COMP 512
Rice University
Spring 2015

The Big Picture

The Programming Assignment & A Taxonomy of Optimizations

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**, **storeAI**

Programming Assignment (*aka "The Lab"*)

Rules of the game

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

- 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

Optimization

The subject is confusing

- Whole notion of optimality
- Incredible number of transformations
- Odd, inconsistent terminology

Cooper McKinley, & Torczon cite 237 distinct papers in their unpublished survey!

Value numbering
Redundancy elimination
Common subexpressions

Maybe this stuff is inherently hard

- Many intractable problems
- Many NP-complete problems
- Much overlap between problems and between solutions
- If optimization <u>wasn't</u> confusing, why take **COMP 512** ?

Optimization

Allen-Cocke Catalog (1971)

Procedure integration
(open, semi-open, semi-closed, closed)

Loop unrolling

Loop fusion

Loop unswitching

Redundancy elimination

Code motion

Constant propagation

Dead-code elimination

Strength reduction

Linear function test replacement

Carry optimization

Instruction scheduling

Register allocation

Storage mapping

Shadow variable optimization

Anchor pointing

Special case code generation

Peephole optimziation

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)

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

And this was before the literature exploded

And I forgot constant propagation …

Optimization

From EaC1e (2006) and EaC2e (2012)

LVN, **SVN**, **DVN**, **GCSE** w/**AVAIL**

DAG construction

Superblock cloning

Eliminating useless code (**DEAD**)

Eliminating unreachable code (**CLEAN**)

Lazy Code Motion

Hoisting

Sinking

Constant propagation

Loop unrolling

Loop unswitching

Renaming

Peephole optimization

Tail-recursion elimination

Operator strength reduction

Linear function test replacement

Procedure abstraction

Procedure placement

Block placement & fluff removal

Instruction scheduling

Register allocation

Tree-height balancing

Leaf-call optimization

Parameter promotion

Procedure cloning

Copy coalescing

Inline substitution

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

Revival
Partially-dead code
Forward propagation

Not all those techniques can possibly be the best !

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

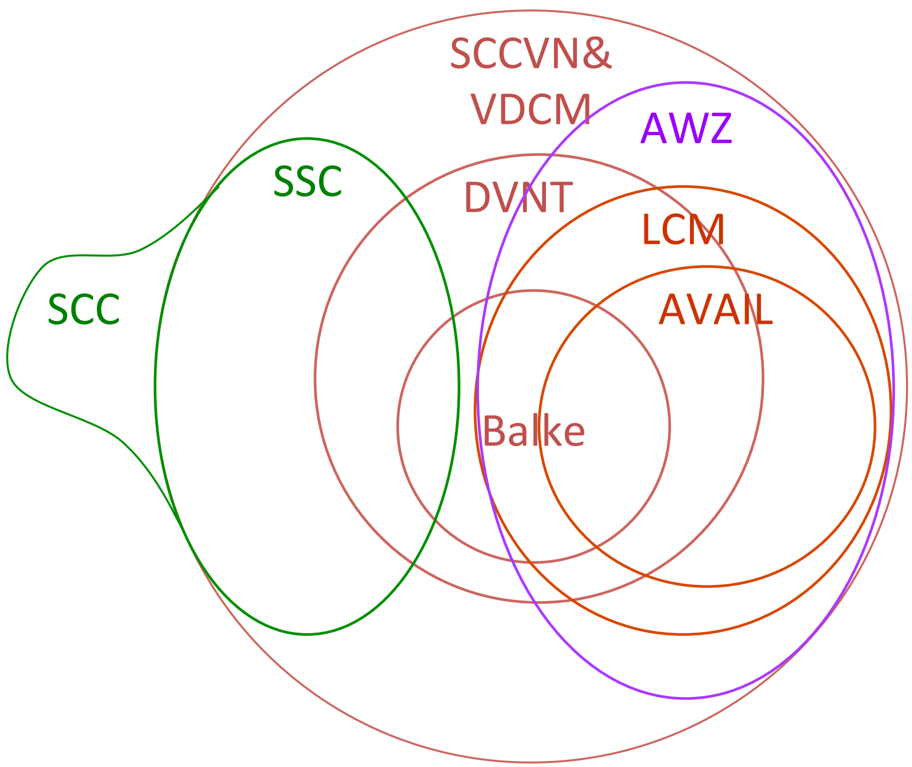
Reality lies somewhere in the middle

⇒Waterman showed that program-specific heuristics win

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 [Click]
  - 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

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

Constant time	***Low-order polynomial time***	***Hard Problems***
LVN, SVN, DVNT Block placement Tree balancing (ILP)	Loop unrolling	Reassociation Inline substitution Spill Choice in RA Copy Coalescing

Decision complexity forms another axis of comparison among transformations

Other Axes: Selection and Order

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*** **-O0**, **-O1**, ***and*** **-O2** ***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