Compiler Construction

Chapter 8: Introduction to optimization

Dittaya Wanvarie

Department of Mathematics and Computer Science
Chulalongkorn University

Second semester, 2024

Introduction

Common goals of optimization

- Less runtime
- Less memory/register requirement
- Shorter codes
- Less energy consumption

Concern

- **Safety**: the transformed program MUST produce the same result
- **Profit**: the transformed program must have an improvement from the original program

Source of optimization

1. Contextual knowledge
2. Target machine architecture

Array-address calculation

`m(i,j)` with column-major ordering

- Row `i` $\in$ `start_row..end_row`
- column `j` $\in$ `start_col..end_col`

```
for i in start_row .. end_row:
    for j in start_col .. end col:
        .. m(i,j) ..
```

- The index is starting from 1
- $m_1 + (j - low_2(m)) \times (high_1(m) - low_1(m)) + 1) \times w + (i - low_1(m)) \times w$
- $m_1 + (j - 1) \times hw + (i - 1) \times w$
- $low_i(m)$ and $high_i(m)$ are the lower and upper bounds of m's ith dimension
- w is the size of an element of m

Strength reduction: from multiplication to addition

Improving a loop nest

$y + x \times m$ for vectors x and y, and matrix m

```
do 60 j = 1, n2
      do 50 i = 1, n1
        y(i) = y(i) + x(j) * m(i,j)
50    continue
60 continue
```

```
      subroutine dmxpy (n1, y, n2, ldm, x, m)
      double precision y(*), x(*), m(ldm,*)
        ...
      jmin = j+16
      do 60 j = jmin, n2, 16
         do 50 i = 1, n1
           y(i) = ((((((((((((((( (y(i))
     $          + x(j-15)*m(i,j-15)) + x(j-14)*m(i,j-14))
     $          + x(j-13)*m(i,j-13)) + x(j-12)*m(i,j-12))
     $          + x(j-11)*m(i,j-11)) + x(j-10)*m(i,j-10))
     $          + x(j- 9)*m(i,j- 9)) + x(j- 8)*m(i,j- 8))
     $          + x(j- 7)*m(i,j- 7)) + x(j- 6)*m(i,j- 6))
     $          + x(j- 5)*m(i,j- 5)) + x(j- 4)*m(i,j- 4))
     $          + x(j- 3)*m(i,j- 3)) + x(j- 2)*m(i,j- 2))
     $          + x(j- 1)*m(i,j- 1)) + x(j) *m(i,j)
 50      continue
 60   continue
        ...
      end
```

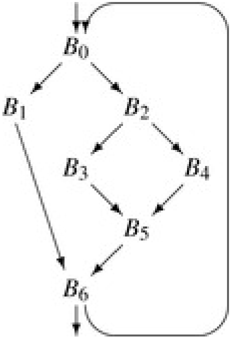
- Loop unrolling: replicates the loop body for distinct iterations and adjusts the index calculations to match
- Utilize the target machine resource: keep some addresses in registers to eliminate load instructions

Opportunities for optimization

1. Reducing the overhead of abstraction
   - E.g. array-address calculation
2. Taking advantage of special cases
3. Matching the code to system resources
   - E.g. eliminating loads instructions, fetching multiple elements into registers

Scope of optimization

- Local methods: a basic block
- Regional methods: a control-flow graph
  - An extended basic block: a set of blocks with one incoming edge
  - A dominator: all paths from the root block to the dominated block.
- Global methods: intraprocedural methods
  - An entire procedure
- Interprocedural methods: a whole-program method



Local optimization

- Local scope - a basic block
- Remove redundancy in the blocks
- E.g. value numbering and tree-height balancing

Local value numbering

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

- The expression `b+c` in the first and the third lines are NOT redundant since we redefine `b` in the second line
- The expression `a-d` in the second and the forth lines are redundant since `a` and `d` are not redefined between these two operations

We have to perform lifetime analysis of each definition (assignment)

Local value numbering algorithm

```
for i ← 0 to n-1, where the block has n operations   "Tᵢ ← Lᵢ Opᵢ Rᵢ"
    1. get the value numbers for Lᵢ and Rᵢ
    2. construct a hash key from Opᵢ and the value numbers for Lᵢ and Rᵢ
    3. if the hash key is already present in the table then
            replace operation i with a copy of the value into Tᵢ and
            associate the value number with Tᵢ
       else
            insert a new value number into the table at the hash key location
            record that new value number for Tᵢ
```

