Vorlesung und Übung

Compiler-Optimierung für eingebettete Systeme

Part 05: Platform Independent Optimizations

Prof. Dr. Sabine Glesner

WS 2008/09

Technische Universität Berlin

Outline – Whole Lecture

- Introduction: Compiler
- Data Flow Analyses
- Control Flow Analyses
- Optimizations
  - Platform-independent
  - Platform-dependent
- Register Allocation
- Code Generation
- Techniques for Parallelization
- Current Research Topics for Embedded Systems
  - MPSoC (Multi-Processor System-on-a-Chip)
  - Dynamical Reconfigurable Systems

Motivation

- Control flow analyses
  - Vital for all optimizations
  - Dominator Analysis essential
    - for code motion
    - for global instruction scheduling
  - Loop Detection important (especially for Embedded Systems)
    - Programs spend most of the time in its execution
    - Optimizations of loop bodies have significant effect on runtime

Outline – This Lecture

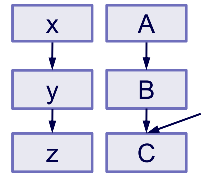
- Control Flow Analysis
  - Dominator Analysis
    - SSA Construction
    - Definition of Control Dependence
  - Loop Detection
- Platform-independent Optimizations
  - Global Value Numbering (GVN)
  - Partial Redundancy Elimination (PRE)
  - Transformations of SSA-Form
  - PRE on SSA-Form (GVN-PRE)

Dominator Analysis

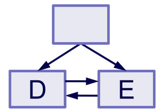
- **Def.**: Given a CFG, a basic block *x* dominates a block *y*, if every path from the entry block to *y* contains *x*
  - Also called "plain dominance"
  - Written *x* **dom** *y*
- Can be formulated as data flow problem
  - N: set of all blocks
  - *Forward, must* analysis

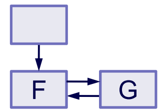
	Dominator Analysis
$\mathcal{P}(\boldsymbol{D})$	power set of N
$\sqcup$	$\cap$
extremal value ι	$\emptyset$
extremal labels E	{init(S_*)}
flow F	flow(S_*)
transfer function $f_B(X)$	$X \cup \{B\}$

Dominance - Properties

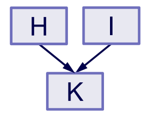


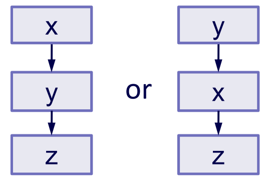
- Reflexivity: *x* dom *x*
- Transitivity: *x* dom *y*, and *y* dom *z* $\Rightarrow$ *x* dom *z*
  - Not A dom C because not B dom C
- Antisymmetry: *x* dom *y*, and *y* dom *x* $\Rightarrow$ *x* = *y*
  - Not D dom E, not E dom D
  - F dom G but not G dom F





- Tree structure: *x* dom *z* and *y* dom *z* $\Rightarrow$ *x* dom *y* or *y* dom *x*
  - Not H dom K, not I dom K



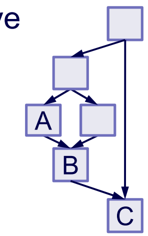


Dominance - Definitions

- Strict dominance
  - Written *x* sdom *y*
  - Consider two basic blocks *x* and *y* such that *x* dom *y*
  - If *x* and *y* are not the same block, then *x* **sdom** *y*
- Immediate dominance
  - Written *x* idom *y*
  - Consider two basic blocks *x* and *y* such that *x* sdom *y*
  - *x* **idom** *y* if there is no block *z*, such that *x* sdom *z* and *z* sdom *y*
- Dominator tree
  - Written domtree
  - Each node's children are those blocks it immediately dominates
    - Because the immediate dominator is unique, it is a tree
    - The entry node of the CFG is the top of the tree.

Dominance Frontier

- Dominance frontier
  - Written domfront(*x*)
  - The set of blocks y_S of a block *x* such that *x* plainly dominates a predecessor of y_S but not *x* sdom y_S.
  - *Intuitively*: all direct successors that are merge points of control flow paths not including *x*
- Remark: domfront is not transitive
  - B $\in$ domfront(A)
  - C $\in$ domfront(B)
  - C $\notin$ domfront(A)



- domfront can be used to construct SSA form of IR

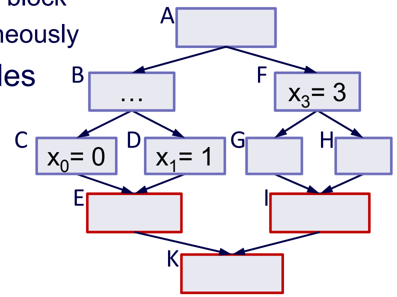
Repetition: SSA

- Use-definition-relation explicit
  - Simplifies optimizations
- Construction
  - Split variables into versions
  - Φ nodes required at joins
    - Placing at beginning of a basic block
    - All Φ nodes evaluated simultaneously
- *Question*: where to place Φ nodes
  - ❖ All merge points of control flow?
  - ➢ No, not minimal
  - ✓ Answer: use dominance frontier

```
c= 0
do{
  c= c + 1
}while(c<10)
```

