

五、目标代码生成 (16. 指令选择)

魏恒峰

hfwei@nju.edu.cn

2024 年 06 月 12 日



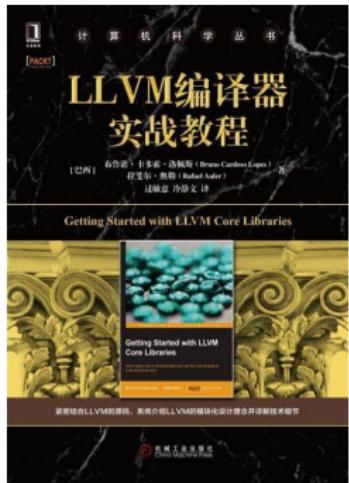
Instruction Selection

Instruction Selection is the process of translating LLVM code presented to the code generator into target-specific machine instructions. There are several well-known ways to do this in the literature. LLVM uses a SelectionDAG based instruction selector.

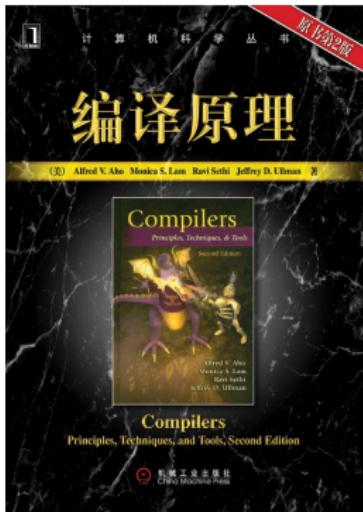
这实际上是一个组合优化问题

“略有耳闻”

遗憾的是，没有“合适的”教材



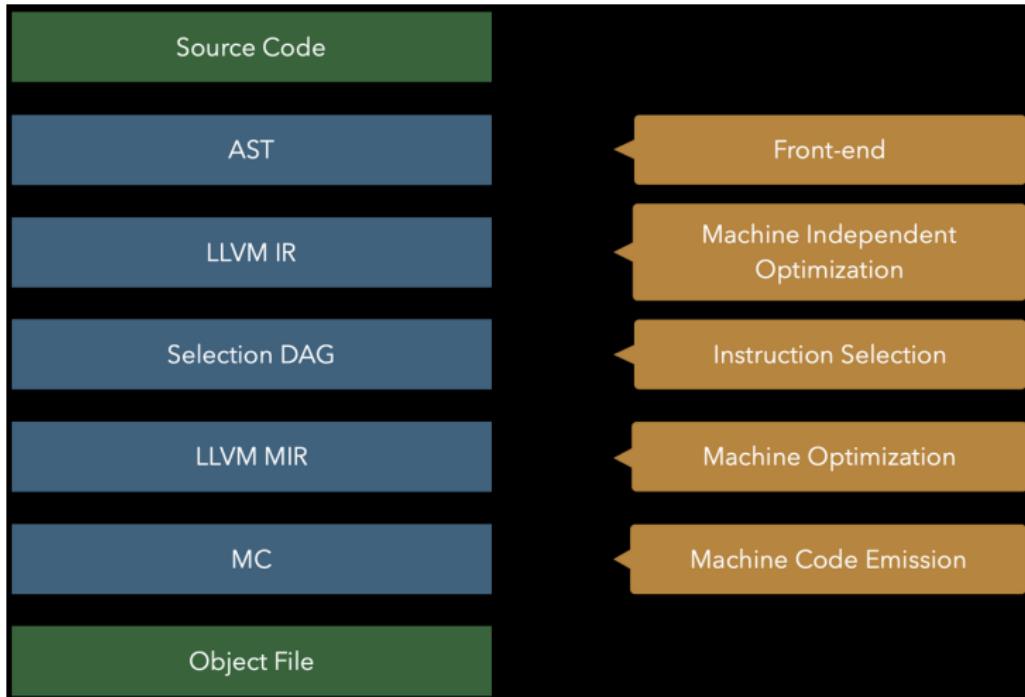
[https://llvm.org/docs/
CodeGenerator.html](https://llvm.org/docs/CodeGenerator.html)



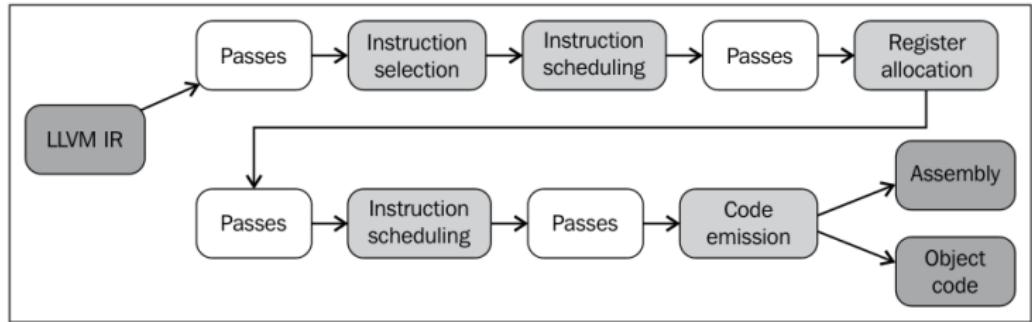
树翻译方案



最佳覆盖与最优覆盖算法



(in-memory) LLVM IR SelectionDAG MachineInstr MCInst



Where is “Prologue/Epilogue Insertion”?

f0-O0 @ Compiler Explorer

The screenshot shows the Compiler Explorer interface with several windows open:

- Left Panel:** Shows the C source code for a factorial function.
- Middle Panel:** Shows the LLVM IR for the same function.
- Right Panels:** Show the Opt Pipeline Viewer and the generated assembly code.

