The Big Picture

The Programming Assignment &
A Taxonomy of Optimizations

Copyright 2015, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 512 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Citation numbers refer to entries in the EaC2e bibliography.
Programming Assignment (aka “The Lab”)  

**You will build an ILOC optimizer**

- Larger set of operations than in the COMP 412 assignments
  - Documents will be posted on CLEAR later today
    - *Simulator document describes set of operations*
    - *Lab document lays down the ground rules*
  - To get started, build a scanner, parser, & design an IR

**The ILOC Simulator**

- One functional unit
- All the simulator operations, except the character operations
  - *cload, cloadAO, cloadAI, cstore, cstoreAO, cstoreAI, i2c, c2i, c2c*
- Operation latencies:
  - Most operations take a single cycle
  - Multiplies take three cycles (*mult, multI*)
  - Load and store operations take five cycles
    - *load, loadAO, loadAI, store, storeAO, storeAl*
Programming Assignment  

(aka “The Lab”)

Rules of the game

• You will choose & implement four (4) optimizations
• I will provide a set of test codes (*six to ten codes*)
  ♦ Several input data sets for each code
• You will report operation counts for the codes & data sets
  ♦ We will have a mini-competition
  ♦ The goal is to minimize cycles measured by the simulator on the various inputs
• Your code should work on **CLEAR**, but you can write & test it anywhere

The software  

(*simulator & compiler that generates examples*)

• Simulator (source & executable) will be posted under **comp512** on **CLEAR**
• The code is, in some aspects, still under development
  
  → *Simulator version number appears at the head of each trace output*
• I will post notices on Piazza when I distribute new versions of simulator

All code must be your own code. You can reuse code that you wrote for other classes.
Optimization

The subject is confusing
• Whole notion of optimality
• Incredible number of transformations
• Odd, inconsistent terminology

Maybe this stuff is inherently hard
• Many intractable problems
• Many NP-complete problems
• Much overlap between problems and between solutions
• If optimization wasn’t confusing, why take COMP 512?

Cooper McKinley, & Torczon cite 237 distinct papers in their unpublished survey!

Value numbering
Redundancy elimination
Common subexpressions
## Optimization

### Allen-Cocke Catalog (1971)

<table>
<thead>
<tr>
<th>Optimization Technique</th>
<th>Optimization Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure integration (open, semi-open, semi-closed, closed)</td>
<td>Linear function test replacement</td>
</tr>
<tr>
<td>Loop unrolling</td>
<td>Carry optimization</td>
</tr>
<tr>
<td>Loop fusion</td>
<td>Instruction scheduling</td>
</tr>
<tr>
<td>Loop unswitching</td>
<td>Register allocation</td>
</tr>
<tr>
<td>Redundancy elimination</td>
<td>Storage mapping</td>
</tr>
<tr>
<td>Code motion</td>
<td>Shadow variable optimization</td>
</tr>
<tr>
<td>Constant propagation</td>
<td>Anchor pointing</td>
</tr>
<tr>
<td>Dead-code elimination</td>
<td>Special case code generation</td>
</tr>
<tr>
<td>Strength reduction</td>
<td>Peephole optimization</td>
</tr>
</tbody>
</table>

*And a discussion of parsing methods*

*And this was before the literature exploded*
Optimization

Sites & Perkins (1979)

- Stack-height reduction
- Constant-valued arithmetic
- Operator simplification
- Local common subexpressions
- Global common subexpressions
- Procedure merging
- Activation-record merging
- Loop induction expressions
- Moving subscript range test

- Zero iteration test
- Code motion out of loops
- Code hoisting
- Live range shrinkage
- Storage (register) allocation
- Forced copies
- Unreachable code
- Branch logic
- Test ellision

