TU WIEN Informatics

Advanced Computer Architecture

B2: Static Code Analysis and Optimization

Daniel Mueller-Gritschneder

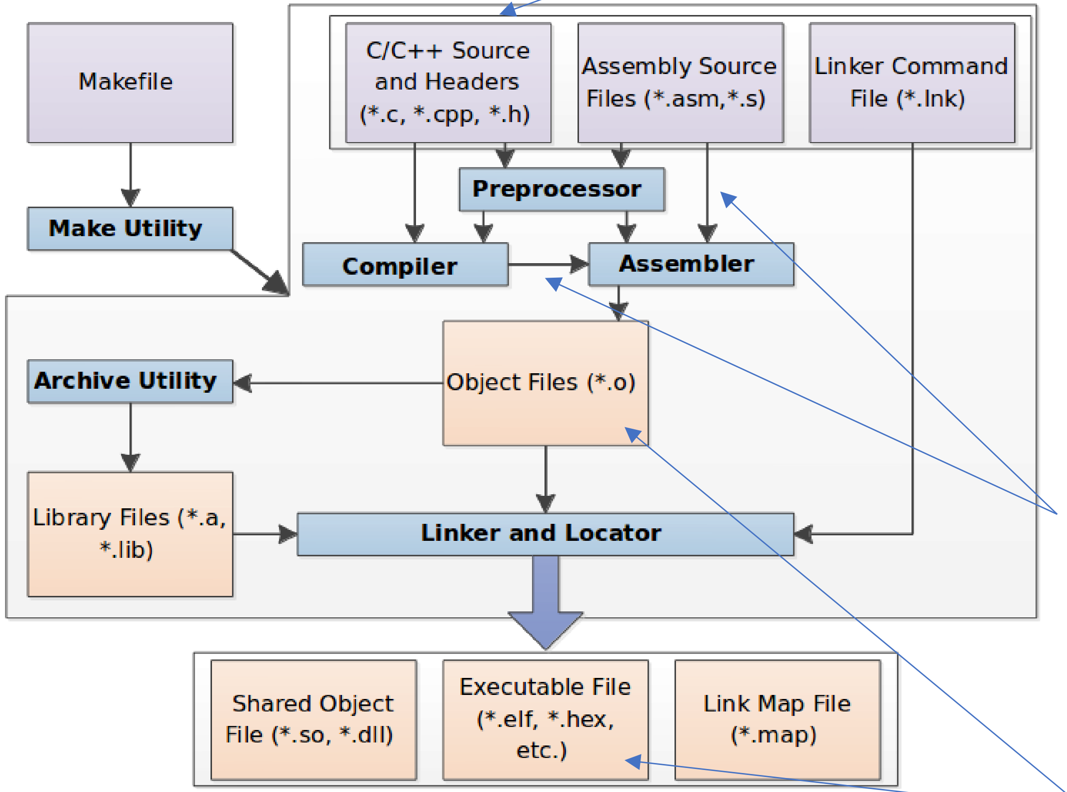
Agenda

- Control and Data Flow
- Code Optimization
- Live Variable Analysis

B2-0 Recap - Compiler Basics

Compilation

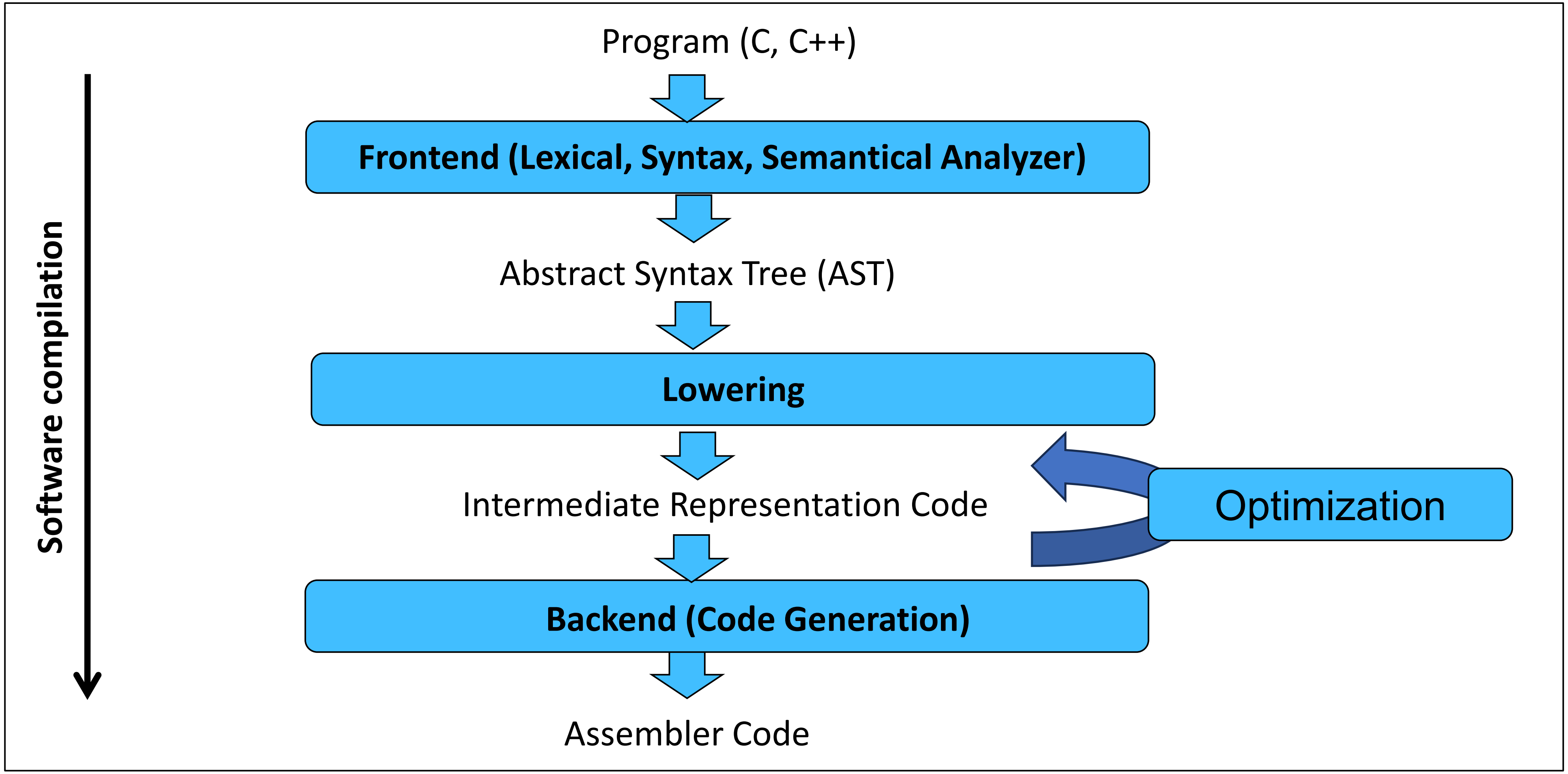
C-code:
`val1=val1+4;`



Assembly-code:
`ADDI x10,x10,4`

Machine code:
`0x00450513`

Compilation Flow



B2-1 Control and Data Flow Analysis

Running Example - Goertzel-Algorithm in C

- Computes power spectrum for one frequency component.

- C-Code:

```c
float goertzel (float freq, const  float x[])
{
int i;
float coeff, s, s_prev1 ,s_prev2, power;
s_prev1 = 0.0;
s_prev2 = 0.0;
coeff = 2.0 * cos(2.0 * 3.14 * freq);
for (i = 0; i < 64; i++) {
        s = x[i] + (coeff * s_prev1) - s_prev2;
        s_prev2 = s_prev1;
        s_prev1 = s;
}
power = (s_prev1*s_prev1) + (s_prev2*s_prev2) - (s_prev1*s_prev2*coeff);
return power;
}
```

Basic Blocks

- A basic block is a maximal sequence of instructions with:
  - no jump target labels (except at the first instruction), and
  - no jumps (except in the last instruction)

- Idea:
  - Cannot jump into a basic block (except at beginning)
  - Cannot jump out of a basic block (except at end)
  - A basic block is a single-entry, single-exit, straight-line code segment

Control Flow Graph (CFG) (1/2)

