中山大學 SUN YAT-SEN UNIVERSITY 计算机学院（软件学院） SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle
编 译 原 理

第21讲：中间代码(3)

张献伟

xianweiz.github.io

DCS290, 5/16/2023

中山大學 SUN YAT-SEN UNIVERSITY

Review Questions

- Explain: br i1 %9, label %2, label %10.

  Check the value of %9, if T jump to %2 block, otherwise to %10.

- What is SSA?

  Single Static Assignment.

  Give variable different version name on every assignment.

- Benefits of SSA?

  Make dataflow explicit, facilitating IR optimizations.

- Explain LLVM Phi: %5 = phi i32 [%7, %4], [%1, %2].

  Value of %5 either from block %4 (reg: %7) or %2 (reg: %1).

- Is it possible to generate IR during syntax analysis?

  YES. Syntax-directed translation.

Array References

- Type(a) = array(10, int)
  - c = a[i];

addr(a[i]) = base + i*4

t_1 = i * 4
t_2 = a[t_1]
c = t_2

addr(a[i_1][i_2]) = base + i_1*20 + i_2*4

- Type(a) = array(3, array(5, int))
  - c = a[i_1][i_2];

t_1 = i_1 * 20
t_2 = i_2 * 4
t_3 = t_1 + t_2
t_4 = a[t_3]
c = t_4

- Type(a) = array(3, array(5, array(8, int)))
  - c = a[i_1][i_2][i_3]

addr(a[i_1][i_2][i_3]) = base + i_1*w_1 + i_2*w_2 + i_3*w_3
= base + i_1*160 + i_2*32 + i_3*4

Example: LLVM

```c
double x;
int arr[3][5][8];

void foo() {
    char a;
    int b = 0;
    long long c;
    int d;

    int x = arr[2][3][4];
}
```

```llvm
@arr = dso_local global [3 x [5 x [8 x i32]]] zeroinitializer, align 4
@x = dso_local global double 0.000000e+00, align 8

; Function Attrs: noinline nounwind optnone
define dso_local void @foo() #0 {
  %1 = alloca i8, align 1
  %2 = alloca i32, align 4
  %3 = alloca i64, align 8
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 0, i32* %2, align 4
  %6 = load i32, i32* getelementptr inbounds ([3 x [5 x [8 x i32]]], [3 x [5 x [8 x i32]]]* @arr, i64 0, i64 2, i64 3, i64 4), align 4
  store i32 %6, i32* %5, align 4
  ret void
}
```

// addr(@arr + 4x(0 + 2*3*4 + 3*4 + 4))

Builder.CreateInBoundsGEP(addr, …);

Translation of Array References

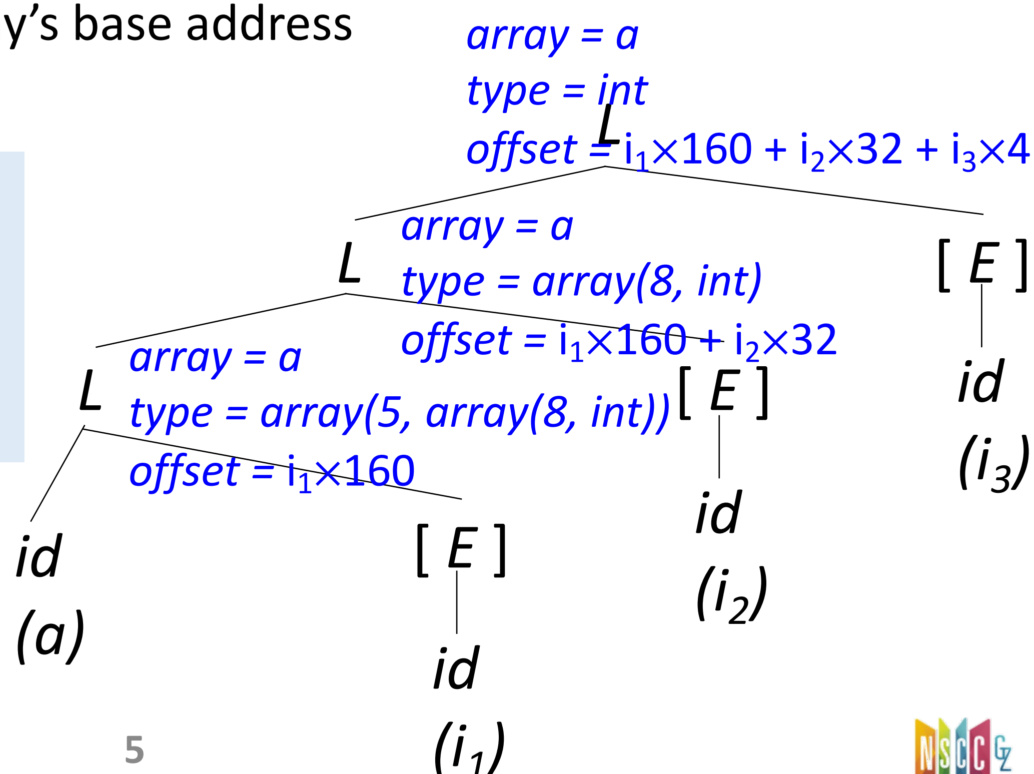
- $A[i_1][i_2][i_3]$, type(a) = array(3, array(5, array(8, int)))
  - *L.type*: the type of the subarray generated by *L*
  - *L.addr*: a temporary that is used while computing the offset for the array reference by summing the terms $i_j \times w_j$
  - *L.array*: a pointer to the symbol-table entry for the array name
    - *L.array.base* gives the array's base address

① S -> id = E; | $L = E$;

② E -> $E_1 + E_2$ | - E_1 | (E_1) | id | L

③ L -> id [E] | L_1 [E]

$base + i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$



Translation of Array References (cont.)