⇒

```
c0= 0
do{
  c1= Φ(c0, c2)
  c2= c1 + 1
}while(c2<10)
```

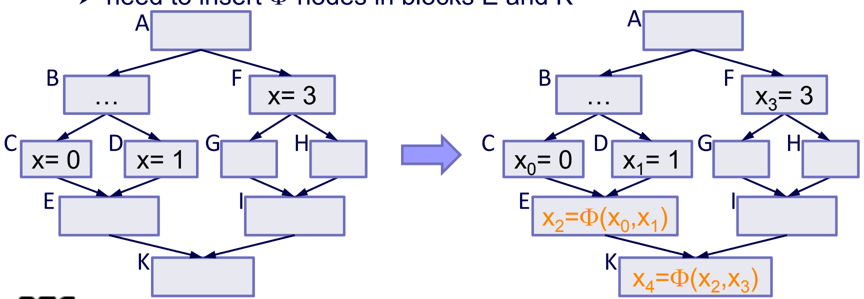


Placing of Φ Nodes

- Extend frontier relationship to set of nodes
  domfront(S)= $\cup_{x \in S}$ domfront(x)
- Build transitive, irreflexive closure of domfront relation:
  domfront$_1$= domfront(S)
  domfront$_{i+1}$=domfront($S \cup$ domfront$_i$)
  - Called *iterated* dominance frontier
- **For any variable *v*:**
  - The set of basic blocks that needs Φ-nodes is the iterated dominance frontier of *S*, where *S* is the set of basic blocks with assignments to *v*

Placing of Φ Nodes - Example

- *S*={C,D,F}
  - domfront(F) = ∅
  - domfront(C) = domfront(D) = {E} = domfront(*S*) = domfront$_1$
  - domfront(E) = {K}
  - domfront(K) = ∅
  - ➢ *iterated* domfront({C,D,F,E,K}) = {E,K}
  - ➢ need to insert Φ-nodes in blocks E and K



SSA Construction

- What is left?
- ➢ Renaming the variables
  - Required data structures
    - Array of stacks
      - one for each variable, initial empty
      - holds the subscript of the most recent definition
    - Array of counters
      - one for each variable, initial 0
      - holds the number of assignments to V processed

```
procedure GenName(Variable V)
    i ← Counter[V]
    replace V by Vi
    Push i onto Stack[V]
    Counter[V] ← i + 1
```

SSA Construction (2)

- Renaming walks preorder over the dominator tree

```
procedure Rename(Block X)
    for each Φ-node P in X :  GenName(LHS(P))    //first process Φ-nodes
    for each statement A in X :   //process statements in block X
        for each variable V ∈ RHS(A) :  replace V by V_i where i = Top(Stack[V])
        for each variable V ∈ LHS(A) :  GenName(V)
    for each Y ∈ SUCC(X) :    //update any Φ-function in CFG successors of X
        j ← position in Y's Φ-nodes corresponding to X
        for each Φ-node P in Y :
            replace the j^th operand of RHS(P) by V_i  (i = Top(Stack[V]))
    for each Y ∈ CHILDS(X) :  Rename(Y) //recursively visit children of X in domtree
    //when backing out of X, pop variables defined in X
    for each Φ-node or statement A in X :
        for each V_i ∈ LHS(A) :  Pop (Stack[V])
```

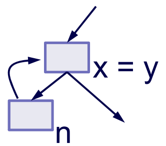

Post-dominance

- Dual to dominance:
  - Given a CFG, a basic block *y* post-dominates a block *x*, if every path from *y* to the exit block contains *x*
  - Written *x* pdom *y*
- Analog: strict, immediate, frontier, tree
- Caveat: *x* dom *y* does not imply *y* pdom *x*
- Post-dominance is dominance in reverse CFG
  - Reversing direction of all edges
  - Interchanging roles of entry and exit block
- Can be used to define Control Dependence

Control Dependence

- Node *y* is control-dependent on a node *x* if
  - *Intuitive*: node *x* determines whether *y* is executed
  - *Formal*: there exists a path *P* from *x* to *y* such that
    - every node *n* ≠ *x* within *P* will be followed by *y* in each possible path to the end of the program and
    - *x* will not necessarily be followed by *y*, i.e. there is an execution path from *x* to the end of the program that does not go through *y*
  - *Equivalent to*:
    - *y* post-dominates all nodes *n* within *P*
    - if *y* is not *x*, *y* does not post-dominate *x*

Required, because pdom is reflexive

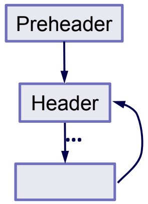


Outline – This Lecture

- Control Flow Analysis
  - Dominator Analysis
    - SSA Construction
    - Definition of Control Dependence
  - Loop Detection
- Platform-independent Optimizations
  - Global Value Numbering (GVN)
  - Partial Redundancy Elimination (PRE)
  - Transformations of SSA-Form
  - PRE on SSA-Form (GVN-PRE)

Loop Concepts

- Loop Entry Edge
  - Source of edge outside the loop and target in loop
- Loop Exit Edge
  - Target of edge outside the loop and source in loop
- Loop Header
  - Block inside loop that dominates all loop blocks
- Loop Preheader
  - Single block that is source of the loop entry edge
    - Only successor is the Loop Header
    - Executed once while invoking the loop
    - Never executed while the loop iterates
  - Not all loops have a preheader
    - Useful to create them (for invariant code motion)



Loop Variables

- Induction Variable
  - A variable that gets increased or decreased by a fixed amount on every iteration of a loop
- Basic Induction Variable
  - Induction variable *v* whose only assignments within a loop are of the form *v* += *C*, where *C* is a constant (loop invariant expression)
- Primary Induction Variable
  - Basic induction variable that controls the loop execution
- Derived Induction Variable
  - A variable that is a *constant* linear function of a basic induction variable

