University of Amsterdam

Introduction to Compiler Design: optimization and backend issues

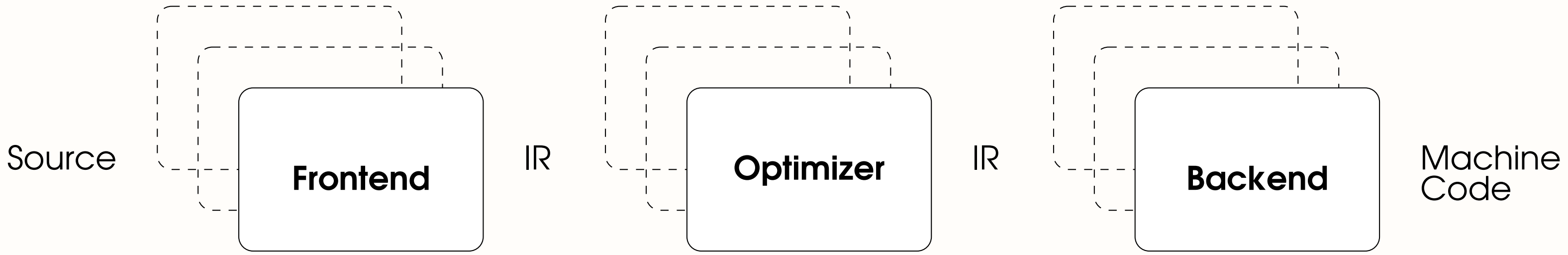
Andy Pimentel

Computer Systems Architecture group
andy@science.uva.nl

CSA
Computer Systems Architecture

University of Amsterdam

Compilers: Organization Revisited



- Optimizer
  - Independent part of compiler
  - Different optimizations possible
  - IR to IR translation

CSA
Computer Systems Architecture

Intermediate Representation (IR)

- Flow graph
  - Nodes are basic blocks
    - Basic blocks are single entry and single exit
  - Edges represent control-flow
- Abstract Machine Code
  - Including the notion of functions and procedures
- Symbol table(s) keep track of scope and binding information about names

University of Amsterdam

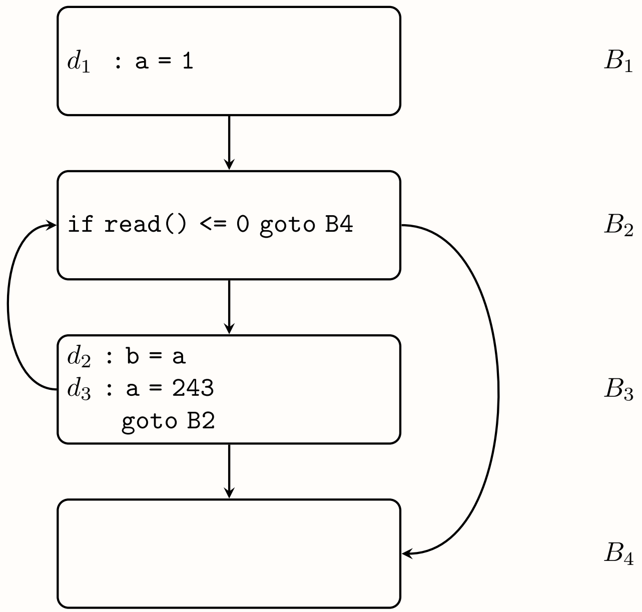
Partitioning into basic blocks

1. Determine the leaders, which are:
   - The first statement
   - Any statement that is the target of a jump
   - Any statement that immediately follows a jump
2. For each leader its basic block consists of the leader and all statements up to but not including the next leader

CSA
Computer Systems Architecture

University of Amsterdam

Partitioning into basic blocks (cont'd)



CSA
Computer Systems Architecture

Intermediate Representation (cont'd)

Structure within a basic block:

- Abstract Syntax Tree (AST)
  - Leaves are labeled by variable names or constants
  - Interior nodes are labeled by an operator
- Directed Acyclic Graph (DAG)
- C-like
- 3 address statements (like we have already seen)

Directed Acyclic Graph

- Like ASTs:
  - Leaves are labeled by variable names or constants
  - Interior nodes are labeled by an operator
- Nodes can have variable names attached that contain the value of that expression
- Common subexpressions are represented by multiple edges to the same expression

DAG creation

Suppose the following three address statements:

1. $x = y \text{ op } z$
2. $x = \text{op } y$
3. $x = y$

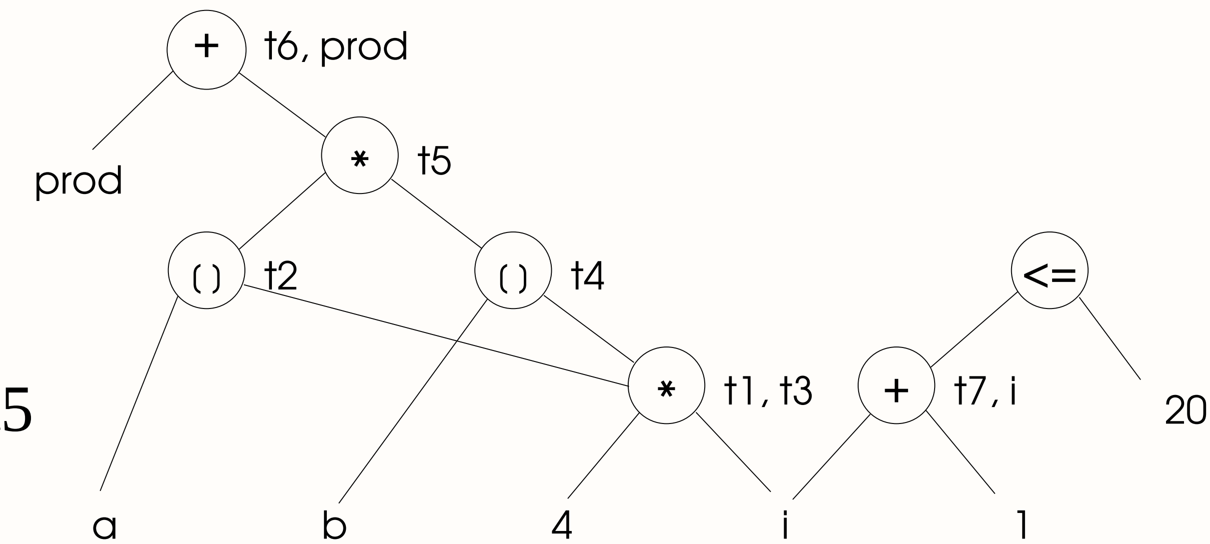
$if(i <= 20)$ ... will be treated like case 1 with x undefined

DAG creation (cont'd)

- If $node(y)$ is undefined, create leaf labeled y, same for z if applicable
- Find node n labeled op with children $node(y)$ and $node(z)$ if applicable. When not found, create node n. In case 3 let n be $node(y)$
- Make $node(x)$ point to n and update the attached identifiers for x

DAG example

```
1    t1 = 4 * i
2    t2 = a[t1]
3    t3 = 4 * i
4    t4 = b[t3]
5    t5 = t2 * t4
6    t6 = prod + t5
7    prod = t6
8    t7 = i + 1
9    i = t7
10   if (i <= 20) goto 1
```



University of Amsterdam

Local optimizations

- On basic blocks in the intermediate representation
  - Machine independent optimizations
- As a post code-generation step (often called peephole optimization)
  - On a small “instruction window” (often a basic block)
  - Includes machine specific optimizations

CSA
Computer Systems Architecture

University of Amsterdam

Transformations on basic blocks

Examples

- Function-preserving transformations
  - Common subexpression elimination
  - Constant folding
  - Copy propagation
  - Dead-code elimination
  - Temporary variable renaming
  - Interchange of independent statements

CSA
Computer Systems Architecture

University of Amsterdam

Transformations on basic blocks (cont'd)

- Algebraic transformations
- Machine dependent eliminations/transformations
  - Removal of redundant loads/stores
  - Use of machine idioms

CSA Computer Systems Architecture

Common subexpression elimination

- If the same expression is computed more than once it is called a common subexpression
- If the result of the expression is stored, we don't have to recompute it
- Moving to a DAG as IR, common subexpressions are automatically detected!

$$\begin{array}{lcl} x = a + b & & x = a + b \\ \ldots & \Rightarrow & \ldots \\ y = a + b & & y = x \end{array}$$

University of Amsterdam

CSA
Computer Systems Architecture

Constant folding

- Compute constant expression at compile time
- May require some emulation support

$$
\begin{array}{lcl}
x = 3 + 5 & & x = 8 \\
\ldots & \Rightarrow & \ldots \\
y = x * 2 & & y = 16
\end{array}
$$

Copy propagation

- Propagate original values when copied
- Target for dead-code elimination

$$
\begin{array}{lcl}
x = y & & x = y \\
\ldots & \Rightarrow & \ldots \\
z = x * 2 & & z = y * 2
\end{array}
$$

University of Amsterdam

Dead-code elimination

- A variable x is dead at a statement if it is not used after that statement
- An assignment $x = y + z$ where x is dead can be safely eliminated
- Requires live-variable analysis (discussed later on)

CSA
Computer Systems Architecture

Temporary variable renaming

$$\begin{array}{lcl} t1 = a + b & & t1 = a + b \\ t2 = t1 * 2 & & t2 = t1 * 2 \\ \ldots & \Rightarrow & \ldots \\ t1 = d - e & & t3 = d - e \\ c = t1 + 1 & & c = t3 + 1 \end{array}$$

- If each statement that defines a temporary defines a new temporary, then the basic block is in normal-form
  - Makes some optimizations at BB level a lot simpler (e.g. common subexpression elimination, copy propagation, etc.)

Algebraic transformations

- There are many possible algebraic transformations
- Usually only the common ones are implemented
- $x = x + 0$
- $x = x * 1$
- $x = x * 2 \Rightarrow x = x << 1$
- $x = x^2 \Rightarrow x = x * x$

University of Amsterdam

Machine dependent eliminations/transformations

- Removal of redundant loads/stores

```
1   mov R0, a
2   mov a, R0              // can be removed
```

- Removal of redundant jumps, for example

```
1           beq ...,$Lx                 bne ...,$Ly
2           j $Ly          ⇒    $Lx:    ...
3   $Lx:    ...
```

- Use of machine idioms, e.g.,
  - Auto increment/decrement addressing modes
  - SIMD instructions
- Etc., etc. (see practical assignment)

CSA
Computer Systems Architecture

Other sources of optimizations

- Global optimizations
  - Global common subexpression elimination
  - Global constant folding
  - Global copy propagation, etc.
- Loop optimizations
- They all need some dataflow analysis on the flow graph

Code motion

- Decrease amount of code inside loop
- Take a loop-invariant expression and place it before the loop

$$\text{while } (i <= limit - 2) \quad \Rightarrow \quad \begin{array}{l} t = limit - 2 \\ \text{while } (i <= t) \end{array}$$

Loop optimizations (cont'd)

Induction variable elimination

- Variables that are locked to the iteration of the loop are called induction variables
- Example: in `for (i = 0; i < 10; i++)` *i* is an induction variable
- Loops can contain more than one induction variable, for example, hidden in an array lookup computation
- Often, we can eliminate these extra induction variables

University of Amsterdam

Loop optimizations (cont'd)

Strength reduction

- Strength reduction is the replacement of expensive operations by cheaper ones (algebraic transformation)
- Its use is not limited to loops but can be helpful for induction variable elimination

$$
\begin{array}{lcl}
i = i + 1 & & i = i + 1 \\
t1 = i * 4 & \Rightarrow & t1 = t1 + 4 \\
t2 = a[t1] & & t2 = a[t1] \\
\text{if } (i < 10) \text{ goto top} & & \text{if } (i < 10) \text{ goto top}
\end{array}
$$

CSA

Computer Systems Architecture

University of Amsterdam

Loop optimizations (cont'd)

Induction variable elimination (2)

- Note that in the previous strength reduction we have to initialize $t1$ before the loop
- After such strength reductions we can eliminate an induction variable

$$\begin{array}{lcl} i = i + 1 & & t1 = t1 + 4 \\ t1 = t1 + 4 & \Rightarrow & t2 = a[t1] \\ t2 = a[t1] & & \text{if } (t1 < 40) \text{ goto top} \\ \text{if } (i < 10) \text{ goto top} & & \end{array}$$

CSA
Computer Systems Architecture

Finding loops in flow graphs

Dominator relation

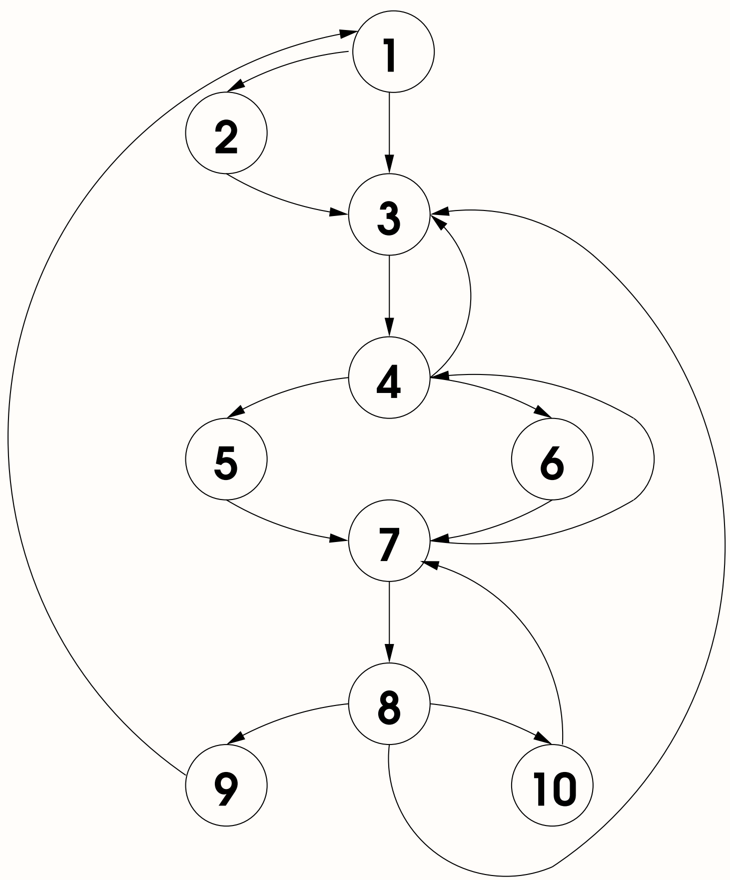
- Node A dominates node B if all paths to node B go through node A
- A node always dominates itself

We can construct a tree using this relation: the Dominator tree

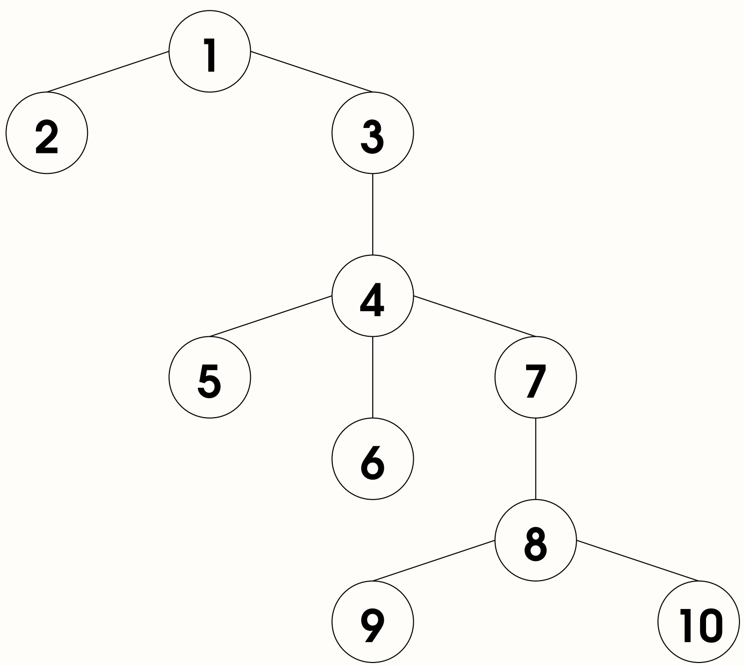
University of Amsterdam

Dominator tree example

Flow graph



Dominator tree



CSA
Computer Systems Architecture

Natural loops

- A loop has a single entry point, the header, which dominates the loop
- There must be a path back to the header
- Loops can be found by searching for edges of which their heads dominate their tails, called the backedges
- Given a backedge $n \rightarrow d$, the natural loop is d plus the nodes that can reach n without going through d