- $A[i_1][i_2][i_3]$, type(a) = array(3, array(5, array(8, int)))

① *S* -> id = *E*; | *L* = *E*; { *gen(L.array.base'['L.addr']' '=' E.addr);* }
② *E* -> $E_1 + E_2$ | - E_1 | (E_1) | id | *L* { *E.addr = newtemp();*
gen(E.addr '=' L.array.base'['L.addr']'); }
③ *L* -> id [*E*] { *L.array = lookup(id.lexeme); if !L.array then error;*
L.type = L.array.type.elem;
L.offset = newtemp();
gen(L.addr '=' E.addr '' L.type.width); }*
| L_1 [*E*] { *L.array =* L_1*.array;*
L.type = L_1*.type.elem;*
t = newtemp();
gen(t '=' E.addr '' L.type.width);*
L.addr = newtemp();
gen(L.addr '=' L_1.addr '+' t; }

$t_1 = i_1 * 160$
$t_2 = i_2 * 32$
$t_3 = t_1 + t_2$
$t_4 = i_3 * 4$
$t_5 = t_3 + t_4$
$c = a[t_5]$

中山大學 SUN YAT-SEN UNIVERSITY

CodeGen: Boolean Expressions

- Boolean expression: *a op b*
  - where op can be <, <=, = !=, > or >=, &&, ||, …
- **Short-circuit** evaluation[短路计算]: to skip evaluation of the rest of a boolean expression once a boolean value is known
  - Given following C code: *if (flag || foo()) { bar(); };*
    - If *flag* is true, *foo()* never executes
    - Equivalent to: *if (flag) { bar(); } else if (foo()) { bar(); };*
  - Given following C code: *if (flag && foo()) { bar(); };*
    - If *flag* is false, *foo()* never executes
    - Equivalent to: *if (!flag) { } else if (foo()) { bar(); };*
  - Used to alter control flow, or compute logical values
    - Examples: *if (x < 5) x = 1*; *x = true*; *x = a < b*
    - For control flow, boolean operators translate to ***jump*** statements

Example: LLVM

```c
double x;

void foo() {
    char a;
    int b = 0;
    long long c;
    int d;

    if (b < 5) b = 1;
    b = d < b;
}
```

```llvm
@x = dso_local global double 0.000000e+00, align 8

; Function Attrs: noinline nounwind optnone
define dso_local void @foo() #0 {
  %1 = alloca i8, align 1
  %2 = alloca i32, align 4
  %3 = alloca i64, align 8
  %4 = alloca i32, align 4
  store i32 0, i32* %2, align 4
  %5 = load i32, i32* %2, align 4
  %6 = icmp slt i32 %5, 5               // %6 = (b < 5)
  br i1 %6, label %7, label %8          // true: '7', false: '8'

7:                                                ; preds = %0
  store i32 1, i32* %2, align 4         // b = 1
  br label %8                           // jump to '8'

8:                                                ; preds = %7, %0
  %9 = load i32, i32* %4, align 4       // %9 = d
  %10 = load i32, i32* %2, align 4      // %10 = b
  %11 = icmp slt i32 %9, %10            // %11 = d < b
  %12 = zext i1 %11 to i32              // %12 = %11
  store i32 %12, i32* %2, align 4       // b = %12
  ret void
}
```

llvm::BasicBlock::Create(…);

Builder.CreateCondBr(…); // Create a conditional 'br Cond, TrueDest, FalseDest' instruction.

Builder.SetInsertPoint(…);

Boolean Exprs (w/o Short-Circuiting)

- Computed just like any other arithmetic expression

E -> (a < b) or (c < d and e < f)

t_1 = a < b
t_2 = c < d
t_3 = e < f
t_4 = t_2 && t_3
t_5 = t_1 || t_4

- Then, used in control-flow statements
  - *S.next*: label for code generated after *S*

S -> if E S_1

// t_5=F, skip S_1
if (!t_5) goto *S.next*
S_1.code
S.next: ...