- Assign a distinct number to each value that the block computes
- Use a hash table to find defined values (`L_i` and `R_i`)
- Insert a new value `L_i Op R_i` to the hash table

LVN example

From the original code block, the value numbering becomes

```
a_2 = b_0 + c_1
b_4 = a_2 - d_3
c_5 = b_4 + c_1
d_4 = a_2 - d_3
```

`b_4` and `d_4` refer to the same value (number 4)

Then, we can re-write the block as follows

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

Extending the algorithm

We may use LVN to perform several other local optimizations

- Commutative operations: $a \times b$ and $b \times a$ should receive the same value number.
- Constant folding: find the evaluated expression in the hash table
- Algebraic identities: e.g. $x + 0$ and x should receive the same value number. However, we need a set of rules to test for these identities

Example rules

$a + 0 = a$	$a - 0 = a$	$a - a = 0$	$2 \times a = a + a$
$a \times 1 = a$	$a \times 0 = 0$	$a \div 1 = a$	$a \div a = 1, a \neq 0$
$a^1 = a$	$a^2 = a \times a$	$a \gg 0 = a$	$a \ll 0 = a$
a AND a = a	a OR a = a	MAX (a,a) = a	MIN (a,a) = a

Extending the algorithm

for $i \leftarrow 0$ to $n-1$, where the block has n operations "$T_i \leftarrow L_i\ Op_i\ R_i$"

1. get the value numbers for L_i and R_i
2. if L_i and R_i are both constant then evaluate $L_i\ Op_i\ R_i$,
   assign the result to T_i, and mark T_i as constant
3. if $L_i\ Op_i\ R_i$ matches an identity in Figure 8.3, then replace it with
   a copy operation or an assignment
4. construct a hash key from Op_i and the value numbers for L_i and R_i,
   using the value numbers in ascending order, if Op_i commutes
5. if the hash key is already present in the table then
   replace operation i with a copy into T_i and
   associate the value number with T_i

   else

   insert a new value number into the table at the hash key location
   record that new value number for T_i

The role of naming

```
a_3 = x_1 + y_2
b_3 = x_1 + y_2
a_4 = 17_4
c_3 = x_1 + y_2
```

We can see that `x_1 + y_2` are redundant.

- We can rewrite `b = x + y` with `b = a` since they receive the same value number
- However, we cannot rewrite `c = x + y` with `c = a` because `a` does not have value number 4 at the forth line

We may remove the name from the list if it is redefined

The role of naming

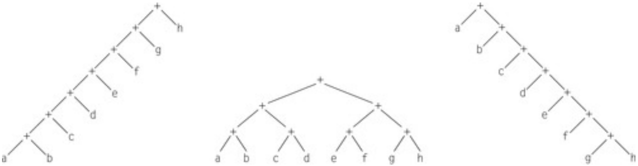
Another solution is the static single-assignment form (SSA)

```
a_0_3 = x_0_1 + y_0_2
b_0_3 = x_0_1 + y_0_2
a_1_4 = 17_4
c_0_3 = x_0_1 + y_0_2
```

We will also assign (definition) number to each name apart from the value number.

Tree-height balancing

Parallel processing



(a) Left-Associative Tree (b) Balanced Tree (c) Right-Associative Tree

```
t1 = a + b
t2 = t1 + c
t3 = t2 + d
t4 = t3 + e
t5 = t4 + f
t6 = t5 + g
t7 = t6 + h
```

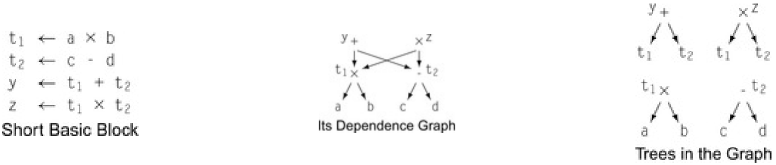
Ideas

- Write dependence graph
- Larger candidate trees provide more opportunities for rearrangement

Tree-height balancing algorithm

1. Identify candidate expression tress in the block
2. For each candidate tree, assign operands with rank and insert the tree into a priority queue

