CSE 504: Compiler Design

Optimizations

Pradipta De
pradipta.de@sunykorea.ac.kr
Current Topic

• Purpose of Optimization Phase
• Optimization Techniques
Optimization Basics

- Classic three-phase compiler: Optimizer is the middle phase
- Optimizer converts from IR to IR with a goal to improve some objective
  - Execution time, space, energy consumption
- Must ensure correctness of code (safety) while improving performance (profitability)
Scope of Optimization

• Local Methods
  – Operate over a single basic block

• Regional Methods
  – Operates over scope larger than a single basic block, but smaller than a procedure

• Global Methods
  – Focus on an entire procedure (also called intraprocedural methods)

• Interprocedural Methods
  – Whole program methods optimizes across procedures and entire program
Local Value Numbering

Compiler rewrites to compute \((a-d)\) only once \(\Rightarrow\) eliminate common sub-expression

Pros: The code runs faster \(\Rightarrow\) profitable
Cons: May increase the register demand (depending on precise details)

How to spot that sub-expression \((b \times c)\) and \((d \times c)\) produce the same result?
The algorithm traverses a basic block and assigns a distinct number to each value that the block computes ➔ Choose numbers such that two expressions $e_i$ and $e_j$ have same value number iff $e_i$ and $e_j$ have equal values for all possible operands for the expressions.

for $i \leftarrow 0$ to $n-1$, where the block has $n$ operations \( "T_i \leftarrow L_i \text{ Op}_i R_i" \)

1. get the value numbers for $L_i$ and $R_i$
2. construct a hash key from $\text{Op}_i$ and the value numbers for $L_i$ and $R_i$
3. if the hash key is already present in the table then
   replace operation $i$ with a copy of the value 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 $T_i$
Input is a basic block with \( n \) binary operations of the form \( T_i \leftarrow L_i \ O \ O \ R_i \).

LVN uses a hash table to map names, constants and expressions into distinct value numbers.

Look for value number of \( L_i \) and \( R_i \) in hash table \( \Rightarrow \) \( \text{VN}(L_i) \) and \( \text{VN}(R_i) \)

Given value numbers for \( L_i \) and \( R_i \), called \( \text{VN}(L_i) \) and \( \text{VN}(R_i) \), LVN constructs a hash key from \( \langle \text{VN}(L_i), O, \text{VN}(R_i) \rangle \) and looks up that key in the table.

\[
\begin{align*}
a^2 & \leftarrow b^0 + c^1 \\
b^4 & \leftarrow a^2 - d^3 \\
c^5 & \leftarrow b^4 + c^1 \\
d^4 & \leftarrow a^2 - d^3
\end{align*}
\]
Extending LVN

• Commutative Operations
  – Should receive the same value number

• Constant Folding
  – If all operands have constant value, then perform the operation and fold the answer into code
  – Ex: if (2 < 0) jmp L ⇒ can be eliminated

• Algebraic Identities
  – Apply to simplify code ⇒ need special case code for each identity

\[
\begin{align*}
a + 0 &= a \\
a - 0 &= a \\
a - a &= 0 \\
2 \times a &= a + a \\
a \times 1 &= a \\
a \times 0 &= 0 \\
a \div 1 &= a \\
a \div a &= 1, a \neq 0 \\
a^1 &= a \\
a^2 &= a \times a \\
a \gg 0 &= a \\
a \ll 0 &= a \\
a \text{AND} a &= a \\
a \text{OR} a &= a \\
\text{MAX} (a,a) &= a \\
\text{MIN} (a,a) &= a
\end{align*}
\]
Naming matters in LVN

\[ a^3 \leftarrow x^1 + y^2 \]
\[ b^3 \leftarrow x^1 + y^2 \]
\[ a^4 \leftarrow 17^4 \]
\[ c^3 \leftarrow x^1 + y^2 \]

Step-1: LVN assigns 1 and 2 to \( x \) and \( y \), and 3 to \((x+y)\)

Step-2: Identifies \( x+y \) is redundant, assigns 3, rewrite \( b \leftarrow a \)

Step-3: nonredundant entry

Step-4: Identifies \((x+y)\), but cannot rewrite it as a

Solution: Add a subscript to each name

\[ a^3_0 \leftarrow x^1_0 + y^2_0 \]
\[ b^3_0 \leftarrow x^1_0 + y^2_0 \]
\[ a^4 \leftarrow 17^4 \]
\[ c^3_0 \leftarrow x^1_0 + y^2_0 \]
Regional Optimization

• Limited to a single block with straightline code
  – all operations execute in a consistent manner such that prior context can be used to expose redundancies and constant valued expressions

• Extend the LVN to larger regions to find more context
Superlocal Value Numbering

• Compiler extends the scope to Extended Basic Blocks

• To process \((B_0, B_1)\), the compiler can apply LVN to \(B_0\) and use the resulting hash table as a starting point when it applies LVN to \(B_1\)

EBB:
\{B_0, B_1, B_2, B_3, B_4\}, \{B_5\}, \{B_6\}
Paths:
(B_0, B_1), (B_0, B_2, B_3), (B_0, B_2, B_4)
Superlocal Value Numbering

• Compiler must reuse the results of blocks that occur as prefixes on multiple paths through the EBB
  – Undo effects of processing a block \( \rightarrow \) after processing \((B0,B2,B3)\), recreate state after \((B0,B2)\) to use in B4

• One way to implement
  – Implement the value table using similar ideas as lexically scoped hash tables
Loop Unrolling

- Replicate the loops body and adjust the logic that controls the number of iterations
  - Can unroll the inner or the outer loop
  - Requires additional prologue code
  - Reduces the number of operations required to complete loop
  - Increases program size

- Inner loop unrolling produces code that reduces execution of test-and-branch sequences

- Outer loop unrolling changes the ratio of arithmetic operations to memory operations
  - Reduces number of test-and-branch sequences
  - Reuse of variables
  - Sequential access to variables
do 60 j = 1, n2
  do 50 i = 1, n1
    y(i) = y(i) + x(j) * m(i,j)
  50 continue
60 continue

nextra = mod(n2,4)
if (nextra .ge. 1) then
  do 59 j = 1, nextra
    do 49 i = 1, nextra
      y(i) = y(i) + x(j) * m(i,j)
    49 continue
  59 continue

  do 50 i = nextra + 1, n1, 4
    y(i) = y(i) + x(j) * m(i,j)
    y(i+1) = y(i+1) + x(j) * m(i+1,j)
    y(i+2) = y(i+2) + x(j) * m(i+2,j)
    y(i+3) = y(i+3) + x(j) * m(i+3,j)
  50 continue
60 continue

(a) Unroll Inner Loop by Four

(b) Unroll Outer Loop by Four, Fuse Inner Loops
Pros and Cons of Loop Unrolling

• Pros:
  – Better instruction schedules
    • Increases the number of independent operations in the loop body
  – Consecutive memory accesses can move into the same loop iteration
  – Can expose cross-iteration redundancies

• Cons:
  – Demand for registers may increase leading to spill and higher memory traffic
Summary

- Presented Local and Regional Optimizations

- Global and Interprocedural optimizations are less frequently used than local and regional optimizations