UNIVERSITÄT
BERN

9. Optimization

Marcus Denker

Roadmap

- Introduction
- Optimizations in the Back-end
- The Optimizer
- SSA Optimizations
- Advanced Optimizations

Roadmap

- > **Introduction**
- > Optimizations in the Back-end
- > The Optimizer
- > SSA Optimizations
- > Advanced Optimizations

Optimization: The Idea

- Transform the program to improve efficiency

- **Performance**: faster execution
- **Size**: smaller executable, smaller memory footprint

Tradeoffs: 1) **Performance** vs**. Size**

2) **Compilation speed** and **memory**

No Magic Bullet!

- There is no perfect optimizer
- Example: optimize for simplicity

Opt(P): Smallest Program

Q: Program with no output, does not stop

Opt(Q)?

No Magic Bullet!

- There is no perfect optimizer
- Example: optimize for simplicity

Opt(P): Smallest Program

Q: Program with no output, does not stop

Opt(Q)?

```
L1 goto L1
```

No Magic Bullet!

- There is no perfect optimizer
- Example: optimize for simplicity

Opt(P): Smallest ProgramQ: Program with no output, does not stop

Opt(Q)?

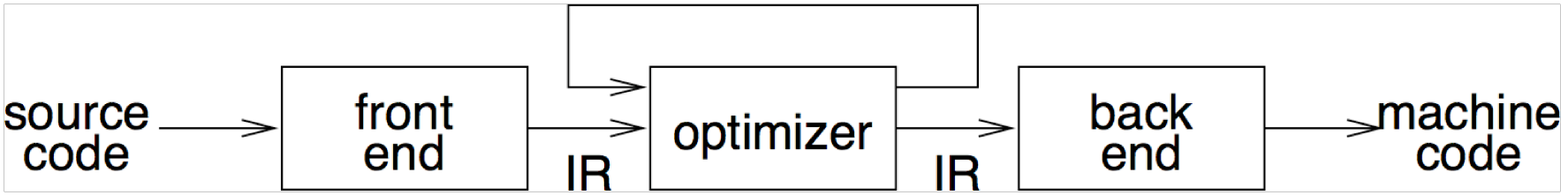
```
L1 goto L1
```

Halting problem!

Another way to look at it...

- Rice (1953): For every compiler there is a modified compiler that generates shorter code.
- Proof: Assume there is a compiler U that generates the shortest optimized program Opt(P) for all P.
  - Assume P to be a program that does not stop and has no output
  - Opt(P) will be L1 goto L1
  - Halting problem. Thus: U does not exist.
- There will be always a better optimizer!
  - Job guarantee for compiler architects :-)

Optimization on many levels

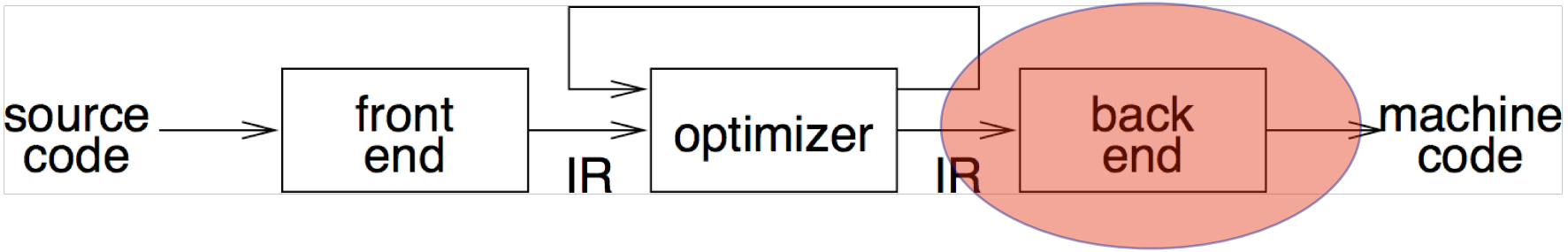


> Optimizations both in the optimizer and back-end

Roadmap

- Introduction
- **Optimizations in the Back-end**
- The Optimizer
- SSA Optimizations
- Advanced Optimizations

