Lecture 18: Optimization
1 Apr 2010

Nate Nystrom
University of Texas at Arlington
Optimization

Compiler has primary responsibility for performance

Optimizations
- set of transformations intended to improve performance of the generated code

Issues:
- safety
- profitability
- risk
Safety

Must be careful to preserve meaning of the program

- two programs are equivalent if they produce identical results

Large part of analysis goes toward proving optimizations are safe
Profitability

“Optimization” is a misnomer
“Code improvement” is probably a better term

Will a given optimization improve performance?
By what metric?
  - time, space, ...

Often not clear:
  - little comparative data that is believable
  - papers give conflicting advice
Profitability

Determining whether an optimization is profitable is hard:

- linear-time heuristics for hard problems
- unforeseen consequences and poorly understood interactions
- “obvious wins” have non-obvious downsides
- multiple ways to achieve same end
Inlining

Replace function call with body of the function

```c
int length(Node n) {
    int len = 0;
    while (n != null) {
        n = n.next;
        len++;
    }
    return len;
}
```

Node m = ...;
Node n = m;
int a = 0;
while (n != null) {
    n = n.next;
    a++;
}

Node m = ...;
int a = length(m);
```
Is inlining profitable?

Holler: almost always helps

Hall: occasionally helps, but has lots of problems

MacFarland: introduces cache misses

The reality is somewhere in the middle

- Waterman: program-specific heuristics win
Risk

Transformations can hurt ability to generate good code

- loop unrolling, inlining can increase icache misses
- unrolling introduces redundant computation
- redundancy elimination can increase register pressure

One optimization can enable another, or can undo the benefits of another

Ordering of optimizations can affect performance
Some optimizations

Activation record merging  Loop fusion
Adjacency analysis  Loop unswitching
Anchor pointing  Operator simplification
Carry optimization  Operator strength reduction
Common subexpression elimination  Peephole optimization
Constant folding  Procedure integration
Dead code elimination  Special case code generation
Dead space reclamation  Register allocation
Detection of parallelism  Register assignment
Inserting forced copies  Reassociation
Instruction scheduling  Shadow variable introduction
Linear function test replacement  Stack height reduction
Live range shrinking  Test elision
Loop unrolling  Variable folding
Look fission  Zero iteration test

This list is c. 1982. Even more optimizations now!
Classic taxonomy

Machine independent transformations
- deliberately ignore machine-specific constraints
- applicable across broad range of machines
- decrease ration of overhead to real work
- reduce running time or space
- example: dead code elimination

Machine dependent transformations
- explicitly consider machine-specific constraints
- capitalize on machine-specific properties
- improve mapping from IR to this machine
- might use an exotic instruction (e.g., register window shifting)
- example: instruction scheduling
But distinction is not always clear
- e.g., replacing multiple with shifts and adds
- e.g., eliminating a redundant expression

Redundancy elimination might fit into either category
- simple machine independent version
  vs. versions that consider register pressure
Better taxonomy (Cooper)

Effects-based classification

- five machine-independent ways to speed up code
  - eliminate a redundant computation
  - move code to a place where it executes less often
  - eliminate dead code
  - specialize a computation based on context
  - enable another transformation

- three machine-dependent ways to speed up code
  - manage or hide latency
  - take advantage of special hardware features
  - manage finite resources

This covers most scalar optimizations
Machine independent

- redundancy
  - redundancy elimination (CSE)
  - partial redundancy elimination (PRE)
  - consolidation

- dead code
  - dead code elimination (DCE)
  - partial DCE
  - constant propagation
  - algebraic identities

- code motion
  - loop-invariant code motion
  - PRE
  - consolidation
  - global scheduling
  - constant propagation

- specialization
  - replication
  - strength reduction
  - method caching
  - heap->stack allocation
  - tail recursion elimination

- create opportunities
  - reassociation
  - replication
**Machine dependent**

hide latency
- scheduling
- blocking references
- prefetching
- code layout
- data packing

manage resources
- allocation (registers, tlb slots)
- scheduling
- data packing
- coloring memory locations

special features
- instruction selection
- peephole optimization
Scope of optimization

Local
- handles individual basic blocks
  - maximal length sequence of straight line code
- basic blocks are easy to analyze (no branches)
- can prove strongest results
