = 5;
int main()
{
    putf("hello world\n");
    return 0;
}
```

```
Program
  VarDecls    Type: Int
    VarDef    name:a    scope:0
      init:
        VarInitVal_exp
          Intconst    val:5    Type: Int    ConstValue: 5
  FuncDef    Name:main    ReturnType: Int
    Block    Size:2
      ExpressionStmt:    Type: Void
        FuncCall    name:putf    Type: Void
          FuncRParams:
            StringConst    type:string    val:hello world\n
      ReturnStmt:
        Lval    Type: Int        name:a    scope:0
```

- 识别每个中间变量的类型
- 判断每个变量的作用域
- 预先计算所有常量表达式
- 语义错误检查
  - 使用未定义变量
  - *continue*出现在循环外

中端优化

Mid-End optimizations

Analysis Pass

- ControlFlowGraph
- DomTree
- AliasAnalysis
- MemoryDependencyAnalysis
- LoopCarriedDependencyAnalysis
- LoopBasicInformation
- ScalarEvolution

Transform Pass

Redundent-elimination

- AggressiveDeadCodeElimination
- CommonSubexpressionElimination
- DeadStoreElimination
- RedundantBranchElimination

loop-optimizition

- LoopSimplify
- LCSSA
- LoopInvariantCodeMotion
- LoopFullUnroll
- LoopIdomRecognize
- LoopInvariantCodeMotion
- LoopFusion
- LoopParallel
- LoopStrengthReduce
- LoopUnroll

others

- Mem2reg
- SparseConditionalConstantProgation
- Reassociate
- FunctionInline
- TailRecursiveEliminate
- SimplifyCFG
- InstSimplify
- InstCombine

ScalarEvolution

```c
int i = 1;
while(i <= n){
    a[k+i][3*i+4] = 5*i+5+p;
    i = i+1;
}
```

```
header
{1,+,1}        %i = phi i32 [1,%entry],[%inc,%latch]
               %icmp = icmp sle i32 %i, %n
               br i1 %icmp, label %body, label %exit

body
{%k+1,+,1}     %idx1 = add i32 %k, %i
{3,+,3}        %idx2mul = mul i32 3, %i
{7,+,3}        %idx2 = add i32 %idx2mul, 4
               %gep = getelementptr [200x[200xi32]],ptr @a,i32 0,i32 %idx1,i32 %idx2
{5,+,5}        %mul = mul i32 5, %i
{10,+,5}       %add1 = add i32 %mul, 5
{10+%p,+,5}    %add2 = add i32 %add1, %p
               store i32 %add2, ptr %gep
               br label %latch

latch
{2,+,1}        %inc = add i32 %i, 1
               br label %header
```

1. 寻找基本归纳变量： 查找header中的phi

`%i = phi i32 [1,%entry],[%inc,%latch]`

2. 识别间接归纳变量：反复遍历循环每条指令进行判断

`%idx1 = add i32 %k, %i`

3. 判断简单的 for 循环：

   (1) 只有一个exit

   (2) 只有一个exiting

   (3) exiting分支判断中有一个是归纳变量，另一个是循环不变量

循环强度削弱

```c
int a[200][200] = {};
int i = 1;
while(i <= n){
    a[k+i][3*i+4] = 5*i+5+p;
    i = i+1;
}
```

```
preheader
%n = load i32 ptr @n (invariant)
br label %header
```

SCEV:{1,+,1}

```
header
%i = phi i32 [1,%entry],[%inc,%latch]
%icmp = icmp sle i32 %i, %n
br i1 %icmp, label %body, label %exit
```

SCEV:{%k+1,+,1}

SCEV:{7,+,3}

```
body
%idx1 = add i32 %k, %i
%idx2mul = mul i32 3, %i
%idx2 = add i32 %idx2mul, 4
%gep = getelementptr [200x[200xi32]],ptr @a,i32 0,i32 %idx1,i32 %idx2
%mul = mul i32 5, %i
%add1 = add i32 %mul, 5
%add2 = add i32 %add1, %p
store i32 %add2, ptr %gep
br label %latch
```

