GENEROVÁNÍ KÓDU

8. ÚVOD DO PROBLÉMU OPTIMALIZACÍ, KONVERZE 3AC <-> AST (DAG), LOKÁLNÍ OPTIMALIZACE (V RÁMCI ZÁKLADNÍHO BLOKU)

2011 Jan Janoušek
MI-GEN

Evropský sociální fond
Praha & EU:
Investujeme do vaší budoucnosti

OPTIMIZATIONS - INTRODUCTION

Introduction

- Optimization is a transformation of a code - the resulting code is to work correctly and to work better from a specific point of view (speed, memory and register usage,…).

- Optimization are mostly performed over various levels of IR code – from high-levels IRs to low-level IRs. However, even optimizations over a code in the target language are possible.

- Optimization is a huge problem – many types of optimizations have been introduced.

Introduction, contd.

- A most important optimization is the register allocation, presented in the previous lecture.

- The code selection algorithm is also very important when we consider optimizations:
  - naïve (direct) code generation from an IR often produces a poor code.
  - a sophisticated code selection (e.g. by an intelligent tiling of the tree) produces a good code (here, the resulting code can be seen as code produced by a naive code selection + various optimizations).

Clasification - Levels of Optimizations

- **Local**
  - inside a basic block, based on analysis of basic blocks
- **Global**
  - across basic blocks, based on analysis of whole procedures
- **Interprocedural**
  - Across procedures, based on analysis of whole program

Clasification – another point of view

- **Machine-dependent optimization**
- **Machine-independent optimizations**

Various types of optimizations which can be joined into particular groups exist…

The Golden Rules of Optimization:

Premature Optimization is Evil

- Donald Knuth: "… ***premature optimization is the root of all evil…***"
  - Optimization can introduce new, subtle bugs
  - Optimization usually makes code harder to understand and maintain

- Get your code right first, then optimize it

The Golden Rules of Optimization: the 80/20 Rule

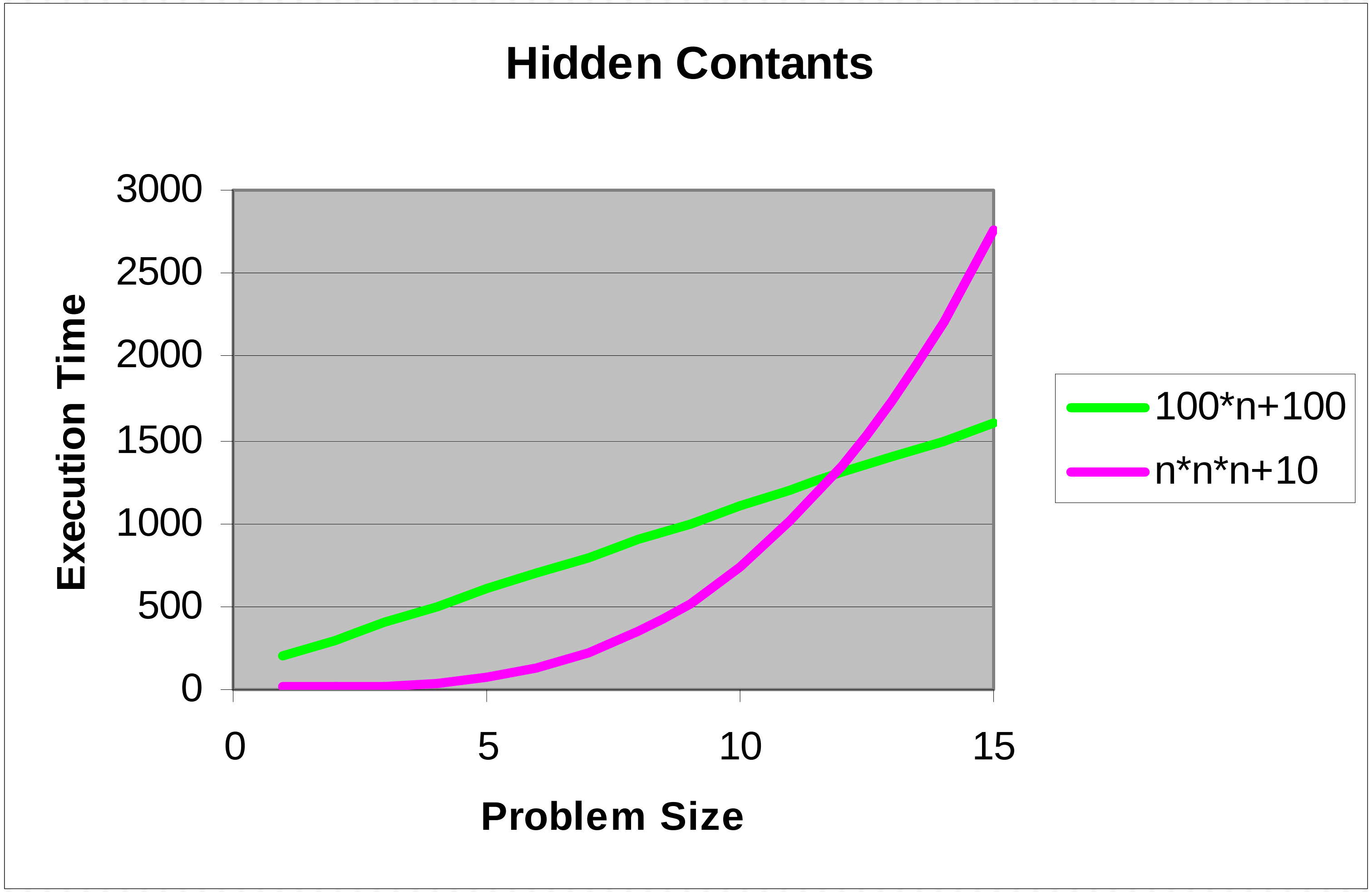
- In general, ***80% percent of a program's execution time is spent executing 20% of the code*** (mainly in loops)
- 90%-10% for performance-hungry programs