Loop Characteristics

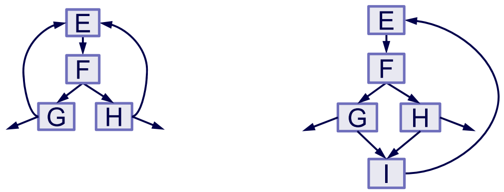
- Nesting
  - Inner Loop: contains no other loop
  - Outer Loop: contained within no other loop
  - Depth: starts at 1 (outer loop), +1 per nesting

- (Average) Trip Count
  - How many times (on average) does the loop iterate

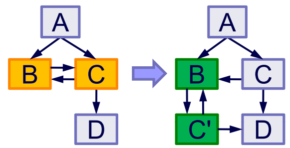
- Natural Loop
  - connected subgraph of CFG with a *single* entry point
  - Unless two loops have the same header
    - they are either disjoint or
    - one is entirely contained (nested within) the other

Natural Loop

- When two loops have the same header but neither is nested within each other, they are treated as a single loop
  - E-F-G-H single loop, because equivalent to loop E-F-G-H-I



- Not all CFG cycles are natural loops
  - Less potential for optimizations with entry points into middle of cycle
  - Cycle B-C is not a natural loop
  - Can be converted with duplicating
    - Cycle B-C' now a natural loop



Reducible CFG

- CFG is called *reducible* iff every cycle is a natural loop
  - goto-less programs always reducible
  - **Formal Definition**: A CFG is reducible iff we can partition its edges into 2 disjoint sets, the forward and the back edges, such that
    - The forward edges form an acyclic graph in which every node can be reached from the entry
    - The back edges consist only of edges whose targets dominate their sources
- Reducibility often required for optimizations

Loop Detection

1. Identify all backedges in the CFG using dominance info
   - All edges $x \rightarrow y$, where y dom x
2. Each backedge defines a natural loop
   - Source block called *Loop Tail*
3. Compute loop blocks
   - The natural loop of a back edge $x \rightarrow y$ is the set of nodes b_L such that y dominates b_L and there is a path from b_L to x not containing y
4. Merge loops with same header

Outline – This Lecture

- Control Flow Analysis
  - Dominator Analysis
    - SSA Construction
    - Definition of Control Dependence
  - Loop Detection
- Platform-independent Optimizations
  - ➡ Global Value Numbering (GVN)
  - Partial Redundancy Elimination (PRE)
  - Transformations of SSA-Form
  - PRE on SSA-Form (GVN-PRE)

Motivation

- Platform independent optimizations
  - perform high-level code-improving transformations
  - require little or no knowledge about the Instruction Set Architecture
  - operate on high-level IR
  - ➢ are highly retargetable
- GVN eliminates redundant expressions (like CSE)
  - Finds redundancy which CSE cannot

Global Value Numbering GVN

- Objectives
  - Assign a unique number to each variable, expression, and constant
    - Value numbers used to represent symbolic expressions
    - Same number ⇒ same value
  - Algebraic identities to simplify expressions
  - Discover redundant computations and replace them
  - Fold and propagate constant values
- History
  - Local version using hashing: Cocke & Schwartz, 1969
  - Further algorithms for extended blocks, dominator regions, procedures
  - Alpern, Wegman & Zadeck, "Detecting Equality of Variables in Programs", *POPL 1988*

Local GVN

- Equivalence based solely on facts from within block
  - When encountering a variable, expression or constant, see if it's already been assigned a number
    - If so, use the value
    - If not, assign a new number (storing in hash table)
  - ➢ If an instruction's value number is already defined, it can be eliminated and subsequent references subsumed
  - ➢ Constant Folding / Propagation is simple

Local GVN - Example

- Cannot be found with CSE alone
  1. only in conjunction with Copy Propagation

```
a= x + y        x ← #1       z ← #1      z+y ← #3        t= x + y
z= x       ⇒    y ← #2                   b ← #3     ⇒    a= t
b= z + y        x+y ← #3                                 z= x
                a ← #3                                   b= t
```

  2. only in conjunction with Constant Propagation

```
a= x + 3        x ← #1       z ← #2      x+z ← #3        t= x + 3
z= 3       ⇒    3 ← #2                   b ← #3     ⇒    a= t
b= x + z        x+3 ← #3                                 z= 3
                a ← #3                                   b= t
```


Local GVN – Algorithm

- For each instruction i : x = y op z in the block

  $V_1 \leftarrow$ ValNum[y] create if necessary

  $V_2 \leftarrow$ ValNum[z] create if necessary

  let v = Hash(*op*, V_1, V_2)

  if (v exists in hash table)

      replace RHS with Name[v]

  else

      enter v in hash table

      Name[v] $\leftarrow t_i$ (new temporary)

      replace instruction with: t_i = y *op* z; x = t_i

  ValNum[x] $\leftarrow$ v

Local GVN – Algebraic Identities

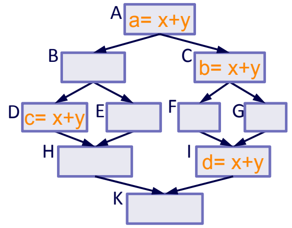
- Check many special cases
  - Commutativity of operations
    - Search for all possible orderings of operands

      Or
    - Order operands before
  - $x \pm 0$
  - $x * 1$, $x / 1$, $x * 0$
  - max(x, x) , min(x, x)
  - x - x
- Replace with corresponding value number / simpler construct