Algorithm example



Short Basic Block

Its Dependence Graph

Trees in the Graph

- **Uses**(T) is the set of blocks that need (use) the definition of T
- **UEVar**(b) is the set of variables whose values are necessary to block b

Finding the root

1. If the name is used more than once, then the (operation) node must be marked as a root to ensure that the value is available for all of its uses
2. If the name is used just once in another operation but the operators are not the same, then the name must be the root

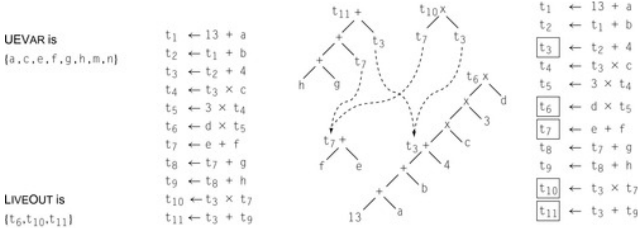
Finding the root

```
// Rebalance a block b of n operations, each of form "Ti ← Li Opi Ri"
// Phase 1: build a queue, Roots, of the candidate trees
Roots ← new queue of names
for i ← 0 to n-1
    Rank(Ti) ← -1;
    if Opi is commutative and associative and
       (|USES(Ti)| > 1 or (|USES(Ti)| = 1 and OpUSES(Ti) ≠ Opi)) then
          mark Ti as a root
          Enqueue(Roots, Ti, precedence of Opi)

// Phase 2: remove a tree from Roots and rebalance it
while (Roots is not empty)
    var ← Dequeue(Roots)
    Balance(var)

Balance(root)    // Create balanced tree from its root, Ti in "Ti ← Li Opi Ri"
    if Rank(root) ≥ 0
       then return    // have already processed this tree

    q ← new queue of names                  // First, flatten the tree
    Rank(root) ← Flatten(Li,q) + Flatten(Ri,q)

    Rebuild(q,Opi)                          //Then, rebuild a balanced tree

Flatten(var,q)    // Flatten computes a rank for var & builds the queue
    if var is a constant                    // Cannot recur further
       then
          Rank(var) ← 0
          Enqueue(q,var,Rank(var))
       else if var∈UEVAR(b)                 // Cannot recur past top of block
              then
                  Rank(var) ← 1
                  Enqueue(q,var,Rank(var))
              else if var is a root
                     then                   // New queue for new root
                        Balance(var)        // Recur to find its rank
                        Enqueue(q,var,Rank(var))
                     else                   // var is Tj in jth op in block
                        Flatten(Lj,q)       // Recur on left operand
                        Flatten(Rj,q)       // Recur on right operand
    return Rank(var)
```

Rebuilding the tree

```
Rebuild(q,op)                                  // Build a balanced expression
    while (q is not empty)
        NL ← Dequeue(q)                        // Get a left operand
        NR ← Dequeue(q)                        // Get a right operand
        if NL and NR are both constants then   // Fold expression if constant
            NT ← Fold(op, NL, NR)
            if q is empty
                then
                    Emit("root ← NT")
                    Rank(root) = 0;
                else
                    Enqueue(q, NT, 0)
                    Rank(NT) = 0;
        else                                   // op is not a constant expression
            if q is empty                      // Get a name for result
                then NT ← root
                else NT ← new name
            Emit("NT ← NL op NR")
            Rank(NT) ← Rank(NL) + Rank(NR)     // Compute its rank
            if q is not empty                  // More ops in q ⇒ add NT to q
                then Enqueue(q, NT, r)
```

Example of tree-height balancing

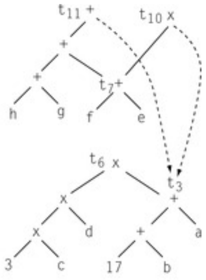


(a) Original Code (b) Trees in the Code (c) Finding Roots

Example of tree-height balancing

$$
\begin{aligned}
n_0 &\leftarrow 17 + b \\
t_3 &\leftarrow n_0 + a \\
t_7 &\leftarrow f + e \\
n_1 &\leftarrow h + g \\
n_2 &\leftarrow n_1 + t_7 \\
t_{11} &\leftarrow n_2 + t_3 \\
t_{10} &\leftarrow t_7 \times t_3 \\
n_3 &\leftarrow 3 \times c \\
n_4 &\leftarrow n_3 \times d \\
t_6 &\leftarrow n_4 \times t_3
\end{aligned}
$$