```
latch
%inc = add i32 %i, 1
br label %header
```

⇨

```
preheader
%n = load i32 ptr @n (invariant)
%idx1_start = add i32 %k, 1
%gep_start = getelementptr [200x[200xi32]],ptr @a,i32 0,i32 %idx1_start,i32 7
br label %header
```

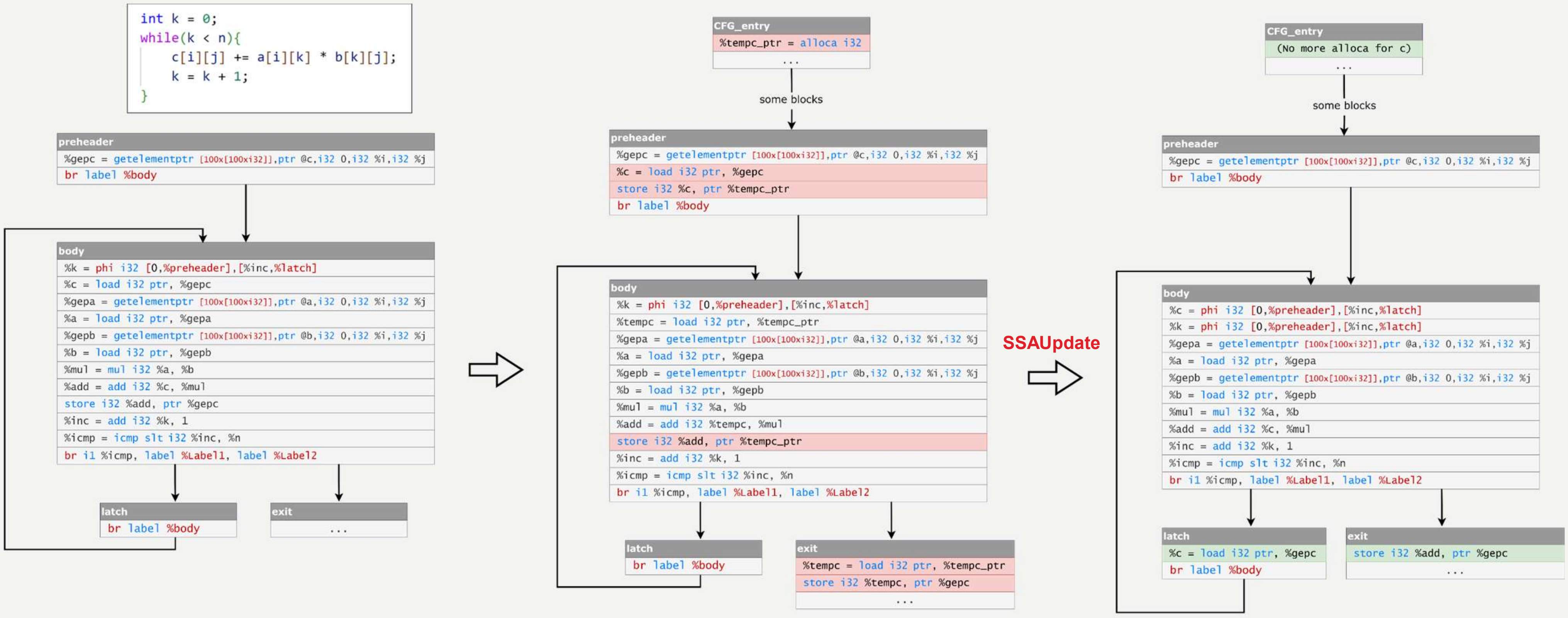
```
header
%i = phi i32 [1,%entry],[%inc,%latch]
%gep = phi i32 [%gep_start,%preheader],[%gep_next,%latch]
%icmp = icmp sle i32 %i, %n
br i1 %icmp, label %body, label %exit
```