Extension to Global GVN

- Local version: new hash table per basic block

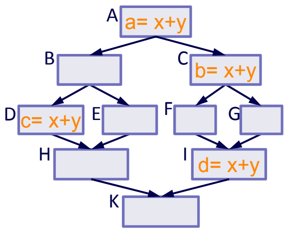
1. Idea: Consider "Extended Basic Blocks" (EBB)
   - Set of blocks b_i: b_1 with >=1 predecessor, b_i 1predecessor (i>1)
     - Tree with control flow join as root
     - Here: blocks A – G
   - Reuse hash table from unique predecessor in EBB
     - B's and C's hash table initialized with A's
     - D's and E's hash table initialized with AB's and so on
   - No reuse for block I
     - Initialization with ACF's or ACG's??
2. Idea: Use idom's hash table at joins



Extension to Global GVN

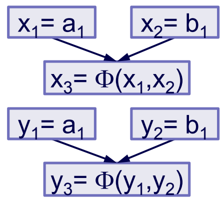
- Local version: new hash table per basic block

1. Idea: Consider "Extended Basic Blocks" (EBB)
2. Idea: Use idom's hash table at joins
   - Remove expressions killed in between
   - Problem with new defined variables in between
     - which to use??
   - called Dominator VN Technique (DVNT)



GVN

- Alpern, Wegman & Zadeck (1988)
  - Partition the *n* program variables into congruence classes
    - Variables in a particular congruence class have the same value
    - SSA form is helpful
      - Allows to avoid data flow analysis
      - Variables correspond to values
  - Instructions *i* and *j* are congruent iff they have the same operator and their operands are congruent
    - Problem: not true for Φ-functions
      - x_1 and y_1 congruent
      - x_2 and y_2 congruent
      - x_3 and y_3 NOT congruent
        - because Φ is dynamic choice
    - Solution: Label Φ-functions with join point number



GVN – Congruence

- Approaches to computing congruence classes
  - Pessimistic
    - Assume no variables are congruent (start with *n* classes)
    - Iteratively coalesce classes that are determined to be congruent
      - Consider each assignment (reverse postorder in CFG)
      - Update LHS value number with hash of RHS
      - Identical value number ⇒ congruence
    - Reverse postorder
      - Ensures that while considering an assignment, the definitions that reach the RHS operands where already considered
      - Problem: Can't deal with code containing loops
      - Solution
        - Ignore back edges
        - Make conservative (worst case) assumption for previously unseen variable (i.e., assume it is in its own congruence class)
  - Optimistic

GVN – Congruence

- Approaches to computing congruence classes
  - Pessimistic
  - Optimistic
    - Assume all variables are congruent (start with one class)
    - Iteratively partition variables that contradict assumption
    - Split classes when evidence of non-congruence arises
      - Variables that are computed using different functions
      - Variables that are computed using functions with non-congruent operands
    - Slower but better results

GVN – Optimistic Algorithm

1. Partition instructions into congruence classes by opcode
2. *worklist* ← all classes
3. while (*worklist* is not empty)

   remove a class *C* from worklist

   for each class *S* that uses some x ∈ *C*

   split S into S & S′: all users of *C* in one class

   put smaller of S & S′ onto worklist
4. Select a representative for each partition & perform replacement

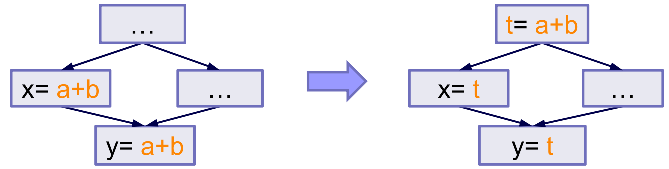
Outline – This Lecture

- Control Flow Analysis
  - Dominator Analysis
    - SSA Construction
    - Definition of Control Dependence
  - Loop Detection
- Platform-independent Optimizations
  - Global Value Numbering (GVN)
  - Partial Redundancy Elimination (PRE)
  - Transformations of SSA-Form
  - PRE on SSA-Form (GVN-PRE)

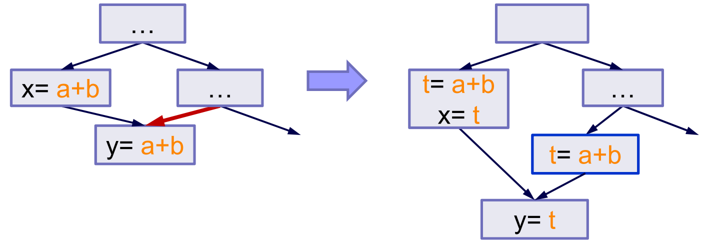
Partial Redundancy Elimination PRE

- Expression is partially redundant if already available on some but not necessarily all paths to that expression
- Goal:
  - Place calculations such that no path re-executes same expression
- Constraints on placement
  - No wasted operation
  - No branches that lead to exit without use
  - Calculation as late as possible
    (shortens liveness ranges ➔ minimizes register usage)
- Subsumes
  - Global Common Subexpression Elimination (full redundancy)
  - Loop Invariant Code Motion (partial redundancy for loops)