- Control flow graph: $G_c(V, E)$

- Nodes are basic blocks of the algorithm

$$V = \{B_i : i = 1, ..., n_B\}$$

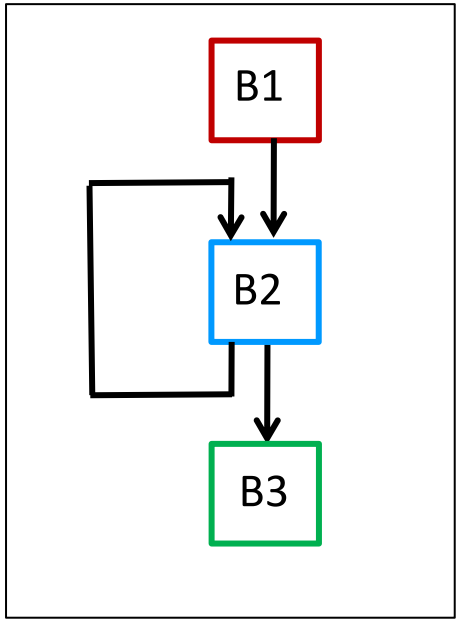
- Edges: Next possible basic blocks
  - Branches in the CFG: Conditional constructs
  - Cycles in the CFG: Loop constructs

$$E = \{(B_i, B_j) : i, j = 1, ..., n_B\}$$

- Alternative paths in the CFG describe **alternative** flow of control (only one of them is executed!)

Control Flow Graph (CFG) (2/2)

- Example 1: Goertzel Algorithm (IR Code)




```
B1: s_prev1 := 0.0
    s_prev2 := 0.0
    i:=0
    t1  := 2*3.14
    f := t1 * freq
    param f
    t2 := call cos,1
    coeff:=2.0*t2
```

```
B2: t3:= coeff * s_prev1
    t4:= x[i]
    t5  := t4 - s_prev2
    s  := t3 + t5
    s_prev2  := s_prev1
    s_prev1  := s
    i:=i+1
    if i < 64 goto B2
```

```
B3: t6:= s_prev1 * s_prev1
    t7:= s_prev2 * s_prev2
    t8:= s_prev1 * s_prev2
    t9:= t8 * coeff
    t10:= t6+t7
    power:= t10 - t9
    return power
```

Data Flow Graph (DFG) (1/3)

- DFG for basic block Bx: $G_{d,B1}(V, E)$
- DFG is directed acyclic graph
- Nodes: Operations op_i inside one basic block Bx
- Number of Operations: $n_{d,Bx}$

$$V = \{op_i : i = 1, ..., n_{d,Bx}\}$$

- Edges describe data dependencies between operations

$$E = \{(op_i, op_j) : i, j = 1, ..., n_{d,Bx}\}$$

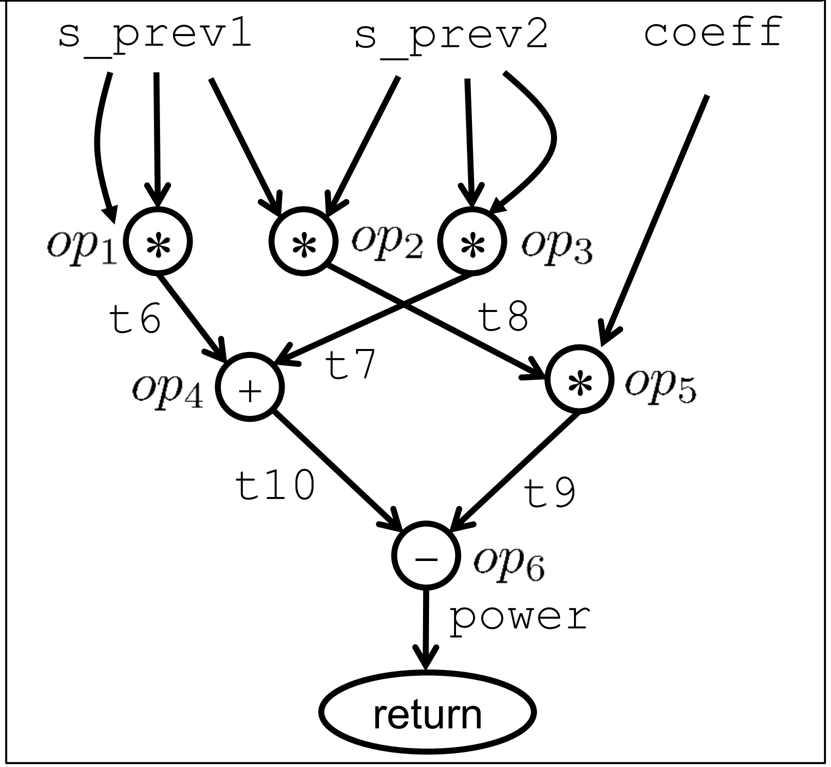
- Paths in the DFG describe **concurrent** operations, that may be executed in **parallel**.