And this was before the literature exploded
## Optimization

**Cooper’s thesis proposal**  
*(circa 1982)*

<table>
<thead>
<tr>
<th>Activation record merging</th>
<th>Loop unswitching</th>
<th>Loop unrolling</th>
<th>Loop fusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjacency analysis</td>
<td>Operator simplification</td>
<td>Operator strength reduction</td>
<td>Procedure integration (inlining)</td>
</tr>
<tr>
<td>Anchor pointing</td>
<td>Operator strength reduction</td>
<td>Peephole optimization</td>
<td>Special case code generation</td>
</tr>
<tr>
<td>Carry optimization</td>
<td></td>
<td></td>
<td>Register allocation</td>
</tr>
<tr>
<td>Dead code elimination</td>
<td></td>
<td></td>
<td>Reassociation</td>
</tr>
<tr>
<td>Dead space reclamation</td>
<td></td>
<td></td>
<td>Shadow variable introduction</td>
</tr>
<tr>
<td>Detection of parallelism</td>
<td></td>
<td></td>
<td>Stack height reduction</td>
</tr>
<tr>
<td>Instruction Scheduling</td>
<td></td>
<td></td>
<td>Test elision</td>
</tr>
<tr>
<td>Linear function test replacement</td>
<td></td>
<td></td>
<td>Zero iteration test</td>
</tr>
<tr>
<td>Live range shrinking</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop unrolling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop unrolling</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*And this was before the literature exploded*

And I forgot constant propagation ...
## Optimization

**From EaC1e (2006) and EaC2e (2012)**

<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVN, SVN, DVN, GCSE w/AVAIL</td>
<td>Operator strength reduction</td>
</tr>
<tr>
<td>DAG construction</td>
<td>Linear function test replacement</td>
</tr>
<tr>
<td>Superblock cloning</td>
<td>Procedure abstraction</td>
</tr>
<tr>
<td>Eliminating useless code (DEAD)</td>
<td>Procedure placement</td>
</tr>
<tr>
<td>Eliminating unreachable code (CLEAN)</td>
<td>Block placement &amp; fluff removal</td>
</tr>
<tr>
<td>Lazy Code Motion</td>
<td>Instruction scheduling</td>
</tr>
<tr>
<td>Hoisting</td>
<td>Register allocation</td>
</tr>
<tr>
<td>Sinking</td>
<td>Tree-height balancing</td>
</tr>
<tr>
<td>Constant propagation</td>
<td>Leaf-call optimization</td>
</tr>
<tr>
<td>Loop unrolling</td>
<td>Parameter promotion</td>
</tr>
<tr>
<td>Loop unswitching</td>
<td>Procedure cloning</td>
</tr>
<tr>
<td>Renaming</td>
<td>Copy coalescing</td>
</tr>
<tr>
<td>Peephole optimization</td>
<td>Inline substitution</td>
</tr>
<tr>
<td>Tail-recursion elimination</td>
<td></td>
</tr>
</tbody>
</table>
Optimization

The literature throws fuel on the fire
• Terminology is non-standard & non-intuitive
• Explanations are terse and incomplete
• Little comparative data that is believable
• No sense of perspective
• Papers give conflicting advice

An example – Is inline substitution profitable?
• Holler’s thesis: it almost always helps
• Hall’s thesis: it occasionally helps, but has lots of problems
• MacFarland’s thesis: it causes instruction cache misses

Reality lies somewhere in the middle
⇒ Waterman showed that program-specific heuristics win

Revival
Partially-dead code
Forward propagation

Not all those techniques can possibly be the best!
Optimization

To make matters worse

- Individual optimizations often have multiple effects
- The effects of those optimizations can overlap

Balke - Value Numbering
DVNT - Dominator VN
SCCVN & VDCM - Global VN
AVAIL - Classic CSE
LCM - Lazy Code Motion
AWZ - Partitioning algorithm
SSC - Sparse Simple Constant
SCC - Sparse Cond. Constant

And, there are many others ...
Optimization

**Improvement should be objective**

- Easy to quantify
- Produce concrete improvements
- Taking measurements seems easy

*Code either gets better or it gets worse*

**But, ...**

- Linear-time heuristics for hard problems
- Unforeseen consequences & poorly understood interactions
- “Obvious wins” have non-obvious downsides
- Multiple ways to achieve the same end

*Experimental computer science takes a lot of work*
The Role of Comp 512

**Bringing order out of chaos**

- Provide a framework for thinking about optimization
- Differentiate analysis from transformation†
- Think about how things help, not what they do

**Goal: a rational approach to the subject matter**

- Objective criteria for evaluating ideas & papers
- Bring high school level science back into the game

† The Comp 512 Motto:

*Knowledge alone does not make code run faster.*
*You have to change the code to make it run faster.*
Classic Taxonomy

**Machine independent transformations**

- Applicable across a broad range of machines
- Decrease ratio of overhead to real work
- Reduce running time or space
- Examples: dead code elimination