PRE - Example



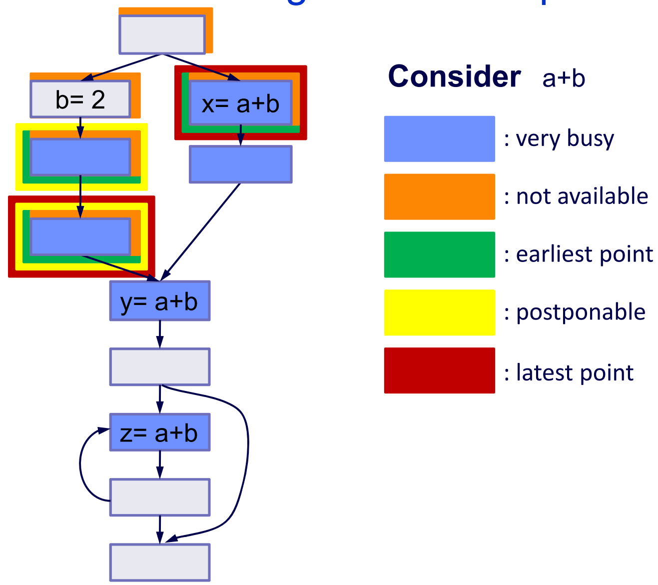
- Restricted by Critical Edges
  - Source block has multiple successors, target multiple predecessors
- Solution: Insertion of a Basic Block



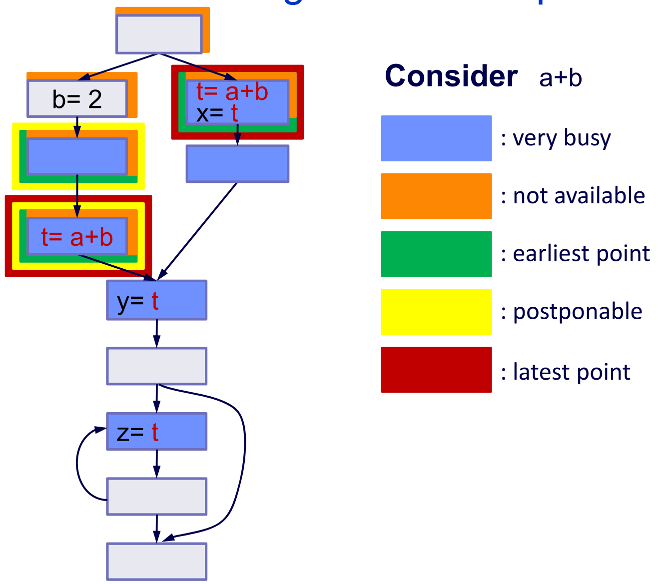
PRE - Algorithm

1. Collect expressions which are "Very Busy"
2. Place expression at the earliest point
3. Delay computation as much as possible
4. Remove temporary assignments unused afterwards
5. Code Transformation

PRE – Algorithm Example



PRE – Algorithm Example



PRE - Algorithm

1. Collect expressions which are "Very Busy"
   - Also called "Anticipated"
   - *Backward*, *must* analysis
     - Its computed value will be used along *all* subsequent paths
     - $Antic[B].IN = (Antic[B].OUT \setminus expr_{Killed}[B]) \cup expr_{Used}[B]$
     - Antic[B].IN: set of very busy expressions at entry of B
   - ➢ Cannot execute any operations not executed originally
   - ➢ Range of code motion
2. Place expression at the earliest point
3. Delay computation as much as possible
4. Remove temporary assignments unused afterwards
5. Code Transformation

PRE - Algorithm

1. Collect expressions which are "Very Busy"
2. Place expression at the earliest point
   - Very busy and not already available
     - Requires modified "Available Expression"-Analysis
       - *Forward, must* Analysis
       - Depends on result of anticipated expressions
       - $Avail[B].OUT = (Avail[B].IN \cup Antic[B].IN) \setminus expr_{Killed}[B]$
     - $Earliest[B] = Antic[B].IN \setminus Avail[B].IN$
   - ➢ Eliminate as many redundant calculations of an expression as possible
3. Delay computation as much as possible
4. Remove temporary assignments unused afterwards
5. Code Transformation

PRE - Algorithm

1. Collect expressions which are "Very Busy"
2. Place expression at the earliest point
3. Delay computation as much as possible
   - Place it at latest point p where it is "Postponable"
     - Postponable
       - All paths leading to p have seen the earliest placement but not a subsequent use
       - *Forward, must* analysis
       - $Postpon[B].OUT = (Postpon[B].IN \cup Earliest[B]) - expr_{Used}[B]$
     - $Latest[B] = (\boxed{Earliest[B] \cup Postpon[B].IN) \cap}$ $\boxed{(expr_{Used}[B] \cup \neg(\cap_{S \in Succ(B)}(Earliest[B] \cup Postpon[B].IN}))$
   - ➢ Move it down unless it creates redundancy (Lazy Code Motion)
   - ➢ Minimizes register lifetimes

OK to place: earliest or postponable

Need to place: used in B or not OK to place in one of its successors

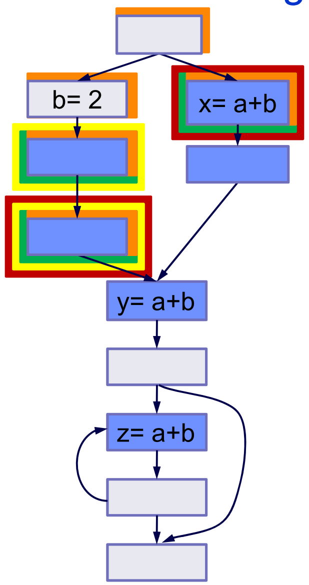
PRE - Algorithm

1. Collect expressions which are "Very Busy"
2. Place expression at the earliest point
3. Delay computation as much as possible
4. Remove temporary assignments unused afterwards
   - Compute sets of used (live) expressions
     - *Backward*, *may* analysis
     - Used[B].IN= (Used[B].OUT ∪ $expr_{Used}$[B]) \ Latest[B]
   - Used[B].OUT: set of live expressions at exit of B
   - Insert assignment only if in Used[B].OUT
- Code Transformation