Boolean Exprs (w/ Short-Circuiting)

- Implemented via a series of jumps[利用跳转]
  - Each relational op converted to two gotos (*true* and *false*)
  - Remaining evaluation skipped when result known in middle
- Example
  - *E.true*: label for code to execute when *E* is *'true'*
  - *E.false*: label for code to execute when *E* is *'false'*
  - E.g. if above is condition for a *while* loop
    - *E.true* would be label at beginning of loop body
    - *E.false* would be label for code after the loop

```
while (E) {
  // E.true
}
// E.false
...
```

E -> (a < b) or (c < d and e < f)

if (a < b) goto *E.true*	E为真：只要a < b真
goto L_1	a < b假：继续评估
L_1: if (c < d) goto L_2	a < b假、 c < d真：继续评估
goto *E.false*	E为假：a < b假，c < d假
L_2: if (e < f) goto *E.true*	E为真：a < b假，c < d真，e < f真
goto *E.false*	E为假：a < b假，c < d真，e < f假

SDT Translation of Booleans[布尔表达式]

- B -> B_1 || B_2
  - B_1.*true* is same as B.*true*, B_2 must be evaluated if B_1 is false[B_1假才评估B_2]
  - The true and false exits of B_2 are the same as B[B_2与B同真假]
- B -> E_1 *relop* E_2
  - Translated directly into a comparison TAC inst with jumps

B_1为真，跳转到B.true　　　　B_1为假，跳转到别处（需要继续评估B_2）

① *B* -> { *B_1.true = B.true; B_1.false = newlabel();* } B_1
　|| { *label(B_1.false); B_2.true = B.true; B_2.false = B.false;* } B_2

② *B* -> { *B_1.true = newlabel(); B_1.false = B.false;* } B_1
　&& { *label(B_1.true); B_2.true = B.true; B_2.false = B.false;* } B_2

③ *B* -> E_1 *relop* E_2 { *gen('if' E_1.addr relop E_2.addr 'goto' B.true);*
　gen('goto' B.false); }

④ *B* -> ! { *B_1.true = B.false; B_1.false = B.true;* } B_1

⑤ *B* -> true { *gen('goto' B.true);* }

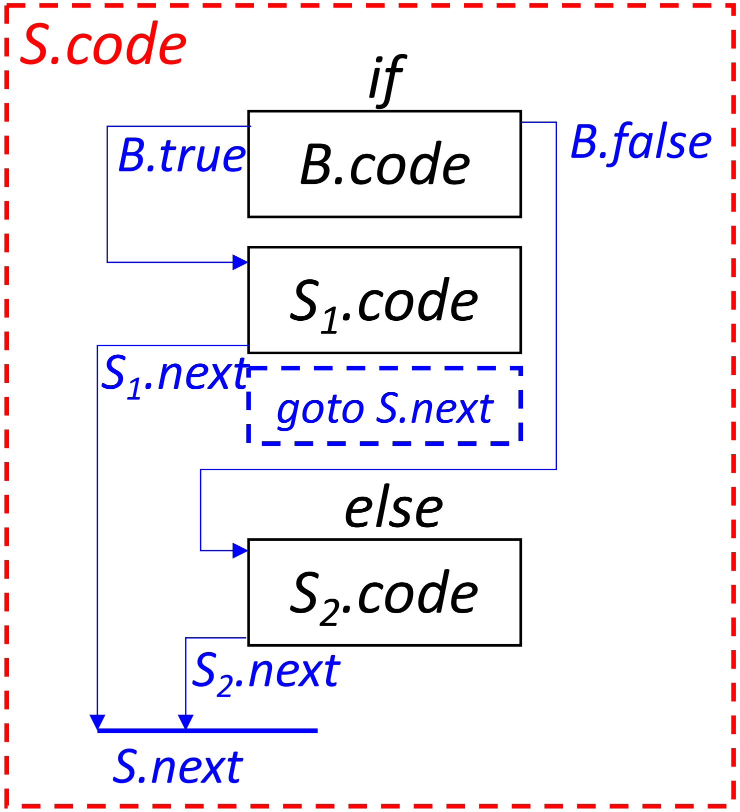
⑥ *B* -> false { *gen('goto' B.false);* }

B: a boolean expression
S: a statement

Dragon Book, Chapter 6.6.1

CodeGen: Control Statement[控制语句]

① S -> if (B) S_1
② S -> if (B) S_1 else S_2
③ S -> while (B) S_1



- Inherited attributes[继承属性]
  - *B.true*: the label to which control flows if *B* is true(依赖于S_1)
  - *B.false*: the label to which control flows if *B* is false(依赖于S_2)
  - *S.next*: a label for the instruction immediately after the code of *S*