Optimizations in the Backend



- Register Allocation
- Instruction Selection
- Peep-hole Optimization

Register Allocation

- Processor has only finite amount of registers
  - Can be very small (x86)
- Temporary variables
  - non-overlapping temporaries can share one register
- Passing arguments via registers
- Optimizing register allocation very important for good performance
  - Especially on x86

Instruction Selection

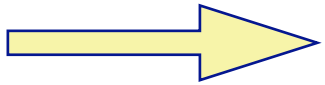
- For every expression, there are many ways to realize them for a processor
- Example: Multiplication*2 can be done by bit-shift

Instruction selection is a form of optimization

Peephole Optimization

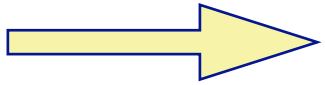
- Simple local optimization
- Look at code “through a hole”
  - replace sequences by known shorter ones
  - table pre-computed

```
store R,a;
load a,R
```



```
store R,a;
```

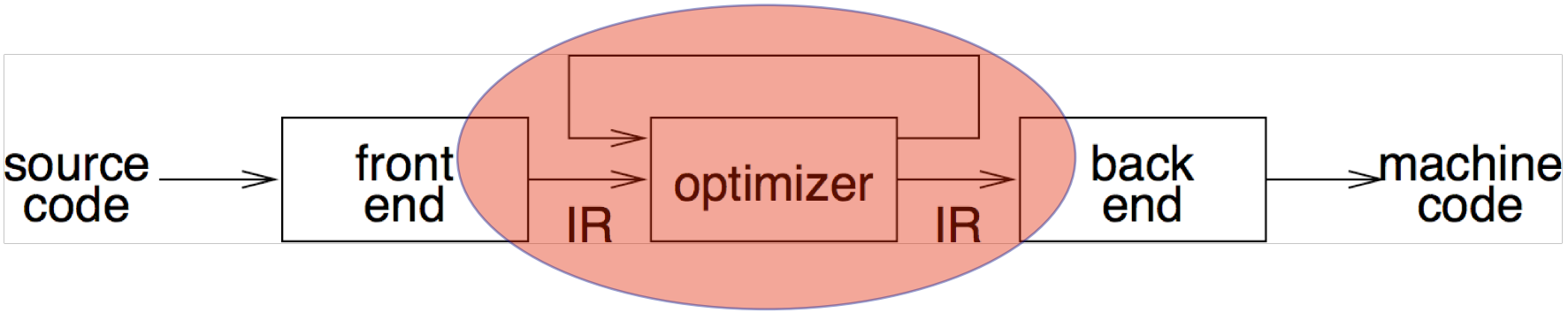
```
imul 2,R;
```



```
ashl 2,R;
```

Important when using simple instruction selection!

Optimization on many levels



Major work of optimization done in a special phase

Focus of this lecture

Different levels of IR

- Different levels of IR for different optimizations
- Example:
  - Array access as direct memory manipulation
  - We generate many simple to optimize integer expressions
- We focus on high-level optimizations

Roadmap

- Introduction
- Optimizations in the Back-end
- **The Optimizer**
- SSA Optimizations
- Advanced Optimizations

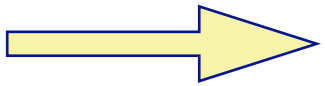
Examples for Optimizations

- Constant Folding / Propagation
- Copy Propagation
- Algebraic Simplifications
- Strength Reduction
- Dead Code Elimination
  - Structure Simplifications
- Loop Optimizations
- Partial Redundancy Elimination
- Code Inlining

Constant Folding

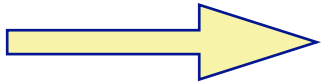
- Evaluate constant expressions at compile time
- Only possible when side-effect freeness guaranteed

```
c:= 1 + 3
```



```
c:= 4
```

```
true not
```



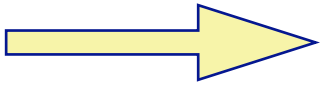
```
false
```

Caveat: Floats — implementation could be different between machines!

