Optimization

ASU Textbook Chapter 9

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

http://www.iis.sinica.edu.tw/~tshsu

Machine-dependent code generation

- **For some compiler, the intermediate code is a pseudo code of a virtual machine.**
    - **Interpreter of the virtual machine is invoked to execute the intermediate code.**
    - **No machine-dependent code generation is needed.**
    - **usually with great overhead.**
    - **Example:**
        - ***Pascal: P-code for the virtual P machine.***
        - ***JAVA: Byte code for the virtual JAVA machine.***

Machine-dependent issues (1/2)

- **Input and output format:**
  - **The format of the intermediate code and the target program.**
- **Memory management:**
  - **Alignment, indirect addressing, paging, segment, . . .**
  - **Those you learned from your assembly language class.**
- **Instruction cost:**
  - **Special machine instructions to speed up execution.**
  - **Example:**
    - ***Increment by 1.***
    - ***Multiplying or dividing by 2.***
    - ***Bit-wise manipulation.***
    - ***Operators applied on a block of memory space.***
  - **Pick the fastest instruction combination for a certain target machine.**

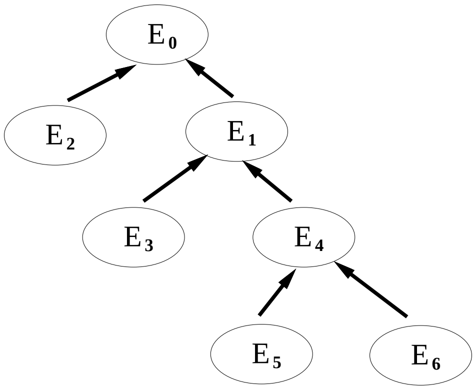
Machine-dependent issues (2/2)

- **Register allocation:**
  - **C language allows the user to management a pool of registers.**
  - **Some language leaves the task to compiler.**
  - **Idea: save mostly used intermediate result in a register. However, finding an optimal solution for using a limited set of registers is NP-hard.**
  - **Example:**

    ```
    t := a + b           load  R0,a         load   R0,a
                         load  R1,b         add    R0,b
                         add   R0,R1        store  R0,T
                         store R0,T
    ```
  - **Solutions using heuristics: similar to swapping.**
- **Optimization.**

Optimization

- Issues:
  - In an expression, assume its dependence graph is given.
  - We can evaluate it using any topological ordering.
  - There are many legal topological orderings.
  - Picking a right one will increase its efficiency.
- Example:

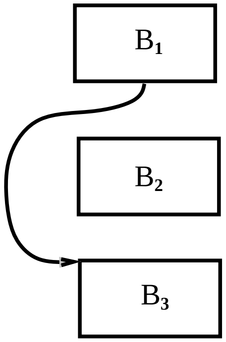


order#1	reg#	order#2	reg#
E2	1	E6	1
E3	2	E5	2
E5	3	E4	1
E6	4	E3	2
E4	3	E1	1
E1	2	E2	2
E0	1	E0	1

- On a machine with only 2 free registers, some of the intermediate results in ordering 1 must be stored in temporary space.
  - STORE/LOAD takes time.

Basic blocks and flow graphs

- **Assumption: the input is an intermediate code program.**
- **Basic block: a sequence of intermediate code such that**
    - **Jump statements, if any, are at the end of the sequence.**
    - **Codes in other basic block can only jump to the beginning of this sequence, but not in the middle.**
    - **Example:**
        - $t_1 := a * a$
        - $t_2 := a * b$
        - $t_3 := 2 * t_2$
        - $\mathit{goto\ outter}$
- **Flow graph: represent the program using a flow chart-like graph where nodes are basic blocks and edges are flow of control.**



How to find basic blocks

- **How to find leaders, which are the first statements of basic blocks.**
  - **The first statement of a program is a leader.**
  - **For all conditional and unconditional goto:**
    - ▷ ***Its target is a leader.***
    - ▷ ***Its next statement is also a leader.***
- **Using leaders to partition the program into basic blocks.**
- **Ideas for optimization:**
  - **Two basic blocks are equivalent if they compute the same expressions.**
  - **Use transformation techniques below to perform machine-dependent optimization.**

