DAC718 - Compiler Design

Optimizations – An Introduction

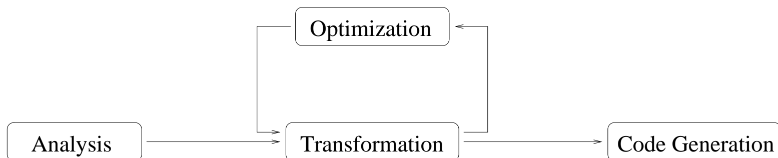
Jonas Lundberg

Jonas.Lundberg@msi.vxu.se

http://w3.msi.vxu.se/users/jonasl/dac718

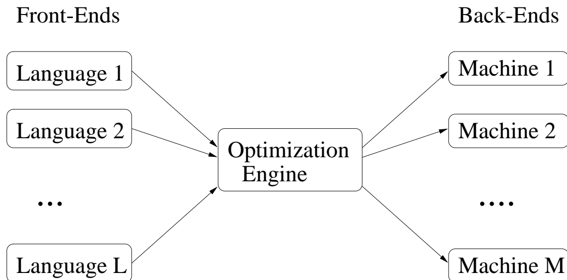
19 mars 2007

Optimization



- **Today**
  - An suitable program representation (IMR)
  - Various levels of optimization
  - Different types of optimization
- We focus on what the different methods do
- We disregard how they do it $\Rightarrow$ Our course Compilation II describes optimization in detail.

Compiler Modules



By separating the compiler into three different modules (front-end, transformer,back-end) we can:

- Build $L \times M$ compilers at the price of implementing $L + M + 1$ modules
- Only implement one transformer/optimizer (the most difficult part)
- Easily reuse the modules (especially the front-ends) in other types of applications (e.g editors and analysis tools).

Optimization: General

- **Requirement:** Program semantics must not be changed
- **Goal:**
  - Make generated program faster
  - Reduce run-time memory requirements
  - Make generated program smaller
  - Ease programming by identifying (and replacing) expensive constructs $\Rightarrow$ Programmers can focus on algorithms and design.
- **Limitations:** A bad algorithm is always a bad algorithm $\Rightarrow$ poor design can not be detected and removed.
- **Consideration:** An optimization must be worth the effort $\Rightarrow$ cost vs gain must be considered. (This may depend on the purpose of the program to be compiled.)
- **Notice:** "optimization" is an exaggeration.
  There is no guarantee that resulting code is the best possible.
  Major reason: Programs are not a closed world - they often take some input.

The three-address code representation

- The **three-address code** representation of a program is a sequence of assembly like statements (one sequence per procedure) of the general form:

  $$x := y \ op \ z$$

  where x, y, and z are identifiers, constants, or compiler generated **temporaries**; *op* stands for **operators** like $+, *, >$, *jump*, *param*, *call*.
- Examples:

```
Program
   a = (b+c) * (3+4);

Code
(1) t1 = b + c
(2) t2 = 3 + 4
(3) t3 = t1 * t2
(4)  a = t3
```

- It is called "three-address code" since each statement usually contains tree addresses, two for the operands and one for the results
- The short format requires that longer expressions (or more complicated statements) are broken into multiple codes. Temporaries (e.g. `t1`, `t2`, and `t3`) are used to store intermediate results.

Quadruples

- Three-address code is internally represented as a **quadruple**:
  `< op, arg1, arg2, res >`.
- Here `arg1`, `arg2`, and `res` are pointers to the symbol-table entries for the names or literals. The op field contains an internal code (usually an integer) for the operator.
- Example

```
Program          Code             Quadruples
=======          ====             ==========
a = x+y*z        t1 := y*z        <mul, y , z , t1>
                 t2 := x+t1       <add, x , t1, t2>
                 a  := t2         <cpy, t2,   , a >
```

- Quadruples has a fixed size and are usually stored in a large array where they efficiently can be accessed, traversed, and manipulated.
- We will use the abstract three-address code notation, implementations work with quadruples.