Constant Propagation

- Variables that have constant value, e.g. c := 3
  - Later uses of c can be replaced by the constant
  - If no change of c between!

```
b := 3
c := 1 + b
d := b + c
```



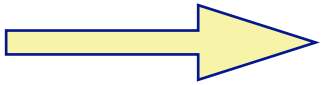
```
b := 3
c := 1 + 3
d := 3 + c
```

Analysis needed, as b can be assigned more than once!

Copy Propagation

- for a statement x := y
- replace later uses of x with y, if x and y have not been changed.

```
x := y
c := 1 + x
d := x + c
```

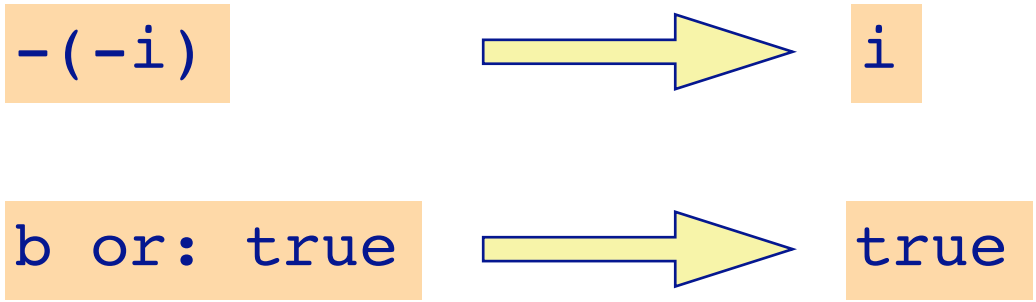


```
x := y
c := 1 + y
d := y + c
```

Analysis needed, as y and x can be assigned more than once!

Algebraic Simplifications

- Use algebraic properties to simplify expressions

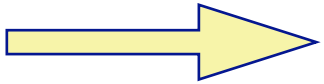


Important to simplify code for later optimizations

Strength Reduction

- Replace expensive operations with simpler ones
- Example: Multiplications replaced by additions

```
y := x * 2
```



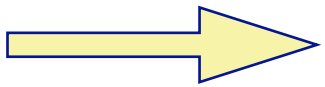
```
y := x + x
```

Peephole optimizations are often strength reductions

Dead Code

- Remove unnecessary code
  - e.g. variables assigned but never read

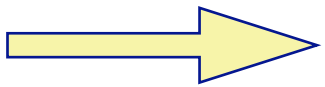
```
b := 3
c := 1 + 3
d := 3 + c
```



```
c := 1 + 3
d := 3 + c
```