Finding natural loop of $n \rightarrow d$

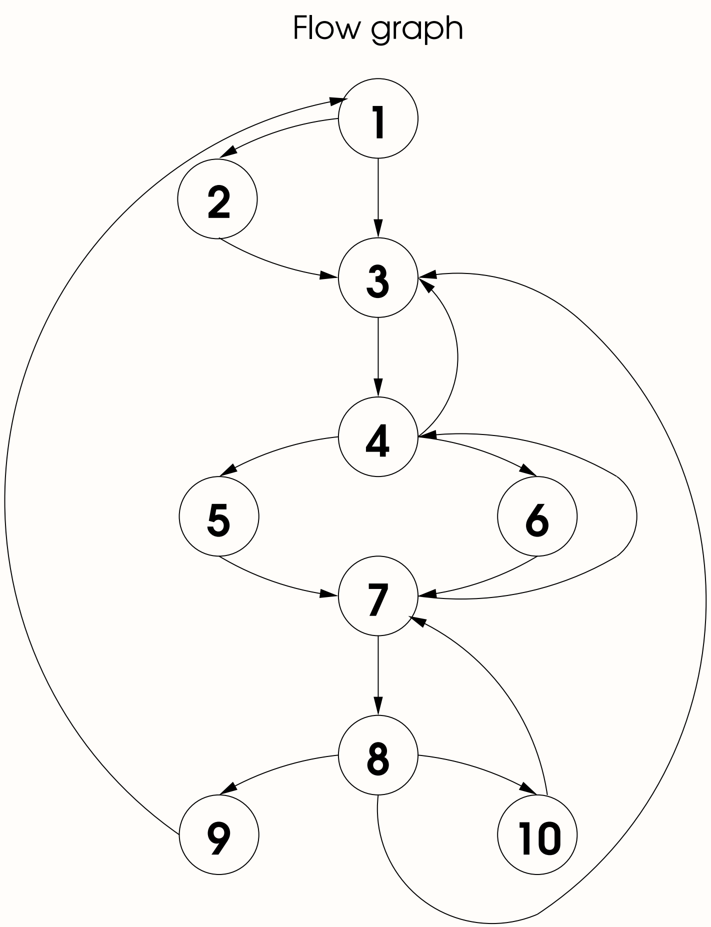
```
procedure insert(m) {
   if (not m ∈ loop) {
      loop = loop ∪ m
      push(m)
   }
}

stack = ∅
loop = {d}
insert(n)
while (stack ≠ ∅) {
   m = pop()
   for (p ∈ pred(m)) insert(p)
}
```

Natural loops (cont’d)

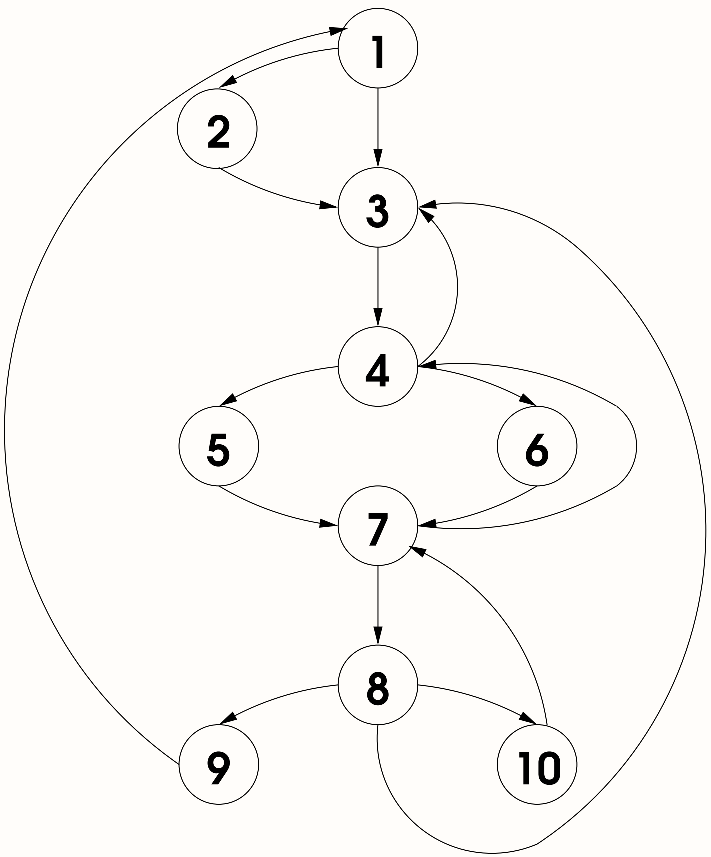
- When two backedges go to the same header node, we may join the resulting loops
- When we consider two natural loops, they are either completely disjoint or one is nested inside the other
- The nested loop is called an inner loop
- A program spends most of its time inside loops, so loops are a target for optimizations. This especially holds for inner loops!

Our example revisited



University of Amsterdam

Our example revisited



Natural loops:

1. backedge 10 −> 7: {7,8,10} (the inner loop)
2. backedge 7 −> 4: {4,5,6,7,8,10}
3. backedges 4 −> 3 and 8 −> 3: {3,4,5,6,7,8,10}
4. backedge 9 −> 1: the entire flow graph

CSA Computer Systems Architecture

University of Amsterdam

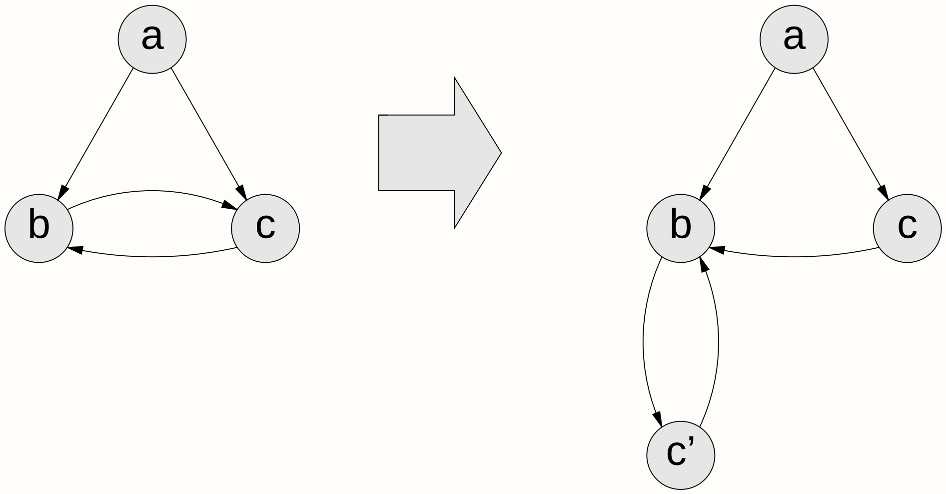
Reducible flow graphs

- A flow graph is reducible when the edges can be partitioned into forward edges and backedges
- The forward edges must form an acyclic graph in which every node can be reached from the initial node
- Exclusive use of structured control-flow statements such as `if-then-else`, `while` and `break` produces reducible control-flow
- Irreducible control-flow can create loops that cannot be optimized

CSA
Computer Systems Architecture

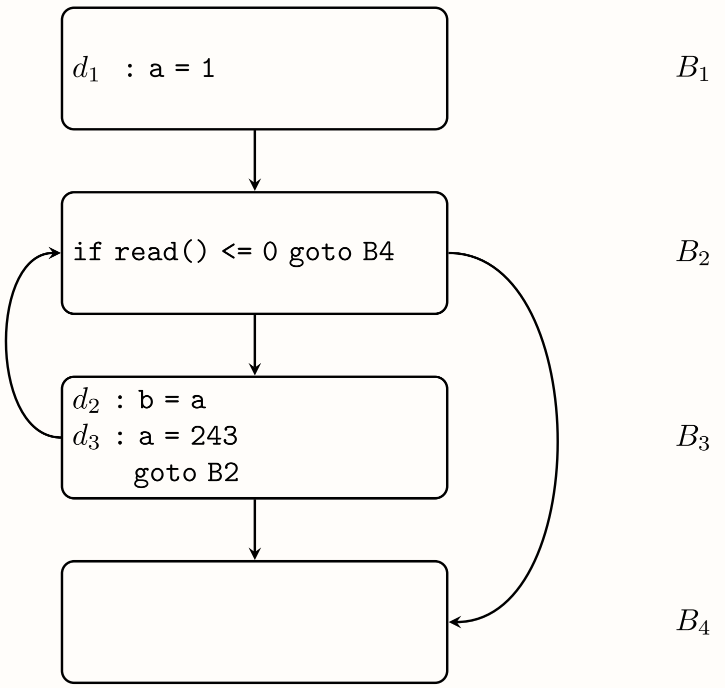
Reducible flow graphs (cont'd)

- Irreducible control-flow graphs can always be made reducible
- This usually involves some duplication of code



Dataflow abstraction

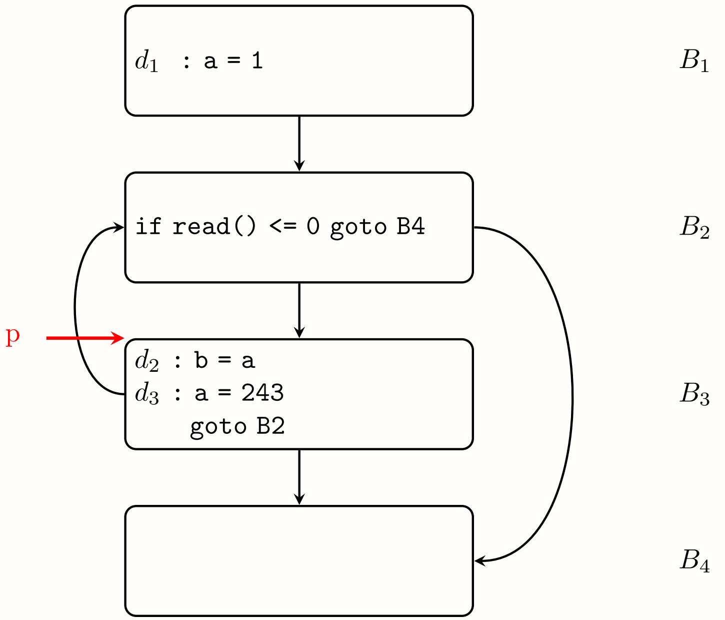
- Unbounded number of execution paths
- skip loop, 1 iteration etc..
- So abstract details



University of Amsterdam

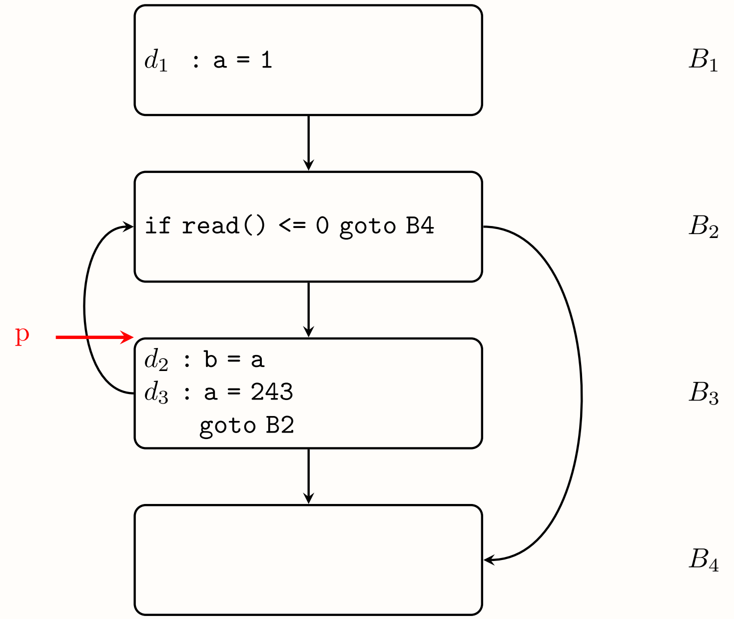
Abstraction for reaching definitions

- Possible values and definitions of variable *a* at point *p*.
- Abstraction:
  - Values of *a* at point *p*: {1,243}
  - Definitions reaching *p*: $\{d_1, d_3\}$



Abstraction for constant folding

- Check if:
  - Variable *x* is reached by one defintion
  - Definition assigns a constant to *x*
- Abstraction: *a* is not a constant at *p* (NAC)



Dataflow analysis example

```
int foo(int x, int y) {
    if (x)
        y += 2;
    else
        y += 3;
    return y;
}
```

No dataflow analysis (-O0)

```
      sw   $4,8($fp)    # x
      sw   $5,12($fp)   # y
      lw   $2,8($fp)
      beq  $2,$0,$L2
      lw   $2,12($fp)
      addu     $3,$2,2
      sw   $3,12($fp)
      j    $L3
$L2:  lw   $2,12($fp)
      addu     $3,$2,3
      sw   $3,12($fp)
$L3:  lw   $2,12($fp) # return val
```

With dataflow analysis (-O1)

```
       move     $2,$5   # y
       beq $4,$0,$L14
       addu     $2,$2,2
       j    $L15
$L14:  addu     $2,$2,3 # return val
$L15:  j    $31
```

University of Amsterdam

Dataflow analysis

- Data analysis is needed for global code optimization, e.g.:
  - Is a variable live on exit from a block? Does a definition reach a certain point in the code?
- Dataflow equations are used to collect dataflow information
  - A typical dataflow equation has the form $out[S] = gen[S] \cup (in[S] - kill[S])$
- The notion of generation and killing depends on the dataflow analysis problem to be solved
- Let's first consider Reaching Definitions analysis for structured programs

CSA Computer Systems Architecture

Reaching definitions

- A definition of a variable x is a statement that assigns or may assign a value to x
- An assignment to x is an unambiguous definition of x
- An ambiguous assignment to x can be an assignment to a pointer or a function call where x is passed by reference

Reaching definitions (cont'd)

- When x is defined, we say the definition is generated
- An unambiguous definition of x kills all other definitions of x
- When all definitions of x are the same at a certain point, we can use this information to do some optimizations
- Example: all definitions of x define x to be 1. Now, by performing constant folding, we can do strength reduction if x is used in $z = y * x$

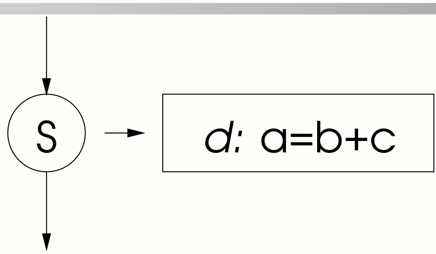
University of Amsterdam

Dataflow analysis for reaching definitions

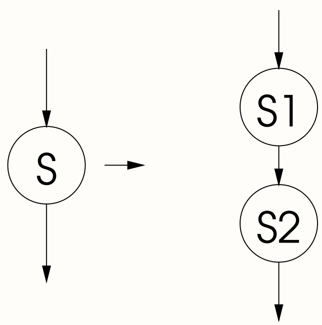
- During dataflow analysis we have to examine every path that can be taken to see which definitions reach a point in the code
- Sometimes a certain path will never be taken, even if it is part of the flow graph
- Since it is undecidable whether a path can be taken, we simply examine all paths
- This won't cause false assumptions to be made for the code: it is a conservative simplification
  - It merely causes optimizations not to be performed

CSA
Computer Systems Architecture

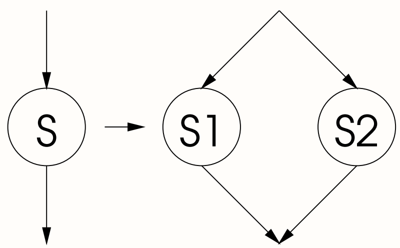
The building blocks



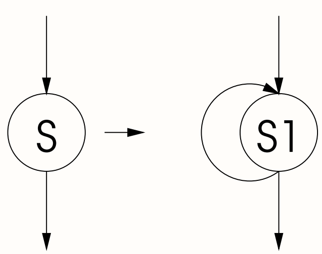
gen[S]=$\{d\}$
kill[S]=$D_a - \{d\}$
out[S]=gen[S]$\cup$(in[S]-kill[S])



gen[S]=gen[S2]$\cup$(gen[S1]-kill[S2])
kill[S]=kill[S2]$\cup$(kill[S1]-gen[S2])
in[S1]=in[S]
in[S2]=out[S1]
out[S]=out[S2]



gen[S]=gen[S1]$\cup$gen[S2]
kill[S]=kill[S1]$\cap$kill[S2]
in[S1]=in[S2]=in[S]
out[S]=out[S1]$\cup$out[S2]



gen[S]=gen[S1]
kill[S]=kill[S1]
in[S1]=in[S]$\cup$gen[S1]
out[S]=out[S1]