PRE - Algorithm

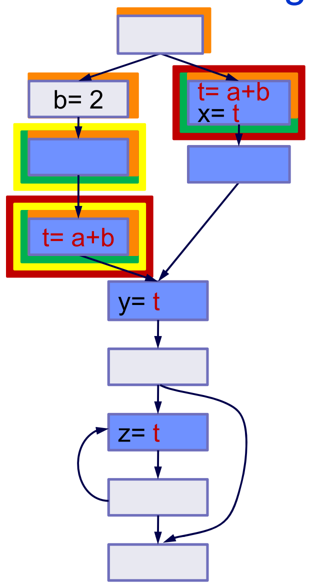
1. Collect expressions which are "Very Busy"
2. Place expression at the earliest point
3. Delay computation as much as possible
4. Remove temporary assignments unused afterwards
5. Code Transformation
   - For all basic blocks B
     - if (x+y) ∈ (Latest[b] ∩ Used[B].OUT)
       - at beginning of b: add new t = x+y
     - if (x+y) ∈ ($expr_{Used}$[B] ∩ ¬(Latest[b] ∩ ¬Used[B].OUT))
       - replace every original x+y by t

PRE – Algorithm Example



Consider a+b

PRE – Algorithm Example



Consider a+b

Outline – This Lecture

- Control Flow Analysis
  - Dominator Analysis
    - SSA Construction
    - Definition of Control Dependence
  - Loop Detection
- Platform-independent Optimizations
  - Global Value Numbering (GVN)
  - Partial Redundancy Elimination (PRE)
  - Transformations of SSA-Form
  - PRE on SSA-Form (GVN-PRE)

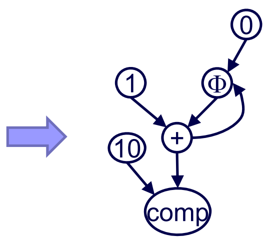
SSA Transformations

- Goal: normalize expressions to make those comparable, which are written differently
  - Observation: syntactically equivalent expressions are also semantically equivalent
    - Not true without SSA
- Kinds:
  - Algebraic equivalence operations
    - Commutativity
    - Distributivity
  - Eliminate operations
  - Constant Folding
- Operate on SSA graph

SSA Graph

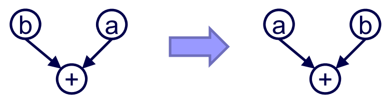
- Nodes are constants, operations or Φ-constructs
  - Defines abstract values
- Edges point from definitions to uses
  - Reversal of edges corresponds to data dependences

```
c0= 0
do{
  c1= Φ(c0, c2)
  c2= c1 + 1
}while(c2<10)
```



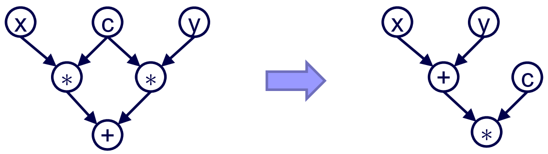
Commutative Transformation

- Define order $\mathcal{N}$ on SSA nodes
- Reorder commutative operations γ(a,b) such that $\mathcal{N}$(a)>$\mathcal{N}$(b)



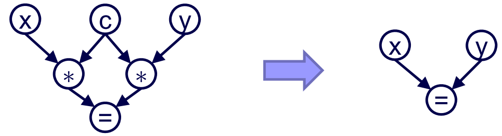
Distributive Transformation

- Define order $\mathcal{O}$ on operations
- Reorder operations γ,δ such that $\mathcal{O}(\gamma) > \mathcal{O}(\delta)$

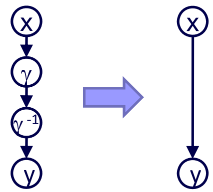


Eliminate Operations

- Values only used for comparisons
  – Eliminate side-effect-free operations

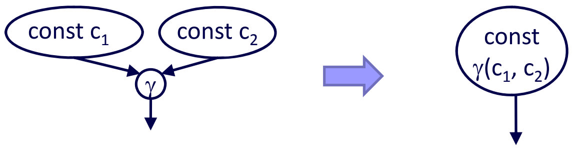


- Eliminate inverse operations
  – Often necessary after other transformations

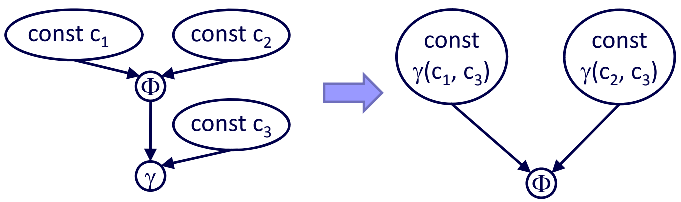


Constant Folding

- Propagate constants and evaluate constant expressions



- Also possible across Φ nodes



Outline – This Lecture

- Control Flow Analysis
  – Dominator Analysis
    - SSA Construction
    - Definition of Control Dependence
  – Loop Detection
- Platform-independent Optimizations
  – Global Value Numbering (GVN)
  – Partial Redundancy Elimination (PRE)
  – Transformations of SSA-Form
  – PRE on SSA-Form (GVN-PRE)