- Remove code never reached

```
if (false)
{a := 5}
```

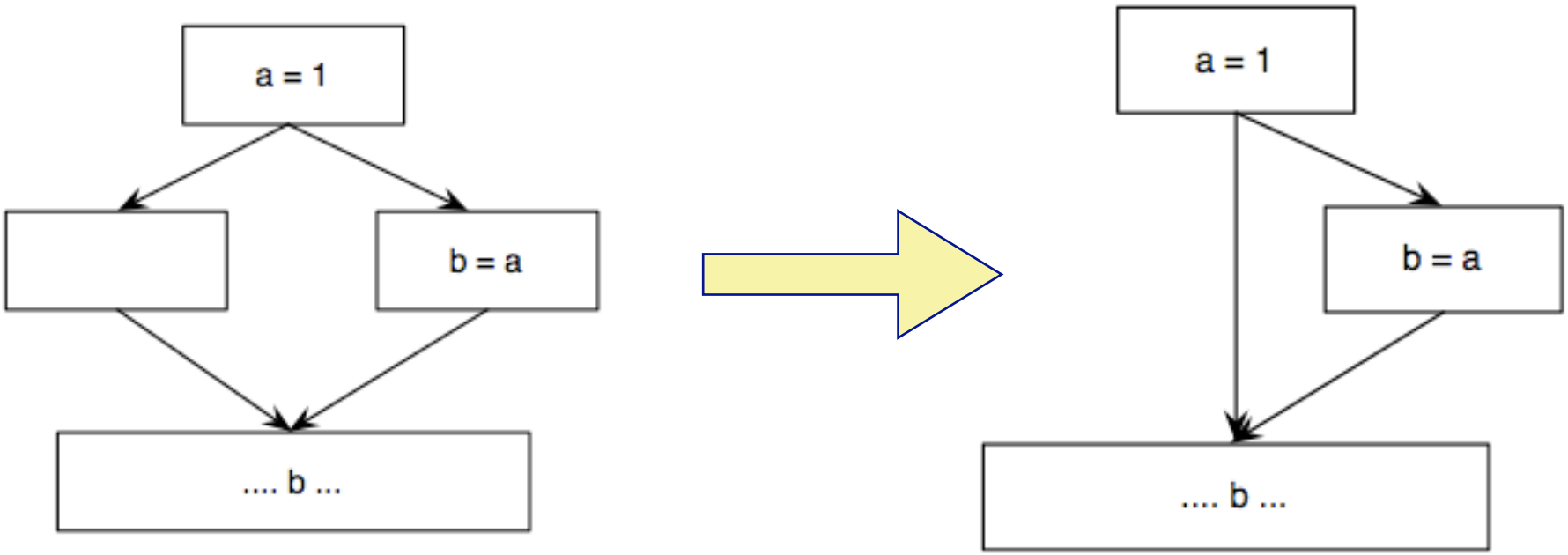


```
if (false)
{}
```

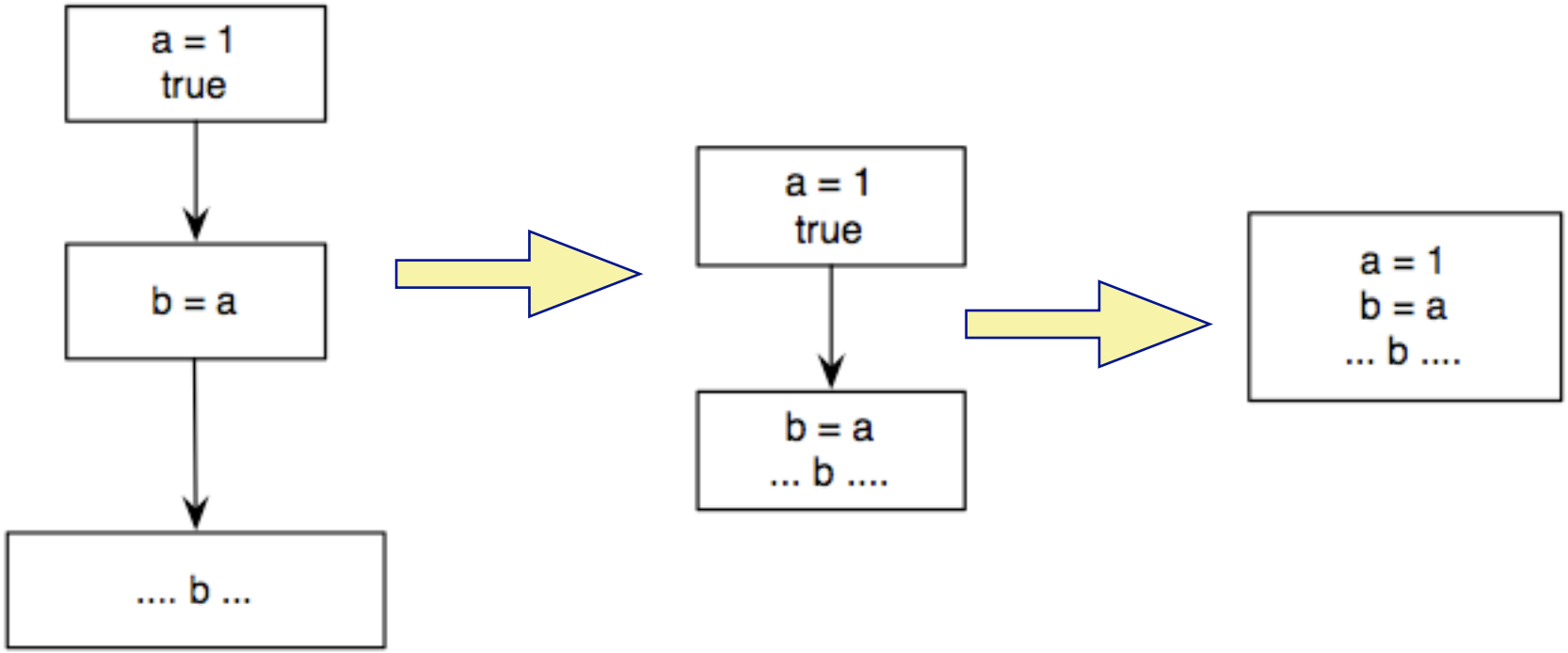
Simplify Structure

- Similar to dead code: Simplify CFG Structure