```
body
%mul = mul i32 5, %i
%add1 = add i32 %mul, 5
%add2 = add i32 %add1, %p
store i32 %add2, ptr %gep
br label %latch
```

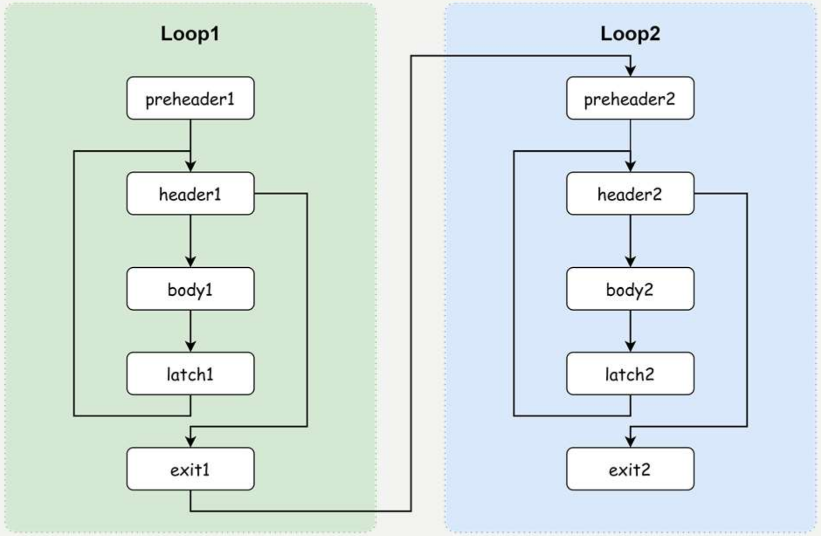
```
latch
%inc = add i32 %i, 1
%gep_next = getelementptr i32, ptr %gep, 203
br label %header
```

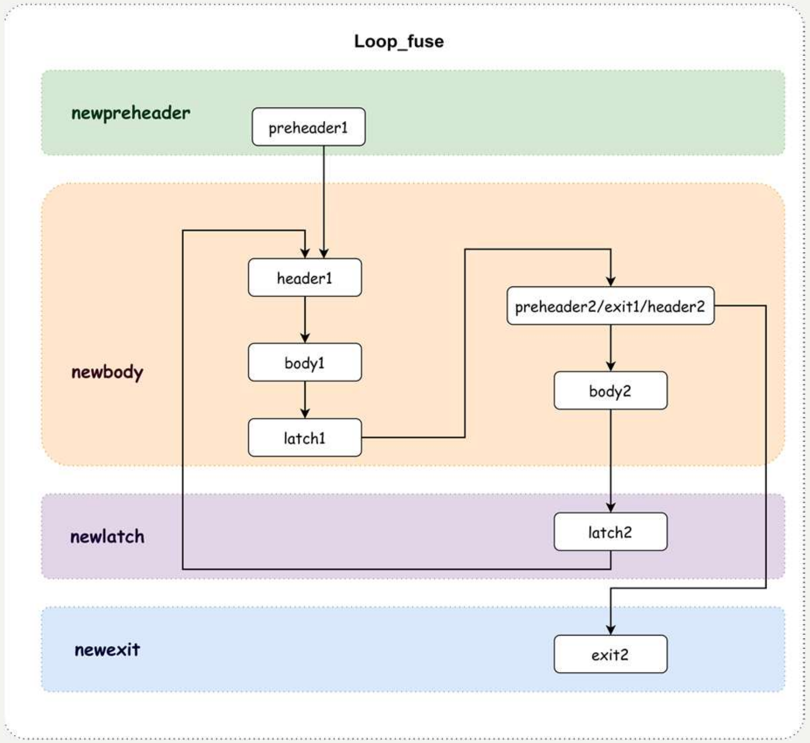