Dealing with loops

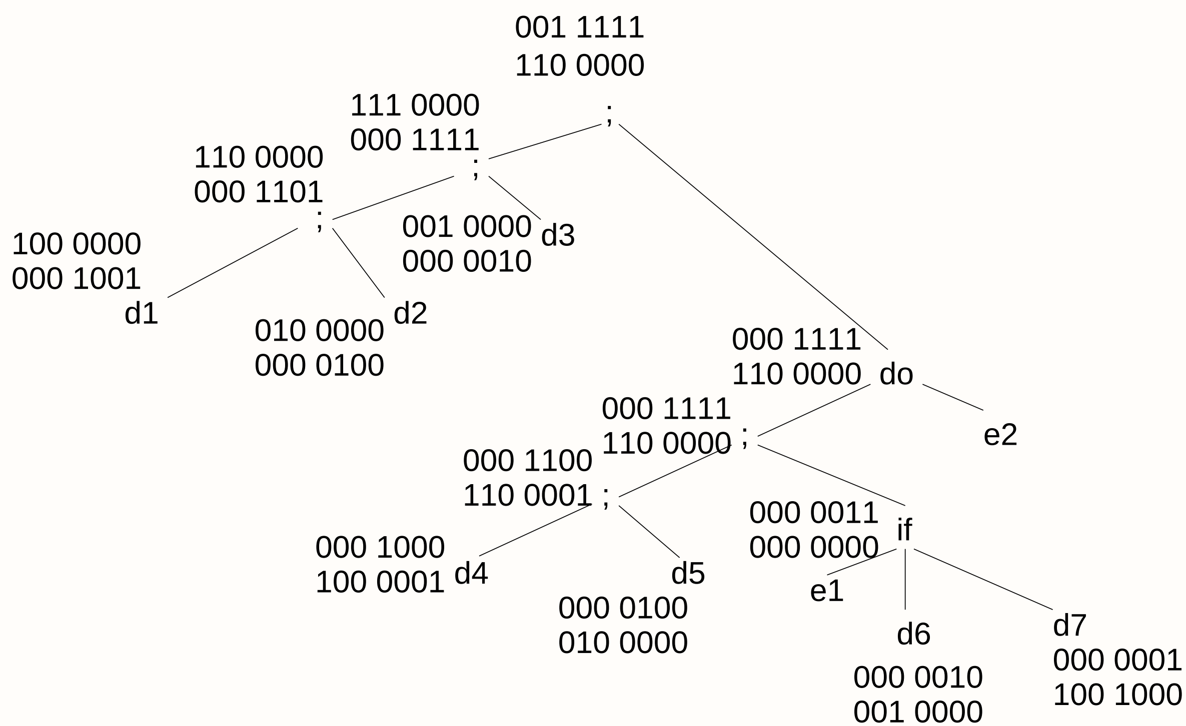
- The in-set to the code inside the loop is the in-set of the loop plus the out-set of the loop: $in[S1] = in[S] \cup out[S1]$
- The out-set of the loop is the out-set of the code inside: $out[S] = out[S1]$
- Fortunately, we can also compute $out[S1]$ in terms of $in[S1]$: $out[S1] = gen[S1] \cup (in[S1] - kill[S1])$

Dealing with loops (cont'd)

- $I = in[S1], O = out[S1], J = in[S], G = gen[S1]$ and $K = kill[S1]$
- $I = J \cup O$
- $O = G \cup (I - K)$
- Assume $O = \emptyset$, then $I^1 = J$
- $O^1 = G \cup (I^1 - K) = G \cup (J - K)$
- $I^2 = J \cup O^1 = J \cup G \cup (J - K) = J \cup G$
- $O^2 = G \cup (I^2 - K) = G \cup (J \cup G - K) = G \cup (J - K)$
- $O^1 = O^2$ so $in[S1] = in[S] \cup gen[S1]$ and $out[S] = out[S1]$

Reaching definitions example

d_1 i = m - 1
d_2 j = n
d_3 a = u1
do
d_4 i = i + 1
d_5 j = j - 1
if (e1)
d_6 a = u2
else
d_7 i = u3
while (e2)



In reality, dataflow analysis is often performed at the granularity of basic blocks rather than statements

Iterative solutions

- Programs in general need not be made up out of structured control-flow statements
- We can do dataflow analysis on these programs using an iterative algorithm
- The equations (at basic block level) for reaching definitions are:

$$in[B] = \bigcup_{P \in pred(B)} out[P]$$

$$out[B] = gen[B] \cup (in[B] - kill[B])$$

Iterative algorithm for reaching definitions

for (each block B) $out[B] = gen[B]$

do {

  change = false

  for (each block B) {

    $in[B] = \bigcup_{P \in pred(B)} out[P]$

    oldout = $out[B]$

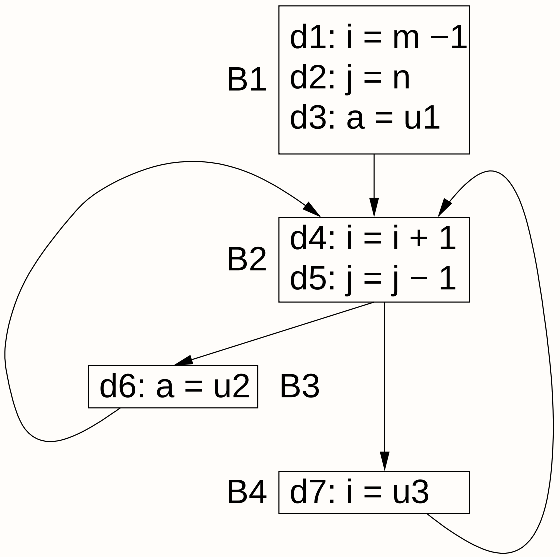
    $out[B] = gen[B] \cup (in[B] - kill[B])$

    if ($out[B] \neq$ oldout) change = true

  }

} while (change)

Reaching definitions: an example



gen[B1] = {d1,d2,d3}
kill[B1] = {d4,d5,d6,d7}

gen[B2] = {d4,d5}
kill[B2] = {d1,d2,d7}

gen[B3] = {d6}
kill[B3] = {d3}

gen[B4] = {d7}
kill[B4] = {d1,d4}

Block B	Initial		Pass 1		Pass 2	
	$in[B]$	$out[B]$	$in[B]$	$out[B]$	$in[B]$	$out[B]$
B1	000 0000	111 0000	000 0000	111 0000	000 0000	111 0000
B2	000 0000	000 1100	111 0011	001 1110	111 1111	001 1110
B3	000 0000	000 0010	001 1110	000 1110	001 1110	000 1110
B4	000 0000	000 0001	001 1110	001 0111	001 1110	001 0111

Available expressions

- An expression *e* is available at a point *p* if every path from the initial node to *p* evaluates *e*, and the variables used by *e* are not changed after the last evaluations
- An available expression *e* is killed if one of the variables used by *e* is assigned to
- An available expression *e* is generated if it is evaluated
- Note that if an expression *e* is assigned to a variable used by *e*, this expression will not be generated

Available expression: example

Instruction sequence	Available expressions
	$\emptyset$
`a = b + c`	
	$\{b+c\}$
`b = a - d`	
	$\{a-d\}$
`c = b + c`	
	$\{a-d\}$
`d = a - d`	
	$\emptyset$

University of Amsterdam

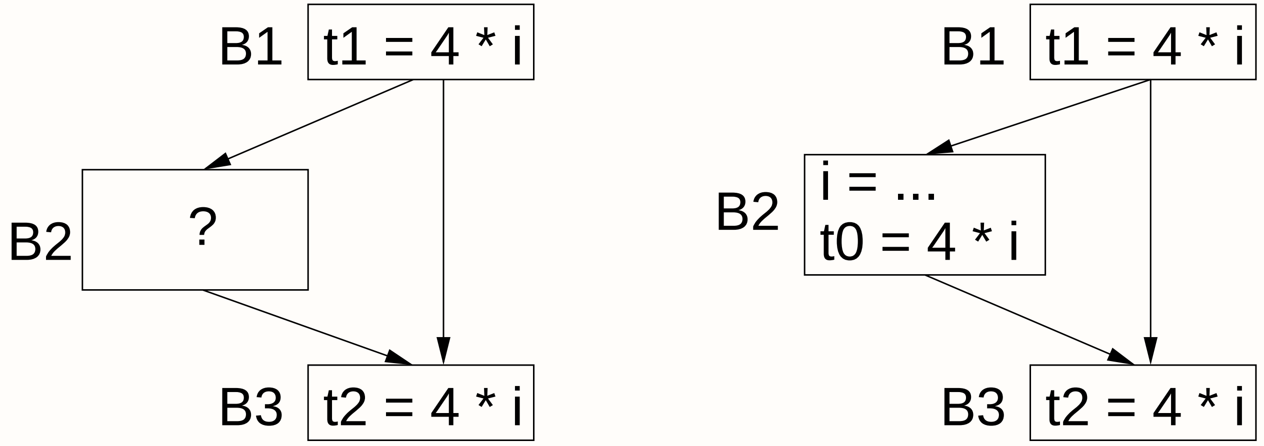
Input sets for Available expressions

- e_gen set for B:
  - Loop through instructions in B in order
  - For every instruction `i: x = y + z`
  - Add expression `y + z` to e_gen set
  - Remove all expressions containing x from e_gen set
- e_kill set for B
  - All expressions containing variables defined in B.

CSA
Computer Systems Architecture

Available expressions (cont'd)

- Available expressions are mainly used to find common subexpressions



Available expressions (cont'd)

- Dataflow equations:

$$out[B] = e\_gen[B] \cup (in[B] - e\_kill[B])$$

$$in[B] = \bigcap_{P \in pred(B)} out[P] \;\; \text{for B not initial}$$

$$in[B1] = \emptyset \;\; \text{where B1 is the initial block}$$

- The confluence operator is intersection instead of the union!

Liveness analysis

- A variable is live at a certain point in the code if it holds a value that may be needed in the future
- Solve backwards:
  - Find use of a variable
  - This variable is live between statements that have found use as next statement
  - Recurse until you find a definition of the variable

University of Amsterdam

Dataflow for liveness

- Using the sets $use[B]$ and $def[B]$
  - $def[B]$ is the set of variables assigned values in B prior to any use of that variable in B
  - $use[B]$ is the set of variables whose values may be used in B prior to any definition of the variable
- A variable comes live into a block (in $in[B]$), if it is either used before redefinition or it is live coming out of the block and is not redefined in the block
- A variable comes live out of a block (in $out[B]$) if and only if it is live coming into one of its successors

CSA
Computer Systems Architecture

Dataflow equations for liveness

$$in[B] = use[B] \cup (out[B] - def[B])$$

$$out[B] = \bigcup_{S \in succ[B]} in[S]$$

- Note the relation between reaching-definitions equations: the roles of *in* and *out* are interchanged

Data flow analysis summary

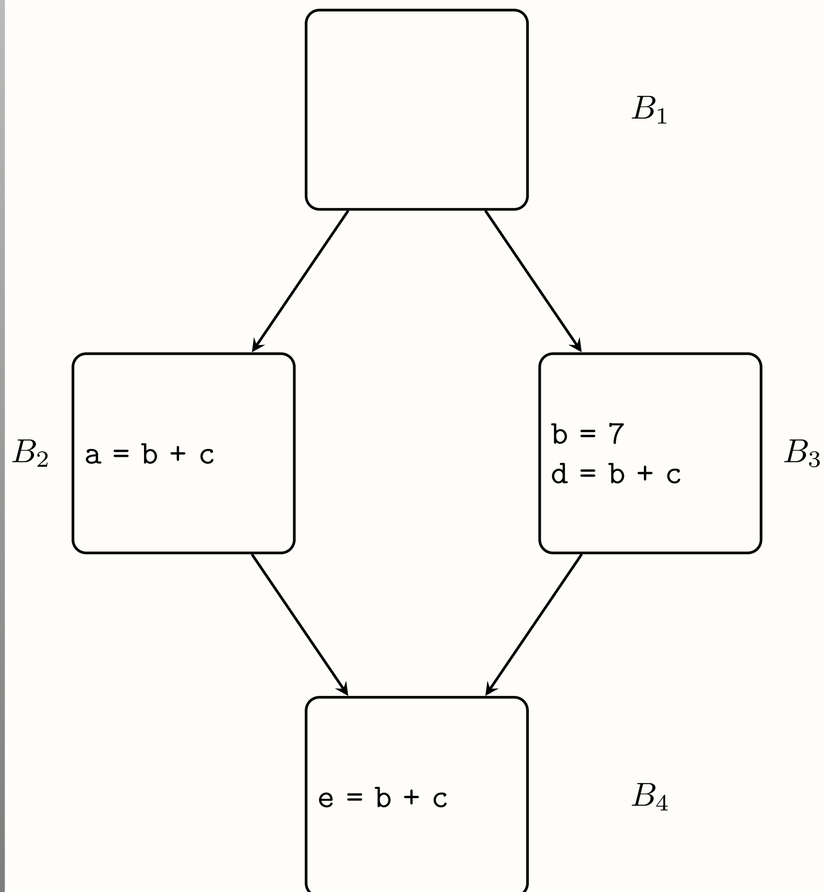
	Reaching definitions	Available expressions	Live variables
Domain	Set of definitions	Set of expressions	Set of variables
Direction	Forward	Forward	Backwards
Transfer function	$gen[B] \cup (x - kill[B])$	$e\_gen[B] \cup (x - e\_kill[B])$	$use[B] \cup (x - def[B])$
Boundary	$OUT[entry] = \emptyset$	$OUT[entry] = \emptyset$	$IN[exit] = \emptyset$
Meet operator	$\cup$	$\cup$	$\cap$
Initialize	$OUT[B] = \emptyset$	$OUT[B] = U$	$IN[B] = \emptyset$

Algorithms for global optimizations

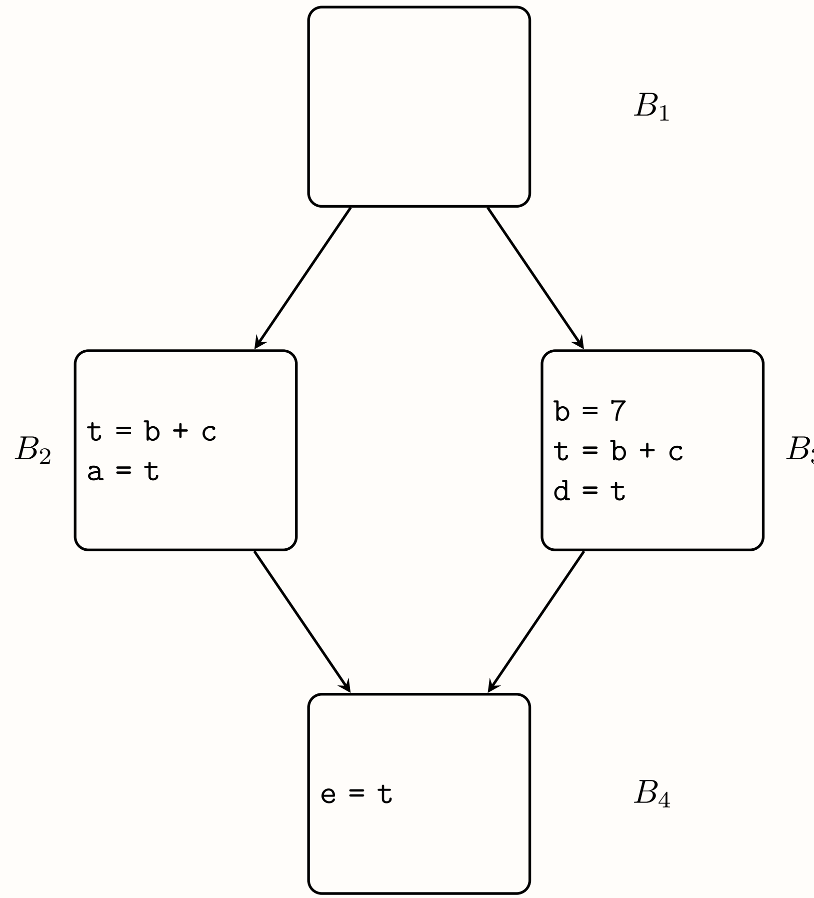
Global common subexpression elimination

- First calculate the sets of available expressions
- For every statement s of the form $x = y + z$ where $y + z$ is available do the following
  - Search backwards in the graph for the evaluations of $y + z$
  - Create a new variable u
  - Replace statements $w = y + z$ by $u = y + z;\ w = u$
  - Replace statement s by $x = u$

GCSE example



Before global common subexpression elimination



After global common subexpression elimination

University of Amsterdam

Copy propagation

- Suppose a copy statement s of the form $x = y$ is encountered. We may now substitute a use of x by a use of y if
  - Statement s is the only definition of x reaching the use
  - On every path from statement s to the use, there are no assignments to y

CSA
Computer Systems Architecture

Copy propagation (cont'd)