- Optimize the common case even at the cost of making the uncommon case slower

Good Algorithms Rule

- The best and most important way of optimizing a program is using **good algorithms**
  - E.g. O(n*log) rather than $O(n^2)$
- However, we still need lower level optimization to get more of our programs
- Remark: **asymptotic complexity** is not always an appropriate metric of efficiency
  - Hidden constant may be misleading
  - *E.g.* a linear time algorithm than runs in $100*n+100$ time is slower than a cubic time algorithm than runs in n^3+10 time **if the problem size is small:**

Asymptotic Complexity
Hidden Constants



LOCAL OPTIMIZATIONS (within basic blocks)

3AC -> DAG

Sometimes it is useful to construct 3AC code or DAG for particular optimizations.

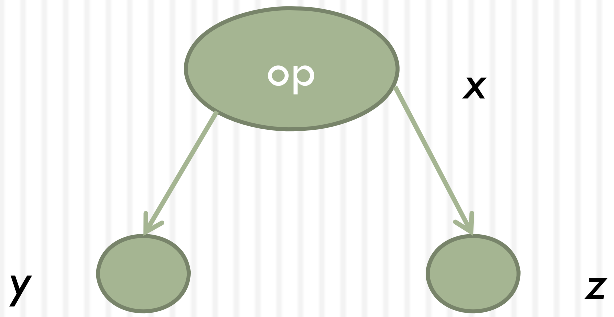
We can even convert 3AC ->DAG, then we can perform optimizations on DAG, and then convert DAG ->3AC.

3AC -> DAG construction

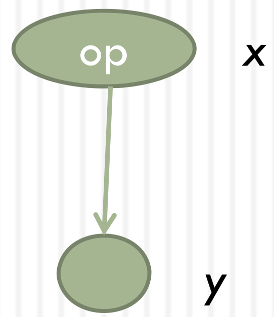
- Forward pass over basic block
- Array curr[x] = nil initiailly
- For *x = y op z* (x=op y)
  - Find node labeled y (curr[y]), or create one
  - Find node labeled z (curr[z]), or create one
  - Create new node for *op*, or find an existing one with descendants y, z (based on curr[y] and curr[z])
  - Add x to list of labels for new node, curr[x]=pointed to the node
  - Remove label x from node on which it appeared previously
- For *x = y;*
  - Add x to list of labels of node which currently holds y