The assembly code window displays the following assembly instructions:

```
1 int factorial()
2 {
3     int main()
4     {
5         return
6     }
7
8     main:
9         addi    $a0, $zero, 1
10    sw      $a0, -4($sp)
11    addi    $a1, $zero, 0
12    sw      $a1, -8($sp)
13    slli   $a1, $a1, 2
14    sub    $a1, $a1, 1
15    addi   $a1, $a1, 1
16    seqz   $a1, $a1
17    lw      $a1, 4($sp)
18    lw      $a1, 8($sp)
19    addi   $a1, $a1, 1
20    ret
```

The assembly code is annotated with various compiler passes and optimizations:

- Passes: main
- RISC-V DAG->DAG Pattern Selection (riscv-isel)
- Eliminate PHI nodes for register allocation (phi-node-elimination)
- Two-Address instruction pass (twoaddressinstruction)
- Fast Register Allocator (regallocfast)
- Slot index numbering (slotindexes)
- Live Interval Analysis (liveintervals)
- Fast Register Allocator (regallocfast)
- Prologue/Epilogue Insertion & Frame Finalization (prologepilog)
- Post-RA pseudo instruction expansion pass (postrapseudos)

The assembly code itself contains many annotations:

- Annotations for `$x1`, `$x2`, `$x8`, `$x10`, `$x11`, `$x12`, `$x13`, `$x14`, `$x15`, `$x16`, `$x17`, `$x18`, `$x19`, `$x20`, `$x21`, `$x22`, `$x23`, `$x24`, `$x25`, `$x26`, `$x27`, `$x28`, `$x29`, `$x30`, `$x31`, `$x32`, `$x33`, `$x34`, `$x35`.
- Annotations for `(load $s32)`, `(frame-destroy ADDI $x2, 3)`, `(PseudoRET implicit killed $x10);`.
- Annotations for `28 ::`, `24 ::`, `32 ::`, `3 ::`.

f0-O1 @ Compiler Explorer

The screenshot shows the Compiler Explorer interface with two main panes. The left pane displays the LLVM IR Viewer for RISC-V code, and the right pane displays the Opt Pipeline Viewer.

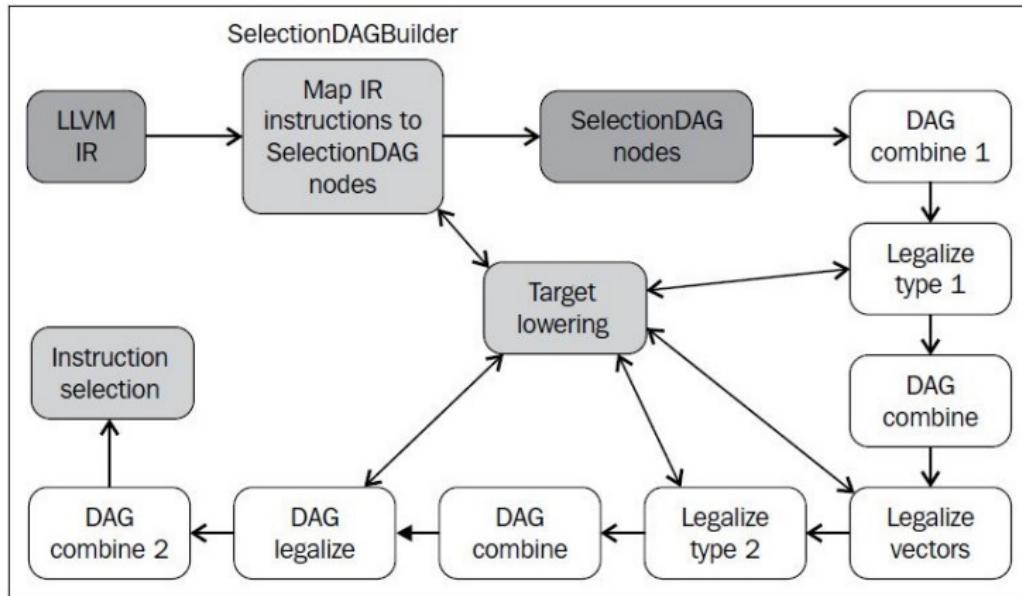
LLVM IR Viewer (RISC-V rv32gc clang (trunk)):

```
define dso_local range(i32 0, 2)
entry:
%call = tail call i32 @_factorial(%call)
%cmp = icmp eq i32 %call, 6
%conv = zext i1 %cmp to i32
ret i32 %conv
}
# Machine code for function main:
bb.0.entry:
ADDCALLSTACKDOWN 0, 0, implicit
%2:gpr = ADDI $x0, 2
$x10 = COPY %2:gpr; example.c:4
PseudoCALL target-flags(riscv-c)
ADDCALLSTACKUP 0, 0, implicit-c
%3:gpr = COPY $x10; example.c:4
%4:gpr = ADDI %3:gpr, -6; example.c:4
%5:gpr = SLTIU killed %4:gpr, 1
$x10 = COPY %5:gpr; example.c:4
PseudoRET implicit $x10; example.c:4
# End machine code for function main:
```