(a) Transformed Code



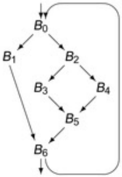
(b) Trees in the Code

Chula Chulalongkorn University

Superlocal value numbering

```
B0: m0 ← a0 + b0
    n0 ← a0 + b0
    (a0>b0) → B1, B2

B1: p0 ← c0 + d0
    r0 ← c0 + d0
    → B6

B2: q0 ← a0 + b0
    r1 ← c0 + d0
    (a0>b0) → B3, B4

B3: e0 ← b0 + 18
    s0 ← a0 + b0
    u0 ← e0 + f0
    → B5

B4: e1 ← a0 + 17
    t0 ← c0 + d0
    u1 ← e1 + f0
    → B5

B5: e2 ← φ(e0,e1)
    u2 ← φ(u0,u1)
    v0 ← a0 + b0
    w0 ← c0 + d0
    x0 ← e2 + f0
    → B6

B6: r2 ← φ(r0,r1)
    y0 ← a0 + b0
    z0 ← c0 + d0
```

(a) Original Code



(b) The CFG

```
B0: m0 ← a0 + b0
    n0 ← a0 + b0
    q0 ← a0 + b0
    r1 ← c0 + d0
    e0 ← b0 + 18
    s0 ← a0 + b0
    u0 ← e0 + f0
```

(c) Path (B_0,B_2,B_3)

1. *Create scope for B_0*
2. *Apply* LVN *to B_0*
3. *Create scope for B_1*
4. *Apply* LVN *to B_1*
5. *Add B_6 to* `WorkList`
6. *Delete B_1's scope*
7. *Create scope for B_2*
8. *Apply* LVN *to B_2*
9. *Create scope for B_3*
10. *Apply* LVN *to B_3*
11. *Add B_5 to* `WorkList`
12. *Delete B_3's scope*
13. *Create scope for B_4*
14. *Apply* LVN *to B_4*
15. *Delete B_4's scope*
16. *Delete B_2's scope*
17. *Delete B_0's scope*
18. *Create scope for B_5*
19. *Apply* LVN *to B_5*
20. *Delete B_5's scope*
21. *Create scope for B_6*
22. *Apply* LVN *to B_6*
23. *Delete B_6's scope*

(d) Scope Manipulations

Global optimization

Data-flow analysis: Live analysis

- **LiveOut**(n) is the set of variables (names) that are live on exit from block n
- **UEVar**(m) is the set of variables whose values are necessary to block m
- **VarKill**(m) is the set of variables (names) that are killed (redefined) in block m
- $succ(n)$ is the set of successor blocks of n

$$\textbf{LiveOut}(n) = \cup_{m \in succ(n)} (\textbf{UEVar}(m) \cup (\textbf{LiveOut}(m) \cap \overline{\textbf{VarKill}(m)}))$$

Live analysis

$$\textbf{LiveOut}(n) = \cup_{m \in succ(n)} (\textbf{UEVar}(m) \cup (\textbf{LiveOut}(m) \cap \overline{\textbf{VarKill}(m)}))$$

A backward data-flow problem: a variable v is live on entry to m under two conditions

1. It is used in m before it is redefined in $m \rightarrow v \in \textbf{UEVar}(m)$
2. It is live on exit from m and pass $\overline{\text{through } m}$ without any new definition $\rightarrow v \in \textbf{LiveOut}(m) \cap \textbf{VarKill}(m)$

Algorithm

$$\text{LiveOut}(n) = \cup_{m \in succ(n)} (\text{UEVar}(m) \cup (\text{LiveOut}(m) \cap \overline{\text{VarKill}(m)}))$$

```
// assume block b has k operations
// of form "x ← y op z"
for each block b
    Init(b)

Init(b)
    UEVAR(b) ← ∅
    VARKILL(b) ← ∅
    for i ← 1 to k
        if y ∉ VARKILL(b)
            then add y to UEVAR(b)
        if z ∉ VARKILL(b)
            then add z to UEVAR(b)
        add x to VARKILL(b)
```