循环不变量外提

- ptr 是循环不变量
- 循环中的 store, load, call不影响ptr



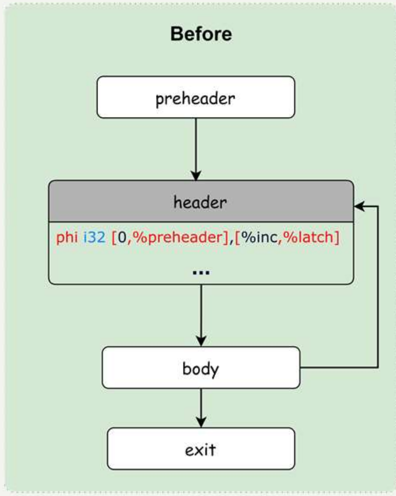
循环合并

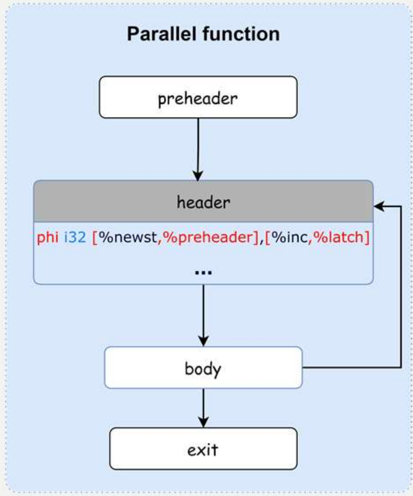




- 循环步长和上下界相同
- Loop1支配Loop2，Loop2后向支配Loop1
- 循环需要相邻或中间指令可外提至Loop1的preheader
- 合并后不存在后向数据依赖

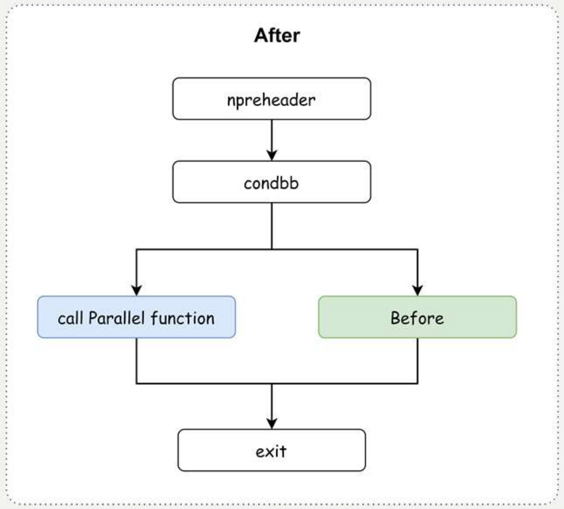
自动并行化