A Simple Set of Quadruples

```
Quadruples                      Interpretation
==========                      ==============
< add  | a1 | a2 | r >          r := a1 + a2
< sub  | a1 | a2 | r >          r := a1 - a2
< mult | a1 | a2 | r >          r := a1 * a2
< div  | a1 | a2 | r >          r := a1 / a2
< usub | a1 |    | r >          r := - a1
< lt   | a1 | a2 | r >          r := a1 < a2
< eq   | a1 | a2 | r >          r := a1 = a2

< cpy  | a1 |    | r >          r := a1
< []=  | a1 | a2 | r >          r[a1] := a2
< =[]  | a1 | a2 | r >          r := a1[a2]

< jmp  |    | a2 |   >          goto a2
< cjmp | a1 | a2 |   >          if a1 = false goto a2

< par  | a1 |    |   >          arg := a1
< call | a1 |    | r >          r := call a1
```

Examples with Conditions and Calls

```
Program
=======
   while ( i < 10 ) {
     s = s + i;
     i = i + 1;
   }

Code
======
(1) t1 = i < 10
(2) if_false t1 goto (8)
(3) t2 = s + i
(4) s = t2
(5) t3 = i + 1
(6) i = t3
(7) goto (1)
(8)
```

```
Program
=======
   x = 1;
   y = 2;
   z = p(x+y,y*x);

Code
======
(1) x = 1
(2) y = 2
(3) t1 = x + y
(4) t2 = y * x
(5) arg t1
(6) arg t2
(7) z = call p
```

Syntax Driven Quadruple Generation

- The quadruple representation can be generated using a single traversal of the AST.
- Each production generates a specific sequence of quadruples

Productions	Action
$Assgn \rightarrow Id = Exp$	$Id.symbol = Exp.val$
$Exp \rightarrow Exp1 + Exp2$	$Exp.val = newTemp(),\ Exp.val = Exp1.val + Exp2.val$
$Exp \rightarrow Exp1 * Exp2$	$Exp.val = newTemp(),\ Exp.val = Exp1.val * Exp2.val$
$Exp \rightarrow Id$	$Exp.val = Id.symbol$

- Basic idea: The result of each subexpression is assigned to a new temporary.

```
Program                 Generated Quadruples
=======                 ====================
a = x+y*z               t1 = y * z
                        t2 = x + t1
                        a = t2
```

- Control statements like `if` or `while` requires backpatching.

Peephole Optimizations 1

- The generated code is often of poor quality and can easily be improved.
- Hence, the code generation is often followed by a so-called **peephole optimization** where obvious suboptimal code constructs are removed.
- It is called "peephole" since we scan the generated code and examine short sequences (peepholes) which we try to replace by a shorter (or faster) sequence.
- **Redundant Assignments:** The assignment instruction can often be merged with the previous instruction.

```
Program          Generated Quadruples     Peephole Optimized
=======          ====================     ==================
a = x + y        t1 = x + y               a = x + y
                 a = t1
```

Peephole Optimizations 2

- Simple **algebraic identities** like $x * 1 = 1 * x = x + 0 = 0 + x = x$ can also be used.

```
Generated Quadruples          Peephole Optimized
====================          ==================
(4) t5 = t4 * 1              (4) t5 = t4
```

  The Redundant Assignment optimization will (maybe) later remove `t5 = t4`.

- **Repeated jumps** can be simplified.

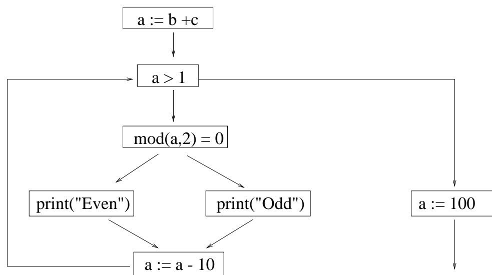
```
Generated Quadruples          Peephole Optimized
====================          ==================
(4) if_false t4 goto (8)      (4) if_false t4 goto (16)
...
(8) goto (16)
```

- A simple peephole optimization can often reduce the number of instruction by 15-20% if the generated code is of poor quality.
- The peephole optimization is often considered to be a part (the last part) of the code generation.