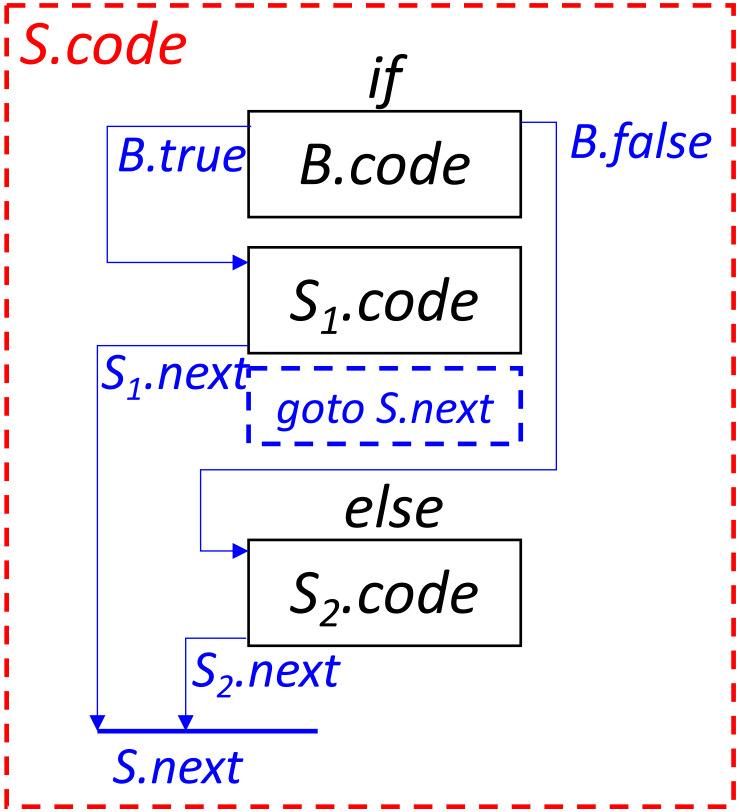
Translation of Controls

① S -> if (B) S_1

② S -> if (B) S_1 else S_2

③ S -> while (B) S_1

S -> *if* { *B.true = newlabel();*
B.false = newlabel(); }
(*B*) { *label(B.true); S_1.next = S.next;* }
S_1 { *gen('goto' S.next);* }
else { *label(B.false); S_2.next = S.next;* } S_2



- Helper functions[辅助函数]
  - *newlabel()*: creates a new label
  - *label(L)*: attaches label *L* to the next three-address inst to be generated

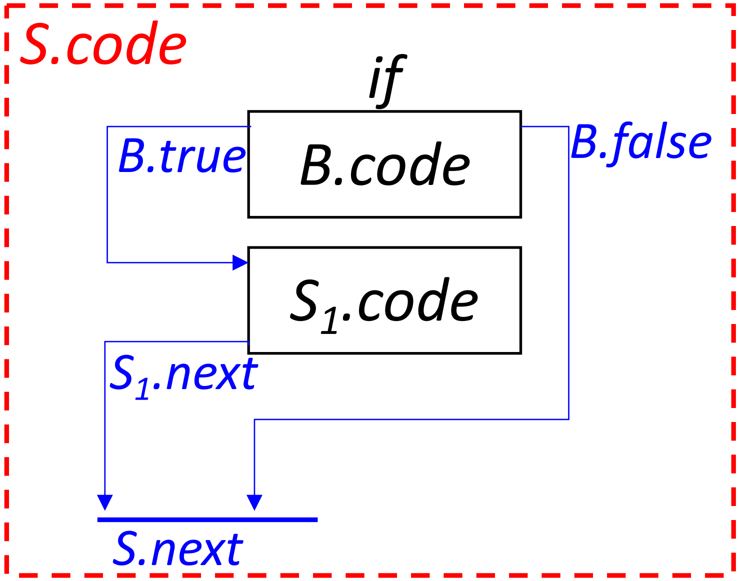
IfFalse *B* goto *B.false*
B.true:
S_1*.code*
goto *S.next*
B.false:
S_2*.code*
S.next:

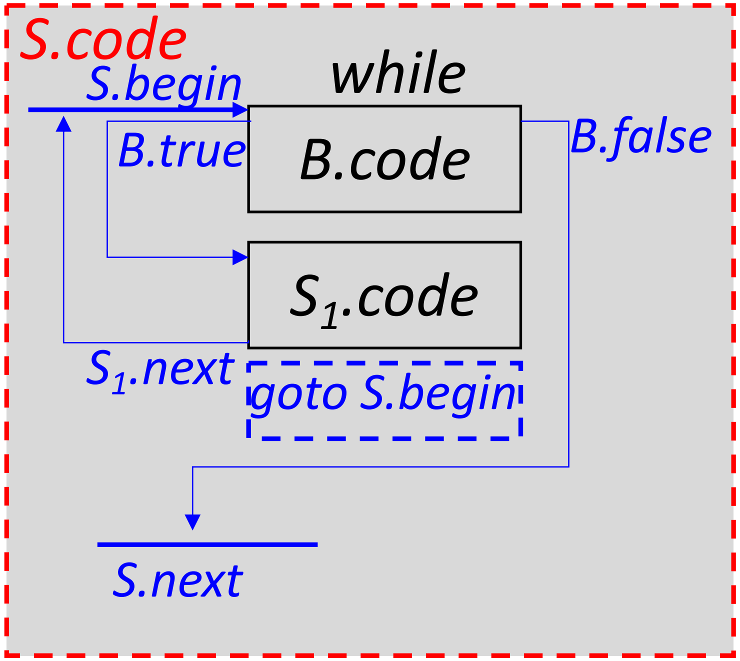
Translation of Controls (cont.)