- To find the set of copy statements we can use, we define a new dataflow problem
- An occurrence of a copy statement generates this statement
- An assignment to x or y kills the copy statement $x = y$
- Dataflow equations:

$$out[B] = c\_gen[B] \cup (in[B] - c\_kill[B])$$

$$in[B] = \bigcap_{P \in pred(B)} out[P] \text{ for B not initial}$$

$$in[B1] = \emptyset \text{ where B1 is the initial block}$$

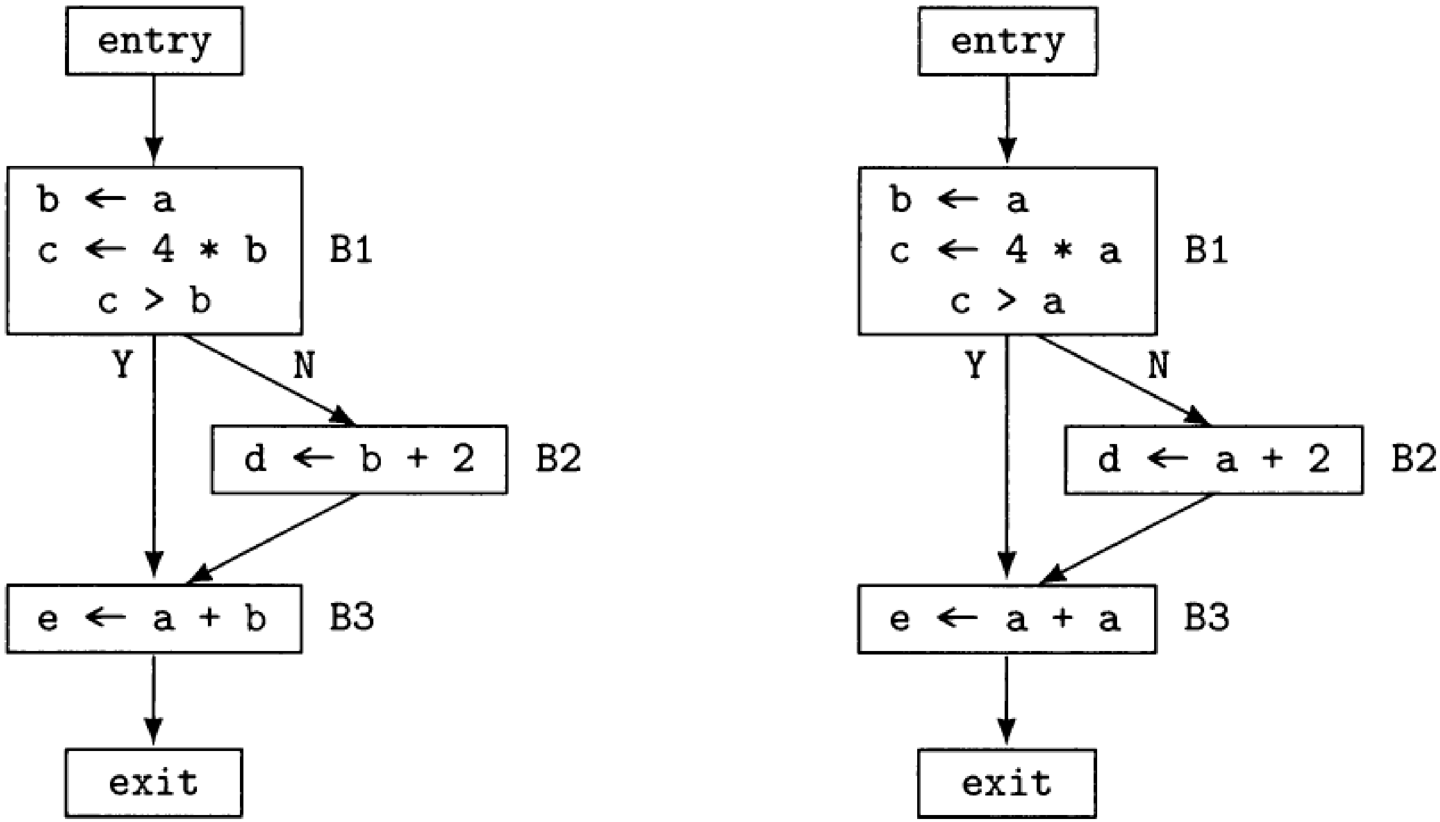
University of Amsterdam

Copy propagation (cont'd)

- For each copy statement s: $x = y$ do
  - Determine the uses of x reached by this definition of x
  - Determine if for each of those uses this is the only definition reaching it ($\rightarrow s \in in[B_{use}]$)
  - If so, remove s and replace the uses of x by uses of y

CSA
Computer Systems Architecture

Copy propagation example



(a) (b)

University of Amsterdam

Detection of loop-invariant computations

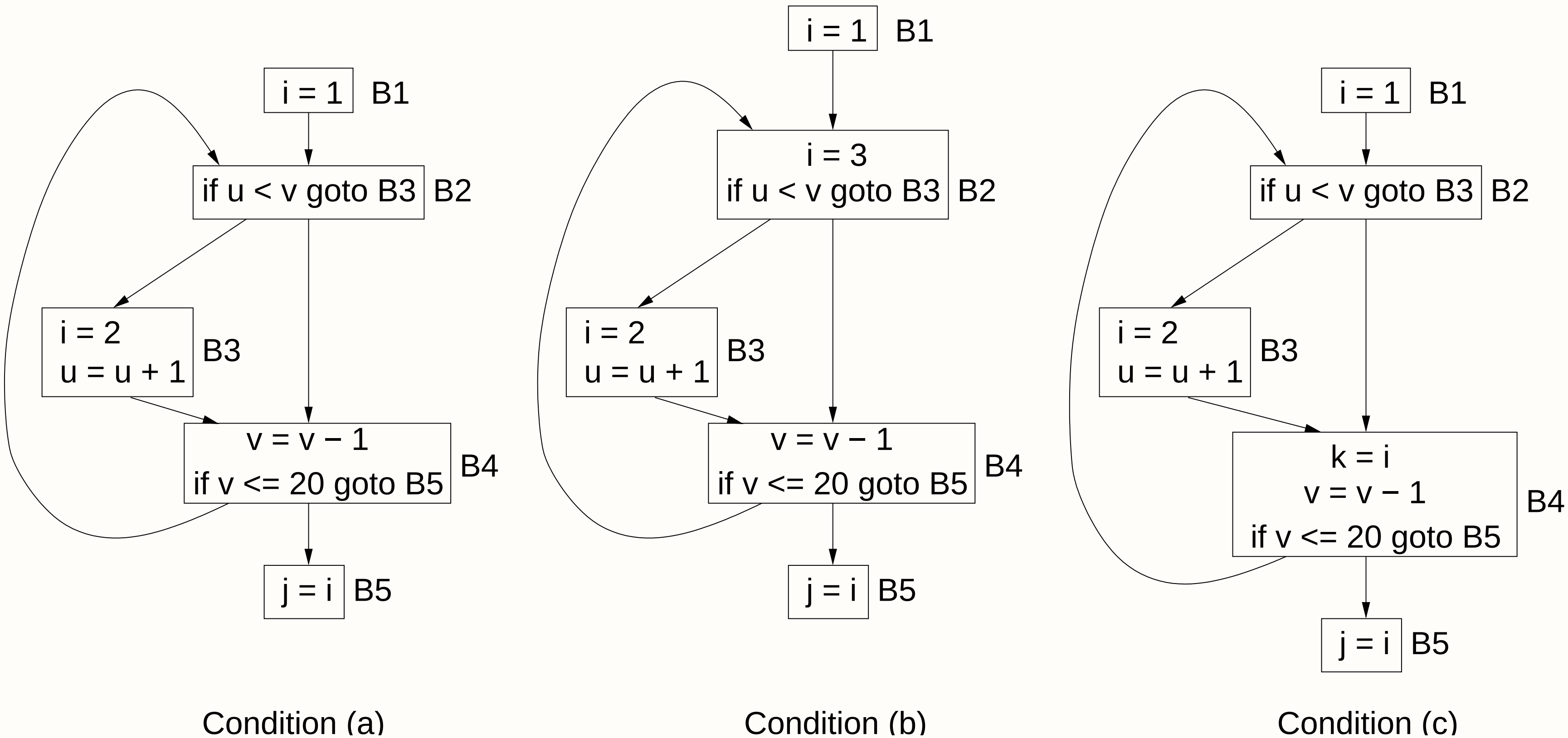
1. Mark invariant those statements whose operands are constant or have all reaching definitions outside the loop
2. Repeat step 3 until no new statements are marked invariant
3. Mark invariant those statements whose operands either are constant, have reaching definitions outside the loop, or have exactly one reaching definition that is marked invariant

CSA Computer Systems Architecture

Code motion

1. Create a pre-header for the loop
2. Find loop-invariant statements
3. For each statement *s* defining *x* found in step 2, check that
   (a) it is in a block that dominate all exits of the loop
   (b) *x* is not defined elsewhere in the loop
   (c) all uses of *x* in the loop can only be reached from this statement *s*
4. Move the statements that conform to the pre-header

Code motion (cont'd)



Detection of induction variables

- A basic induction variable i is a variable that only has assignments of the form $i = i \pm c$
- Associated with each induction variable j is a triple (i, c, d) where i is a basic induction variable and c and d are constants such that $j = c * i + d$
- In this case j belongs to the family of i
- The basic induction variable i belongs to its own family, with the associated triple $(i, 1, 0)$

Detection of induction variables (cont'd)

- Find all basic induction variables in the loop
- Find variables k with a single assignment in the loop with one of the following forms:
  - $k = j * b$, $k = b * j$, $k = j/b$, $k = j + b$, $k = b + j$, where b is a constant and j is an induction variable
- If j is not basic and in the family of i then there must be
  - No assignment of i between the assignment of j and k
  - No definition of j outside the loop that reaches k

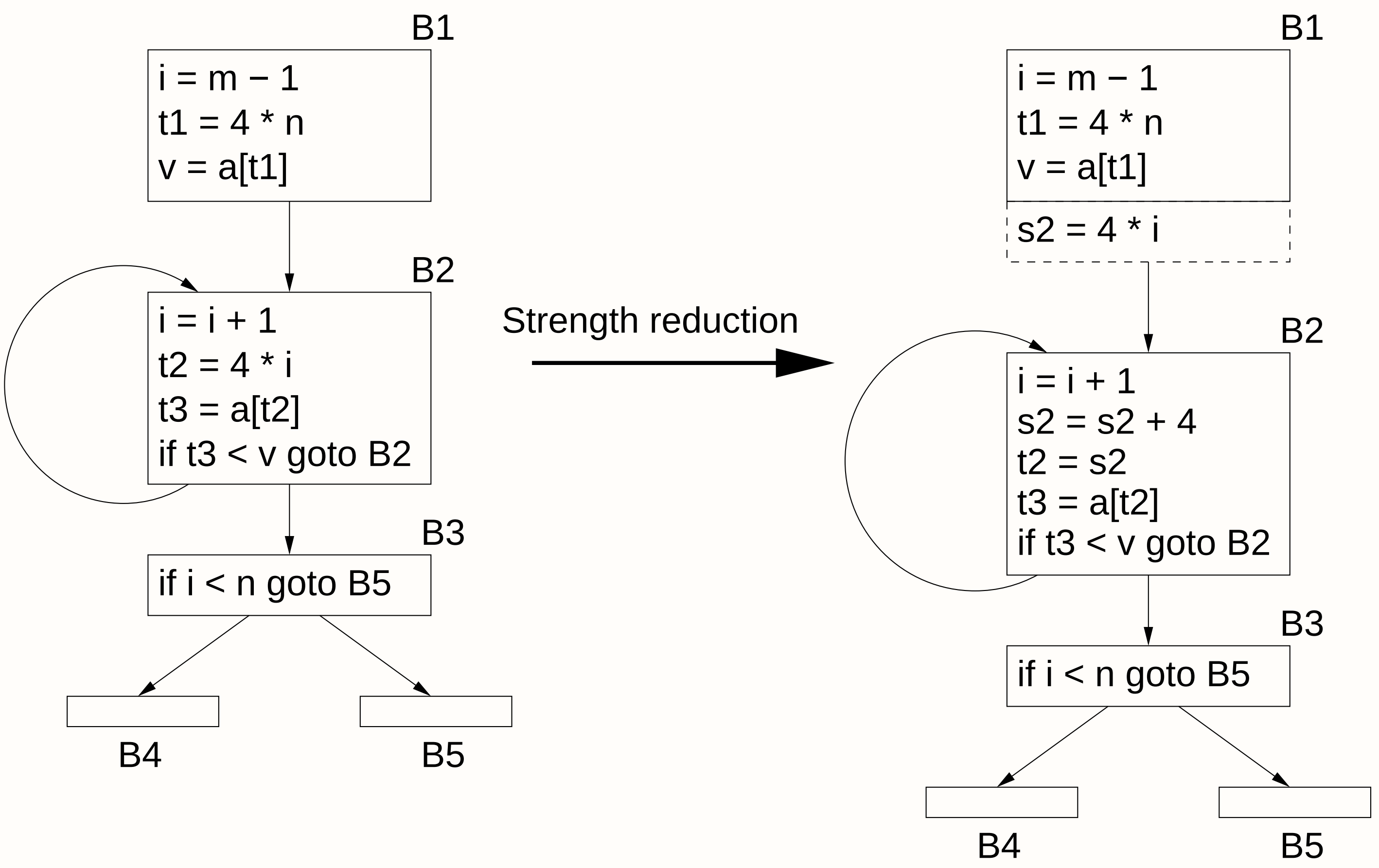
University of Amsterdam

Strength reduction for induction variables

- Consider each basic induction variable i in turn. For each variable j in the family of i with triple (i,c,d):
  - Create a new variable s
  - Replace the assignment to j by $j = s$
  - Immediately after each assignment $i = i \pm n$ append $s = s + c * n$
  - Place s in the family of i with triple (i,c,d)
  - Initialize s in the preheader: $s = c * i + d$

CSA
Computer Systems Architecture

Strength reduction for induction variables (cont'd)



Elimination of induction variables

- Consider each basic induction variable i only used to compute other induction variables and tests
- Take some j in i's family such that c and d from the triple (i, c, d) are simple
- Rewrite tests `if (`i `relop` x`)` to

  $r = c * x + d$; `if (`j `relop` r`)`
- Delete assignments to i from the loop
- Do some copy propagation to eliminate $j = s$ assignments formed during strength reduction

University of Amsterdam

Introducing Aliases

- Aliases, e.g. caused by pointers, make dataflow analysis more complex (uncertainty regarding what is defined and used: $x = *p$ might use any variable)
- Call by reference parameters will also introduce aliases
- Use dataflow analysis to determine what a pointer might point to
- Without alias analysis optimization possibilities will be limited

CSA
Computer Systems Architecture

Alias analysis example

a. Program fragment

$$\begin{array}{lcl} u & \leftarrow & M[t] \\ M[x] & \leftarrow & r \\ w & \leftarrow & M[t] \\ b & \leftarrow & u+w \end{array}$$

University of Amsterdam

Alias example

a. Program fragment

$$
\begin{aligned}
u &\leftarrow M[t] \\
M[x] &\leftarrow r \\
w &\leftarrow M[t] \\
b &\leftarrow u + w
\end{aligned}
$$

b. After GCSE

$$
\begin{aligned}
1:\quad z &\leftarrow M[t] \\
2:\quad u &\leftarrow z \\
3:\quad M[x] &\leftarrow r \\
4:\quad w &\leftarrow z \\
5:\quad b &\leftarrow u + w
\end{aligned}
$$

CSA Computer Systems Architecture

Alias example

a. Program fragment

$$
\begin{array}{lll}
u & \leftarrow & M[t] \\
M[x] & \leftarrow & r \\
w & \leftarrow & M[t] \\
b & \leftarrow & u+w
\end{array}
$$

b. After GCSE

$$
\begin{array}{llll}
1: & z & \leftarrow & M[t] \\
2: & u & \leftarrow & z \\
3: & M[x] & \leftarrow & r \\
4: & w & \leftarrow & z \\
5: & b & \leftarrow & u+w
\end{array}
$$

c. Copy Prop on $u \leftarrow z$

$$
\begin{array}{llll}
1: & z & \leftarrow & M[t] \\
3: & M[x] & \leftarrow & r \\
4: & w & \leftarrow & z \\
5: & b & \leftarrow & z+w
\end{array}
$$

Alias example

a. Program fragment

$$\begin{array}{lcl} u & \leftarrow & M[t] \\ M[x] & \leftarrow & r \\ w & \leftarrow & M[t] \\ b & \leftarrow & u+w \end{array}$$

b. After GCSE

$$\begin{array}{llcl} 1: & z & \leftarrow & M[t] \\ 2: & u & \leftarrow & z \\ 3: & M[x] & \leftarrow & r \\ 4: & w & \leftarrow & z \\ 5: & b & \leftarrow & u+w \end{array}$$

c. Copy Prop on $u \leftarrow z$

$$\begin{array}{llcl} 1: & z & \leftarrow & M[t] \\ 3: & M[x] & \leftarrow & r \\ 4: & w & \leftarrow & z \\ 5: & b & \leftarrow & z+w \end{array}$$

d. Copy Prop on $w \leftarrow z$

$$\begin{array}{llcl} 1: & z & \leftarrow & M[t] \\ 3: & M[x] & \leftarrow & r \\ 5: & b & \leftarrow & z+z \end{array}$$

Alias Analysis

- $in[B]$ contains for each pointer p the set of variables to which p could point at the beginning of block B
  - Elements of $in[B]$ are pairs (p,a) where p is a pointer and a a variable, meaning that p might point to a
- $out[B]$ is defined similarly for the end of B

Alias Analysis (cont'd)