```
void ___parallel_loop_constant_100(void *fn, int st, int ed,int len1, int len2, ...)
```

- fn - 并行函数　　st - 循环起点　　ed - 循环终点
- len1 - 4字节参数数量　　len2 - 8字节参数数量
- 按字节读取可变参数：va_start, va_arg, va_end
- pthread创建四线程：pthread_create, pthread_join



其他优化

- 交换运算次序

将 (a+b)+c 转化为 a+(b+c)
当 (b+c) 已经计算或为循环不变量时可以优化

Before

```
int b = getint();
int c = getint();
int i = 1;
while(i <= n){
    int t1 = i + b;
    int t2 = t1 + c;
    putint(t2);
    i = i+1;
}
```

⇨

After

```
int b = getint();
int c = getint();
int i = 1;
int t1 = b + c;
while(i <= n){
    int t2 = t1 + i;
    putint(t2);
    i = i+1;
}
```

- 模2判断转与判断

将条件判断中的%2转化为&
有符号不能只与1

Before

```
if(n%2 == 1){
    ...
}
if(n%2 != 1){
    ...
}
if(n%2 == 0){
    ...
}
if(n%2 != 0){
    ...
}
```

⇨

After

```
if(n&-2147483647 == 1){
    ...
}
if(n&-2147483647 != 1){
    ...
}
if(n&1 == 0){
    ...
}
if(n&1 != 0){
    ...
}
```

其他优化

- Min-Max识别

  识别可以转化为Min-Max的判断语句
  变量和数组之间在实现上略有不同

Before

```
if(b > c){
    b = c;
}
if(a[i][j] >= a[i-1][j-1]){
    a[i][j] = a[i-1][j-1];
}
```

After

```
b=min(b,c);

a[i][j]=min(a[i][j],a[i-1][j-1]);
```

- 常量除法判断转乘法判断

  在条件判断中，将常量除法的大小判断
  转化为常量乘法的大小判断

Before

```
const int c1 = 15;
const int c2 = 25;
if(n / c1 > c2){
    ...
}
if(n / c1 >= c2){
    ...
}
if(n / c1 < c2){
    ...
}
if(n / c1 <= c2){
    ...
}
```

After

```
const int c1 = 15;
const int c2 = 25;
if(n > (c2+1)*c1-1){
    ...
}
if(n >= c2 * c1){
    ...
}
if(n < c2 * c1){
    ...
}
if(n <= (c2+1)*c1-1){
    ...
}
```

后端总体流程

- 指令选择
- Phi消除前代码优化
  - 乘除模常量优化
  - SSA窥孔优化
  - 公共子表达式消除
  - 循环不变量外提
- Phi消除
- 指令调度
- 寄存器分配
- 寄存器分配后的窥孔优化
- 插入ld/sd
- 软件分支预测

指令选择

- 基于窥孔的指令选择
  - 分两次pass进行，第一个pass翻译函数内IR
  - 第二个pass在开头插入实参

```
define void @fillline(ptr %r0,ptr %r1,i32 %r2)
{
L0:  ;
    %r5 = getelementptr i32, ptr %r1, i32 0
    store i32 1, ptr %r5
    %r9 = icmp eq i32 %r2,1
    br i1 %r9, label %L3, label %L1
L1:  ;
    %r14 = add i32 %r2,-1
    %r44 = getelementptr i32, ptr %r0, i32 0
    store i32 1, ptr %r44
    %r45 = icmp eq i32 %r14,1
    br i1 %r45, label %L7, label %L5
```

```
fillline:
.fillline_0:
    %2 = COPY a2, i64
    %1 = COPY a1, i64
    %0 = COPY a0, i64
    %4 = COPY %1, i64
    %6 = COPY 1, i64
    sw          %6,0(%4)
    %8 = COPY 1, i64 # Can't schedule
    beq         %2,%8,.fillline_3 # Can't schedule
.fillline_1:
    addiw       %9,%2,-1
    %11 = COPY %0, i64
    %13 = COPY 1, i64
    sw          %13,0(%11)
    %15 = COPY 1, i64 # Can't schedule
    beq         %9,%15,.fillline_7 # Can't schedule
```

SSA窥孔优化

- Phi消除前代码优化
  - 乘除模常量优化
  - SSA窥孔优化
  - 公共子表达式消除
  - 循环不变量外提

```
%2 = COPY 8, i64
mulw          %1,%0,%2
%4 = COPY 2, i64
divw          %3,%0,%4
addw          %5,%1,%3
%7 = COPY 3, i64
divw          %6,%0,%7
addw          %8,%5,%6
```

```
%2 = COPY 8, i64
slliw         %1,%0,3
%4 = COPY 2, i64
srliw         %37,%0,31
add           %38,%0,%37
sraiw         %3,%38,1
addw          %5,%1,%3
%7 = COPY 3, i64
%39 = COPY 1431655766, i64
mul           %40,%0,%39
srli          %41,%40,63
srli          %42,%40,32
add           %6,%42,%41
addw          %8,%5,%6
```