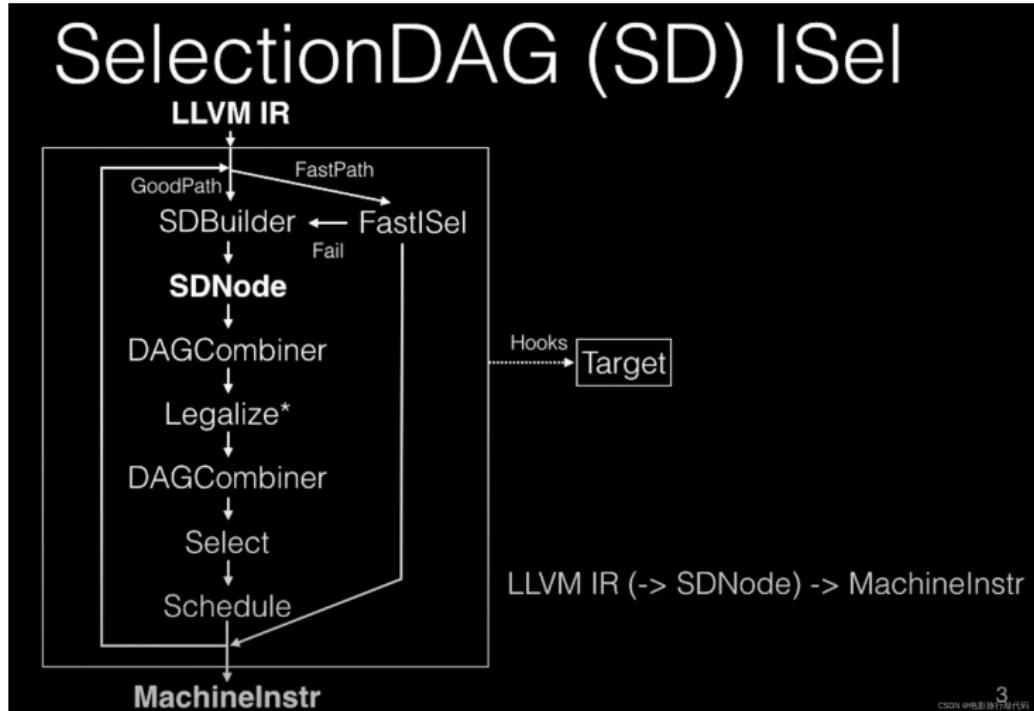
Opt Pipeline Viewer (RISC-V rv32gc clang (trunk)):

The pipeline viewer shows the execution flow through various compiler passes:

- AssignmentTrackingPass on [module]
- SROAPass on main
- IPSCCPPass on [module]
- GlobalOptPass on [module]
- InstCombinePass on main
- PostOrderFunctionAttrsPass on (main)
- TailCallElimPass on main
- RISC-V DAG->DAG Pattern
- Instruction Selection (riscv-isel)
- Slot index numbering (slotindexes)
- Merge disjoint stack slots (stack-coloring)
- Machine code sinking (machine-sink)
- Live Variable Analysis (livevars)
- Eliminate PHI nodes for register allocation (phi-node-elimination)
- Two-Address instruction pass (twoaddressinstruction)
- Slot index numbering (slotindexes)
- Live Interval Analysis



SDISel FastISel (per basic block) GlobalISel (per function)



GlobalISel (per function)

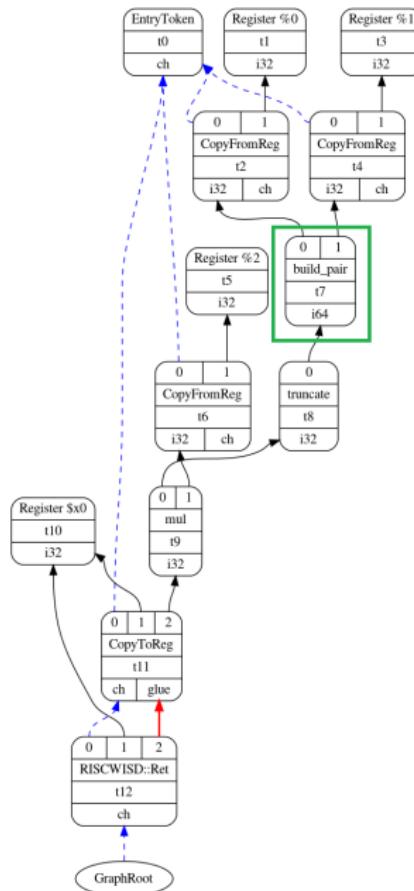
```
unsigned int MUL(unsigned long long int x, unsigned int y)
{
    return x * y;
}
```

