

Memory Allocation & Code Optimization

Target Code, Basic Blocks, Flow Graphs and DAGs

K.R. Chowdhary, Professor of CS

Compiler Construction, Fall 2019 | Lecture 17

17.1 Introduction: The Code Generator



- ▶ Input: IR plus symbol-table information; output: a semantically equivalent target program
- ▶ Target code must be high quality (uses CPU, memory, registers well) and the generator itself must run efficiently
- ▶ Optimal code is **undecidable**; subproblems such as register allocation are intractable, so we use heuristics

Three primary tasks

1. **Instruction selection:** choose target instructions for IR statements
2. **Register allocation and assignment:** which values live in which registers
3. **Instruction ordering:** schedule execution order

17.2–17.3 Target Language and Design Issues

Target machine model

Three-address machine, registers $R_0 \dots R_{n-1}$:

- ▶ LD r, x / ST x, r
- ▶ OP $dst, src1, src2$ (ADD, SUB, ...)
- ▶ BR $L, Bcond\ r, L$
- ▶ Modes: register, memory, indirect, absolute, relative

Skeleton for $x = y + z$:

```
LD  R0, y      // R0 = y
ADD R0, z      // R0 = R0 + z
ST  x, R0      // x = R0
```

Design issues

- ▶ **Input:** quadruples, triples, bytecode, postfix, trees, DAGs; assumed error-free with type conversions inserted
- ▶ **Target:** RISC (many registers, simple instructions), CISC (few registers, many modes, variable length), stack machines (e.g. JVM)
- ▶ **Instruction selection:** depends on IR level, instruction set and desired quality; templates give poor code
- ▶ **Registers:** allocation (what to keep) and assignment (which register), NP-complete
- ▶ **Evaluation order:** affects registers needed; best order is NP-complete

17.4 Addresses in Target Code: Static Allocation

The run-time address space has four areas:

1. **Code:** size known at compile time
2. **Static data:** globals, constants; size known
3. **Heap:** dynamic, size unknown
4. **Stack:** activation records, size unknown

Static call (call callee) is two instructions:

```
ST callee.staticArea, #here+20
BR callee.codeArea
```

```
Return: BR *callee.staticArea
```

Example 17.1: call p from c

```
// code for c
100: ACTION1
120: ST 364, #140    // save return addr
132: BR 200          // call p
140: ACTION2
160: HALT            // back to OS
// code for p
200: ACTION3
220: BR *364         // return to 140
// activation records
300: ...             // record for c
364: <return addr>   // record for p
368: <local data of p>
```

Each ACTION is 20 bytes; the 5-word call sequence is 20 bytes (4-byte words).

17.4.2–17.4.3 Stack Allocation and Run-Time Addresses

Activation records use relative addresses; register SP points to the record on top of the stack.

```
LD  SP, #stackStart // init stack
...code for first procedure...
HALT
```

Call sequence

```
ADD SP, SP, #caller.recordSize
ST  *SP, #here+16 // return addr
BR  callee.codeArea // go to callee
```

Return sequence

```
BR  *0(SP) // callee: return
SUB SP, SP, #caller.recordSize // caller
```

Why *0(SP)?

Two levels of indirection: 0(SP) is the first word of the record, *0(SP) is the return address saved there.

Names to addresses

A symbol-table name must finally become a storage access. If *x* has relative address 12 and the static area starts at 100:

$$x = 0 \Rightarrow \text{static}[12] = 0 \Rightarrow \text{LD } 112, \#0$$

Keeping names as symbol-table pointers makes the compiler portable.

17.5–17.6 Basic Blocks and the Leader Algorithm

A **basic block** is a maximal run of three-address instructions entered only at its first instruction and left only at its last.

Algorithm 1: finding leaders

1. First instruction is a leader
2. Target of any jump is a leader
3. Instruction right after a jump is a leader

Block = leader up to (not including) the next leader.

Example 17.2: jumps at 9, 11, 17 have targets 3, 2, 13; successors 10 and 12 are leaders.

Leaders: 1, 2, 3, 10, 12, 13.

Blocks: {1}, {2}, {3–9}, {10–11}, {12}, {13–17}.