Basic Blocks and Flow Graphs

- The actual optimization is often performed on a **flow graph**.
  The nodes of the flow graph are called **basic blocks**
- A basic block B is a sequence of statements in which flow of control enters at the beginning and leaves in the end without any branching.
- If flow of control after leaving block B_1 can proceed with block B_2, then $B_1 \rightarrow B_2$ is an edge in the flow graph.

```
a := b + c;
while (a>1) {
  if ( mod(a,2) = 0 )
    print(‘‘Even’’);
  else
    print(‘‘Odd’’);
  endif
  a := a - 10;
}
a := 100;
```



Fibonacci

```
public int fibonacci(int m)
  int f2, ret;
  int f0 = 0, f1 = 1;
  if (m <= 1)
     ret = m;
  else {
     for (int i=2; i <= m; i++) {
        f2 = f0 + f1;
        f0 = f1;
        f1 = f2;
     }
     ret = f2;
  }
  return ret;
}
```

```
(1)  f0 = 0
(2)  f1 = 1
(3)  t1 = m <= 1
(4)  if_false t1 goto (7)
(5)  ret = m
(6)  goto (16)
(7)  i = 2
(8)  t2 = i <= m
(9)  if t2 goto (15)
(10) f2 = f0 + f1
(11) f0 = f1
(12) f1 = f2
(13) i = i + 1
(14) goto (8)
(15) ret = f2
(16) return ret
```

Block Partitioning Algorithm

1. We first determine the **leaders** (first statement in a block)
   1.1 The first statement is a leader
   1.2 Any target of a jump/cjump is a leader
   1.3 Any statement that immediately follows a jump/cjump is a leader
2. For each leader, its basic blocks consists of the leader and all statements up to but not including the next leader.

We have an edge $BB_1 \rightarrow BB_2$ if

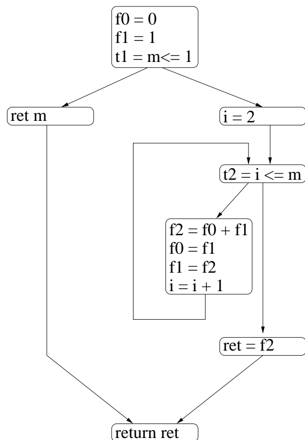
1. there is a goto/jump/cjump from the last statement in BB_1 to the first in BB_2
2. BB_2 immediately follows BB_1 and BB_1 does not end in an unconditional jump.

Fibonacci (2)

```
Code                    Leaders    Blocks   Edges
=====                   =======    ======   =====
(1)  f0 = 0             rule 1a      B1
(2)  f1 = 1                          B1
(3)  t1 = m <= 1                     B1
(4)  if t1 goto (7)                  B1     B1 -> B3, B1 -> B2
(5)  ret = m            rule 1c      B2
(6)  goto (16)                       B2     B2 -> B7
(7)  i = 2              rule 1b      B3     B3 -> B4
(8)  t2 = i <= m        rule 1b      B4
(9)  if t2 goto (15)                 B4     B4 -> B6, B4 -> B5
(10) f2 = f0 + f1       rule 1c      B5
(11) f0 = f1                         B5
(12) f1 = f2                         B5
(13) i = i + 1                       B5
(14) goto (8)                        B5     B5 -> B4
(15) ret = f2           rule 1b      B6     B6 -> B7
(16) return ret         rule 1b      B7
```

Fibonacci

```
public int fibonacci(int m)
  int f2, ret;
  int f0 = 0, f1 = 1;
  if (m <= 1)
     ret = m;
  else {
     for (int i=2; i <= m; i++) {
        f2 = f0 + f1;
        f0 = f1;
        f1 = f2;
     }
     ret = f2;
  }
  return ret;
}
```



Notice: We don't keep the `jump/cjump` statements in the blocks.