3AC -> DAG

- $x = y \; op \; z$



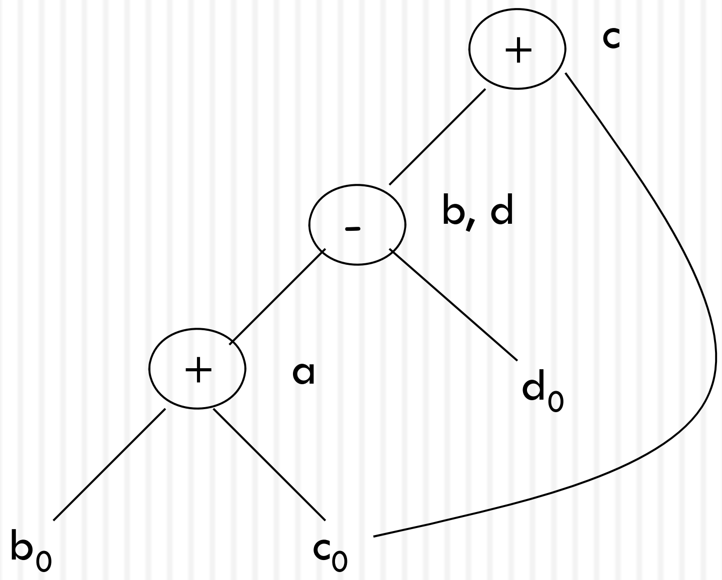
- $x = y$



DAG Example

Transform a basic block into a DAG.

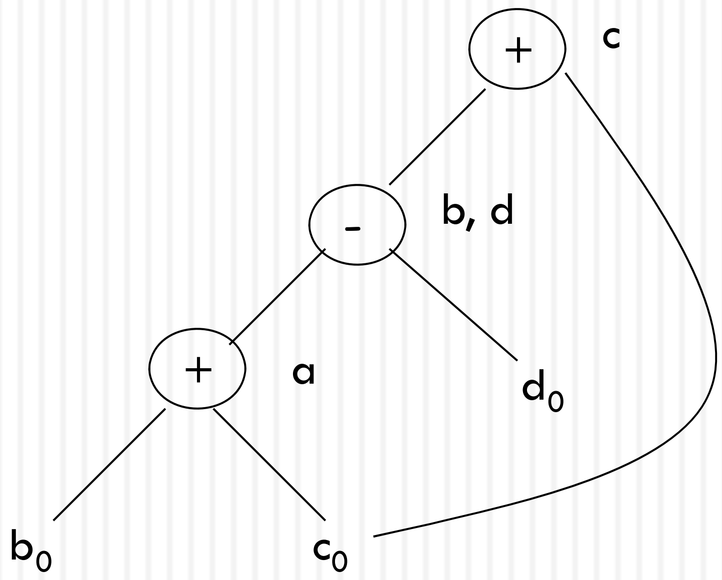
```
a = b + c
b = a - d
c = b + c
d = a - d
```



DAG Example

Transform a basic block into a DAG.

a = b + c
b = a - d
c = b + c
(d = c)



Common subexpressions

DAG -> 3AC

DAG -> 3AC

Reverse operation.

- Numbering nodes of the DAG top-down. We start at the root. Set $n=1$.
  1. Assing the selected node the number n, and remove the node together with all its outcoming edges.
  2. Set $n=n+1$. Select a unnumbered node and repeat step 1 until DAG is empty.

- Producing 3AC code for temporaries in reverse order of numbering.

For a given DAG, more 3AC codes can be produced. It is advised to favor the leftmost upper node in step 2 – the leftmost operand is usually in register.

Particular local optimizations

Some General Optimization Techniques

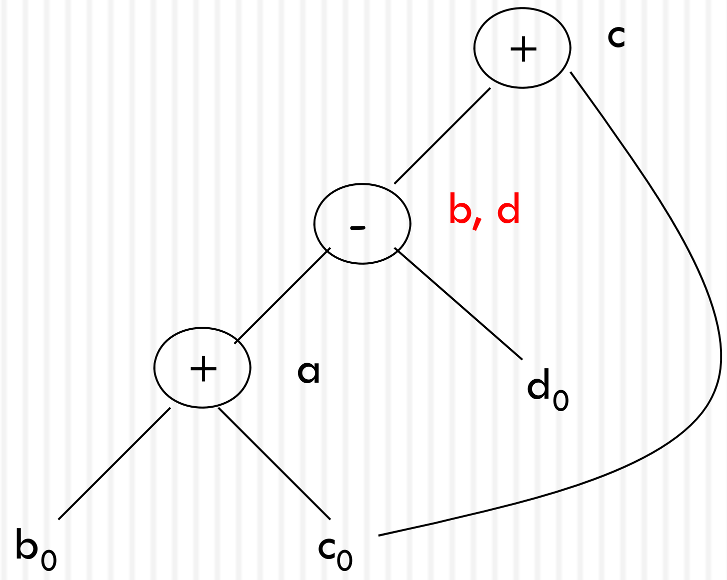
- **Strength reduction**
  - Use the fastest version of an operation
  - *E.g.*
    - `x >> 2` *instead of* `x / 4`
    - `x << 1` *instead of* `x * 2`
- **Common sub expression elimination**
  - Eliminate redundant calculations
  - *E.g.*
    - `double x = d * (lim / max) * sx;`
    - `double y = d * (lim / max) * sy;`
    -
    - `double depth = d * (lim / max);`
    - `double x = depth * sx;`
    - `double y = depth * sy;`