Basic block B3 of the Goertzel algorithm:

```
power =  (s_prev1*s_prev1) + (s_prev2*s_prev2) - (s_prev1*s_prev2*coeff);
return power;
```

Three address code:

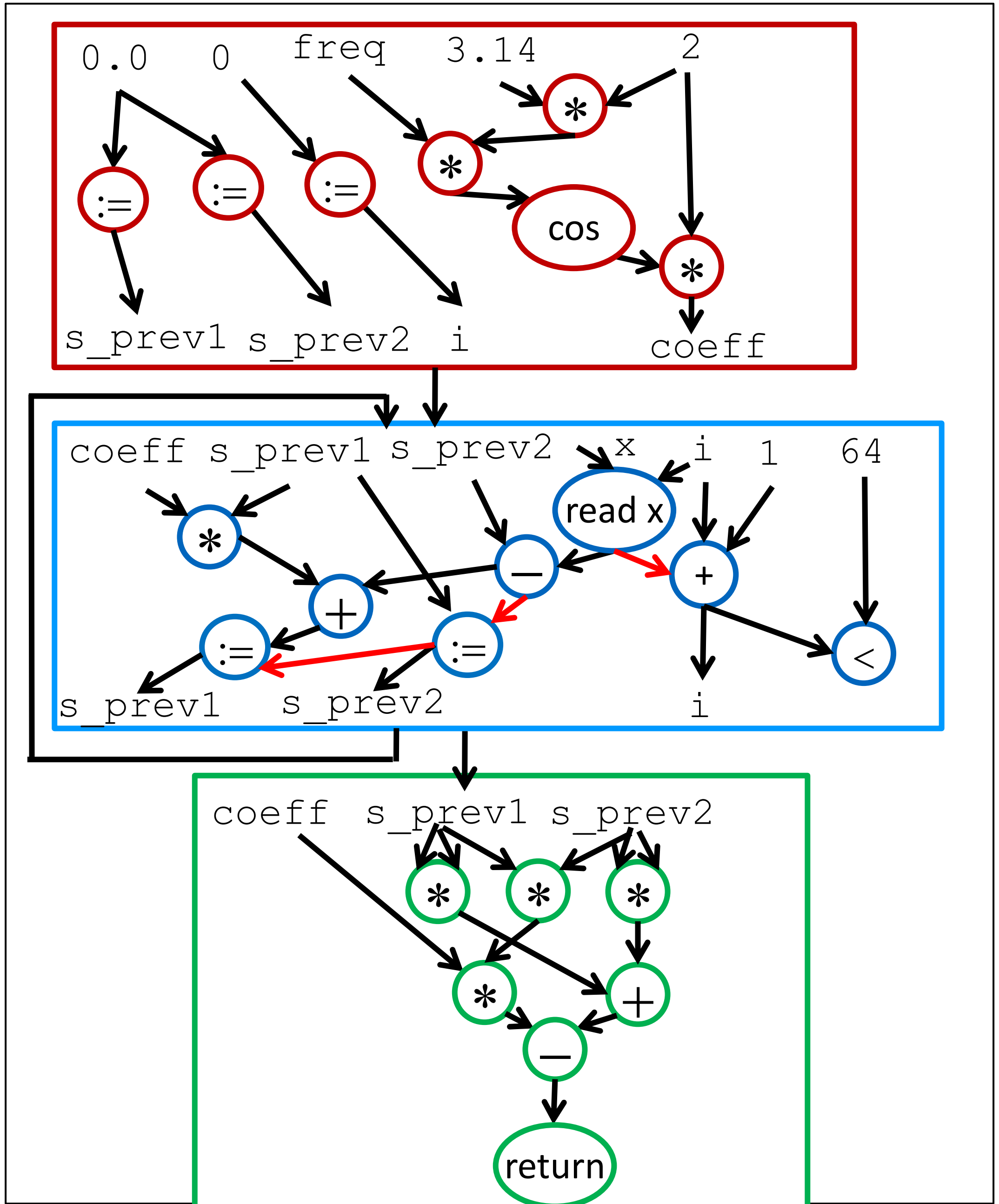
```
B3: t6:=s_prev1 * s_prev1
    t7:=s_prev2 * s_prev2
    t8:=s_prev1 * s_prev2
    t9:=t8 * coeff
    t10:=t6+t7
    power:=t10 - t9
    return power
```