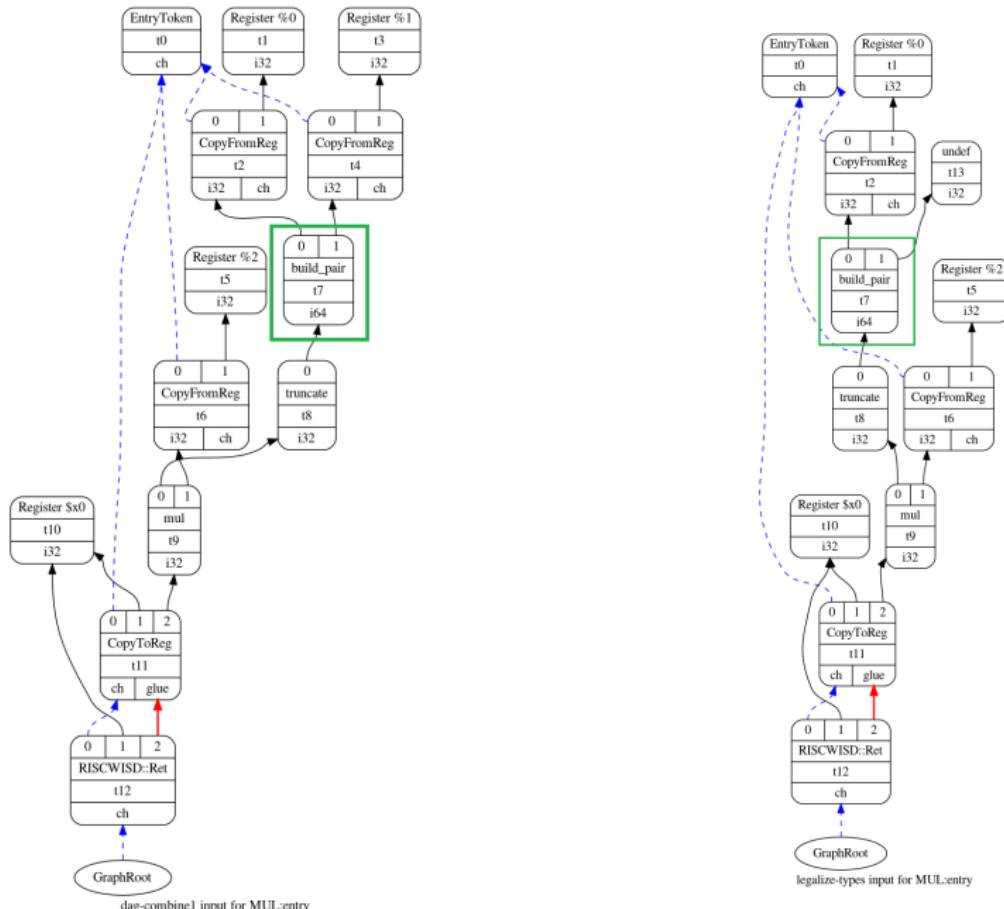
```
define dso_local i32 @MUL(i64 %x, i32 %y) local_unnamed_addr #0 {
entry:
    %0 = trunc i64 %x to i32
    %conv1 = mul i32 %0, %y
    ret i32 %conv1
}
```

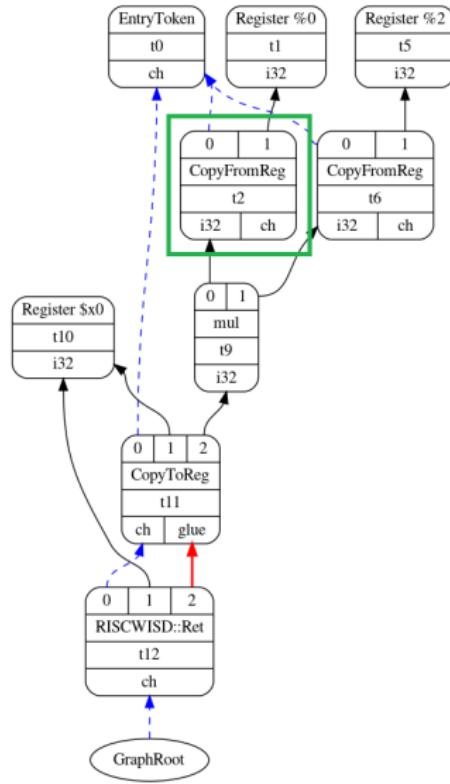
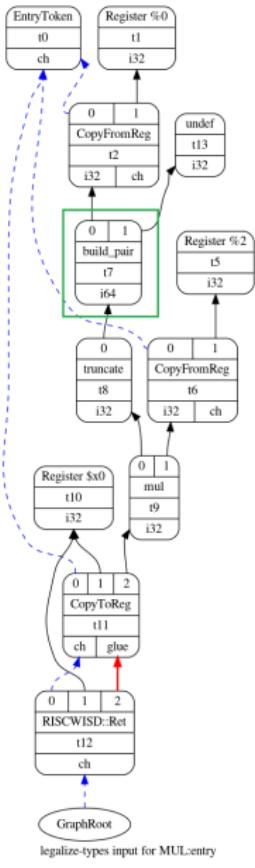
```