```
%2 = COPY 8, i64
slliw         %1,%0,3
%4 = COPY 2, i64
srliw         %37,%0,31
add           %38,%0,%37
sraiw         %3,%38,1
sh3add.uw        %5,%0,%3
%7 = COPY 3, i64
%39 = COPY 1431655766, i64
mul           %40,%0,%39
srli          %41,%40,63
srli          %42,%40,32
add           %6,%42,%41
addw          %8,%5,%6
```

指令调度

- 实例：减少RAW阻塞
- 只考虑块内的指令调度

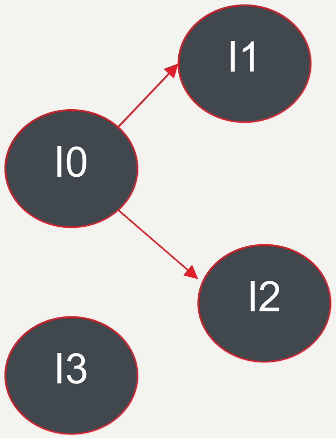
```
addw    %3,%1,%2
addw    %4,%0,%3
addw    %5,%4,%3
addw    %6,%0,%2
```

```
addw    %3,%1,%2
addw    %6,%0,%2
addw    %4,%0,%3
addw    %5,%4,%3
```

指令调度

- 算法：List scheduling
- ①建立数据优先图（有向无环）
- ②找到更好的拓扑序

```
I0  addw  %3,%1,%2
I1  addw  %4,%0,%3
I2  addw  %5,%4,%3
I3  addw  %6,%0,%2
```



- 模拟指令的发射与延迟
  - 模拟处理器状态，维护指令执行完成的倒计时；模拟双发射
  - 指令发射后，数据优先图不会立即删除相应结点，而是等指令倒计时结束才删除
- 尽可能先发射最长路径更长的指令（以指令延迟为路径长度）

- 如果发现发射某条指令后，寄存器压力过大以致溢出，会尝试发射其它指令，如果可发射的任意指令发射完都会溢出，仍选择最长路径更长的指令发射

寄存器分配

- 算法：线性扫描；活跃区间可以是不连续的多段
- 溢出权重：被指令引用的总次数/区间总长；优先溢出权重低的
- 循环外，每遇到一个被引用的操作数，引用次数+1
- 循环内，每遇到一个被引用的操作数，引用次数+$2^{min(5,loopdepth)} + loopdepth + 1$

- 溢出处理
- 在溢出的位置插入ld/sd，重新分配，迭代至不动点

- 区间合并（Coalesce）
- 采用激进的合并策略（只要不冲突并且有COPY语句连接，就合并）
- 分配时，如果有类似%0 = COPY a0的语句，会优先尝试给%0分配a0

活跃区间分段的例子

- 多个def会导致分段
- 在不连续的基本块内活跃也会导致分段