Control Data Flow Graph (CDFG) (1/2)

- Hierarchical graph: Each node of CFG holds DFG of basic block.
- CDFG: $G_{c,d}(V, E)$
- Paths for edges between basic blocks: Alternative execution of control flow
- Path for edges inside one basic block: Concurrency for possible parallel execution of operations

Control Data Flow Graph (CDFG) (2/2)




```
B1: s_prev1 := 0.0
    s_prev2 := 0.0
    i:=0
    t1 := 2*3.14
    f := t1 * freq
    param f
    t2 := call cos,1
    coeff:=2.0*t2
```

```
B2: t3:= coeff * s_prev1
    t4:= x[i]
    t5 := t4 - s_prev2
    s := t3 + t5
    s_prev2 := s_prev1
    s_prev1 := s
    i:=i+1
    if i < 64 goto B2
```

```
B3: t6:= s_prev1 * s_prev1
    t7:= s_prev2 * s_prev2
    t8:= s_prev1 * s_prev2
    t9:= t8 * coeff
    t10:= t6+t7
    power:= t10 - t9
    return power
```

B2-2 Code Optimization Steps

Local Code Optimization Techniques

- Local optimization techniques take into consideration only one single basic block:
- Common techniques:
  - Common subexpression elimination
  - Dead code elimination
  - Arithmetic identities
  - Constant folding (propagation)
  - Strength reduction
  - Tree height reduction

Common Subexpression Elimination

- Two instructions execute same operation on the same operands.
- One operation can be replaced by a copy statement.
- SSA form shows common subexpressions

Three address code:

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

Optimized Three address code:

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

SSA:

```
a$1 := b$1 + c$1
b$2 := a$1 - d$1
c$2 := b$2 + c$1
d$2 := a$1 - d$1
```

Optimized SSA:

```
a$1 := b$1 + c$1
b$2 := a$1 - d$1
c$2 := b$2 + c$1
d$2 := b$2
```

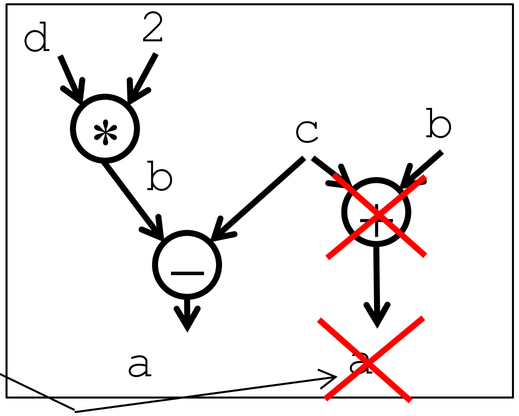
- Live variable: Value of variable is used as program output or input to other operation.
- Lifetime of variable determined by live variable analysis.
- Delete any node from DFG that has no live variable attached to one of its leaving edges.

Three address code:

```
a := b + c
b := 2*d
a := b - c
```

SSA:

```
a$1 := b$1 + c
b$2 := 2*d
a$2 := b$2 - c
```



a$1 dead because value rewritten

Optimized Three address code:

```
b := 2*d
a := b - c
```

Optimized SSA:

```
b$2 := 2*d
a$2 := b$2 - c
```

Algebraic Identities

- Use algebraic identities to reduce number of operations

Three address code:

```
a:=b+0
c:=d*1
f:=g*0
h:=i*1
j:=k/1
```

Optimized Three address code:

```
a:=b
c:=d
f:=0
h:=i
j:=k
```

Three address code:

```
t1:=a*a
t2:=b*b
t3:=a*b
t4:=2*t3
t5:=t1+t2
c:=t4+t5
```

Optimized Three address code:

```
t1:=a+b
c:=t1*t1
```

Strength Reduction

- Replace operation with equivalent operation that is cheaper to execute in hardware.