Table 17.1: identity matrix

1) i = 1	<- leader
2) j = 1	<- leader
3) t1 = 10 * i	<- leader
4) t2 = t1 + j	
5) t3 = 4 * t2	
6) t4 = t3 - 44	
7) a[t4] = 0.0	
8) j = j + 1	
9) if j <= 10 goto (3)	
10) i = i + 1	<- leader
11) if i <= 10 goto (2)	
12) i = 1	<- leader
13) t5 = i - 1	<- leader
14) t6 = 44 * t5	
15) a[t6] = 1.0	
16) i = i + 1	
17) if i <= 10 goto (13)	

17.6.1–17.6.2 Liveness, Next-Use and Flow Graphs

Algorithm 2: liveness and next-use

Input: basic block B ; nontemporaries are live on exit. Scan *backwards* from the last statement. At each $i : x = y + z$:

1. Attach to i the symbol-table info on x, y, z
2. Set x to “not live”, “no next use”
3. Set y, z to “live”, next use = i

Steps 2 and 3 must not be swapped, since x may equal y or z .

A value in a register that is never used again frees that register.

Flow graph

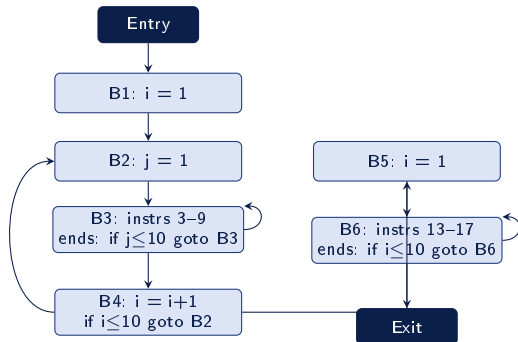
Nodes are basic blocks. Edge $B \rightarrow C$ iff C 's first instruction can immediately follow B 's last:

- ▶ a jump from the end of B to the start of C , or
- ▶ C follows B in order and B does not end in an unconditional jump

Add *entry* and *exit* nodes. Jumps target blocks, not instruction numbers, so blocks can change freely.

Use a linked list of instructions per block.

Example 17.3–17.4: Flow Graph and Loops



Loop definition

A set L of nodes is a loop if

1. one *loop entry* is the only node with predecessors outside L
2. every node has a nonempty path within L to the entry

Loops in the graph

1. B3 by itself
2. B6 by itself
3. {B2, B3, B4}, entry B2

B2 alone is not a loop: no path $B2 \rightarrow B2$ inside $\{B2\}$.

17.7 Optimizing Basic Blocks with DAGs

DAG construction

1. A leaf for each initial variable value
2. A node per statement; children are the last definitions of its operands
3. Label with the operator and the variables it last defines
4. Output nodes are live on exit

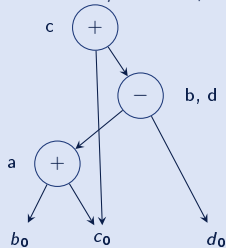
Transformations: common subexpressions, dead code, reordering independent statements, algebraic laws.

Common subexpression: a new node with the same operator and children as an existing one reuses it (value numbering).

Dead code: delete roots with no live variables, repeatedly.

Example 17.5

$a = b + c; b = a - d; c = b + c; d = a - d$



Statement 4 matches the $-$ node, so d joins its list. If b is dead on exit:

$a = b + c; d = a - d; c = d + c.$

Arrays in DAGs and Review Questions

Array references (Example 17.8)

`x = a[i]; a[j] = y; z = a[i]`

- ▶ `x = a[i]`: node = [] with children a_0 and i
- ▶ `a[j] = y`: node [] = with children a_0, j, y ; no variable label
- ▶ That node **kills** all nodes depending on a_0 ; a killed node gets no more labels

So `z = a[i]` is *not* replaced by `z = x`, since j may equal i .

Example 17.6: the DAG misses that $b_0 + c_0 = (b - d) + (c + d)$; algebraic identities are needed to expose it.

Review questions

- ▶ What are the primary tasks of a code generator?
- ▶ Why segment IR into basic blocks? Define block and flow graph.
- ▶ Which optimizations does a DAG enable?
- ▶ How do you decide if a value is live on block exit?
- ▶ Which addresses are fixed at compile time and which are not?
- ▶ Static vs. stack allocation? What is an activation record?