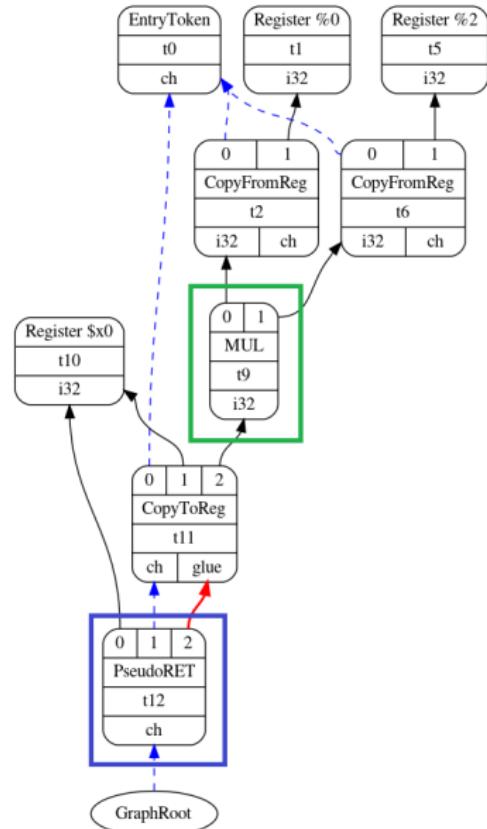
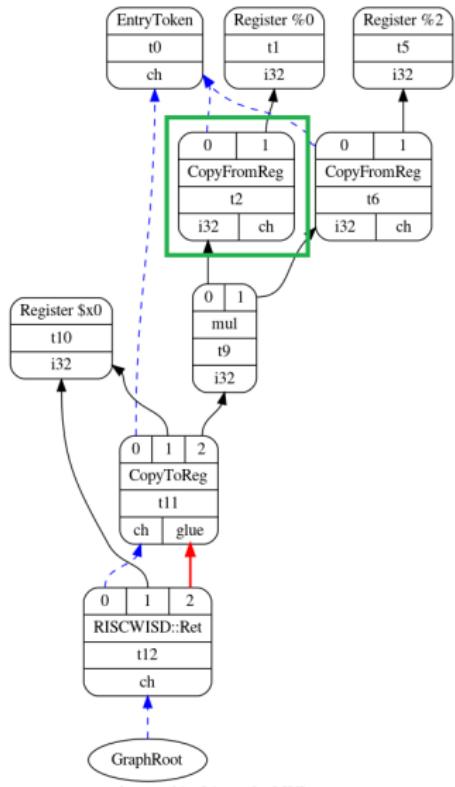
define dso_local i32 @MUL(i64 %x, i32 %y) local_unnamed_addr #0 {
entry:
    %0 = trunc i64 %x to i32
    %conv1 = mul i32 %0, %y
    ret i32 %conv1
}

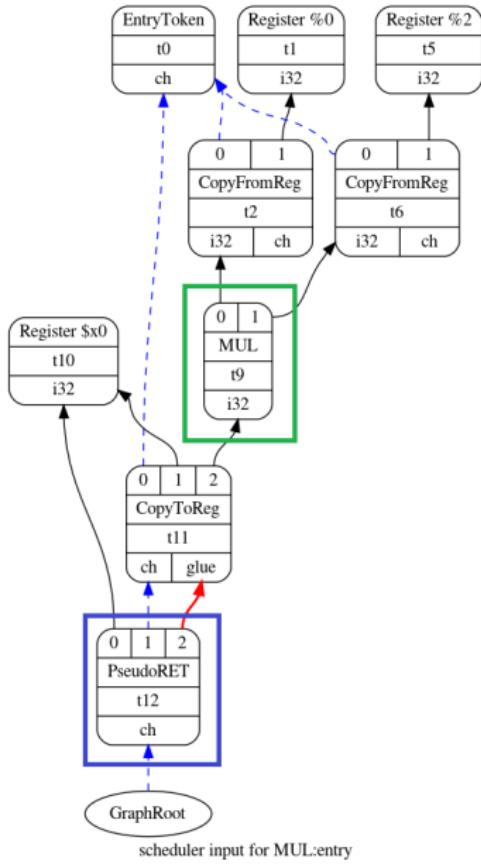
```