Three address code:

```
x:=2*y
v:=w/2
```

Optimized Three address code:

```
x:=y+y
v:=w*0.5
```

Optimized Three address code:

```
x:=y<<1
v:=w>>1
```

Division only if `y` and `w` are unsigned type or signed positive

Constant folding (propagation)

- Operations on two constant values can be computed on compilation time.
- Example: First basic block of Goertzel algorithm

```
B1: s_prev1 := 0.0
    s_prev2 := 0.0
    i:=0
    t1 := 2*3.14
    f := t1 * freq
    param f
    t2 := call cos,1
    coeff:=2.0*t2
```

```
B1: s_prev1 := 0.0
    s_prev2 := 0.0
    i:=0
    f := 6.28 * freq
    param f
    t2 := call cos,1
    coeff:=2.0*t2
```

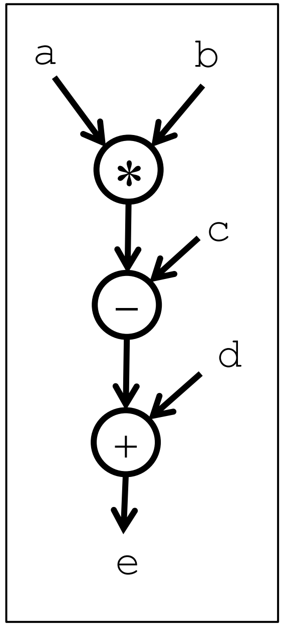
Tree Height Reduction

- Increase possible concurrency by avoiding data dependencies.
- Increases possibilities for parallel execution in hardware implementations or on multi-issue processors.

```
e:=d+(c-a*b)
```

```
t1:= a * b
t2:= c - t1
e:= d + t2
```

No parallel execution due to data dependencies.

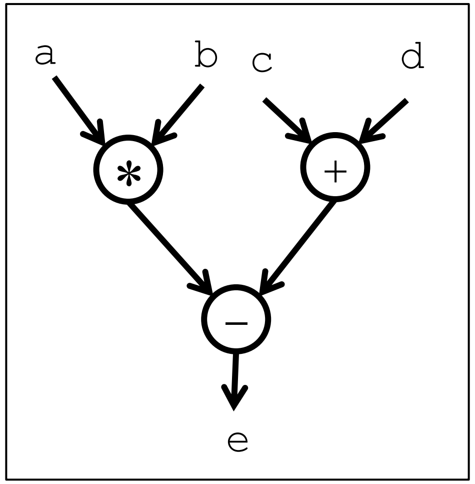


Tree height = 3.

```
e:=(d+c)-a*b
```

```
t1:= a * b
t2:= d + c
e:= t2 – t1
```

Parallel execution of multiplication and addition possible.



Tree height = 2.

Global Code Optimization Techniques

- Global optimization techniques optimize the code by considering more than one basic block
- Assumption: Program spends most time in most inner loops.
- Common techniques:
  - Global common subexpression elimination
  - Global dead code elimination
  - Code Motion
  - Induction variable reduction
  - Loop Unrolling

Code Motion

- Move statements that always compute same value in each loop iteration out of loop body.

Three address code:

```
      i:=0
      c:=0
B2:   c:=c+a
      c:=c+b
      i:=i+1
      if i < 16 goto B2
```

Optimized three address code:

```
      i:=0
      c:=0
      t1:=a+b
B2:   c:=c+t1
      i:=i+1
      if i < 16 goto B2
```

Induction Variable Reduction

- Induction variables change by constant value in each iteration of loop.
- Apply strength reduction and common subexpression elimination on induction variables.

Three address code:

```
i:=0
B2:  j:=i*4
     k:=i*4
     t1:=x[j]
     t2:=k-1
     t3:=c[t2]
     t4:=t1+t3
     c[k]:= t4
     i:=i+1
     if i < 16 goto B2
```

Optimized three address code:

```
i:=0
j:=0
k:=j
B2:  t1:=x[j]
     t2:=k-1
     t3:=c[t2]
     t4:=t1+t3
     c[k]:= t4
     j:=j+4
     k:=j
     i:=i+1
     if i < 16 goto B2
```

Loop classification

- **Do-all loops**: No data dependencies between loop iterations.

```
i=0;
while (i<4){
          c[i]=a[i]+b[i];
          i++;
}
```

```
     i:=0
B2:  t1:=a[i]
     t2:=b[i]
     t3:=t1+t2
     c[i]:=t3
     i:=i+1
     if i < 4 goto B2
```

- **Do-across loops**: There exist possible data dependencies between loop iterations.

```
i=0;
c=0;
while (i<4){
          c=c-b[i];
          i++;
}
```

```
     i:=0
     c:=0
B2:  t1:=b[i]
     c:=c-t1
     i:=i+1
     if i < 4 goto B2
```