① S -> if (B) S_1
② S -> if (B) S_1 else S_2
③ S -> while (B) S_1

S -> *if* { *B.true = newlabel();*
B.false = S.next; }
(*B*) { *label(B.true); S_1.next = S.next;* }
S_1

S -> *while* { *S.begin = newlabel();*
label(S.begin);
B.true = newlabel();
B.false = S.next; }
(*B*) { *label(B.true); S_1.next = S.begin;* }
S_1 { *gen('goto' S.begin);* }





Jumping Labels[跳转标签]

- Key of generating code for Boolean and flow-control: matching a jump inst with the target of jump[跳转指令匹配到跳转目标]
  - Forward jump: a jump to an instruction below you
  - Label for jump target has not yet been generated
  - The labels are **not *L-attributed***[非左属性]

B -> { B_1.true = newlabel(); B_1.false = B.false; } B_1
    && { label(B_1.true); B_2.true = B.true; B_2.false = B.false; } B_2

S -> if { B.true = newlabel();
        B.false = S.next; }
    (B) { label(B.true); S_1.next = S.next; }
    S_1

Handle Non-L-Attribute Labels[处理非左]

- Idea: generate code using <u>dummy labels first</u>, then patch them with <u>addresses later</u> after labels are generated

- **Two-pass** approach: requires two scans of code
  - Pass 1:
    - Generate code creating dummy labels for forward jumps. (Insert label into a hashtable when created)
    - When label emitted, record address in hashtable
  - Pass 2:
    - Replace dummy labels with target addresses (Use previously built hashtable for mapping)
- **One-pass** approach
  - Generate holes when forward jumping to a un-generated label
  - Maintain a list of holes for that label
  - Fill in holes with addresses when label generated later on

Two-Pass Code Generation[两遍生成]

- **newlabel()**: generates a new dummy label
  - Label inserted into hashtable, initially with no address
- Pass 1: generate code with non-address-mapped labels
  - For $S \rightarrow if\ (B)\ S_1$:
    - Dummy labels: *B.true=newlabel(); B.false=S.next;*
    - Generate *B.code* using dummy labels *B.true*, *B.false*
    - Generate label *B.true*: in the process mapping it to an address
    - Generate S_1*.code* using dummy label S_1*.next*
- Pass 2: Replace labels with addresses using hashtable
  - Any forward jumps to dummy labels *B.true*, *B.false* are replaced with jump target addresses

```
S -> if { B.true = newlabel();
          B.false = S.next; }
     ( B ) { label(B.true); S1.next = S.next; }
     S1
```

```
  IfFalse B goto S.next
B.true:
  S1.code
S.next:
```

One-Pass Code Generation[单遍生成]

- If *L-attributed*, grammar can be processed in one pass
- However, forward jumps introduce *non-L-attributes*
  - E.g. $E_1.false = E_2.label$ in $E \rightarrow E_1 \mid\mid E_2$
  - We need to know address of $E_2.label$ to insert jumps in E_1
  - Is there a general solution to this problem?

- Solution: **Backpatching**[回填]
  - Leave holes in IR in place of forward jump addresses
  - Record indices of jump instructions in a hole list
  - When target address of label for jump is eventually known, backpatch holes using the hole list for that particular label
- Can be used to handle any *non-L-attribute* in a grammar

Backpatching[回填]

- Synthesized attributes[综合属性]. *S -> if (B) S_1*
  - *B.truelist*: a list of jump or conditional jump insts into which we must insert the label to which control goes if *B* is true[B为真时控制流应该转向的指令的标号]
  - *B.falselist*: a list of insts that eventually get the label to which control goes when *B* is false[B为假时控制流应该转向的指令的标号]
  - *S.nextlist*: a list of jumps to the inst immediately following the code for *S*[紧跟在S代码之后的指令的标号]

- Functions to implement backpatching
  - *makelist(i)*: creates a new list out of statement index *i*
  - *merge(p_1, p_2)*: returns merged list of p_1 and p_2
  - *backpatch(p, i)*: fill holes in list *p* with statement index *i*

Backpatching (cont.)

- B -> B_1 || M B_2
  - If B_1 is true, then B is also true
  - If B_1 is false, we must next test B_2, so the target for jump B_1.*falselist* must be the beginning of the code of B_2

① *B* -> E_1 *relop* E_2 { *B.truelist = makelist(nextinst);*
B.falselist = makelist(nextinst+1);
gen('if' E_1*.addr relop* E_2*.addr 'goto _');*
gen('goto _'); }

② *B* -> B_1 || *M* B_2 { *backpatch(*B_1*.falselist, M.inst);*
B.truelist = merge(B_1*.truelist,* B_2*.truelist);*
B.falselist = B_2*.falselist;* }

③ *B* -> B_1 && *M* B_2 { *backpatch(*B_1*.truelist, M.inst);*
B.truelist = B_2*.truelist;*
B.falselist = merge(B_1*.falselist,* B_2*.falselist);* }

④ *M* -> *ε* { *M.inst = nextinst;* }

M: causes a semantic action to pick up the index of the next inst to be generated.