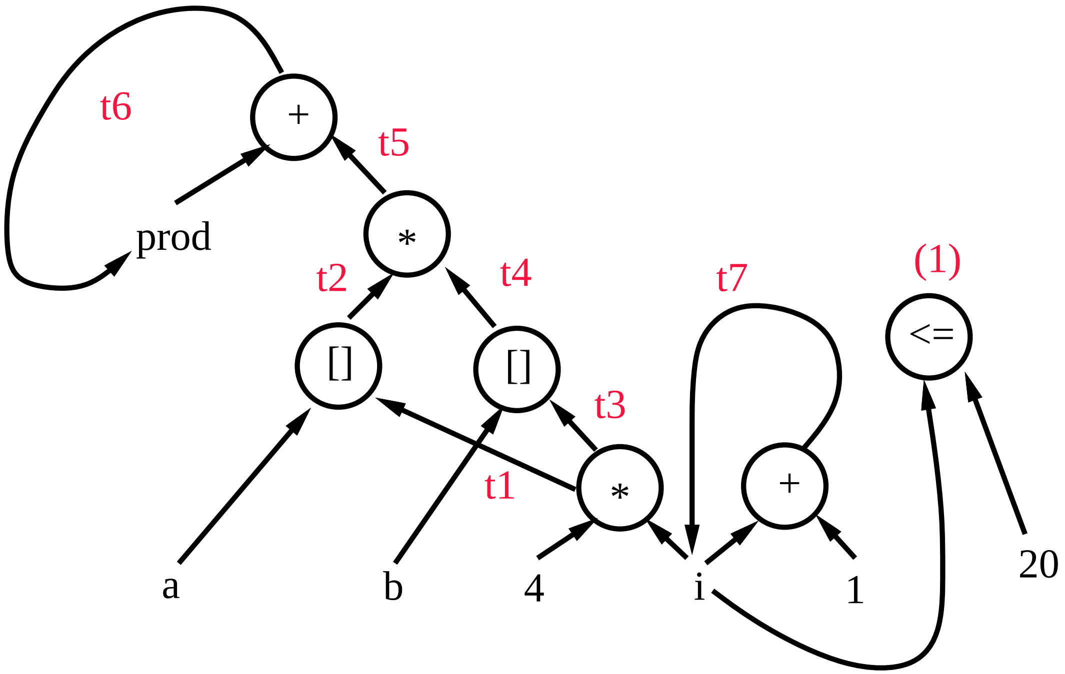
Finding basic blocks — examples

- **Example: Three-address code for computing the dot product of two vectors a and b.**
  - $prod := 0$
  - $i := 1$
  - ***loop:*** $t_1 := 4 * i$
  - $t_2 := a[t_1]$
  - $t_3 := 4 * i$
  - $t_4 := b[t_3]$
  - $t_5 := t_2 * t_4$
  - $t_6 := prod + t_5$
  - $prod := t_6$
  - $t_7 := i + 1$
  - $i := t_7$
  - ***if*** $i \leq 20$ ***goto loop***
  - $\cdots$
- **There are three blocks in the above example.**

DAG representation of a basic block

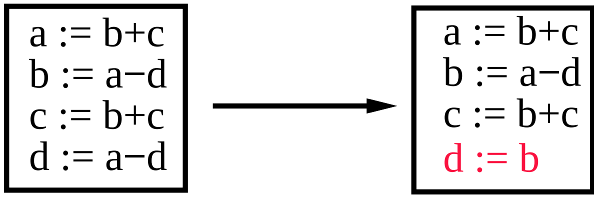
- **Inside a basic block:**
  - **Expressions can be expressed using a DAG that is similar to the idea of a dependence graph.**
  - **Graph might not be connected.**
- **Example:**