Loop Unrolling

- Loop unrolling executes several loop iterations in one single iteration of optimized loop.
- Unroll factor: Number of non-optimized loop iterations executed in one iteration of optimized loop.

```
     i:=0
B2:  t1:=a[i]
     t2:=b[i]
     t3:=t1+t2
     c[i]:=t3
     i:=i+1
     if i < 4 goto B2
```

```
     i:=0
B2:  t1:=a[i]
     t2:=b[i]
     t3:=t1+t2
     c[i]:=t3
     i:=i+1
     t4:=a[i]
     t5:=b[i]
     t6:=t4+t5
     c[i]:=t6
     i:=i+1
     if i < 4 goto B2
```

Unroll factor

```
t1:=a[0]
t2:=b[0]
t3:=t1+t2
c[0]:=t3
t1:=a[1]
t2:=b[1]
t3:=t1+t2
c[1]:=t3
t1:=a[2]
t2:=b[2]
t3:=t1+t2
c[2]:=t3
t1:=a[3]
t2:=b[3]
t3:=t1+t2
c[3]:=t3
```

Unroll factor 4 (fully unrolled)

B2-3 Live Variable Analysis

Data-flow Analysis

- Live Variable Analysis is a data-flow analysis.
- Data-flow analysis determines data-flow values at each point in the program
- Data-flow value:
  - The program state we are interested in
  - For Live Variable Analysis: Is a variable live?
- Set of data-flow values before Intermediate representation (IR) statement s_i: $IN[s_i]$
- Set of data-flow values after Intermediate representation (IR) statement s_i: $OUT[s_i]$

Data-flow Transfer Functions

- Each IR statement applies a transfer function on the data flow values:
  - Forward flow analysis: $OUT[s_i] = f_{s,i}(IN[s_i])$
  - Backward flow analysis: $IN[s_i] = f_{s,i}(OUT[s_i])$
- The transfer function of a complete basic block B_x is the composition of the transfer functions of all IR statements inside the basic block:

$$f_{B,x} = f_{s,1} \circ f_{s,2} \circ ... \circ f_{s,n}$$

  - Forward flow problem: $OUT[B_x] = f_{B,x}(IN[B_x])$
  - Backward flow problem: $IN[B_x] = f_{B,x}(OUT[B_x])$

Control Flow Constraints

- Flow of control places some constraints on the data-flow values.
- Within one basic block, the data flow values after an IR statement are the same as the data-flow values before the next IR statement (same point in the program):

$$OUT[s_i] = IN[s_{i+1}]$$

- Between basic blocks the control flow constraints are:
  - Forward flow problem: The data flow values at the entry to the basic block are the union of the data-flow values at the end of all its predecessor basic blocks P_x :