**Machine dependent transformations**

- Capitalize on specific machine properties
- Improve the mapping from IR to this machine
- Might use an exotic instruction \((shift\ the\ reg.\ window\ for\ a\ loop)\)
- Example: instruction scheduling
Classic Taxonomy

**Distinction between independent & dependent is not always clear**

- Replacing multiply with shifts and adds
- Eliminating a redundant expression

**The truth is somewhat muddled**

- Machine independent means that we deliberately & knowingly ignore target-specific constraints
- Machine dependent means that we explicitly consider target-specific constraints

**Redundancy elimination might fit in either category**

- Versions that consider demand for registers
The Comp 512 Taxonomy

An effects-based classification (for speed)

• 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 the code
  - Manage or hide latency
  - Take advantage of special hardware features
  - Manage finite resources

*For scalar optimization, this covers most of them*
The Comp 512 Taxonomy

Machine Independent

- Redundancy
  - Redundancy elim’n
  - Partial red. elim’n
  - Consolidation

- Code motion
  - Loop-invariant code motion
  - Consolidation
  - Global Scheduling
  - Constant propagation

- Dead code
  - Dead code elimination
  - Partial dead code elim’n
  - Constant propagation
  - Algebraic identities

- Specialization
  - Replication
  - Strength Reduction
  - Method caching
  - Heap→stack allocation
  - Tail recursion elimination

- Create opportunities
  - Reassociation
  - Replication

From Chapter 10 of EaC2e and §6 of the Cooper, McKinley, & Torczon unpublished survey
The Comp 512 Taxonomy

Machine Dependent

- Hide latency
  - Scheduling
  - Blocking references
  - Prefetching
  - Code layout
  - Data packing

- Manage resources
  - Allocate (registers, tlb slots)
  - Schedule
  - Data packing
  - Coloring memory locations

- Special features
  - Instruction selection
  - Peephole optimization

From §6 of Cooper, McKinley, & Torczon
Other Axes: Scope of Optimization

Optimizations are performed at a variety of different scopes

• Local
• Superlocal
• Regional
• Global
• Interprocedural

Scope of optimization forms another axis for comparing analyses and transformations

• Some combinations make no sense, such as local code motion
• Other scopes are possible, such as logical-window peephole optimization
Other Axes: Decision Complexity

The complexity of making decisions varies across optimizations

<table>
<thead>
<tr>
<th>Constant time</th>
<th>Low-order polynomial time</th>
<th>Hard Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVN, SVN, DVNT</td>
<td>Loop unrolling</td>
<td>Reassociation</td>
</tr>
<tr>
<td>Block placement</td>
<td></td>
<td>Inline substitution</td>
</tr>
<tr>
<td>Tree balancing (ILP)</td>
<td></td>
<td>Spill Choice in RA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy Coalescing</td>
</tr>
</tbody>
</table>

Decision complexity forms another axis of comparison among transformations
The “hidden” problem in optimizer design is deciding what to do

• Pick a set of transformations
  ♦ Every project has a limited budget, unless it is an open source compiler that lives forever, uses volunteer labor, and lacks oversight
  ♦ Design under constraint is hard
  ♦ Design without a target market is equally hard
    → *Fortran H and PL.8 had well-defined code bases against which they were tested*

• Pick an order in which to apply the transformations
  ♦ Because of the interactions between techniques, order is a complex issue
    → *PL.8’s -00, -01, and -02 options correspond to different orders and parameters*
  ♦ Each application presents different inefficiencies and may need different transformations and a different order
    → *Each transformation attacks some specific problem*
    → *Code that does not contain that problem will not benefit*

See the “ordering” papers on web site.
What have we seen so far?

**Redundancy elimination**
- LVN, SVN, DVN, GCSE based on AVAIL information
- It is a category by itself in the taxonomy

**Loop Unrolling**
- Form of specialization (replication)

**Block placement**
- Form of latency hiding & resource management

**Inline substitution**
- Form of specialization

**Procedure placement**
- Form of latency hiding & resource management
Near-term Roadmap

**Data-flow analysis**
- 1 lecture look at other data-flow problems
- May come back and do another data-flow solver later

**Static single assignment form**
- Construction and destruction of SSA
- Example algorithms that use SSA

**Populating the Taxonomy**
- More transformations, more transformations, more ...
- Fill in the taxonomy in time for you to make progress on the lab