Example

① $B \text{ -> } E_1 \text{ relop } E_2$ { B.truelist = makelist(nextinst);
B.falselist = makelist(nextinst+1);
gen('if' E_1.addr relop E_2.addr 'goto _');
gen('goto _'); }

② $B \text{ -> } B_1 \text{ || } M\ B_2$ { backpatch(B_1.falselist, M.inst);
B.truelist = merge(B_1.truelist, B_2.truelist);
B.falselist = B_2.falselist; }

③ $B \text{ -> } B_1 \text{ \&\& } M\ B_2$ { backpatch(B_1.truelist, M.inst);
B.truelist = B_2.truelist;
B.falselist = merge(B_1.falselist, B_2.falselist); }

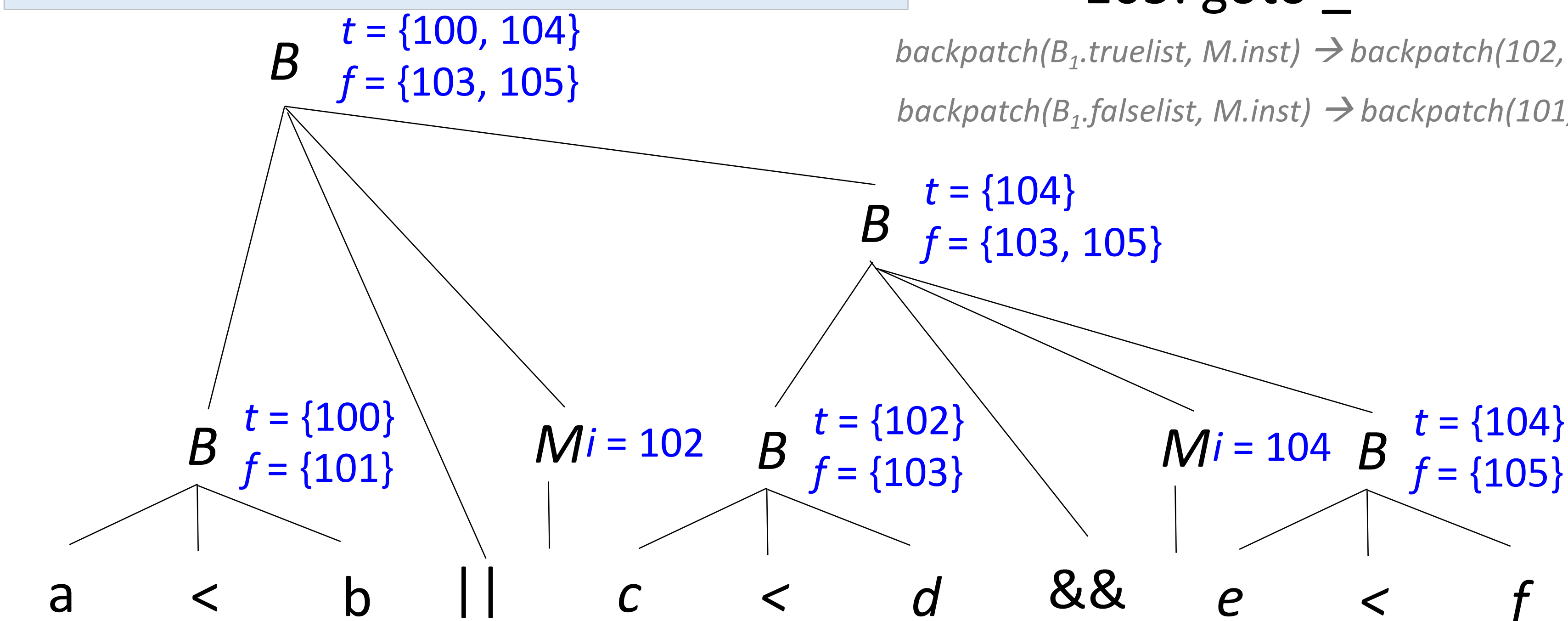
④ $M \text{ -> } \varepsilon$ { M.inst = nextinst; }

Arbitarily start inst numbers at 100

100: if a < b: goto _
101: goto 102
102: if c < d: goto 104
103: goto _
104: if e < f: goto _
105: goto _

backpatch(B_1.truelist, M.inst) → backpatch(102, 104)

backpatch(B_1.falselist, M.inst) → backpatch(101, 102)



Backpatching of Control-Flow

- *S.nextlist*: a list of all jumps to the inst following *S*

① *S -> if (B) M S_1* { *backpatch(B.truelist, M.inst)*
S.nextlist = merge(B.falselist, S_1.nextlist); }

② *S -> if (B) M_1 S_1 N else M_2 S_2* { *backpatch(B.truelist, M_1.inst);*
backpatch(B.falselist, M_2.inst);
temp = merge(S_1.nextlist, N.nextlist);
S.nextlist = merge(temp, S_2.nextlist); }

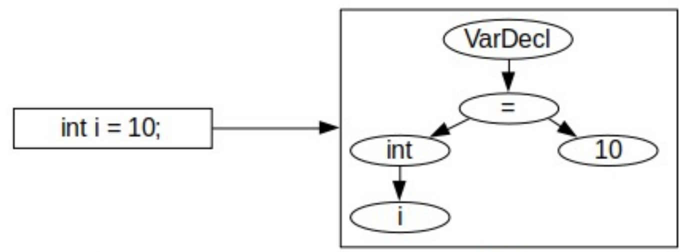
③ *S -> while M_1 (B) M_2 S_1* { *backpatch(S_1.nextlist, M_1.inst);*
backpatch(B.truelist, M_2.inst);
S.nextlist = B.falselist);
gen('goto' M_1.inst); }