- code quality suffers at block boundaries

Local methods
- value numbering
- instruction scheduling
- peephole optimization
Scope of optimization

Superlocal
- handles *extended* basic blocks (aka superblocks)
  - sequence of blocks where each has a unique predecessor
- analysis and transformation over larger region
- fewer rough edges
- can make it efficient by reusing results

Superlocal methods
- value numbering
- instruction scheduling
- peephole optimization
Scope of optimization

Regional
- arbitrary subset of blocks (loop nests, dominator subtrees)
- use results of one block to improve others
- limiting scope can increase focus on performance critical regions
- can eliminate some global impediments

Regional methods
- loop transformations (unroll, fuse, interchange, strip mining, blocking)
- register promotion
- prefetch insertion
- software pipelining
- trace scheduling
Scope of optimization

Whole procedure (global, intra procedural)
- handles entire procedure
- make decisions based on global knowledge & global benefit
- no rough edges inside procedure
- classic dataflow analysis

Global methods
- CSE
- constant propagation
- register allocation
- dead code elimination
- hoisting
- copy coalescing
Scope of optimization

Whole program (interprocedural)
- handles more than one procedure, up to entire program
- creates even larger scopes for optimization
- limited interaction between procedures
  - parameters + global variables
- analysis problems are harder
- opportunities are different
- issues for compiler structure

Interprocedural methods
- inlining
- procedure cloning
- constant propagation
- using whole program analysis to support global transformations
Redundancy elimination

Eliminate redundant expressions in a program

Local common subexpression elimination

\[
\begin{align*}
a &= b + c \\
b &= a - d \\
c &= b + c \\
d &= a - d \\
\end{align*}
\]

\[
\begin{align*}
a &= b + c \\
b &= a - d \\
c &= b + c \\
d &= b \\
\end{align*}
\]

Compiler needs to identify redundancies and transform code
Value numbering

Associate a number with each value

Identify redundant expressions using their value numbers rather than names

\[
\begin{align*}
 a &= b + c \\
 b &= a - d \\
 c &= b + c \\
 d &= a - d \\
 a^3 &= b^1 + c^2 \\
 b^5 &= a^3 - d^4 \\
 c^6 &= b^5 + c^2 \\
 d^5 &= a^3 - d^4 \\
 a &= b + c \\
 b &= a - d \\
 c &= b + c \\
 d &= b
\end{align*}
\]
Value numbering

Use a hash table

table = empty

for each expression in block of the form \( x = a \ op \ b \)

if table(a) not defined, init with next value number

\( v_a = \text{table}(a) \)

if table(b) not defined, init with next value number

\( v_b = \text{table}(b) \)

if table(v_a op v_b) defined, replace a op b with \( \text{table}(v_a \ op \ v_b) \)

else \( \text{table}(v_a \ op \ v_b) = x \)

\( \text{table}(x) = \text{next value number} \)
Value numbering

Can extend for:
- operators of different arity
- commutative operators (1+2 and 2+1 have the same value)
- copy instructions
- algebraic identities
Algebraic identities

- \( a + 0 = a \)
- \( a - 0 = a \)
- \( a - a = 0 \)
- \( 2 \times a = a + a \)
- \( a \times 1 = a \)
- \( a \times 0 = 0 \)
- \( a / 1 = a \)
- \( a / a = 1, a \neq 0 \)
- \( a^1 = a \)
- \( a^2 = a \times a \)
- \( a + a = a \)
- \( a = a \)
- \( a \\& a = a \)
- \( a | a = a \)
- \( \max(a, a) = a \)
- \( \min(a, a) = a \)
- \( a \gg 0 = a \)
- \( a \ll 0 = a \)
Extension to superblocks is trivial

- Continue value numbering through jump instructions
- But, if instruction has > 1 predecessor, discard the hash table
Global redundancy elimination

Classic algorithm:
- global common subexpression elimination [Cocke 1970]

Compute *available expressions*
- (see last lecture)

At instruction $I$: $t = y + z$
- if $y+z$ in $\text{AvailIn}[I]$, then $y+z$ is redundant

Introduce temporary $t$ for each redundant expression
- Replace redundant $y+z$ with $t$
Redundancy elimination (GCSE)
Global value numbering

Same algorithm as GCSE, but:

Instead of identifying expressions by name ("y+z"), use their value numbers
Cloning

Using larger scopes of optimization increases opportunities for redundancy elimination

Idea: transform code to expose more opportunities to eliminate redundancies

Cloning:
- if a block has multiple predecessors, append the block to its predecessors
Cloning

Clone

GCSE

t = y + z

y + z

y + z
Partial redundancy elimination

y + z is partially redundant

Redundant on some, but not all paths
Partial redundancy elimination

$y+z$ is partially redundant

PRE can hoist code out of loops!
“Lazy code motion” [Knoop et al. 1992] PRE that guarantees no path will be lengthened.
Summary

Different approaches to redundancy elimination
- fully redundant (CSE) partially redundant (PRE)
- by name (CSE) vs. by value (value numbering)

Risks:
- Redundancy elimination introduces temporaries
  - may increase register pressure
  - may be cheaper to recompute than to burn a register