Local Common Subexpression elimination – based on DAG

Performing local common subexpression elimination can be based on DAG. Suppose b is not live on exit.

```
a = b + c
b = a - d
c = b + c
d = a - d
```

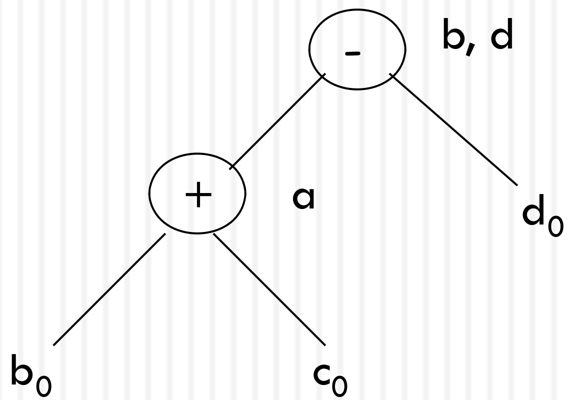
```
a = b + c
d = a - d
c = d + c
```



Dead code elimination – based on DAG

Transform a basic block into a DAG.

a = b + c
b = a – d
(d = b)



Dead code elimination (provided c is not live in the exit of the basic block)
Removing roots which are not live

Some General Optimization Techniques

- **Code motion**
  - ***Invariant* expressions** should be executed only once
  - *E.g.*

```
for (int i = 0; i < x.length; i++)
    x[i] *= Math.PI * Math.cos(y);
```

```
double picosy = Math.PI * Math.cos(y);
for (int i = 0; i < x.length; i++)
    x[i] *= picosy;
```

Some General Optimization Techniques

- **Loop unrolling**
  - The overhead of the loop control code can be reduced by executing more than one iteration in the body of the loop. *E.g.*

```
double picosy = Math.PI * Math.cos(y);
for (int i = 0; i < x.length; i++)
        x[i] *= picosy;
```

```
double picosy = Math.PI * Math.cos(y);
for (int i = 0; i < x.length; i += 2) {
        x[i] *= picosy;
        x[i+1] *= picosy;
}
```

A efficient "+1" in array indexing is required

Some General Optimization Techniques

- **Loop unrolling.** another example

```
for (int i = 0; i < 3; i++)
      x[i] = x[i] + 2;
```

```
x[0] = x[0] + 2;
x[1] = x[1] + 2;
x[2] = x[2] + 2;
```

Some General Optimization Techniques

- **Dead code elimination.**

```
If false then {code}
```

can be omitted.

```
If true then {code}
```

Can be transformed to

```
code
```

Some General Optimization Techniques

- **Dead code elimination.**

Instructions

```
X= {code}
```

, where X is not used anymore (ie. X is not live), can be omitted.

We have seen this above based on DAG.

Peephole optimization techniques

- A group of optimization
- Sub-optimal sequences of instructions that match an optimization pattern are transformed into optimal sequences of instructions
- Peephole optimization usually works by sliding a window of several instructions (a *peephole*)
- Peephole optimization is very fast
  - Small overhead per instruction since they use a small, fixed-size window
- It may be easier to generate naïve code and run peephole optimization than generating good code.

Peephole Optimization

Method:

1. Exam short sequences of target instructions
2. Replacing the sequence by **a more efficient one.**

Note. Often can be done on AST (DAG), or even can be done in the compiler frontend as a part of sematic evaluation!

- constant folding
- redundant-instruction elimination
- algebraic simplifications
- flow-of-control optimizations
- use of machine idioms, strength reduction

Peephole Optimization
Common Techniques

Elimination of redundant loads and stores

$r2 := r1 + 5$ $i := r2$ $r3 := i$ $r4 := r3 \times 3$	becomes	$r2 := r1 + 5$ $i := r2$ $r4 := r2 \times 3$

Constant folding

$r2 := 3 \times 2$ becomes $r2 := 6$

Peephole Optimization
Common Techniques

Constant propagation

r2 := 4 r3 := r1 + r2 r2 := ...	becomes	r2 := 4 r3 := r1 + 4 r2 := ...	and then	r3 := r1 + 4 r2 := ...
r2 := 4 r3 := r1 + r2 r3 := *r3	becomes	r3 := r1 + 4 r3 := *r3	and then	r3 := *(r1+4)
r1 := 3 r2 := r1 × 2	becomes	r1 := 3 r2 := 3 × 2	and then	r1 := 3 r2 := 6