④ *M -> ε* { *M.inst = nextinst;* }

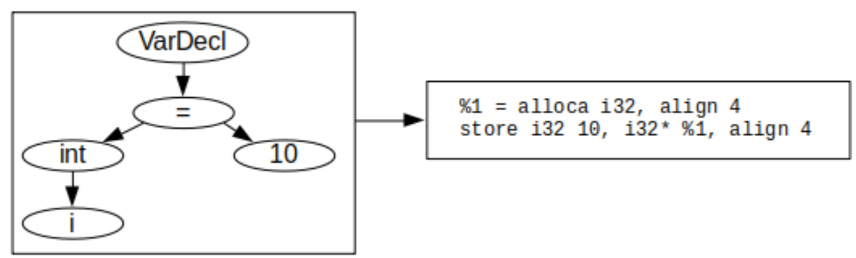
⑤ *N -> ε* { *N.nextlist = makelist(nextinst);*
gen('goto _'); }

Summary

- Code generation: generate TAC instructions using separate AST traversal (LLVM) or syntax directed translation
  - Variable definitions[变量定义]
  - Expressions and statements
    - Assignment[赋值]
    - Array references[数组引用]
    - Boolean expressions[布尔表达式]
    - Control-flow[控制流]



Code AST



AST LLVM IR

- Translations not covered
  - Switch statements[switch语句]
  - Procedure calls[过程调用]

LLVM

```c
int main() {
  int a, b, c;
  a = b + c;
  a = 3;

  if (a > 0) return 1;
  else return 0;
}
```

clang -emit-llvm -S -O0 xx.c

```llvm
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = add nsw i32 %5, %6
  store i32 %7, i32* %2, align 4
  store i32 3, i32* %2, align 4
  %8 = load i32, i32* %2, align 4
  %9 = icmp sgt i32 %8, 0
  br i1 %9, label %10, label %11

10:
  store i32 1, i32* %1, align 4
  br label %12

11:
  store i32 0, i32* %1, align 4
  br label %12

12:
  %13 = load i32, i32* %1, align 4
  ret i32 %13
}
```

clang -emit-llvm -S -O1 xx.c

```llvm
define dso_local i32 @main() local_unnamed_addr #0 {
  ret i32 1
}
```

Compilation Principle
编 译 原 理

第21讲：代码优化(1)

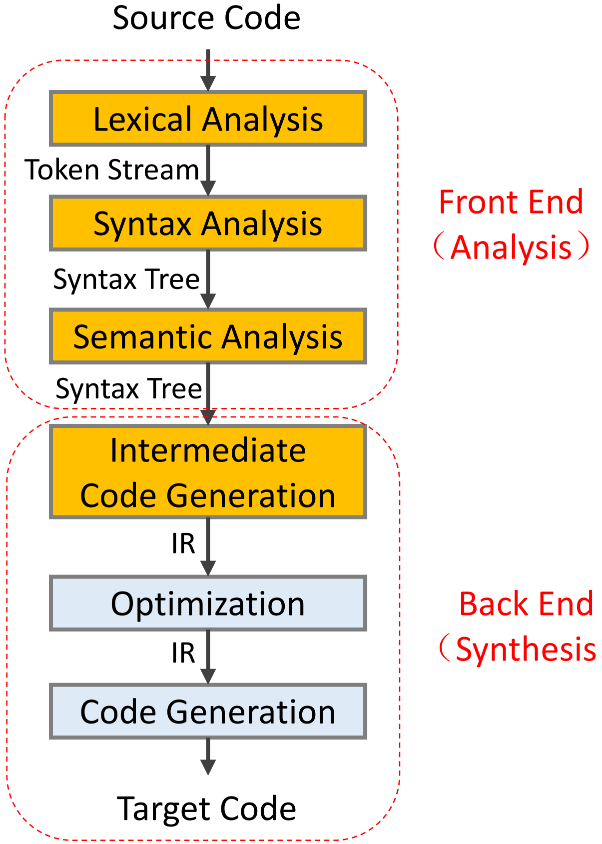
张献伟

xianweiz.github.io

DCS290, 5/16/2023

Optimization[代码优化]

- What we have now
  - IR of the source program (+symbol table)
- Goal of optimization[优化目标]
  - Improve the IR generated by the previous step to take better advantage of resources
- A very active area of research[研究热点]
  - Front end phases are well understood
  - Unoptimized code generation is relatively straightforward
  - Many optimizations are NP-complete
    - Thus usually rely on heuristics and approximations



To Optimize: Who, When, Where?

- Manual: source code[人工, 源码]
  - Select appropriate algorithms and data structures
  - Write code that the compiler can effectively optimize
    - Need to understand the capabilities and limitations of compiler opts
- **Compiler**: intermediate representation[编译器, IR]
  - To generate more efficient TAC instructions

Focus

- **Compiler**: final code generation[编译器, 目标代码]
  - E.g., selecting effective instructions to emit, allocating registers in a better way
- Assembler/Linker: after final code generation[汇编/链接, 目标代码]
  - Attempting to re-work the assembly code itself into something more efficient (e.g., link-time optimization)

Example

```c
int find_min(const int* array, const int len) {
   int min = a[0];
   for (int i = 1; i < len; i++) {
      if (a[i] < min) { min = a[i]; }
   }
   return min;
}
int find_max(const int* array, const int len) {
    int max = a[0];
    for (int i = 1; i < len; i++) {
       if (a[i] > max) { max = a[i]; }
    }
    return min;
}
void main() {
   int* array, len, min, max;
   initialize_array(array, &len);
   min = find_min(array, len);
   max = find_max(array, len);
   ...
}
```