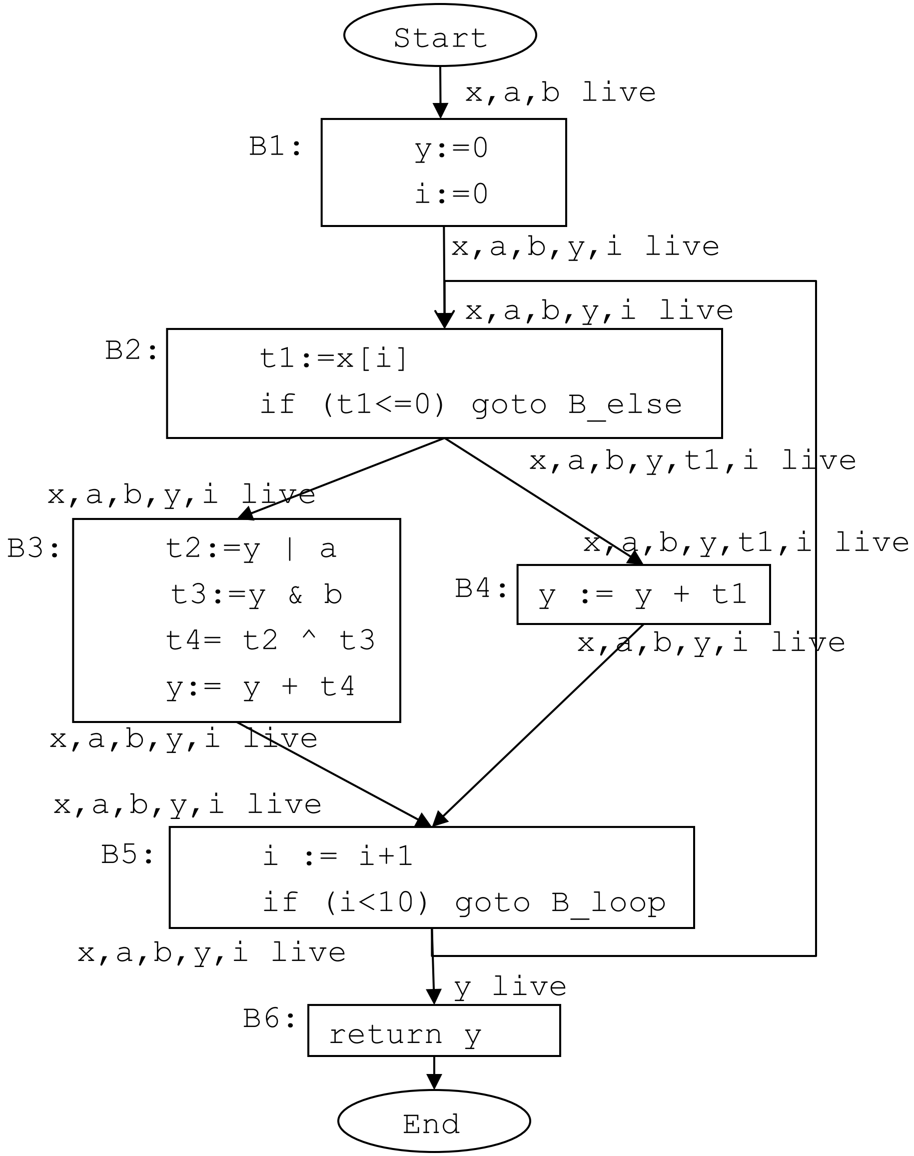
$$IN[B_x] = \bigcup_{P_x} Out[P_x]$$

  - Backward flow problem: The data flow values at the end to the basic block are the union of the data-flow values at the entry of all its successor basic blocks S_x :

$$OUT[B_x] = \bigcup_{S_x} IN[S_x]$$

Variable Liveliness Analysis

- Variables are live from generation until last use as operand.
- For each basic block:
  - Variables live at entry of basic block:
    - All variables used as operands in this basic block, which values are not computed before use inside this basic block.
    - The variables live at end of basic block, which values are not computed inside the basic block.
  - Variables life at end of basic block are the union of all variables live at entry to all its successor basic blocks.



Sets for Live Variable Analysis

- Variables live at entry to basic block B_x:

$$IN[B_x]$$

- Variables live at end of basic block B_x:

$$OUT[B_x]$$

- Set of variables defined (definitely assigned a value) in basic block B_x prior to any use of that variable in B_x

$$DEF[B_x]$$

- Set of variables whose values may be used in B_x prior to any B_x definition of the variable:

$$USE[B_x]$$

Conditions for Live Variable Analysis

- Live Variable Analysis is a backward flow analysis
- **Transfer function**: $IN[B_x] = USE[B_x] \cup (OUT[B_x] - DEF[B_x])$
- **Control flow constraint**: $OUT[B_x] = \bigcup_{S_x} IN[S_x]$

  S_x are all successor basic blocks of B_x
- **Boundary condition**: No variables live at end of function

$$IN[END] = \{\}$$

- Task: Find all sets $IN[B_x]$ and $OUT[B_x]$ such that all three conditions are satisfied.

Algorithm for Live Analysis

- Iterative flow-analysis algorithm:

```
Live_Variable_Analysis(CFG,def[Bx],use[Bx])
{
   for each basic block B_x
        IN[B_x] = {}
   while (changes to any IN[B_x] occur)
   {
        for each basic block B_x other than end
        {
           OUT[B_x] = Union of IN[S_x], S_x all successors of B_x
           IN[B_x] = Union(use[B_x],(OUT[B_x] - def[B_x]));
         }
    }
    return IN[B_x],OUT[B_x]
}
```

Why variable lifetimes are important?

- The compiler does not need to assign registers to all variables during their full lifetime (from generation to last use).
  - We usually only need them in registers when they are generated or used (load-store architectures).
  - But: Keeping them in registers is beneficial to avoid load and store operations.

- The number of required registers (*register pressure*) depend on the number of variables, which are live at a certain point in a program (conflicting variables).

- Registers can be reused in case that the lifetime of variables does not overlap (non-conflicting variables).

- If the compiler runs out of registers, he needs to store values on the stack (register spilling).

Conclusion

- Control and Data Flow
- Code Optimization
- Live Variable Analysis