(1) $t_1 := 4 * i$
(2) $t_2 := a[t_1]$
(3) $t_3 := 4 * i$
(4) $t_4 := b[t_3]$
(5) $t_5 := t_2 * t_4$
(6) $t_6 := prod + t_5$
(7) $prod := t_6$
(8) $t_7 := i + 1$
(9) $i := t_7$
(10) if $i \leq 20$ **goto (1)**



Structure-preserving transformations

- **Techniques: using the flow graph and DAG representation of basic blocks.**
  - **Common sub-expression elimination.**



```
a := b+c
b := a−d
c := b+c
d := a−d
```

```
a := b+c
b := a−d
c := b+c
d := b
```

  - **Dead-code elimination: remove unreachable codes.**
  - **Renaming temporary variables: better usage of registers and avoiding using unneeded temporary variables.**
  - **Interchange of two independent adjacent statements, which might be useful in discovering the above three transformations.**
    - ***Same expressions that are too far away to store E_1 into a register.***
    - ***Example:***

```
t1 := E1
...
tn := E1
```

    - ***Note: two dependent statements cannot be exchanged.***
    - ***Example:***

```
t1 := a + b
...
tn := t1 + c
```

Algebraic transformations

- **Algebraic identities:**
  - $x + 0 = 0 + x = x$
  - $x - 0 = x$
  - $x * 1 = 1 * x = x$
  - $x/1 = x$
- **Reduction in strength:**
  - $x^2 = x * x$
  - $2.0 * x = x + x$
  - $x/2 = x * 0.5$
- **Constant folding:**
  - $2 * 3.14 = 6.28$
- **Standard representation for subexpression by commutativity and associativity:**
  - $n * m = m * n$.
  - $b < a == a > b$.

Peephole optimization (1/2)

- **Idea:**
  - **Statement by statement translation might generate redundant codes.**
  - **Locally improve the target code performance by examine a short sequence of target instructions (called a peephole) and do optimization on this sequence.**
  - **Complexity depends on the "window size".**
- **Techniques:**
  - **Redundant loads and stores:**
    - ▷ $MOV\ R_0, a$
    - ▷ $MOV\ a, R_0$
  - **Unreachable codes:**
    - ▷ ***An unlabeled instruction immediately following an unconditional jump may be removed.***
    - ▷ ***If statements based on constants: If debug then $\cdots$.***

Peephole optimization (2/2)

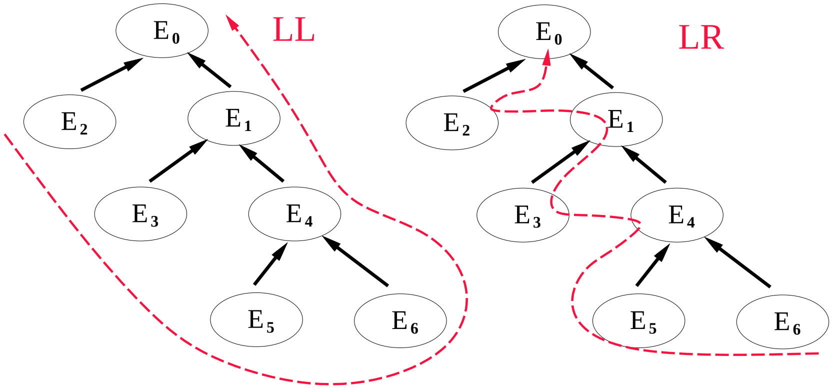
- **More techniques:**
  - **Flow of control optimization:**

goto L1	*goto L2*
. . .	. . .
L1: goto L2	*L1: goto L2*

  - **Algebraic simplification.**
  - **Use of special machine idioms.**
  - **Better usage of registers.**

Correctness after optimization

- When side effects are expected, different evaluation order may produce different results for expressions.



  - Assume E_5 is a procedure call with the side effect of changing some values in E_6.
  - LL and LR parsing produces different results.
- Watch out precisions when doing algebraic simplification.
  - if $(x = 321.123456789 - 321.123456788) > 0$ then $\cdots$
- Need to make sure code before and after optimization produce the same result.
- Complications arise when debugging is involved.