- Define a function $trans_B$ such that $trans_B(in[B]) = out[B]$
- $trans_B$ is composed of $trans_s$, for each stmt s of block B
  - If s is $p = \&a$ or $p = \&a \pm c$ in case a is an array, then
    $$trans_s(S) = (S - \{(p,b) | \text{any variable b}\}) \cup \{(p,a)\}$$
  - If s is $p = q \pm c$ for pointer q and nonzero integer c, then
    $$trans_s(S) = (S - \{(p,b) | \text{any variable b}\}) \cup \{(p,b) | (q,b) \in S \text{ and b is an array variable}\}$$
  - If s is $p = q$, then
    $$trans_s(S) = (S - \{(p,b) | \text{any variable b}\}) \cup \{(p,b) | (q,b) \in S\}$$

Alias Analysis (cont'd)

- If s assigns to pointer p any other expression, then $trans_s(S) = S - \{(p, b) | \text{any variable b}\}$
- If s is not an assignment to a pointer, then $trans_s(S) = S$

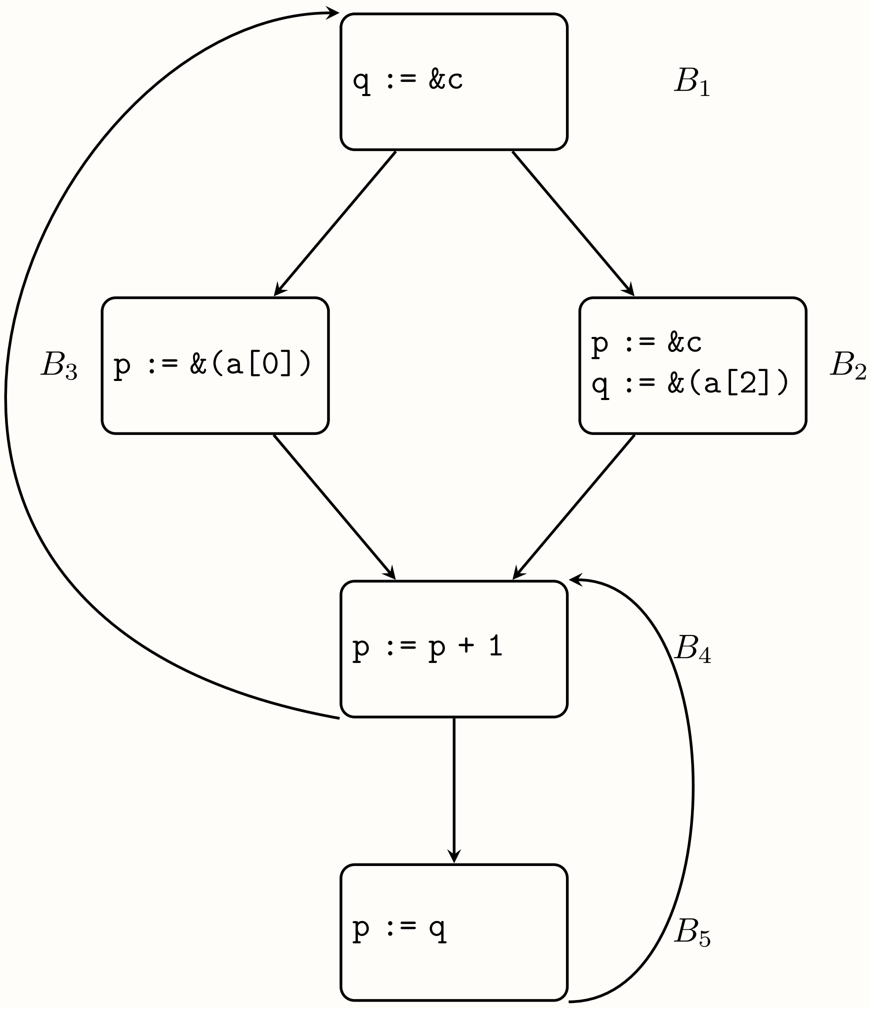
- Dataflow equations for alias analysis:

$$out[B] = trans_B(in[B])$$

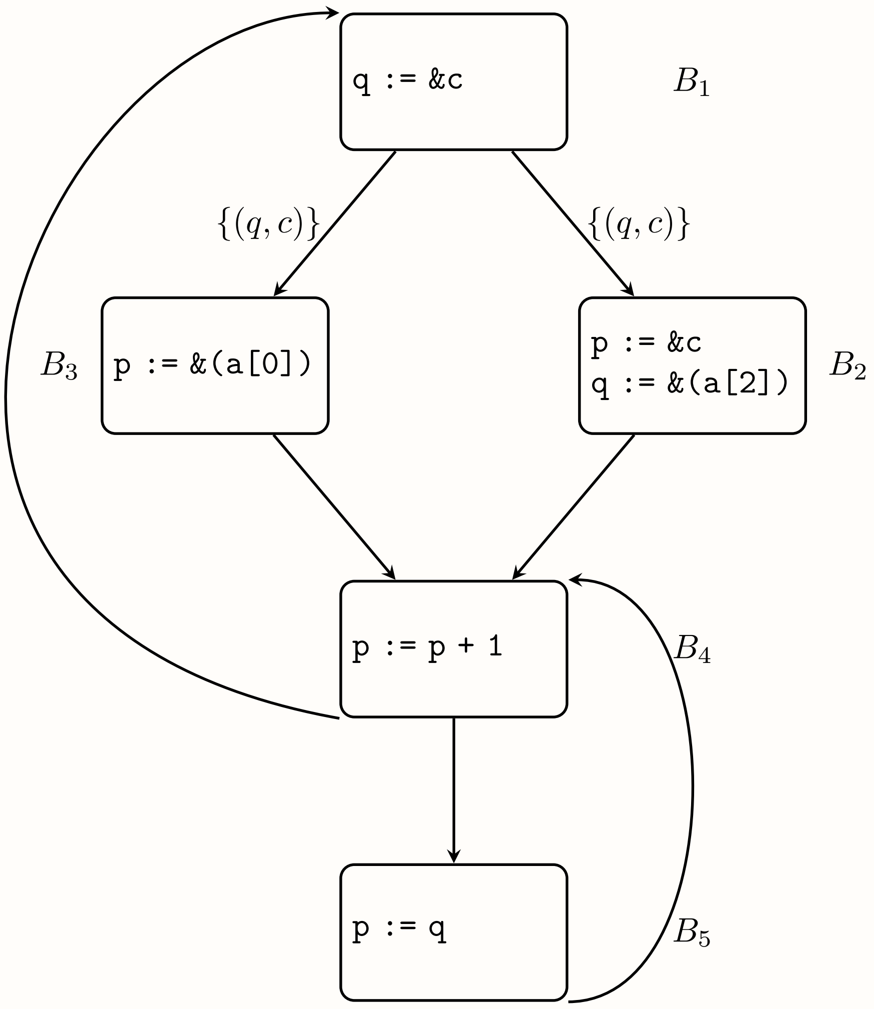
$$in[B] = \bigcup_{P \in pred(B)} out[P]$$