- Optimizations will degenerate CFG
- Needs to be cleaned to simplify further optimization!

Delete Empty Basic Blocks



Fuse Basic Blocks



Common Subexpression Elimination (CSE)

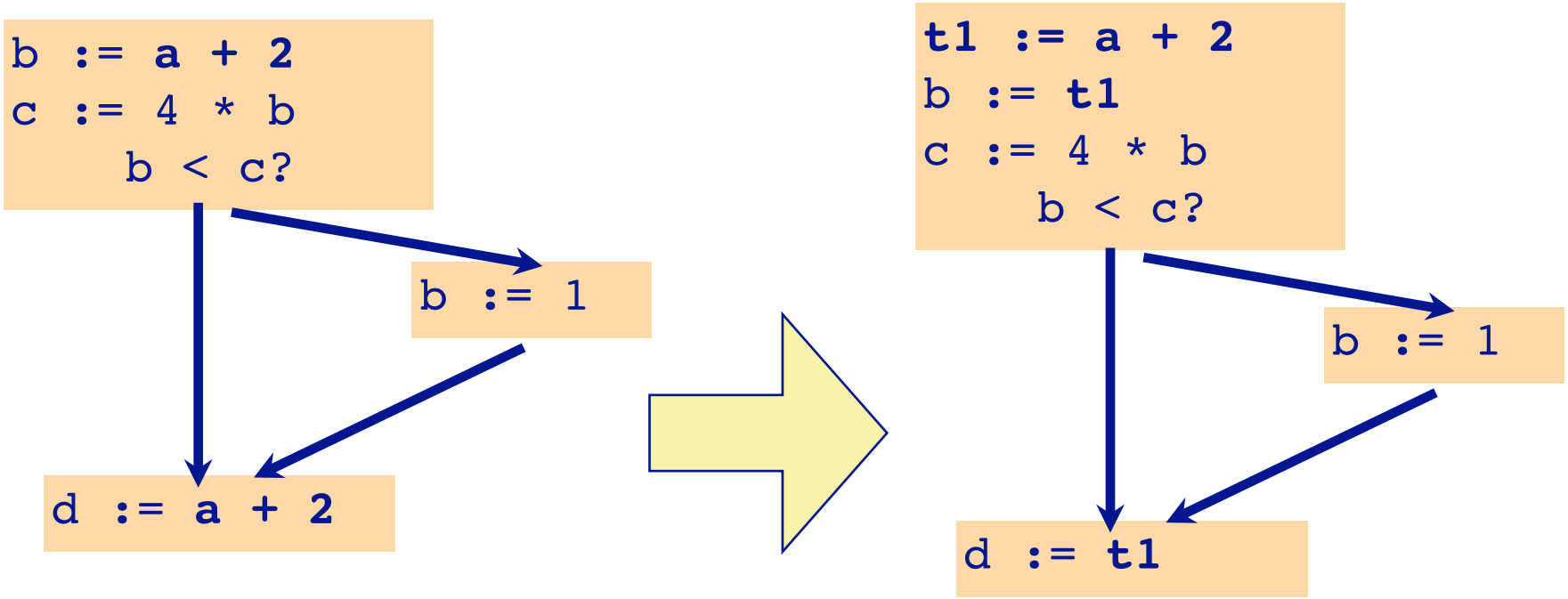
Common Subexpression:

- There is another occurrence of the expression whose evaluation always precedes this one
- operands remain unchanged

Local (inside one basic block): When building IR

Global (complete flow-graph)

Example CSE

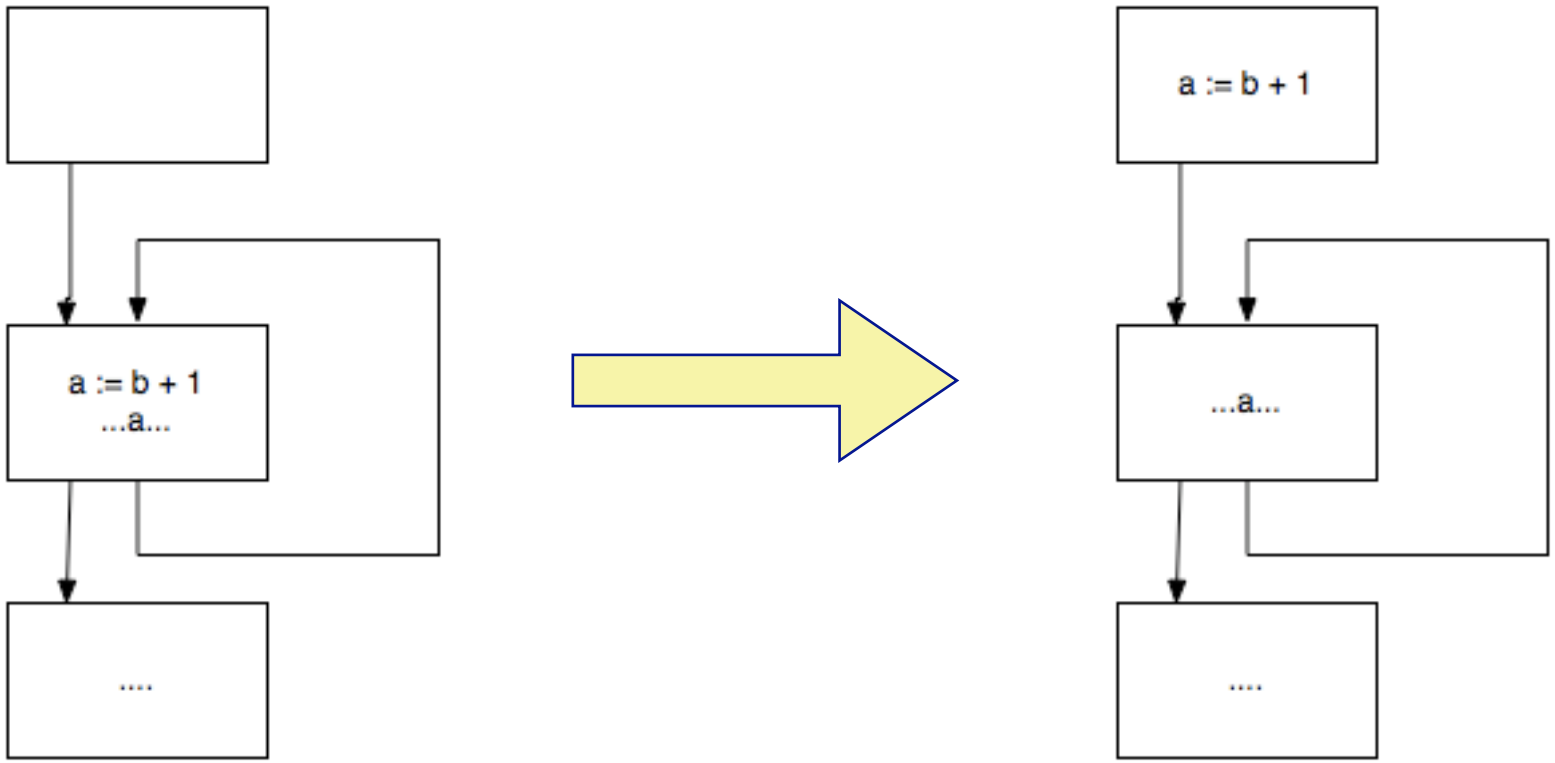


Loop Optimizations

- Optimizing code in loops is important
  - often executed, large payoff
- All optimizations help when applied to loop-bodies
- Some optimizations are loop specific