This information is now encoded in the block edges.

Different Levels of Optimization

- A program can be optimized on many different levels
  1. **Intra-block:** We only look at statements within a single block.
  2. **Intra-procedural:** We look at statements within a single procedure. Here we must take into account contributions from different flow paths. (Much more difficult.)
  3. **Inter-procedural (or whole program):** Information is transported from one procedure to another. This is very tricky when we have polymorphic calls.
- Whole program optimization is a very active research area due to the popularity of OOPL. OO systems have many "expensive" polymorphic methods calls. Many types of optimization could be performed if the polymorphic calls could be resolved at compile time.
  That is, if we could decide the exact target of the polymorphic method calls.
- Many kinds of optimization can be performed at different levels of optimization.
- We start by optimizing individual blocks, then procedures and finally the whole program $\Rightarrow$ A bottom up approach.

Intra-block Optimization

- An occurrence of an expression *E* is called a **common subexpression** if *E* was previously computed, and the value of variables in *E* have not been changed since previously computation.

```
t1 := a + d                                   t1 := a + d
 .....
(no definitions of a and d)      ==>              ....
tn := a + d                                   tn := t1
```

- Deducing at compile time that the value of an expression *E* is constant and using that value instead is known as **constant folding**.

```
  a := 1                               a := 1
  b := 2                               b := 2
  ...                        ===>
(no definitions of a and b)
  c := a + b                           c := 3
```

- Both common expression optimization and constant folding may introduce redundant assignments ⇒ removal of redundant assignments is performed all the time.

Intra Block Optimization (2)

- Replacing "expensive" operations like $2 * x$, $x/4$, and x^2 with "cheaper" operations like $x + x$, $x \gg 2$, and $x * x$ is called **strength reduction**. This kind of optimization is repeated when machine code has been generated.
- Constant folding often gives rise to some **dead code elimination** where code that can't be executed is removed.
- **Example:** Assume that constant folding has shown that `debug` = `false`

```
if (debug) {
  list.printContent();
}
```

We can then remove the whole if-statement and maybe the method `printContent()` as well.

Intra-procedural Optimization

- "90% of the time is spent executing 10% of the code". The frequently used parts are called **hot spots** and are usually within loops, especially the inner loops.
- We use different graph theoretic concepts to identify loops in the flow graph:
  - Back edges are used to identify loops
  - Dominance analysis are used to classify loops as inner or outer
- Once loops have been identified, we try to reduce computation time by moving as much work as possible out of the loops.
- Example: Computations that gives the same result for every iteration of the loop (loop-invariants) should be moved before the loop

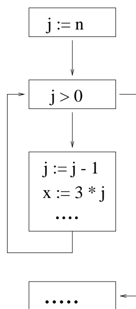
```
while (i <= limit-2) {                          t1 := limit -2
   Code that doesn't change limit       ==>     while (i <= t1) {
}                                                 ...
                                                }
```

Loop Optimization (Strength Reduction)

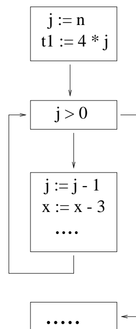
Program

```
j = n;
while (j > 0) {
  j = j - 1;
  x = 3 * j;
   ....
}
```

Before



After



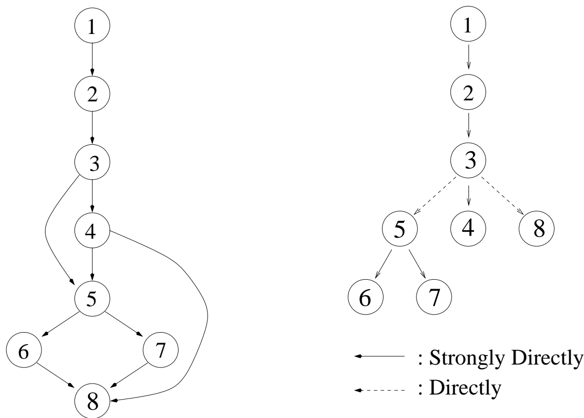
Dominance Analysis: Theory

Dominance analysis is used to identify and classify loops.