where $trans_B(S) = trans_{s_k}(trans_{s_{k-1}}(\cdots(trans_{s_1}(S))))$

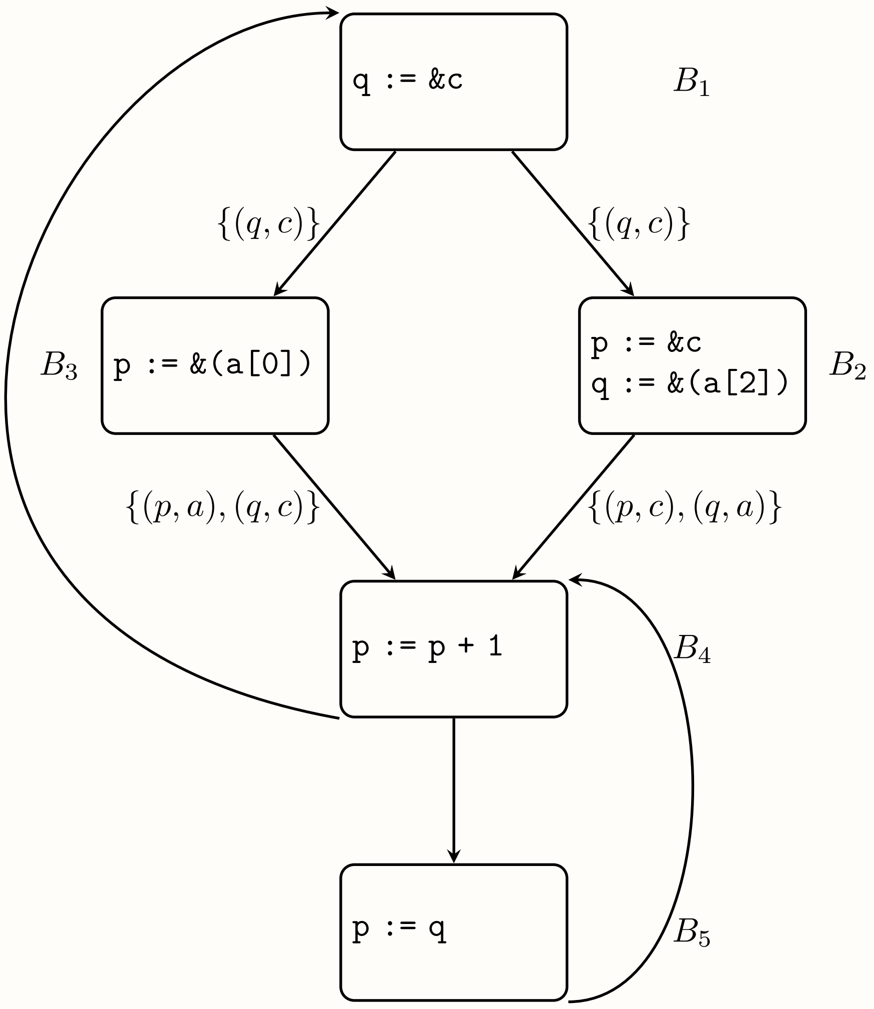
Alias dataflow example



Alias dataflow example

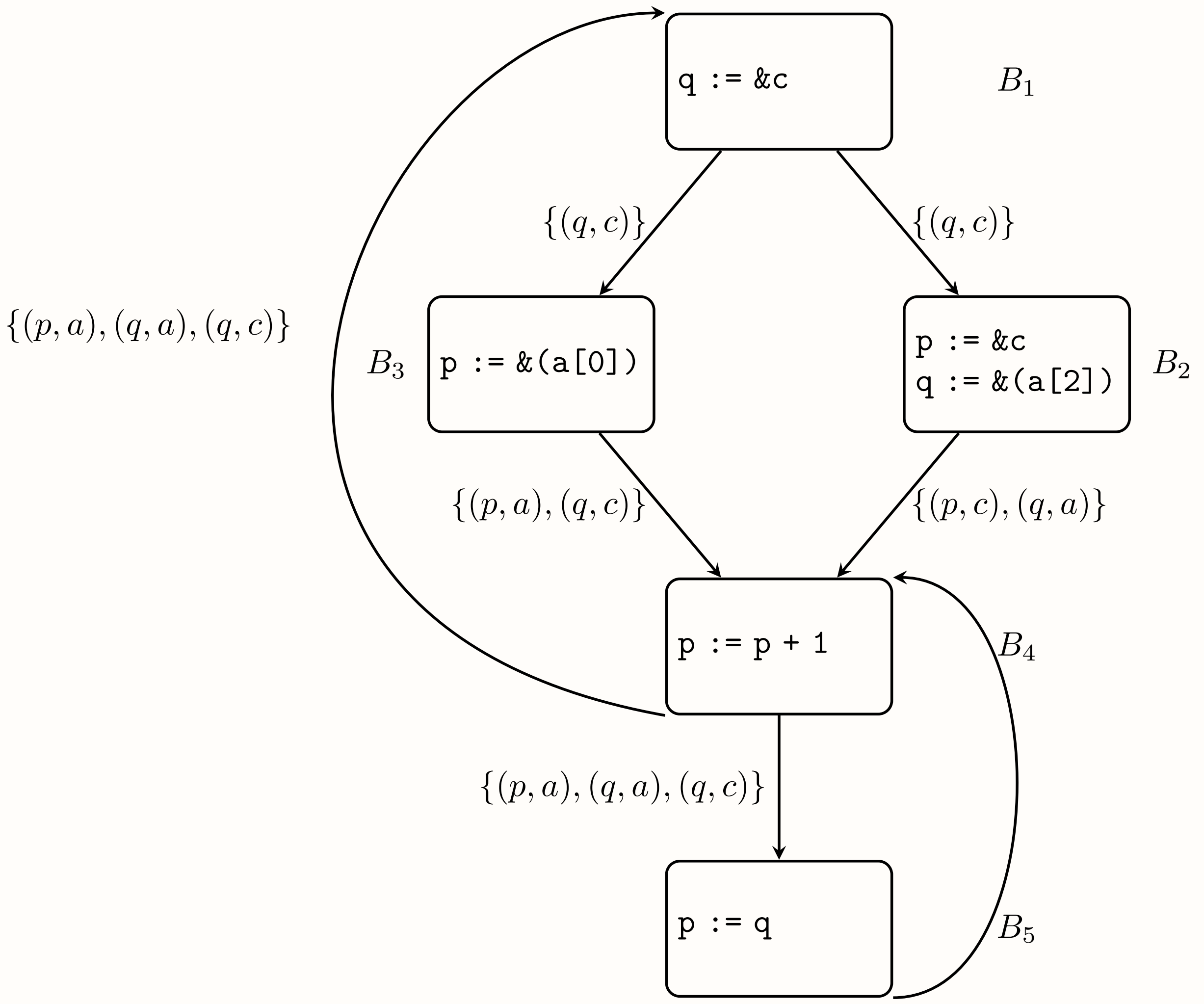


Alias dataflow example



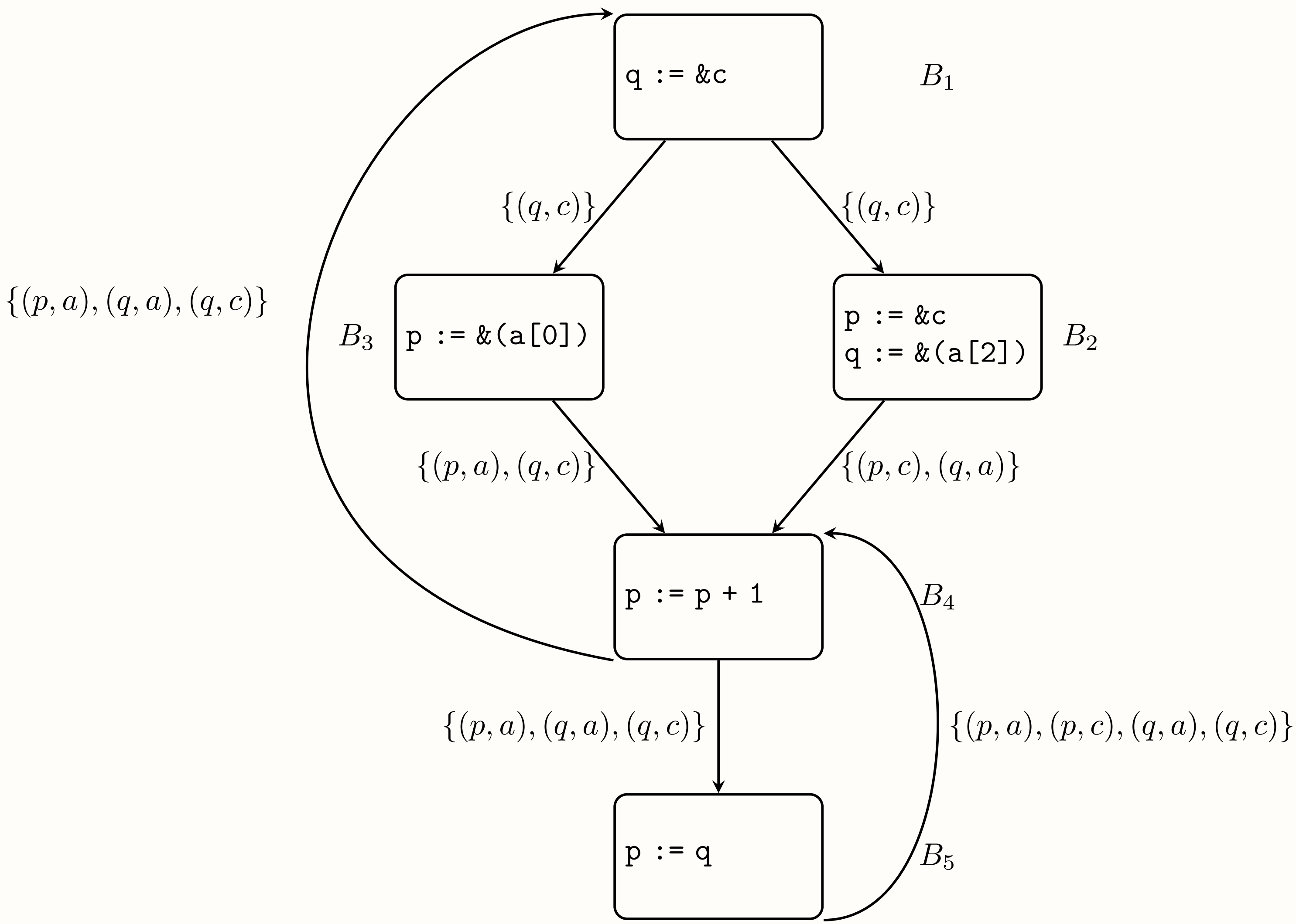
University of Amsterdam

Alias dataflow example

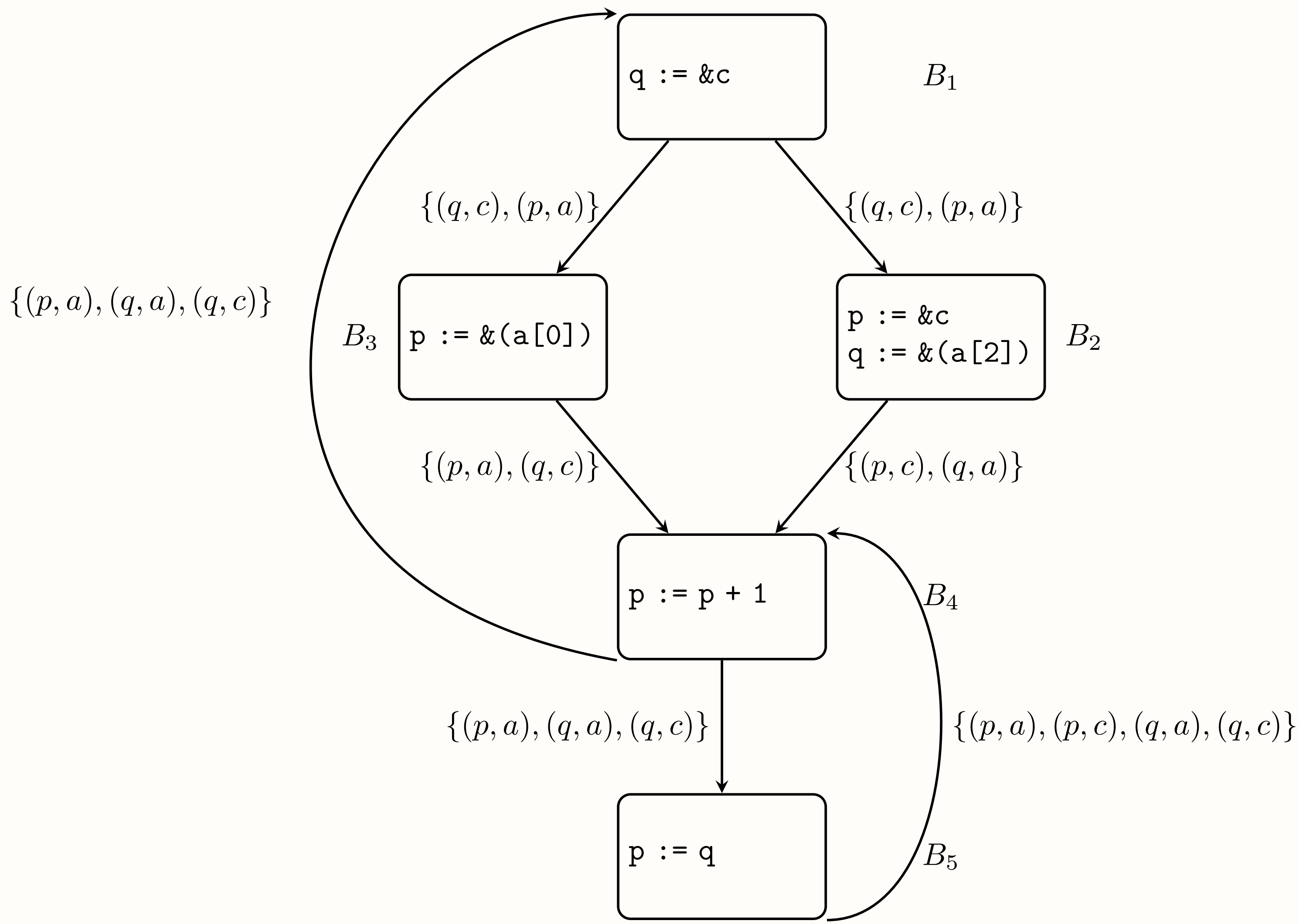


CSA
Computer Systems Architecture

Alias dataflow example



Alias dataflow example



University of Amsterdam

Alias Analysis (cont'd)

- How to use the alias dataflow information? Examples:
  - In reaching definitions analysis (to determine *gen* and *kill*)
    - $\rightarrow$ statement $*p = a$ generates a definition of every variable b such that p could point to b
    - $\rightarrow$ $*p = a$ kills definition of b only if b is not an array and is the only variable p could possibly point to (to be conservative)
  - In liveness analysis (to determine def and *use*)
    - $\rightarrow$ $*p = a$ uses p and a. It defines b only if b is the unique variable that p might point to (to be conservative)
    - $\rightarrow$ $a = *p$ defines a, and represents the use of p and a use of any variable that p could point to

CSA Computer Systems Architecture

MIPS example: CSE

C-code

```
int x,y;
int *p;

int foo() {
   int r;

   r = x + y;
   asm volatile ("#use"::"r"(r));
   *p = 10;
   r = x + y;
   return r;
}
```

MIPS assembly (-O1)

```
        .ent      foo
foo:
        lw      $2,x
        lw      $3,y
        addu      $2,$2,$3
        #use
        lw      $3,p
        li      $2,0x0000000a
        sw      $2,0($3)
        lw      $3,x
        lw      $2,y
        addu      $2,$3,$2
        j     $31
        .end      foo
```

MIPS example: CP

C-code

```
int *p;

int foo(int x, int y) {
   int r;

   x = 1;
   asm volatile ("#use"::"r"(x));
   *p = 10;
   r = x + y;
   return r;
}
```

MIPS assembly (-O1)

```
        .ent    foo
foo:
        li      $4,0x00000001
        #use
        lw      $3,p
        li      $2,0x0000000a
        sw      $2,0($3)
        addu    $2,$5,1
        j       $31
        .end    foo
```

MIPS example: CP cont'd

C-code

```
int *p;

int foo(int x, int y) {
   int r;

   x = 1;
   &x;
   asm volatile ("#use"::"r"(x));
   *p = 10;
   r = x + y;
   return r;
}
```

MIPS assembly (-O1)

```
        .ent    foo
foo:
        li      $2,0x00000001
        sw      $2,0($sp)
        #use
        lw      $3,p
        li      $2,0x0000000a
        sw      $2,0($3)
        lw      $2,0($sp)
        addu    $2,$5,$2
        j       $31
        .end    foo
```

Instruction selection

- Was a problem in the CISC era (e.g., lots of addressing modes)
- RISC instructions mean simpler instruction selection
- However, new instruction sets introduce new, complicated instructions (e.g., multimedia instruction sets)

Instruction selection methods

- Tree-based methods (IR is a tree)
  - Maximal Munch
  - Dynamic programming
  - Tree grammars
    - Input tree treated as string using prefix notation
    - Rewrite string using an LR parser and generate instructions as side effect of rewriting rules
- If the DAG is not a tree, then it can be partitioned into multiple trees

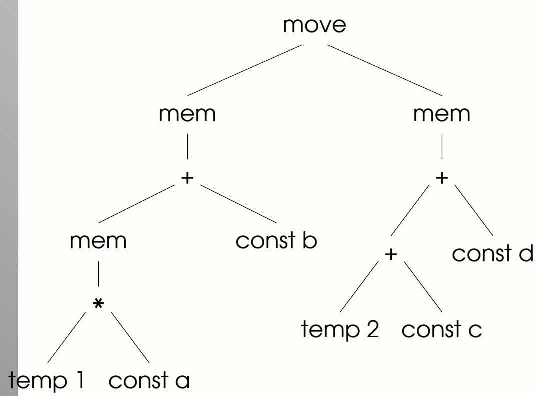
University of Amsterdam

Tree pattern based selection

- Every target instruction is represented by a tree pattern
- Such a tree pattern often has an associated cost
- Instruction selection is done by tiling the IR tree with the instruction tree patterns
- There may be many different ways an IR tree can be tiled, depending on the instruction set

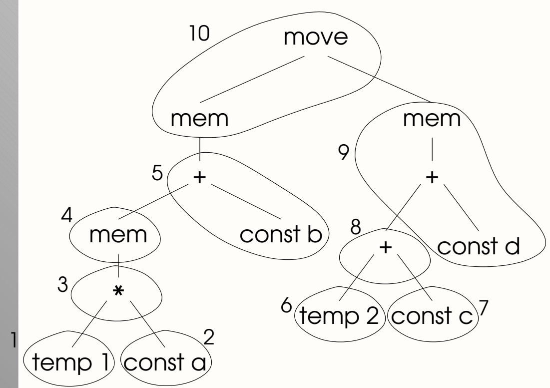
CSA
Computer Systems Architecture

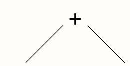
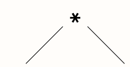
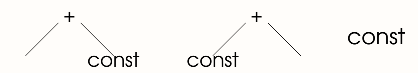
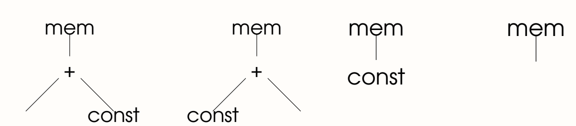
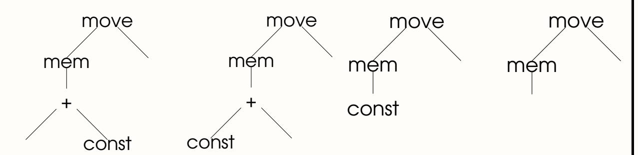
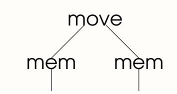
Tree pattern based selection (cont'd)



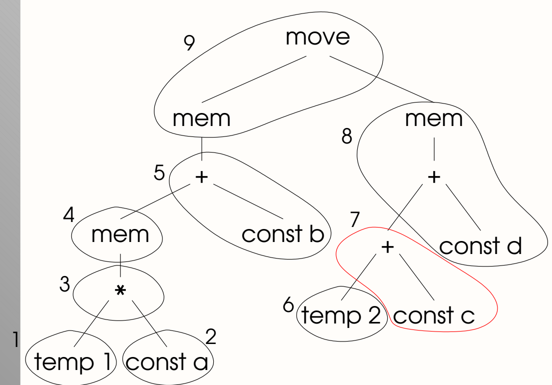
Name	Effect	Trees	Cycles
—	r_i	temp	0
ADD	$r_i \leftarrow r_j + r_k$	+	1
MUL	$r_i \leftarrow r_j * r_k$	*	1
ADDI	$r_i \leftarrow r_j + c$	+ const; + const; const	1
LOAD	$r_i \leftarrow M[r_j + c]$	mem + const; mem + const; mem const; mem	3
STORE	$M[r_j + c] \leftarrow r_i$	move mem + const; move mem + const; move mem const; move mem	3
MOVEM	$M[r_j] \leftarrow M[r_i]$	move mem mem	6

Tree pattern based selection (cont'd)



Name	Effect	Trees	Cycles
—	r_i	temp	0
ADD	$r_i \leftarrow r_j + r_k$		1
MUL	$r_i \leftarrow r_j * r_k$		1
ADDI	$r_i \leftarrow r_j + c$		1
LOAD	$r_i \leftarrow M[r_j + c]$		3
STORE	$M[r_j + c] \leftarrow r_i$		3
MOVEM	$M[r_j] \leftarrow M[r_i]$		6

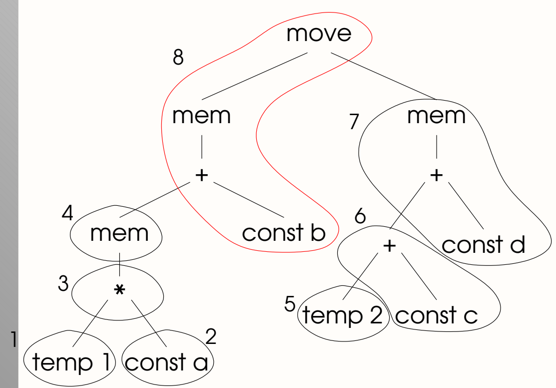
Tree pattern based selection (cont'd)



Name	Effect	Trees	Cycles
—	r_i	temp	0
ADD	$r_i \leftarrow r_j + r_k$	+	1
MUL	$r_i \leftarrow r_j * r_k$	*	1
ADDI	$r_i \leftarrow r_j + c$	+ const; + const; const	1
LOAD	$r_i \leftarrow M[r_j + c]$	mem + const; mem + const; mem const; mem	3
STORE	$M[r_j + c] \leftarrow r_i$	move mem + const; move mem + const; move mem const; move mem	3
MOVEM	$M[r_j] \leftarrow M[r_i]$	move mem mem	6

University of Amsterdam

Tree pattern based selection (cont'd)

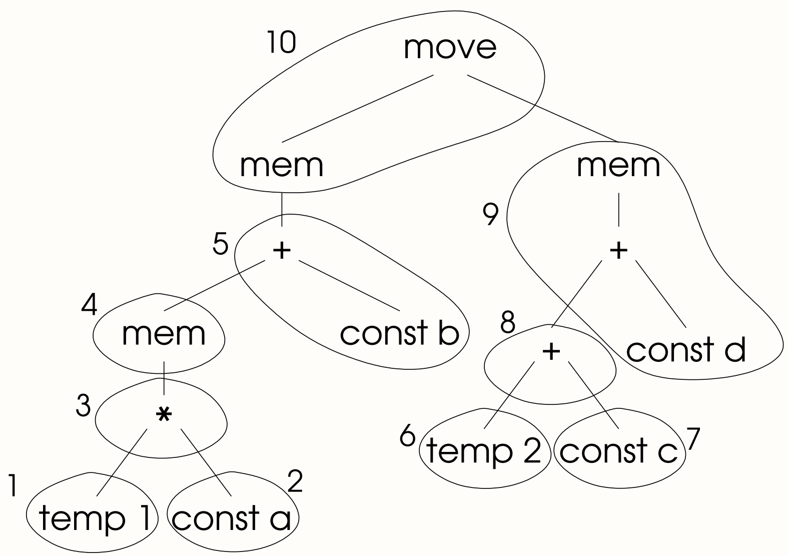


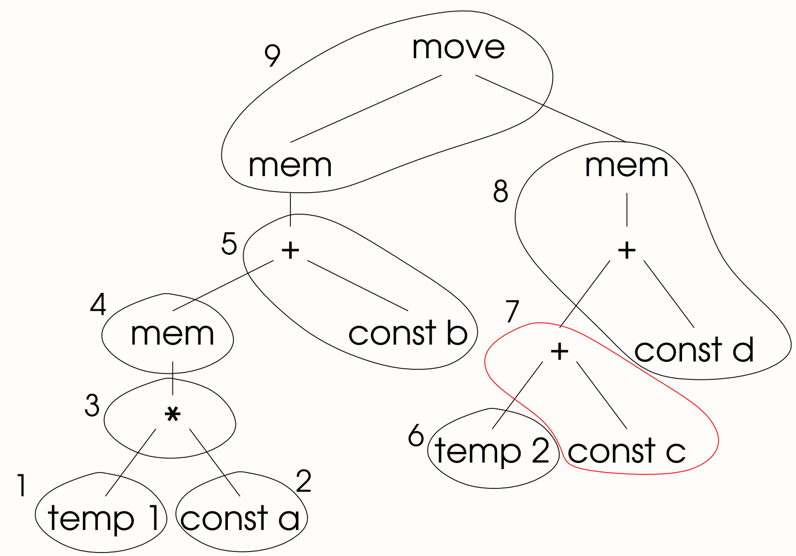
Name	Effect	Trees	Cycles
—	r_i	temp	0
ADD	$r_i \leftarrow r_j + r_k$	+	1
MUL	$r_i \leftarrow r_j * r_k$	*	1
ADDI	$r_i \leftarrow r_j + c$	+ const; + const; const	1
LOAD	$r_i \leftarrow M[r_j + c]$	mem + const; mem + const; mem const; mem	3
STORE	$M[r_j + c] \leftarrow r_i$	move mem + const; move mem + const; move mem const; move mem	3
MOVEM	$M[r_j] \leftarrow M[r_i]$	move mem mem	6