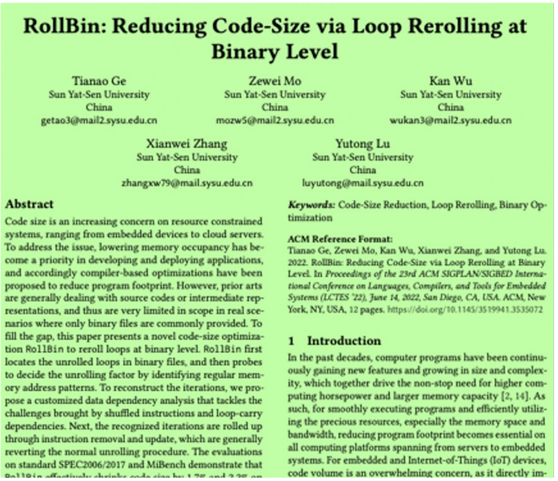
Inline → **Loop merge**

```c
void main() {
    int* array, len, min, max;
    initialize_array(array, &len);
    min = a[0]; max = a[0];
    for (int i = 0; i < len; i++) {
       if (a[i] < min) { min = a[i]; }
       if (a[i] > max) { max = a[i]; }
    }
    ...
}
```


[Link Time Optimizations: New Way to Do Compiler Optimizations](#)

Overview of Optimizations

- Goal of optimization is to generate **better** code[更好的代码]
  - Impossible to generate **optimal** code (so, it is <u>improvement</u>, actually)
    - Factors beyond control of compiler (user input, OS design, HW design) all affect what is optimal
    - Even discounting above, it's still a NP-complete problem
- Better one or more of the following (in the average case)
  - **Execution time**[运行时间]
  - **Memory usage**[内存使用]
  - Energy consumption[能耗]
    - To reduce energy bill in a data center
    - To improve the lifetime of battery powered devices
  - Binary executable size[可执行文件大小]
    - If binary needs to be sent over the network
    - If binary must fit inside small device with limited storage
  - Other criteria[其他]
- Should <u>never</u> change program semantics[正确性是前提]



RollBin: Reducing Code-Size via Loop Rerolling at Binary Level

Tianao Ge, Sun Yat-Sen University, China, getao3@mail2.sysu.edu.cn
Zewei Mo, Sun Yat-Sen University, China, mozw5@mail2.sysu.edu.cn
Kan Wu, Sun Yat-Sen University, China, wukan3@mail2.sysu.edu.cn
Xianwei Zhang, Sun Yat-Sen University, China, zhangxw79@mail.sysu.edu.cn
Yutong Lu, Sun Yat-Sen University, China, luyutong@mail.sysu.edu.cn

Abstract

Code size is an increasing concern on resource constrained systems, ranging from embedded devices to cloud servers. To address the issue, lowering memory occupancy has become a priority in developing and deploying applications, and accordingly compiler-based optimizations have been proposed to reduce program footprint. However, prior arts are generally dealing with source codes or intermediate representations, and thus are very limited in scope in real scenarios where only binary files are commonly provided. To fill the gap, this paper presents a novel code-size optimization RollBin to reroll loops at binary level. RollBin first locates the unrolled loops in binary files, and then probes to decide the unrolling factor by identifying regular memory address patterns. To reconstruct the iterations, we propose a customized data dependency analysis that tackles the challenges brought by shuffled instructions and loop-carry dependencies. Next, the recognized iterations are rolled up through instruction removal and update, which are generally reverting the normal unrolling procedure. The evaluations on standard SPEC2006/2017 and MiBench demonstrate that

Keywords: Code-Size Reduction, Loop Rerolling, Binary Optimization

ACM Reference Format:
Tianao Ge, Zewei Mo, Kan Wu, Xianwei Zhang, and Yutong Lu. 2022. RollBin: Reducing Code-Size via Loop Rerolling at Binary Level. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3519941.3535072

1 Introduction

In the past decades, computer programs have been continuously gaining new features and growing in size and complexity, which together drive the non-stop need for higher computing horsepower and larger memory capacity [2, 14]. As such, for smoothly executing programs and efficiently utilizing the precious resources, especially the memory space and bandwidth, reducing program footprint becomes essential on all computing platforms spanning from servers to embedded systems. For embedded and Internet-of-Things (IoT) devices, code volume is an overwhelming concern, as it directly im-

Types of Optimizations[分类]

- Compiler optimization is essentially a transformation[转换]
  - Delete / Add / Move / Modify something
- **Layout-related** transformations[布局相关]
  - Optimizes *where* in memory code and data is placed
  - Goal: maximize **spatial locality**[空间局部性]
    - Spatial locality: on an access, likelihood that nearby locations will also be accessed soon
    - Increases likelihood subsequent accesses will be faster
      - E.g. If access fetches cache line, later access can reuse
      - E.g. If access page faults, later access can reuse page
- **Code-related** transformations[代码相关]

  Focus
  - Optimizes *what* code is generated
  - Goal: execute least number of most costly instructions