PRE on SSA

- PRE on SSA is current research topic
  - Bodík, Gupta, Soffa: *Complete Removal of Redundant Expressions*, PLDI 1998
  - Kennedy, Chan et al.: *Partial Redundancy Elimination in SSA form,* TOPLAS 1999
  - VanDrunen, Hosking: *Value-Based Partial Redundancy Elimination*, CC'04
    - Known as GVN-PRE
    - Extends dominator based Global Value Numbering
    - ➢ Unifies two of the most powerful redundancy elimination algorithms
      - Classic PRE and SSAPRE can only find *lexically* equivalent partially redundant expressions
      - GVN-PRE uses value numbering to recognize *semantically* equivalent expressions

GVN-PRE

- Value based Partial Redundancy Elimination algorithm
- Requirements
  - All critical edges removed
  - SSA form of IR
- Three steps
  1. Build sets
     (1) Top-down traversal of dominator tree
     (2) Calculate flow sets for each block
  2. Insert
  3. Eliminate

GVN-PRE – 1. Build Sets

(1) Top-down traversal of dominator tree

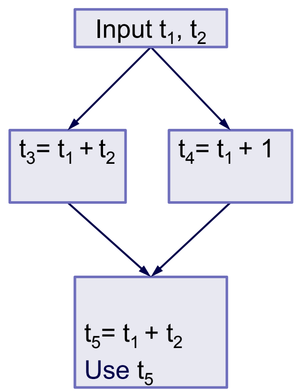
- At each block
  - Global Value Numbering
    - iterate forward over instructions assigning a value to each
  - Build sets
    - EXPGEN: expressions (temporaries and non-simple) that appear in the right hand side of an instruction in *b*
    - PHIGEN: temporaries that are defined by a Φ in *b*
    - TMPGEN: temporaries that are defined by non-Φ instructions in *b*

(2) Calculate flow sets for each block

GVN-PRE – Example

- Global Value Numbering (domtree):

v_0: 1 v_1: t_1 v_2: t_2 v_3: t_3 , v_1+ v_2 , t_5 v_4: t_4 , v_1+ v_0



GVN-PRE – 1. Build Sets

(1) Top-down traversal of dominator tree

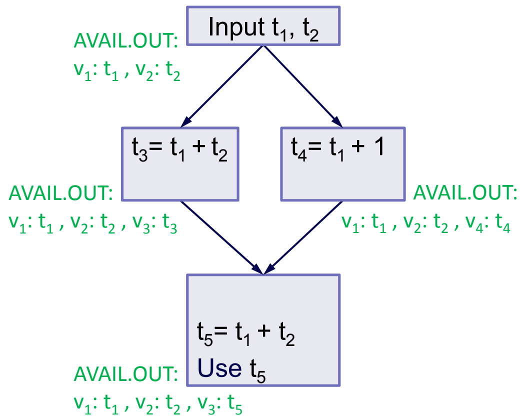
(2) Calculate flow sets for each block (fixed-point iteration)

- Available sets
  - Top-down traversals of the dominator tree
  - AVAIL.IN[*b*] = AVAIL.OUT[idom(*b*)]
  - AVAIL.OUT[*b*] = canon(AVAIL.IN[*b*] ∪ PHIGEN(*b*) ∪ TMPGEN(*b*))
    - canon partitions a set of temporaries into subsets which all have the same value and chooses a leader (representative) from each
- Anticipated sets
  - Top-down traversals of the post-dominator tree

GVN-PRE – Example

- Available Sets (dom):

v_0: 1 v_1: t_1 v_2: t_2 v_3: t_3 , v_1+ v_2 , t_5 v_4: t_4 , v_1+ v_0



GVN-PRE – 1. Build Sets

(1) Top-down traversal of dominator tree

(2) Calculate flow sets for each block (fixed-point iteration)

- Available sets
- Anticipated sets
  - ANTIC.OUT[*b*] =

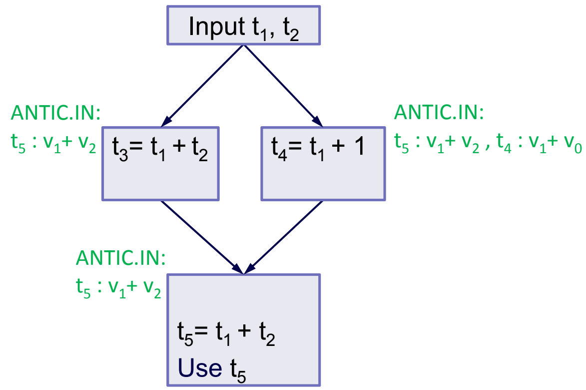
    if |succ(*b*)| = 1: phitrans(ANTIC.IN [Succ(*b*)], *b*, Succ(*b*))
    - phitrans(*S*, *b*, *succ*): ∀ *t* ∈ *S*: if temporary *t* is defined by a Φ at *succ*, it returns the Φ-operand corresponding to *b*, otherwise *t*

    if |succ(*b*)| > 1: $\cap_{S \in Succ(b)}$ ANTIC.IN[S]
    - no Φ's because critical edges removed
  - ANTIC.IN[*b*] = clean(canon$_e$(ANTIC.OUT[*b*] ∪ EXPGEN[*b*] \ TMPGEN(*b*)))
    - canon$_e$ generalizes canon for expressions
    - clean kills expressions that depend on killed values