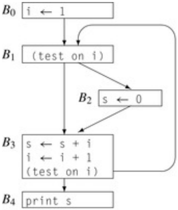
(a) Gathering Initial Information

```
// assume CFG has N blocks
// numbered 0 to N-1
for i ← 0 to N-1
    LIVEOUT(i) ← ∅
changed ← true
while (changed)
    changed ← false
    for i ← 0 to N-1
        recompute LIVEOUT(i)
        if LIVEOUT(i) changed then
            changed ← true
```

(b) Solving the Equations

Live analysis example

$$\mathbf{LiveOut}(n) = \cup_{m \in succ(n)} (\mathbf{UEVar}(m) \cup (\mathbf{LiveOut}(m) \cap \overline{\mathbf{VarKill}(m)}))$$



(a) Example Control-Flow Graph

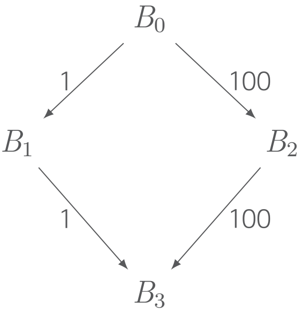
	UEVAR	VARKILL
B_0	∅	{i}
B_1	{i}	∅
B_2	∅	{s}
B_3	{s,i}	{s,i}
B_4	{s}	∅

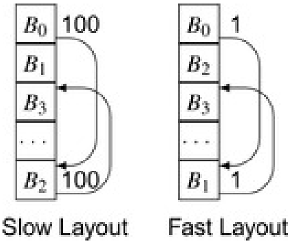
(b) Initial Information

Iteration	LIVEOUT(n) B_0	B_1	B_2	B_3	B_4
Initial	∅	∅	∅	∅	∅
1	{i}	{s,i}	{s,i}	{s,i}	∅
2	{s,i}	{s,i}	{s,i}	{s,i}	∅
3	{s,i}	{s,i}	{s,i}	{s,i}	∅

(c) Progress of the Solution

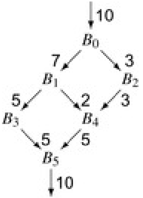
Fall-through branch





Code placement

Greedy algorithm

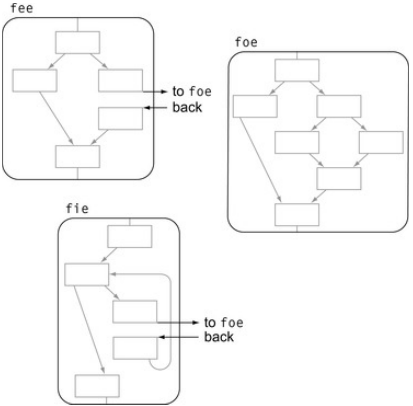


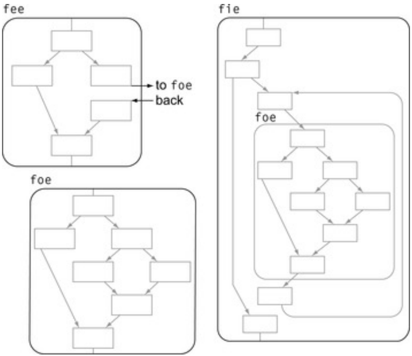
Example CFG

Solution

Edge	Set of chains	Priority
-	$(B_0)_E, (B_1)_E, (B_2)_E, (B_3)_E, (B_4)_E, (B_5)_E$	0
(B_0, B_1)	$(B_0, B_1)_0, (B_2)_E, (B_3)_E, (B_4)_E, (B_5)_E$	1
(B_3, B_5)	$(B_0, B_1)_0, (B_2)_E, (B_3, B_5)_1, (B_4)_E$	2
(B_4, B_5)	$(B_0, B_1)_0, (B_2)_E, (B_3, B_5)_1, (B_4)_E$	2
(B_1, B_3)	$(B_0, B_1, B_3)_0, (B_2)_E, (B_4)_E$	3
(B_0, B_2)	$(B_0, B_1, B_3)_0, (B_2)_E, (B_4)_E$	3
(B_2, B_4)	$(B_0, B_1, B_3)_0, (B_2, B_4)_E$	4
(B_1, B_4)	$(B_0, B_1, B_3)_0, (B_2, B_4)_E$	4

Inline substitution





Interprocedural optimization

Procedure placement

- If procedure p calls q, we would like p and q to occupy adjacent locations in memory.