Dominance is a relation between nodes in a rooted directed graph $G_r = (N, E, r)$. We say that

- $a \in N$ **dominates** $b \in N$, written $a\ dom\ b$, iff every possible path from the root node r to b includes a.
- a is a **direct** dominator of b, written $a\ ddom\ b$, iff $a \neq b \wedge a\ dom\ b$ and there does not exist a node c $(\neq a, b)$ such that $a\ dom\ c \wedge c\ dom\ b$.
- a **strongly directly** dominates b, written $a\ sdom\ b$, iff $a\ ddom\ b$ and a is the only predecessor of b.
- The direct dominance relation identifies, for each single node, a single dominator. This relation can be considered as a parent-child relation and hence, we have a tree, the **dominance tree.**

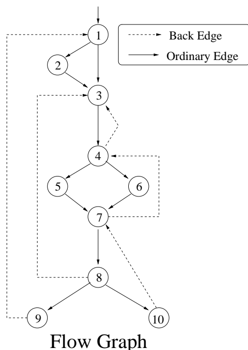
A Simple Example



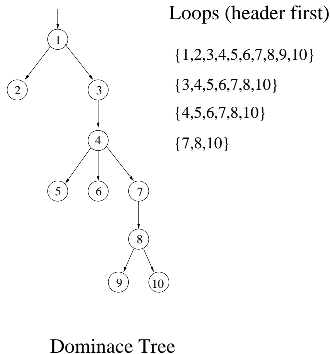
A rooted directed graph (left) with the corresponding dominance tree(right)

Natural Loops

- An edge $a \rightarrow b$ in the flow graph is a **back edge** if b dominates a.
- Given a back edge $a \rightarrow b$ we define the **natural loop** of the edge to be b plus the set of nodes that can reach a without going through b. Node b is the **head** of that natural loop.



Flow Graph



Dominace Tree

- Notice that the back edges $4 \rightarrow 3$ and $8 \rightarrow 3$ generates the same loop $\{3, 4, 5, 6, 7, 8, 10\}$.

Optimization: Summary

- Program optimization consists of a series of (in principle) simple transformations that take place at different levels (peephole, basic block, procedure, whole program).
- Remaining Question: How do we identify and implement these transformations?
- Example: Dead Code Elimination $\Rightarrow$ We can remove parts of the program if we, at compile time, can deduce that they never is executed
- For example, assume that we knew that $a > 5$. Then we can do the following transformation

```
if (a > 0)        \
  call f(a);       |
else               |   ==>   call g(a)   (and maybe drop procedure f complete
  call g(a);       |
fi                /
```

- Deciding all possible values of a variable a is in general impossible.
- All we can do is to find a conservative (safe) approximation of the possible values.
- These problems are called **data flow** problems since they involve tracking flows of data in between different parts (e.g. basic blocks) of the program.
- Conclusion: Optimization is simple in theory but very hard in practice.

Remains: Code Generation and Optimization

- Machine code generation is straight forward since quadruples and machine instructions are very "similar". There is often a ono-to-one mapping between quadruples and machine instructions.
- A processor is similar to a stack machine apart from the fact that we have a finite number of registers. (The stack, which can be considered as "infinite", is the register counterpart.)
- Deciding what to keep in the registers at different times during execution is a non-trivial problem and many elaborate methods are around.
- Once machine code is generated we can
  - Rearrange instruction order to minimize load and store instructions
  - Replace computational "expensive" instructions with "cheaper" ones.
  - Identify and use "machine idioms" (hardware specific instructions to implement certain frequently occurring operations like $i := i + 1$ efficiently).

Thank You and Good Luck!!