bb.0.entry:

liveins: \$x0, \$x2

%2:gpr = COPY \$x2

%0:gpr = COPY \$x0

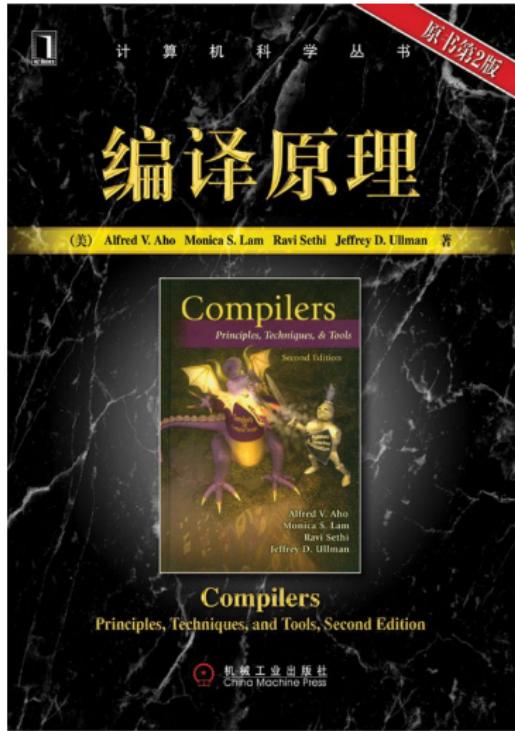
%3:gpr = MUL %0:gpr, %2:gpr

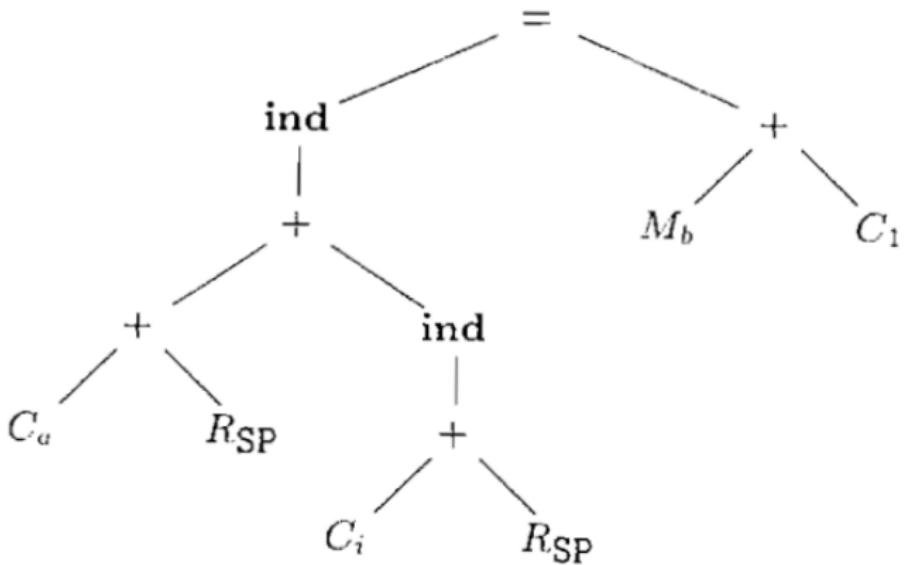
\$x0 = COPY %3:gpr

PseudoRET implicit \$x0

SelectionDAG Select Phase

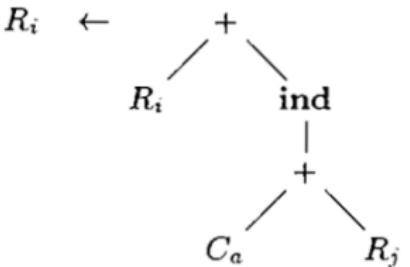
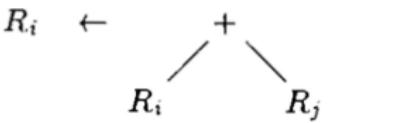
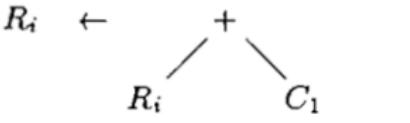
The Select phase is the bulk of the target-specific code for instruction selection. This phase takes a legal SelectionDAG as input, pattern matches the instructions supported by the target to this DAG, and produces a new DAG of target code. For example, consider the following LLVM fragment:

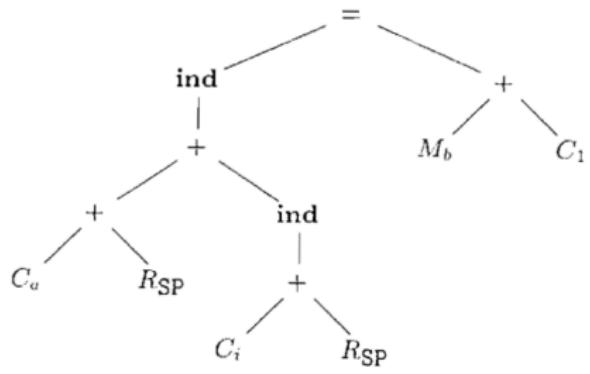




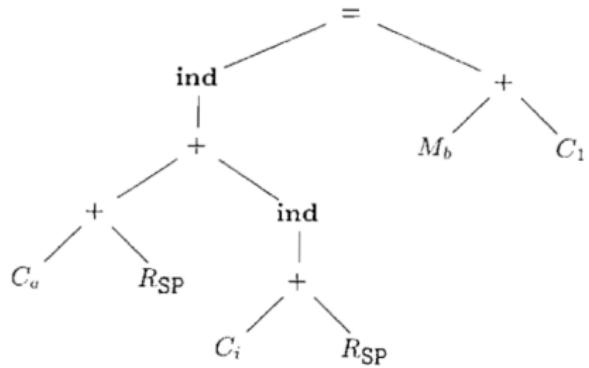
局部变量 a 与 i 的地址: 相对于 SP (栈指针) 的常数偏移量 C_a 和 C_i

1)	$R_i \leftarrow C_a$	{ LD Ri, #a }
2)	$R_i \leftarrow M_x$	{ LD Ri, x }
3)	$M \leftarrow$ $\begin{array}{c} = \\ / \quad \backslash \\ M_x \quad R_i \end{array}$	{ ST x, Ri }
4)	$M \leftarrow$ $\begin{array}{c} = \\ / \quad \backslash \\ \text{ind} \quad R_j \\ \\ R_i \end{array}$	{ ST *Ri, Rj }
5)	$R_i \leftarrow$ $\begin{array}{c} \text{ind} \\ \\ + \\ C_a \quad R_j \end{array}$	{ LD Ri, a(Rj) }

6)	$R_i \leftarrow$ 	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \leftarrow$ 	{ ADD Ri, Ri, Rj }
8)	$R_i \leftarrow$ 	{ INC Ri }



LD R0, #a
 ADD R0, R0, SP
 ADD R0, R0, i(SP)
 LD R1, b
 INC R1
 ST *R0, R1



$= \text{ind} + + C_a R_{SP} \text{ ind} + C_i R_{SP} + M_b C_1$

1)	$R_i \rightarrow c_a$	{ LD Ri, #a }
2)	$R_i \rightarrow M_x$	{ LD Ri, x }
3)	$M \rightarrow = M_x R_i$	{ ST x, Ri }
4)	$M \rightarrow = \text{ind} R_i R_j$	{ ST *Ri, Rj }
5)	$R_i \rightarrow \text{ind} + c_a R_j$	{ LD Ri, a(Rj) }
6)	$R_i \rightarrow + R_i \text{ ind} + c_a R_j$	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \rightarrow + R_i R_j$	{ ADD Ri, Ri, Rj }
8)	$R_i \rightarrow + R_i c_1$	{ INC Ri }
9)	$R \rightarrow \text{sp}$	
10)	$M \rightarrow m$	