GVN-PRE – Example

- Anticipated Sets (pdom):

v_0: 1 v_1: t_1 v_2: t_2 v_3: t_3 , v_1+ v_2 , t_5 v_4: t_4 , v_1+ v_0



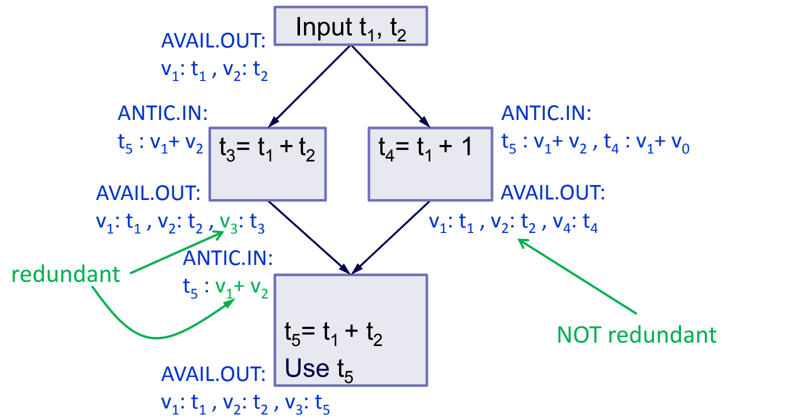
GVN-PRE – 2. Insert

- Top-down traversal of the dominator tree
- Insertions happen only at merge points
  - Iterates over blocks that have more than one predecessor
  - Inspects all expressions anticipated there
- For non-simple expressions
  - Consider the equivalent expressions in the predecessors
  - Look up the value for this equivalent expression and find the leader
    - If there is a leader, then it is available
  - If the expression is available in at least one predecessor
    - (1) Insert it in predecessors where it is not available
      - Generates fresh temporaries
    - (2) create a Φ to merge the predecessors' leaders
  - (1) and (2) implies a new leader for insertions
    - Must be propagated to dominated blocks

GVN-PRE – Example

- Insert – identify partial redundancy

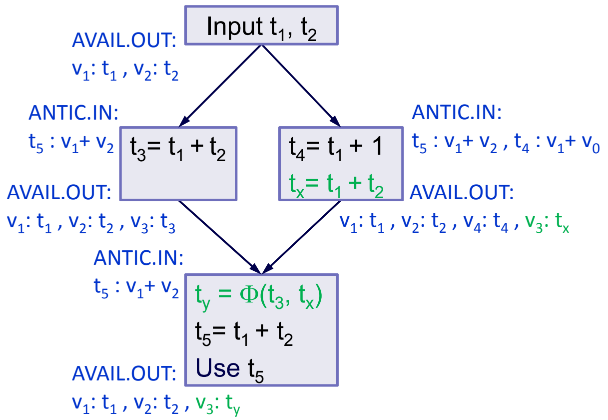
v_0: 1 v_1: t_1 v_2: t_2 v_3: t_3 , v_1+ v_2 , t_5 v_4: t_4 , v_1+ v_0



GVN-PRE – Example

- Insert – convert to full redundancy

v_0: 1 v_1: t_1 v_2: t_2 v_3: t_3 , v_1+ v_2 , t_5 , t_x , t_y v_4: t_4 , v_1+ v_0



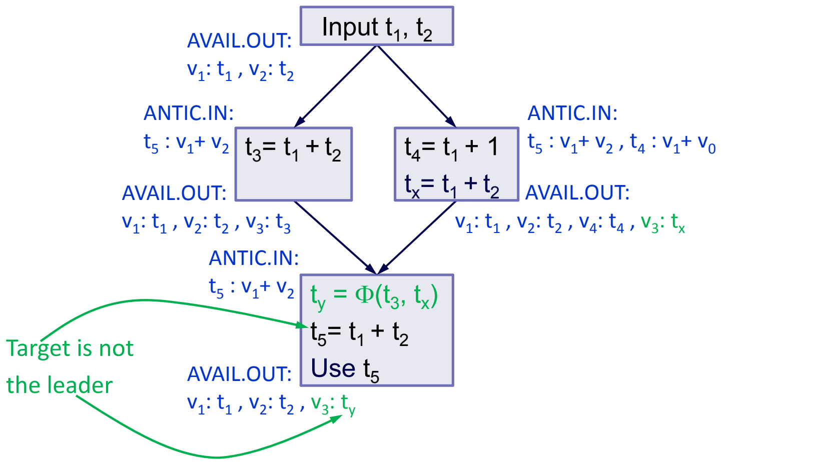
GVN-PRE – 3. Eliminate

- For any instruction
  - find the leader of the target's value
  - If it differs from that target
    - there is a constant or an earlier-defined temporary with the same value
      - current instruction can be replaced by a move from the leader to the target, or
      - current instruction can be removed while replacing all later uses of that target by a use of the leader

GVN-PRE – Example

- Eliminate – identify instructions for removing

v_0: 1 v_1: t_1 v_2: t_2 v_3: t_3 , v_1+ v_2 , t_5 , t_x , t_y v_4: t_4 , v_1+ v_0



GVN-PRE – Example

- Eliminate – remove them and update uses

v_0: 1 v_1: t_1 v_2: t_2 v_3: t_3 , v_1+ v_2 , t_5 , t_x , t_y v_4: t_4 , v_1+ v_0