Peephole Optimization
Common Techniques

Copy propagation

$r2 := r1$ $r3 := r1 + r2$ $r2 := 5$	becomes	$r2 := r1$ $r3 := r1 + r1$ $r2 := 5$	and then	$r3 := r1 + r1$ $r2 := 5$

Strength reduction

$r1 := r2 \times 2$	becomes	$r1 := r2 + r2$	or	$r1 := r2 << 1$
$r1 := r2 / 2$	becomes	$r1 := r2 >> 1$		
$r1 := r2 \times 0$	becomes	$r1 := 0$		

Peephole Optimization
Common Techniques

Elimination of useless instructions

$$r1 := r1 + 0$$

$$r1 := r1 \times 1$$

Algebraic identities

- Worth recognizing single instructions with a constant operand
  - Eliminate computations
    - A * 1 = A
    - A * 0 = 0
    - A / 1 = A
  - Reduce strenth to a more efficient instructions
    - A * 2 = A + A
    - A/2 = A * 0.5
- More delicate with floating-point

Note. Is this ever helpful?

- Why would anyone write X * 1?
- Why bother to correct such obvious junk code?
- In fact one might write

```
#define MAX_TASKS  1
...
a = b * MAX_TASKS;
```

- Also, seemingly redundant code can be produced by other optimizations.

Addition chains for multiplication

- If multiply is very slow (or on a machine with no multiply instruction like the original SPARC), decomposing a constant operand into sum of powers of two can be effective:
  - $X * 125 \quad = \quad x * 128 - x*4 + x$
  - two shifts, one subtract and one add, which may be faster than one multiply
  - Note similarity with efficient exponentiation method

Flow-of-control - jumps

```
    goto L1                         goto L2
     . . .                           . . .
L1: goto L2          →         L1: goto L2
```

```
    if a < b goto L1                if a < b goto L2
     . . .                           . . .
L1: goto L2          →         L1: goto L2
```

```
    goto L1                         if a < b goto L2
     . . .                          goto L3
L1: if a < b goto L2    →           . . .
L3:                            L3:
```

Peephole Opt: an Example

Source Code:

```
debug = 0
. . .
if(debug) {
   print debugging information
  }
```

Intermediate Code:

```
     debug = 0
     . . .
     if debug = 1 goto L1
     goto L2
L1:  print debugging information
L2:
```

Eliminate Jump after Jump

Before:

```
     debug = 0
     . . .
     if debug = 1 goto L1
     goto L2
L1:  print debugging information
L2:
```

After:

```
     debug = 0
     . . .
     if debug ≠ 1 goto L2
     print debugging information
L2:
```

Constant Propagation

Before:

```
    debug = 0
    . . .
    if debug ≠ 1 goto L2
    print debugging information
L2:
```

After:

```
    debug = 0
    . . .
    if 0 ≠ 1 goto L2
    print debugging information
L2:
```

Unreachable Code (dead code elimination)

Before:

```
    debug = 0
    . . .
    if 0 ≠ 1 goto L2
    print debugging information
L2:
```

After:

```
    debug = 0
    . . .
```

A detailed order of optimizations (from the book Muchnick: Advanced Compiler Design and Implementation)

Scalar replacement of array references
Data-cache optimizations

Procedure integration
Tail-call optimization
Scalar replacement of aggregates
Sparse conditional constant propagation
Interprocedural constant propagation
Procedure specialization and cloning
Sparse conditional constant propagation

Global value numbering
Local and global copy propagation
Sparse conditional constant propagation
Dead-code elimination
Local and global common-subexpression elimination
Loop-invariant code motion
Dead-code elimination
Code hoisting
Induction-variable strength reduction
Linear-function test replacement
Induction-variable removal
Unnecessary bounds-checking elimination
Control-flow optimizations

Constant folding
Algebraic simplification and reassociation

In-line expansion
Leaf-routine optimization
Shrink wrapping
Machine idioms
Tail merging
Branch optimizations and conditional moves
Dead-code elimination
Software pipelining, loop unrolling
Basic-block and branch scheduling
Register allocation
Basic-block and branch scheduling
Intraprocedural I-cache optimization
Instruction prefetching
Data prefetching
Branch prediction

Interprocedural register allocation
Aggregation of global references
Interprocedural I-cache optimization