$= \text{ind} + + c_a \text{ sp ind} + c_i \text{ sp} + m_b c_1$

前缀表示

二义性: 偏向于执行较大的归约, 而不是较小的归约

归约/归约冲突 移入/归约冲突

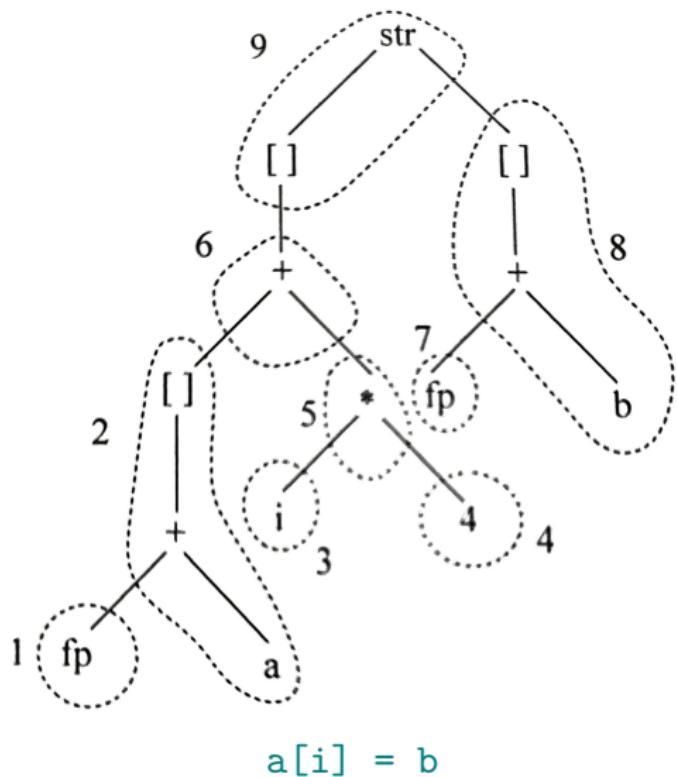
优先选择较长的归约 优先选择移入动作



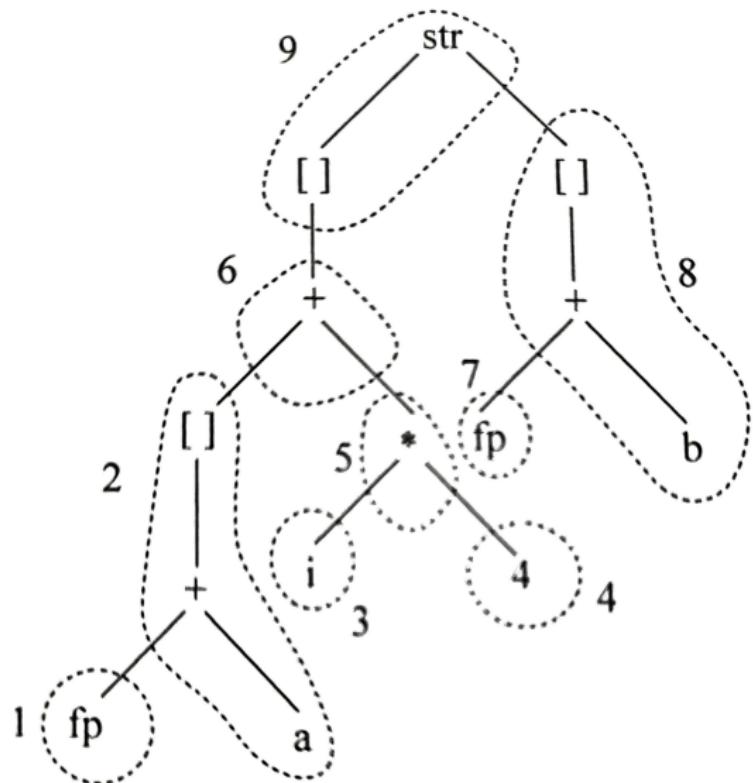
算术与存储指令及树型 (鲲鹏处理器)

指 令	树 型
r_i	x
add r_i, r_j, r_k	$+ \swarrow \searrow$
mul r_i, r_j, r_k	$* \swarrow \searrow$
sub r_i, r_j, r_k	$- \swarrow \searrow$
div r_i, r_j, r_k	$/ \swarrow \searrow$
addi r_i, r_j, c	$+ \swarrow \quad + \swarrow \quad c$ $c \quad c \quad $
subi r_i, r_j, c	$- \swarrow \quad - \swarrow \quad c$ $c \quad c \quad $
ldr $r_i, [r_j, c]$	$[] \quad [] \quad [] \quad []$ $+ \swarrow \quad + \swarrow \quad \quad $ $c \quad c \quad c \quad $
str $[r_j, c], r_i$	$str \quad str \quad str \quad str$ $[] \quad [] \quad [] \quad []$ $+ \swarrow \quad + \swarrow \quad \quad $ $c \quad c \quad c \quad $

瓦片覆盖 (tiling)



```
ldr r1, [fp, a]
addi r2, r0, 4
mul r2, ri, r2
add r1, r1, r2
ldr r2, [fp, b]
str [r1, 0], r2
```