Loop Invariant Code Motion

- Move expressions that are constant over all iterations out of the loop

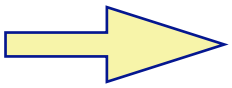


Induction Variable Optimizations

- Values of variables form an arithmetic progression

```
integer a(100)
do i = 1, 100
 a(i) = 202 - 2 * i
endo
```

value assigned to *a* decreases by 2



```
integer a(100)
t1 := 202
do i = 1, 100
  t1 := t1 - 2
  a(i) = t1
endo
```

uses *Strength Reduction*

Partial Redundancy Elimination (PRE)

- Combines multiple optimizations:
  - global common-subexpression elimination
  - loop-invariant code motion
- **Partial Redundancy:** computation done more than once on some path in the flow-graph
- PRE: insert and delete code to minimize redundancy.

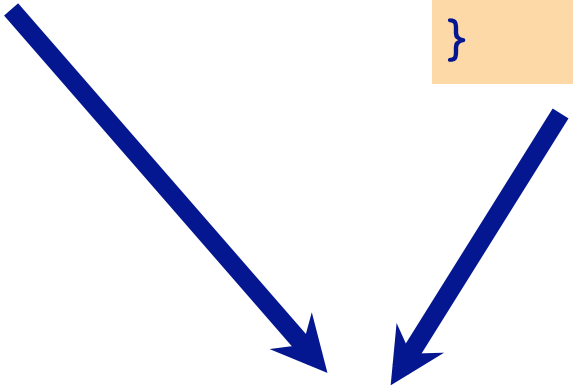
Code Inlining

- All optimization up to know where local to one procedure
- Problem: procedures or functions are very short
  - Especially in good OO code!
- Solution: Copy code of small procedures into the caller
  - OO: Polymorphic calls. Which method is called?

Example: Inlining

```
a := power2(b)
```

```
power2(x) {
    return x*x
}
```



```
a := b * b
```

Roadmap

- Introduction
- Optimizations in the Back-end
- The Optimizer
- **SSA Optimizations**
- Advanced Optimizations

Repeat: SSA

- SSA: Static Single Assignment Form
- **Definition**: Every variable is only assigned once

Properties

- Definitions of variables (assignments) have a list of all uses
- Variable uses (reads) point to the one definition
- CFG of Basic Blocks

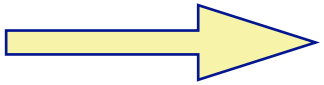
Examples: Optimization on SSA

- We take three simple ones:
  - Constant Propagation
  - Copy Propagation
  - Simple Dead Code Elimination

Repeat: Constant Propagation

- Variables that have constant value, e.g. c := 3
  - Later uses of c can be replaced by the constant
  - If no change of c between!

```
b := 3
c := 1 + b
d := b + c
```



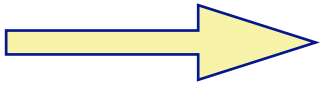
```
b := 3
c := 1 + 3
d := 3 + c
```

Analysis needed, as b can be assigned more than once!

Constant Propagation and SSA

- Variables are assigned once
- We know that we can replace all uses by the constant!

```
b1 := 3
c1 := 1 + b1
d1 := b1 + c1
```

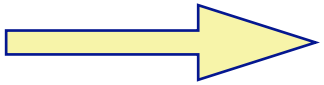


```
b1 := 3
c1 := 1 + 3
d1 := 3 + c
```

Repeat: Copy Propagation

- for a statement x := y
- replace later uses of x with y, if x and y have not been changed.

```
x := y
c := 1 + y
d := y + c
```



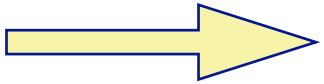
```
x := y
c := 1 + y
d := y + c
```

Analysis needed, as y and x can be assigned more than once!