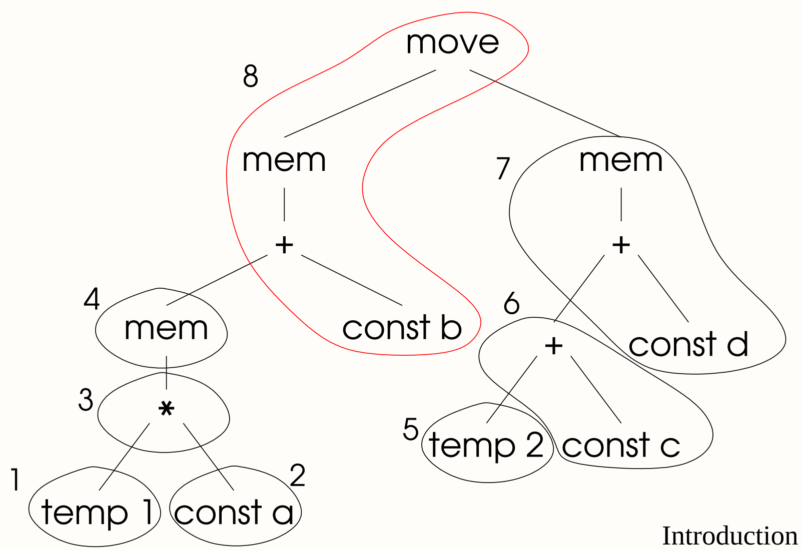
CSA
Computer Systems Architecture

University of Amsterdam

Tiling examples







CSA
Computer Systems Architecture

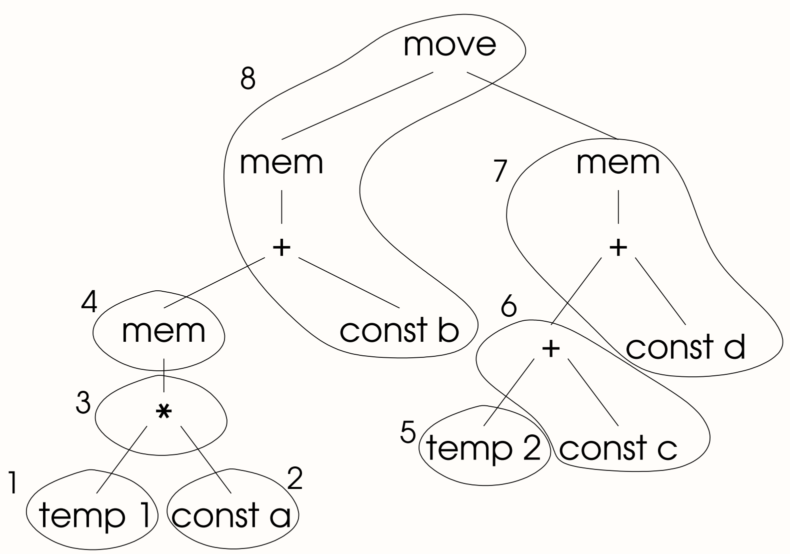
Optimal and optimum tilings

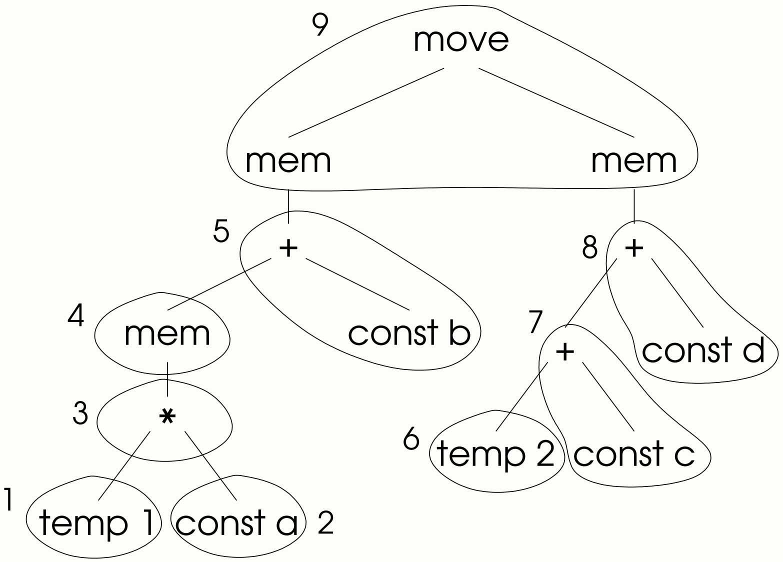
The cost of a tiling is the sum of the costs of the tree patterns

- An optimal tiling is one where no two adjacent tiles can be combined into a single tile of lower cost
- An optimum tiling is a tiling with lowest possible cost

An optimum tiling is also optimal, but not vice-versa

Optimal Tilings

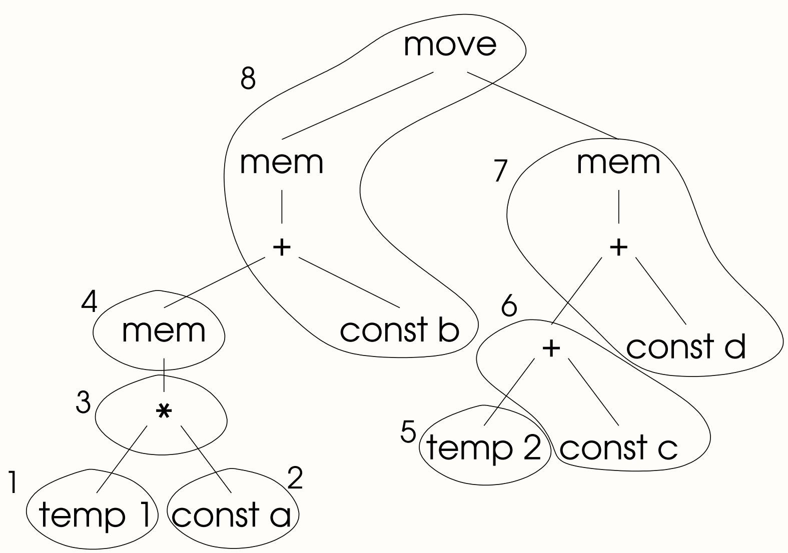


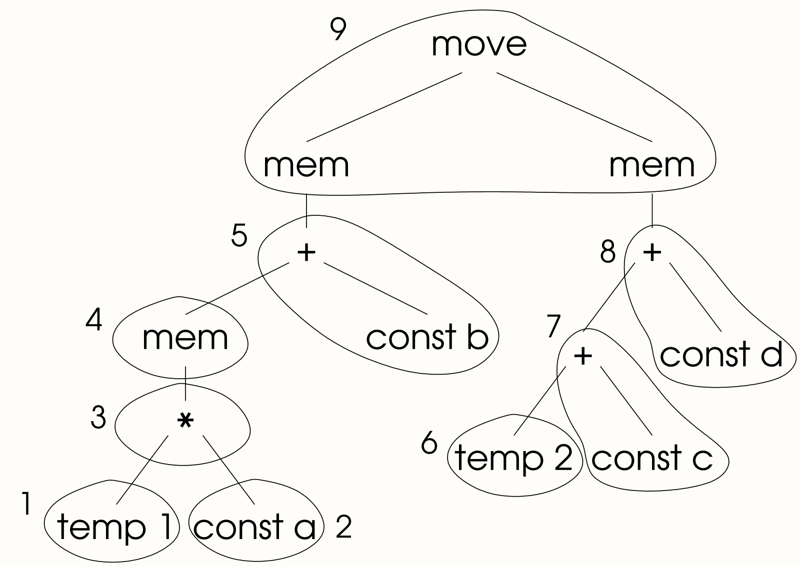



```
2: ADDI    r2 ← r0 + a
3: MUL     r1 ← t1 * r2
4: LOAD    r2 ← M[r1]
6: ADDI    r3 ← t2 + c
7: LOAD    r3 ← M[r3 + d]
8: STORE   M[r2 + b] ← r3
```

```
2: ADDI    r2 ← r0 + a
3: MUL     r1 ← t1 * a
4: LOAD    r2 ← M[r1]
5: ADDI    r3 ← r2 + b
7: ADDI    r2 ← t2 + c
8: ADDI    r2 ← r2 + d
8: MOVEM   M[r3] ← M[r2]
```

Optimum tiling





2: ADDI	r2 ← r0 + a	(1)		2: ADDI	r2 ← r0 + a	(1)
3: MUL	r1 ← t1 * r2	(1)		3: MUL	r1 ← t1 * r2	(1)
4: LOAD	r2 ← M[r1]	(3)		4: LOAD	r2 ← M[r1]	(3)
6: ADDI	r3 ← t2 + c	(1)		5: ADDI	r3 ← r2 + b	(1)
7: LOAD	r3 ← M[r3 + d]	(3)		7: ADDI	r2 ← t2 + c	(1)
8: STORE	M[r2 + b] ← r3	(3)		8: ADDI	r2 ← r2 + d	(1)
				8: MOVEM	M[r3] ← M[r2]	(6)
	total	12			total	14

Maximal Munch

- Maximal Munch is an algorithm for optimal tiling
  - Start at the root of the tree
  - Find the largest pattern that fits
  - Cover the root node plus the other nodes in the pattern; the instruction corresponding to the tile is generated
  - Do the same for the resulting subtrees
- Maximal Munch generates the instructions in reverse order!

Dynamic programming

- Dynamic programming is a technique for finding optimum solutions
  - Bottom up approach
  - For each node *n* the costs of all children are found recursively.
  - Then the minimum cost for node *n* is determined.
- After cost assignment of the entire tree, instruction emission follows:
  - `Emission(node n)`: for each leaves l_i of the tile selected at node *n*, perform `Emission(`l_i`)`. Then emit the instruction matched at node *n*

Register allocation...a graph coloring problem

- First do instruction selection assuming an infinite number of symbolic registers
- Build an interference graph
  - Each node is a symbolic register
  - Two nodes are connected when they are live at the same time
- Color the interference graph
  - Connected nodes cannot have the same color
  - Minimize the number of colors (maximum is the number of actual registers)

Coloring by simplification

- Simplify interference graph G using heuristic method (K-coloring a graph is NP-complete)
  - Find a node m with less than K neighbors
  - Remove node m and its edges from G, resulting in G'. Store m on a stack
  - Color the graph G'
  - Graph G can be colored since m has less than K neighbors

Coloring by simplification (cont'd)

- Spill
  - If a node with less than K neigbors cannot be found in G
    - Mark a node n to be spilled, remove n and its edges from G (and stack n) and continue simplification
- Select
  - Assign colors by popping the stack
  - Arriving at a spill node, check whether it can be colored. If not:
    - The variable represented by this node will reside in memory (i.e. is spilled to memory)
    - Actual spill code is inserted in the program

Coalescing

- If there is no interference edge between the source and destination of a move, the move is redundant
- Removing the move and joining the nodes is called coalescing
- Coalescing increases the degree of a node
- A graph that was K colorable before coalescing might not be afterwards

University of Amsterdam

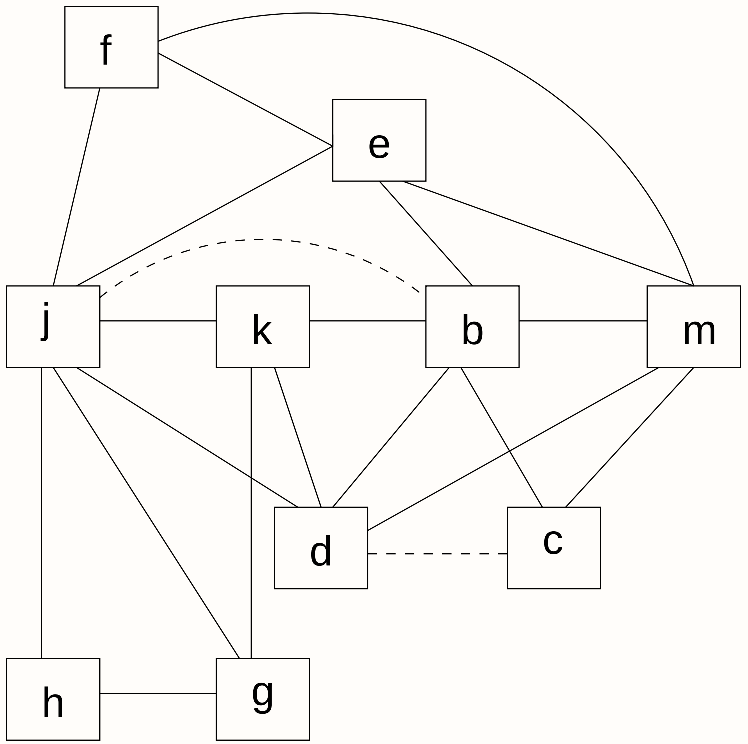
Sketch of the algorithm with coalescing

- Label move-related nodes in interference graph
- While interference graph is nonempty
  - Simplify, using non-move-related nodes
  - Coalesce move-related nodes using conservative coalescing
    - Coalesce only when the resulting node has less than K neighbors with a significant degree
  - No simplifications/coalescings: “freeze” a move-related node of a low degree $\rightarrow$ do not consider its moves for coalescing anymore
  - Spill
- Select

CSA
Computer Systems Architecture

Register allocation: an example

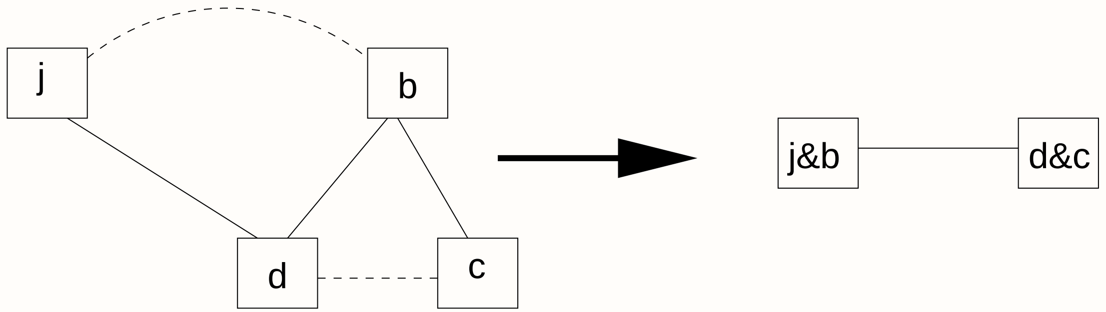
Live in: k,j
g = mem[j+12]
h = k −1
f = g * h
e = mem[j+8]
m = mem[j+16]
b = mem[f]
c = e + 8
d = c
k = m + 4
j = b
goto d
Live out: d,k,j



- Assume a 4-coloring ($K = 4$)
- Simplify by removing and stacking nodes with < 4 neighbors (g,h,k,f,e,m)

Register allocation: an example (cont'd)

- After removing and stacking the nodes g,h,k,f,e,m:

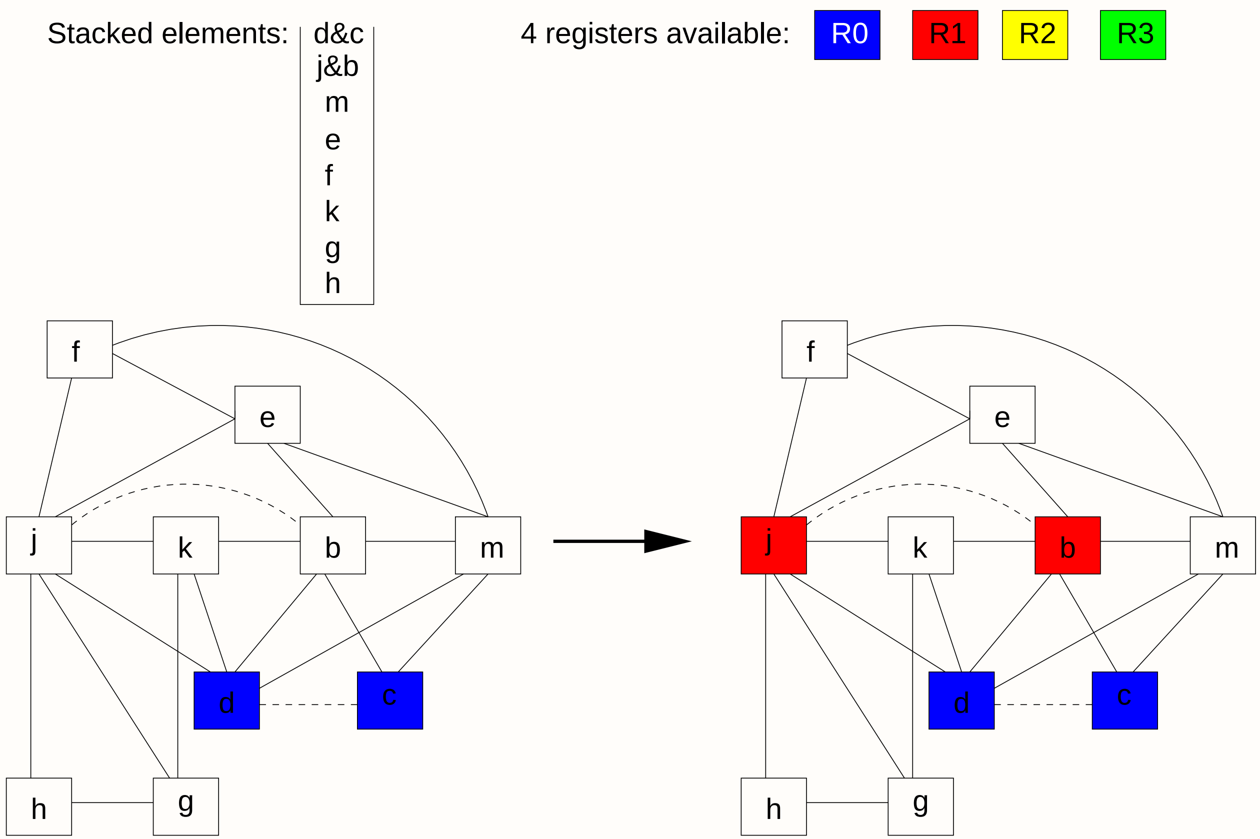


After simplification

After coalescing

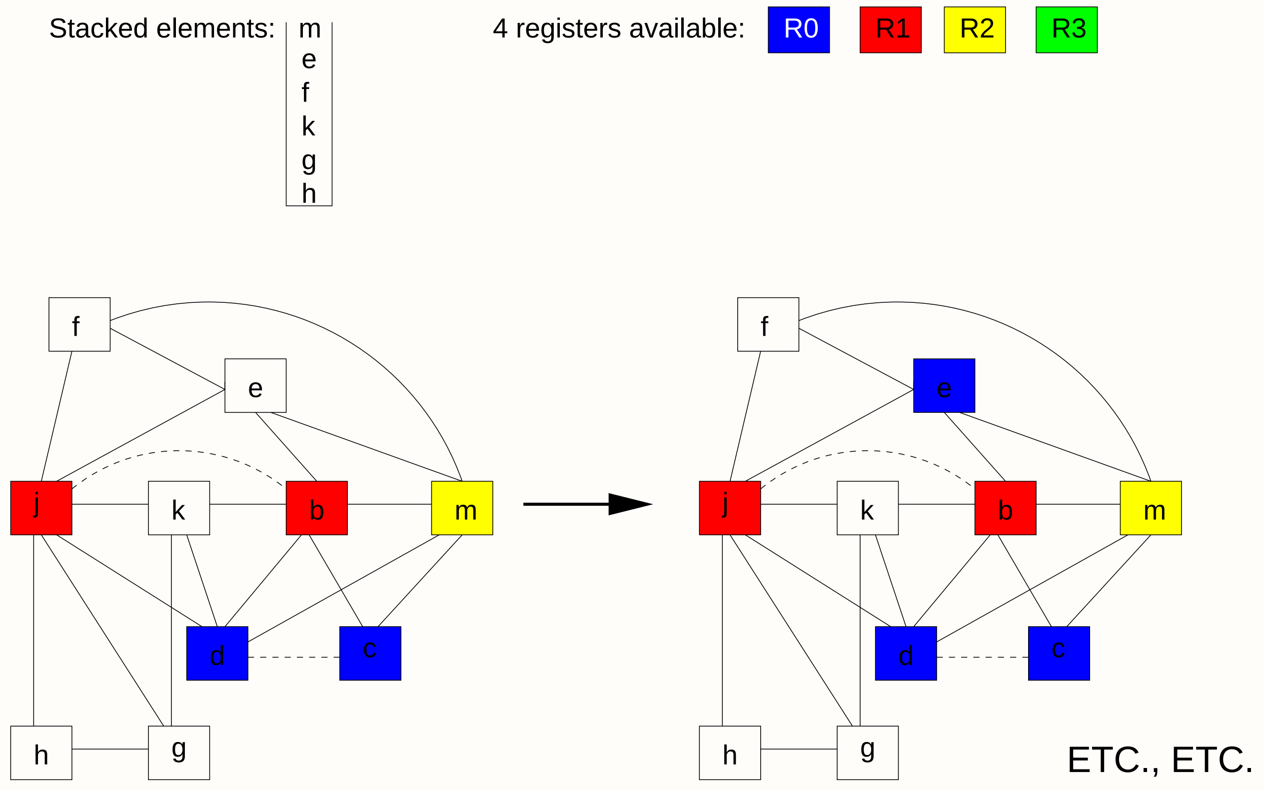
- Coalesce now and simplify again

Register allocation: an example (cont'd)



University of Amsterdam

Register allocation: an example (cont'd)



No spills are required and both moves were optimized away

CSA
Computer Systems Architecture

University of Amsterdam

Instruction scheduling

- Increase ILP (e.g., by avoiding pipeline hazards)
  - Essential for VLIW processors
- Scheduling at basic block level: list scheduling
  - System resources represented by matrix Resources $\times$ Time
  - Position in matrix is true or false, indicating whether the resource is in use at that time
  - Instructions represented by matrices Resources $\times$ Instruction duration
  - Using dependency analysis, the schedule is made by fitting instructions as tight as possible

CSA Computer Systems Architecture

List scheduling (cont'd)

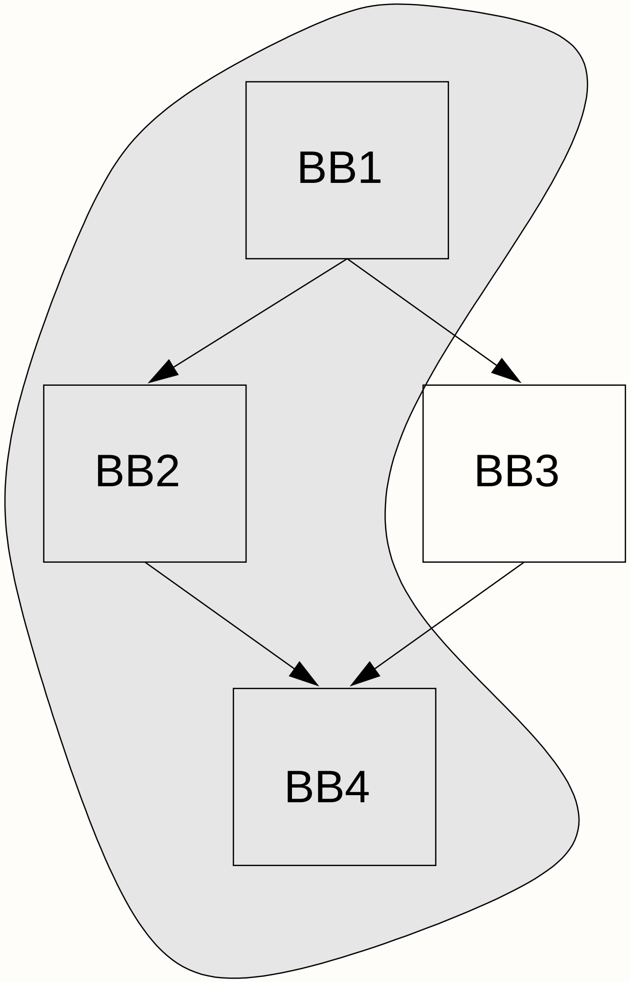
- Finding optimal schedule is NP-complete problem $\Rightarrow$ use heuristics, e.g. at an operation conflict schedule the most time-critical first
- For a VLIW processor, the maximum instruction duration is used for scheduling $\Rightarrow$ painful for memory loads!
- Basic blocks usually are small (5 operations on the average) $\Rightarrow$ benefit of scheduling limited $\Rightarrow$ Trace Scheduling

Trace scheduling

- Schedule instructions over code sections larger than basic blocks, so-called traces
- A trace is a series of basic blocks that does not extend beyond loop boundaries
- Apply list scheduling to whole trace
- Scheduling code inside a trace can move code beyond basic block boundaries $\Rightarrow$ compensate this by adding code to the off-trace edges

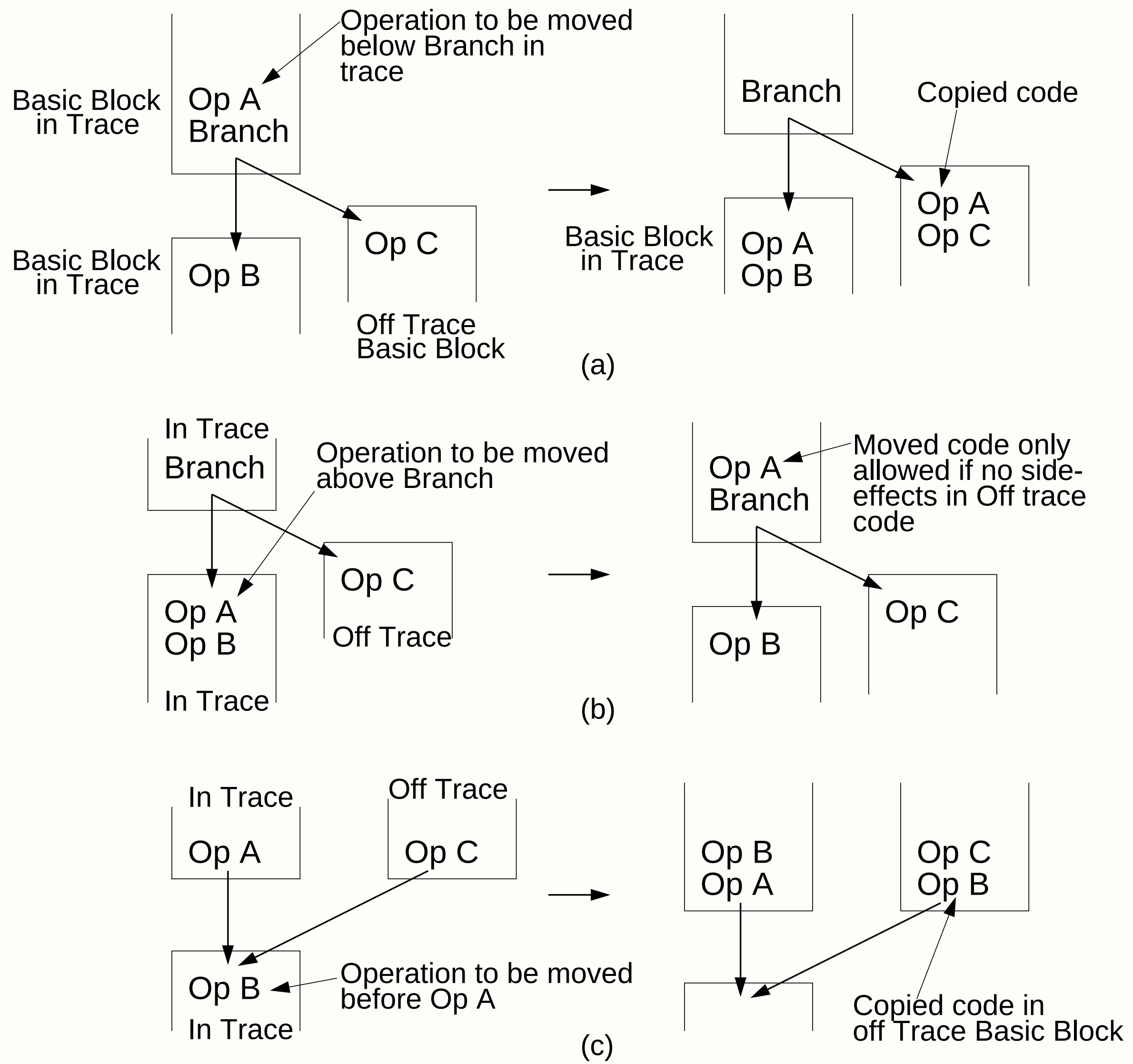
University
of
Amsterdam

Trace scheduling (cont'd)



CSA
Computer
Systems
Architecture

Trace scheduling (cont'd)



Trace scheduling (cont'd)

Trace selection

- Because of the code copies, the trace that is most often executed has to be scheduled first
- A longer trace brings more opportunities for ILP (loop unrolling!)
- Use heuristics about how often a basic block is executed and which paths to and from a block have the most chance of being taken (e.g. inner-loops) or use profiling (input dependent)

University of Amsterdam

Other methods to increase ILP

Loop unrolling

- Technique for increasing the amount of code available inside a loop: make several copies of the loop body
- Reduces loop control overhead and increases ILP (more instructions to schedule)
- When using trace scheduling this results in longer traces and thus more opportunities for better schedules
- In general, the more copies, the better the job the scheduler can do but the gain becomes minimal

CSA
Computer Systems Architecture

Loop unrolling (cont'd)

Example

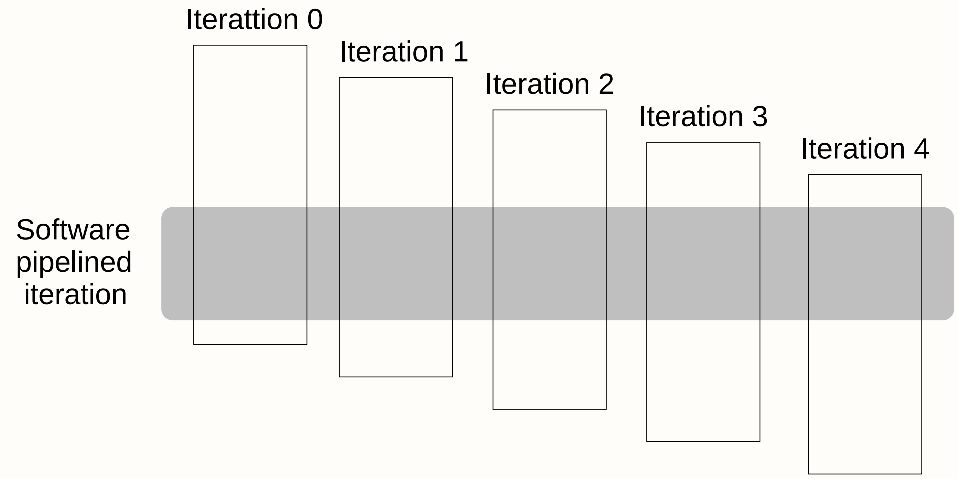
```
for (i = 0; i < 100; i++)
   a[i] = a[i] + b[i];
```

becomes

```
for (i = 0; i < 100; i += 4) {
   a[i]   = a[i]   + b[i];
   a[i+1] = a[i+1] + b[i+1];
   a[i+2] = a[i+2] + b[i+2];
   a[i+3] = a[i+3] + b[i+3];
}
```

Software pipelining

- Also a technique for using the parallelism available in several loop iterations
- Software pipelining simulates a hardware pipeline, hence its name



- There are three phases: Prologue, Steady state and Epilogue

University of Amsterdam

CSA
Computer Systems Architecture

Software pipelining (cont’d)

Loop:	LD	F0,0(R1)	Body
	ADDD	F4,F0,F2	
	SD	0(R1),F4	
	SBGEZ	R1, Loop	Loop control

Prologue	T0	LD				
	T1	·	LD			
	T2	ADDD	·	LD		
Steady state	T... Loop:	SD	ADDD	·	LD	SBGEZ Loop
Epilogue	Tn		SD	ADDD	·	
	Tn+1			SD	ADDD	
	Tn+2				SD	

Modulo scheduling

- Scheduling multiple loop iterations using software pipelining can create false dependencies between variables used in different iterations
- Renaming the variables used in different iterations is called modulo scheduling
- When using *n* variables for representing the same variable, the steady state of the loop has to be unrolled *n* times

Compiler optimizations for cache performance

- Merging arrays (better spatial locality)

int val[SIZE];		struct merge {
int key[SIZE];	$\Rightarrow$	int val, key; };
		struct merge m_array[SIZE]

- Loop interchange
- Loop fusion and fission
- Blocking (better temporal locality)

Loop interchange

- Exchanging of nested loops to change the memory footprint
  - Better spatial locality

```
for (i = 0; i < 50; i++)
   for (j = 0; j < 100; j++)
      a[j][i] = b[j][i] * c[j][i];
```

becomes

```
for (j = 0; j < 100; j++)
   for (i = 0; i < 50; i++)
      a[j][i] = b[j][i] * c[j][i];
```

University of Amsterdam

Loop fusion

- Fuse multiple loops together
  - Less loop control
  - Bigger basic blocks (scheduling)
  - Possibly better temporal locality

```
for (i = 0; i < n; i++)
   c[i] = a[i] + b[i];
for (j = 0; j < n; j++)
   d[j] = a[j] * e[j];
```

becomes

```
for (i = 0; i < n; i++) {
   c[i] = a[i] + b[i];
   d[i] = a[i] * e[i];
}
```

CSA
Computer Systems Architecture

Loop fission

- Split a loop with independent statements into multiple loops
  - Enables other transformations (e.g. vectorization)
  - Results in smaller cache footprint (better temporal locality)

```
for (i = 0; i < n; i++) {
   a[i] = b[i] + c[i];
   d[i] = e[i] * f[i];
}
```

becomes

```
for (i = 0; i < n; i++) {
   a[i] = b[i] + c[i];
}
for (i = 0; i < n; i++) {
   d[i] = e[i] * f[i];
}
```

Blocking

Perform computations on sub-matrices (blocks), e.g. when multiple matrices are accessed both row by row and column by column

Matrix multiplication x = y*z

```
for (i=0; i < N; i++)
    for (j=0; j < N; j++) {
        r = 0;
        for (k = 0; k < N; k++) {
            r = r + y[i][k]*z[k][j];
        };
        x[i][j] = r;
    };
```