

CSC2/455 Software Analysis and Improvement

Value Numbering

Sreepathi Pai

URCS

January 28, 2019

Outline

Review

An Introduction to Code Optimization

Local Value Numbering (LVN)

Postscript

Outline

Review

An Introduction to Code Optimization

Local Value Numbering (LVN)

Postscript

What we know how to do so far

- ▶ ASTs to 3-address code
- ▶ 3-address code to Basic Blocks
- ▶ Basic blocks to Control Flow Graphs
- ▶ (Tested in first assignment, out today/tomorrow)

Some Useful Background Material

- ▶ Not needed if you have CSC 2/454
- ▶ Otherwise, read Cooper and Turczon:
 - ▶ Chapter 6 (“The Procedure Abstraction”)
 - ▶ Chapter 7 (“Code Shape”)

Outline

Review

An Introduction to Code Optimization

Local Value Numbering (LVN)

Postscript

Basic Goals

- ▶ Reduce execution time (“performance”)
 - ▶ Most of our time spent on this topic
- ▶ Reduce code size
 - ▶ Was hugely important, less so in the past decade or so
 - ▶ Increasingly important again (small devices, slow memory, etc.)
- ▶ Code size reduction *often* improves performance
 - ▶ Instructions that do not execute consume no time
 - ▶ Better cache behaviour
 - ▶ But not always...

Can't programmers write fast code?

Major impediments:

- ▶ High-level languages
 - ▶ Productivity is more important than performance(?)
 - ▶ But lose control over machine
- ▶ Complex machines
 - ▶ Out-of-order superscalar machines
- ▶ “Simple” machines
 - ▶ Very-long instruction word (VLIW) machines

Scope of Optimization

- ▶ What should we look at to identify optimization opportunities?
- ▶ What additional information can help us?

Local optimizations

- ▶ Look at basic blocks only
- ▶ Remember:
 - ▶ If one instruction in basic block executes, *all* instructions execute
 - ▶ Can assume instructions execute one after the other
- ▶ Major limitation
 - ▶ Size/Length of basic block

Increasing the size of basic blocks

- ▶ “Region” analysis
- ▶ Many forms:
 - ▶ Extended Basic Blocks
 - ▶ Superblocks
 - ▶ Hyperblocks
- ▶ Basic idea, use a *trace* to build a larger basic block
 - ▶ a trace is an execution path

Global optimizations

- ▶ Global means *procedure* level
 - ▶ A procedure has a single entry and exit
 - ▶ E.g. C functions
 - ▶ *not* global as in global variables
- ▶ A global analysis usually operates on a control-flow graph
- ▶ Sometimes called “intraprocedural” analysis

Interprocedural (Whole program) Optimizations

- ▶ Operates on the entire source code of a program
- ▶ Sometimes called “context-sensitive analysis”
 - ▶ Allows compiler to distinguish invocations of the same function
 - ▶ More opportunities for optimization!
- ▶ Must keep track of function calls
 - ▶ Usually, in a structure called the “call graph”

Link-time Optimization

- ▶ Traditional compiler flow
 - ▶ Program contains multiple translation units (usu. a .c file)
 - ▶ Translation unit to object (.o file)
 - ▶ Combine object files and libraries (“link”) to generate executable
 - ▶ Usually hidden by a compiler driver like gcc
- ▶ Link-time optimization is a form of whole program optimization
 - ▶ Optimization across object boundaries
 - ▶ Often necessary for languages like C++

Runtime (Dynamic) optimization

- ▶ Optimization while a program is running
- ▶ Sometimes a necessity:
 - ▶ E.g., JavaScript
- ▶ Sometimes called Just-in-time (JIT) compilation
 - ▶ because mostly associated with JIT VMs
- ▶ Big advantage:
 - ▶ Have nearly all the information required to optimize program
- ▶ Big disadvantage?

Additional Information: Profiles

- ▶ Most optimizers only look at source code
- ▶ But additional information can help:
 - ▶ What is the most common direction of this branch? (see gcc's `__builtin_expect`)
 - ▶ What is the most likely target of this indirect function call?
- ▶ General technique: *profile-based optimization*
 - ▶ Instrument program to generate statistics (i.e. profile)
 - ▶ Generate statistics on program run
 - ▶ Compiler recompiles code based on gathered profile

Outline

Review

An Introduction to Code Optimization

Local Value Numbering (LVN)

Postscript

Getting rid of redundant calculations

```
ed = (a0 - a1)*(a0 - a1) + (b0 - b1)*(b0 - b1)
```

Optimized code sequence:

```
t1 = a0 - a1  
t2 = t1 * t1  
t3 = b0 - b1  
t4 = t3 * t3  
ed = t2 + t4
```

Where does this code arise?

- ▶ High-level languages
- ▶ AST to 3-address code generation rules
 - ▶ e.g. array indexing
 - ▶ $x = A[i][j] + A[i][j+1]$?

Array indexing example

```
t1 = i * dimsize(A, 1)
t2 = t1 + j
t3 = (A + t2)
```

```
t4 = i * dimsize(A, 1)
t5 = t4 + j
t6 = t5 + 1
t7 = (A + t6)
```

```
x = t3 + t7
```

`dimsize(A, dim)` is a function that returns an integer indicating the size of dimension *dim* of array *A*.

Compiler structure

- ▶ Compilers rewrite and lower your code
- ▶ Can introduce significant redundancy in the process
- ▶ Logically cleaner (and simpler!) to reduce redundancy using dedicated *passes*
 - ▶ Keeps rewrites (incl. translation) simple

Value Numbering

- ▶ Value number is a local optimization
- ▶ Assigns numbers to each value in the basic block
 - ▶ Literals
 - ▶ Variables
 - ▶ $VN(x)$ is the the number assigned to x
- ▶ Design $VN(x)$ such that:
 - ▶ $VN(x) == VN(y)$ if-and-only-if (iff) $x == y$

Example case

$t_1 = a_0 - a_1$

$t_2 = a_0 - a_1$

$t_3 = t_1 * t_2$

$t_4 = b_0 - b_1$

$t_5 = b_0 - b_1$

$t_6 = t_4 * t_5$

$ed = t_3 + t_6$

Example case

```
t1 = a0 - a1      # 2 = 0 - 1
t2 = a0 - a1      # 2 = 0 - 1
t3 = t1 * t2      # 3 = 2 * 2

t4 = b0 - b1      # 6 = 4 - 5
t5 = b0 - b1      # 6 = 4 - 5
t6 = t4 * t5      # 7 = 6 * 6

ed = t3 + t6      # 8 = 3 + 7
```

Algorithms in Figure 8.2, 8.4 of Cooper and Turczon

Why limit ourselves to basic blocks?

What would happen if we ran value numbering across basic blocks?

Outline

Review

An Introduction to Code Optimization

Local Value Numbering (LVN)

Postscript

References

- ▶ Chapter 8 of Cooper and Turczon
 - ▶ Up to (but not including) 8.4.2