$a[i] = b$

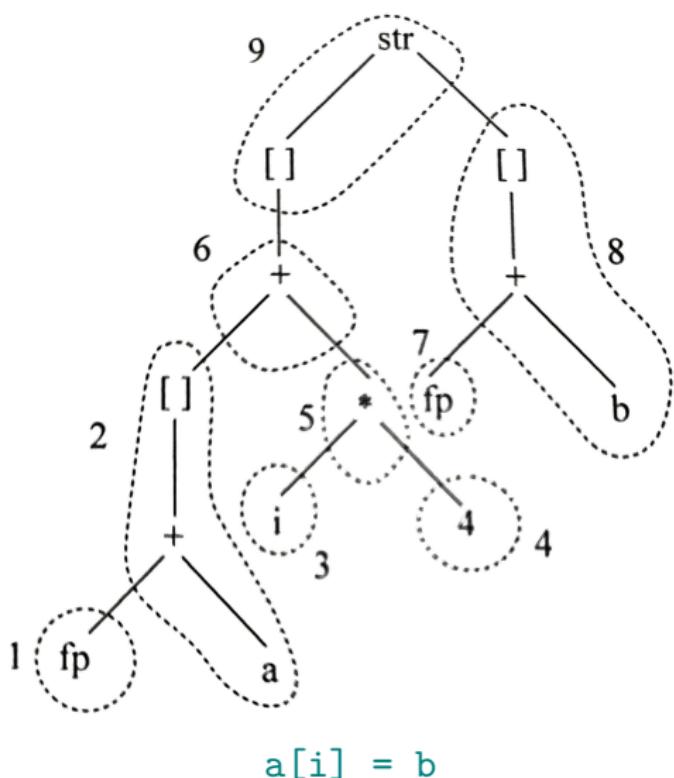
```

addi r1, r0, a
add r1, fp, r1
ldr r1, [r1, 0]
addi r2, r0, 4
mul r2, r1, r2
add r1, r1, r2
addi r2, r0, b
add r2, fp, r2
ldr r2, [r2, 0]
str [r1, 0], r2

```

小树型覆盖方案

瓦片覆盖 (tiling)



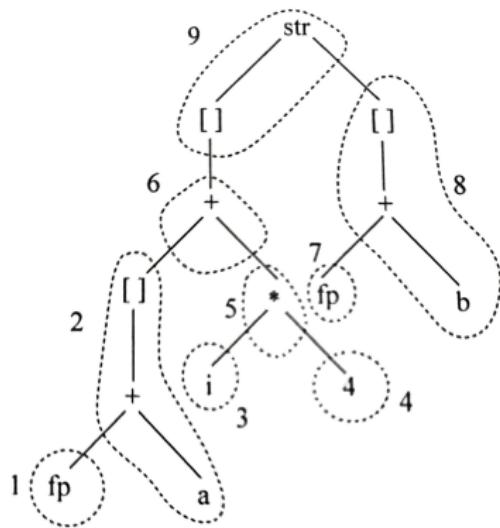
最大吞进算法
(maximal munch)

```
ldr r1, [fp, a]
addi r2, r0, 4
mul r2, ri, r2
add r1, r1, r2
ldr r2, [fp, b]
str [r1, 0], r2
```

最佳覆盖方案

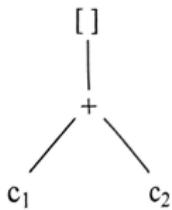
(不存在可以合并成更小代价树型的相邻树型)

使用动态规划思想计算最优覆盖方案



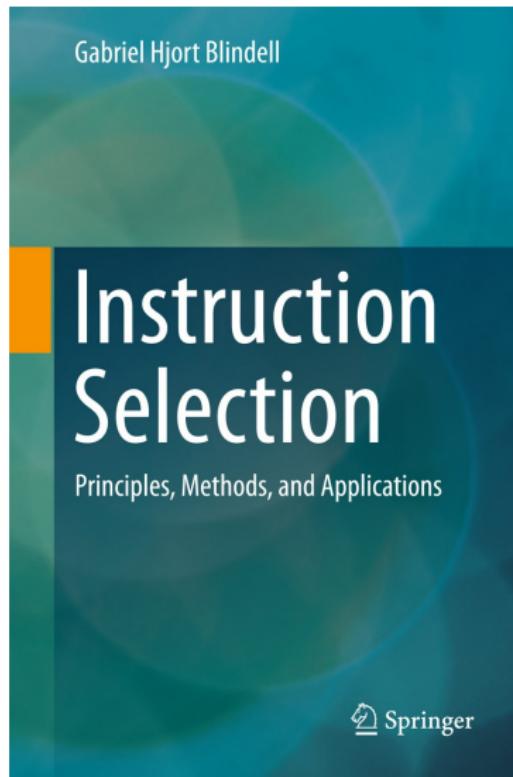
$$\forall n \in Node. c(n) = \min_{t \in tiles(n)} \left(c(t) + \sum_{n_i \in descendants(n)} c(n_i) \right)$$

$$\text{Opt} = c(r) \quad (r \text{ is the root})$$



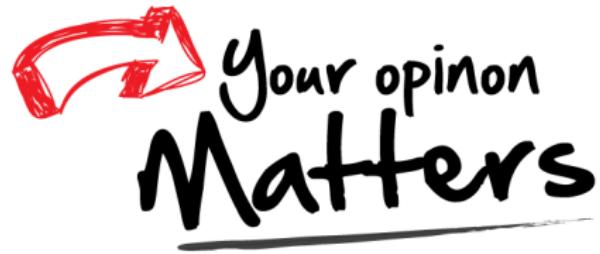
瓦片	指令	瓦片代价	叶子节点代价	总代价
	add	1	$1+1$	3
	addi	1	1	2
	addi	1	1	2

瓦片	指令	瓦片代价	叶子节点代价	总代价
	ldr	1	2	3
	ldr	1	1	2
	ldr	1	1	2



Tree Covering, DAG Covering, Graph Covering

Thank You!



Office 926

hfwei@nju.edu.cn