Copy Propagation And SSA

- for a statement x1 := y1
- replace later uses of x1 with y1

```
x1 := y1
c1 := 1 + x1
d1 := x1 + c1
```



```
x1 := y1
c1 := 1 + y1
d1 := y1 + c1
```

Dead Code Elimination and SSA

- Variable is live if the list of uses is not empty.
- Dead definitions can be deleted
  - (If there is no side-effect)

Roadmap

- Introduction
- Optimizations in the Back-end
- The Optimizer
- SSA Optimizations
- **Advanced Optimizations**

Advanced Optimizations

- Optimizing for using multiple processors
  - Auto parallelization
  - Very active area of research (again)
- Inter-procedural optimizations
  - Global view, not just one procedure
- Profile-guided optimization
- Vectorization
- Dynamic optimization
  - Used in virtual machines (both hardware and language VM)

Iterative Process

- There is no general “right” order of optimizations
- One optimization generates new opportunities for a preceding one.
- Optimization is an iterative process

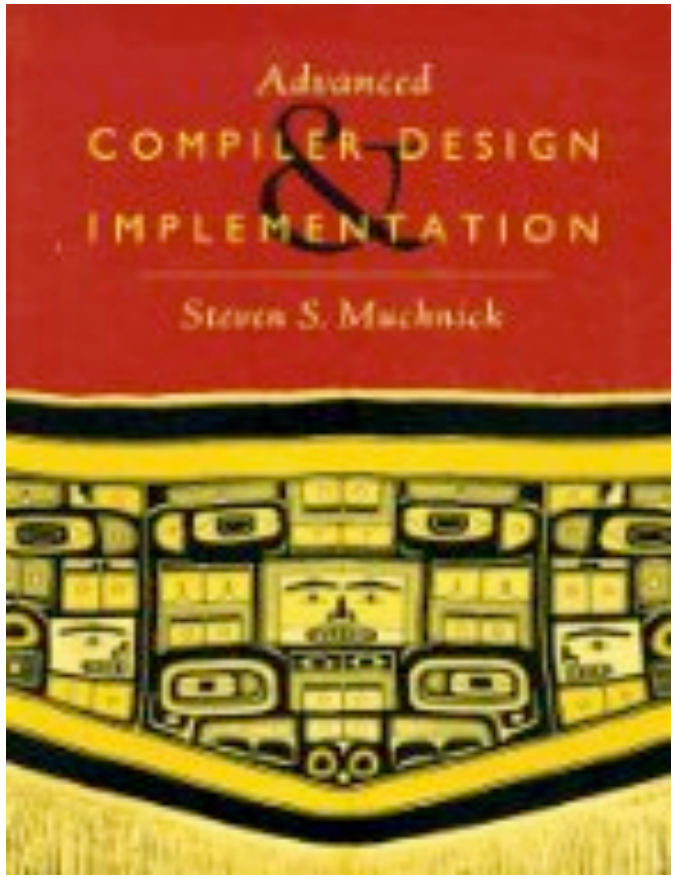
Compile Time vs. **Code Quality**

What we have seen...

- Introduction
- Optimizations in the Back-end
- The Optimizer
- SSA Optimizations
- Advanced Optimizations

Literature

- Muchnick: *Advanced Compiler Design and Implementation*
  - *>600 pages on optimizations*
- Appel: *Modern Compiler Implementation in Java*
  - *The basics*



What you should know!

- *Why do we optimize programs?*
- *Is there an optimal optimizer?*
- *Where in a compiler does optimization happen?*
- *Can you explain constant propagation?*

Can you answer these questions?

- *What makes SSA suitable for optimization?*
- *When is a definition of a variable live in SSA Form?*
- *Why don't we just optimize on the AST?*
- *Why do we need to optimize IR on different levels?*
- *In which order do we run the different optimizations?*

License

> http://creativecommons.org/licenses/by-sa/2.5/

creative commons

COMMONS DEED

Attribution-ShareAlike 2.5

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.