```
1 7 [11,12) [13,35) [42,43) Ref: 42
```

- 活跃区间分段可以大大减少实际上不存在的区间冲突

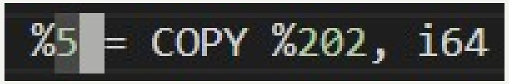
```
main:
.main_0:
    lui        %2,%hi(A)
    addi       %3,%2,%lo(A)
    addi       %1,%3,12
.main_7:
    %202 = COPY %1, i64
    addiw      %203,x0,0
    lui        %206,24 # Can't schedule
    addiw      %38,%206,1688 # Can't schedule
    %5 = COPY %202, i64 # Can't schedule
    %7 = COPY %203, i64 # Can't schedule
```

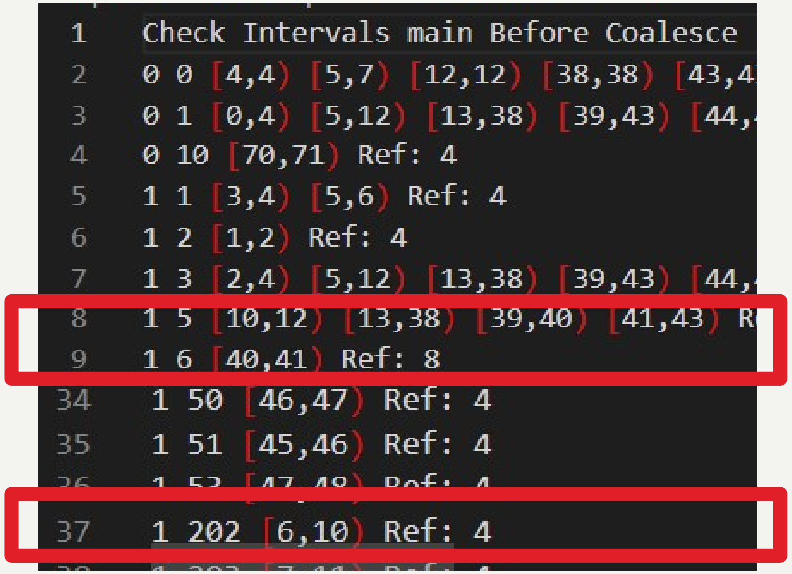
```
.main_5:
    addi       %6,%5,160
    %5 = COPY %6, i64 # Can't schedule
    %7 = COPY %8, i64 # Can't schedule
    jal        x0,.main_1 # Can't schedule
```

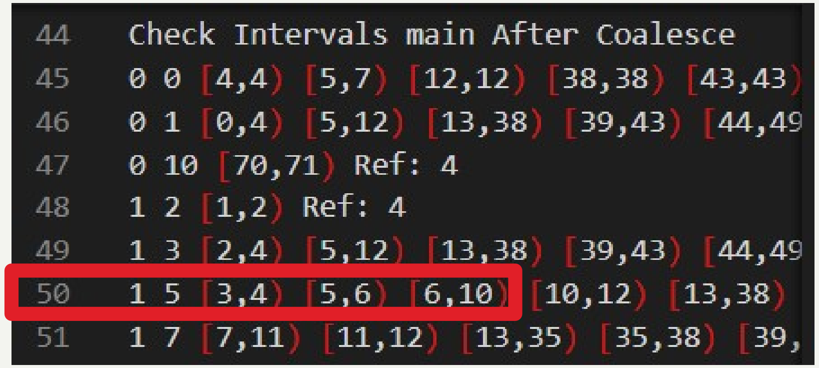
区间合并（Coalesce）的例子

- 激进的合并策略
- 其中一个例子：41个区间->30个区间

- 其中一个区间合并的实例







寄存器分配

- 避免WAW冲突
- 检查虚拟寄存器被def的语句的周围(同一个基本块内[-10,+latency])
- 优先分配指令附近没有被def的物理寄存器

```
t4 = 3
t1 = t2 + t3
%3 = t2 / t3 // %3 应当最后才尝试分配到t1或t5
// ...
t5 = ... // %3 指令latency范围以内
```

- 窥孔优化：删除冗余的读溢出语句

```
addi        t2,t0,16
# Write Spill
sd          t2,304(sp)
# store i32 %r314, ptr %r318
# Read Spill
ld          t2,304(sp)
sw          t3,0(t2)
```

寄存器分配之后的优化

- 插入s寄存器的ld/sd
- 有时，会存在部分控制流，这些控制流不经过部分s寄存器的def。这时，我们不需要将sd语句插入一开头的位置，ld语句也不需要插入结尾的位置，这样可以减少sd/ld的执行次数
- 对于一个s寄存器，找到所有出现的块，在前向/后向支配树求取LCA
- 先在后向支配树的LCA上插入ld，后在前向支配树的LCA上插入sd

- 软件分支预测
- 为了让指令cache能更好地加速取指，高概率进入的分支应当是false分支，否则true和false分支要反转
- 我们认为，凡是进入循环的分支，都是高概率的分支。如果进入循环的分支是true分支，则分支要反转，否则分支不反转。
- 因为while转do-while和循环展开都会生成大量的大概率是true分支的代码，所以我们认为检测不到循环的情况下仍然是进入true分支的几率大