Evolving the Next Generation of Compilers

Keith D. Cooper

Department of Computer Science
Rice University
Houston, Texas

Collaborators:
Phil Schielke, Devika Subramanian, Linda Torczon
Tim Harvey, Steve Reeves,
Alex Grosul, Todd Waterman, L. Alamagor

Funding:
DARPA (early work), DOE LACSI, NSF ITR
Roadmap

• The opportunity for change

• Building slower compilers
  ♦ Randomized iterative repair instruction schedulers
  ♦ A multi-optimizer

• Choosing “good” optimization sequences
  ♦ Understanding the search spaces
  ♦ Design & evaluation of search algorithms
  ♦ Speeding up Evaluations

• Roadmap for Future Work

There are no conclusions; we are far from done
The Opportunity for Change

The structure of compilers has not changed much since 1957

- Front End, Middle Section (Optimizer), Back End
- Series of filter-style passes
- Fixed order of passes
2000: The Pro64 Compiler

Open source optimizing compiler for IA 64
- 3 front ends, 1 back end
- Five-level IR
- Gradual lowering of abstraction level

Evolving the Next Generation of Compilers
Conventional Wisdom

Compilers should

• Use algorithms that take linear (or near-linear) time
• Produce outstanding code from arbitrary inputs
• Build object code that can be assembled like a brick wall

These goals limit the designs of our compilers
The Opportunity for Change

Over the years, computers have become markedly faster

Compilers have not taken advantage of the quantum increases in compute power provided by Moore’s law
The Opportunity for Change

Over the years, computers have become markedly faster.

Compilers have not taken advantage of the quantum increases in compute power provided by Moore’s law.

We can afford slower compilers if they do something useful with the extra time.
The Need for Change

For forty five years, we’ve been doing research on compilation and we’ve been building compilers …

- Hundreds of papers on transformations
- Hundreds of papers on analysis
- A few useful papers on experience …

Unfortunately, the compilers that we use still don’t deliver the performance that we were promised

Research has focused on transformations & analysis

Maybe we need to look at other aspects of compiler construction → such as the structure of our compilers
Building Slower Compilers

In 1996, we began to look at what a compiler might do with 10x or 100x more compilation time

- Most compilers would finish the job early & declare victory
- We began looking at the opportunities
  - More expensive analysis (n^6 pointer analysis?)
  - Many more transformations (what & when)
  - Compile code 10 ways & keep the best version

This inquiry led to an in-depth study of instruction scheduling

- How good is list scheduling?
- Can we do better? (see Sebastian Winkel’s talk, next)
Iterative Repair Scheduling

Search technique from AI based on randomization & restart

• Used for (small) scheduling problems in other domains

• Assume a simple invalid schedule & fix it
  ♦ Pick an error at random & reschedule that operation
  ♦ Different runs find different valid schedules

• Finds better schedules for hard problems (at higher cost)
  ♦ Schedules are often better in secondary criteria (register use)

How often are schedules hard enough to justify IR?

• Examined blocks & extended blocks from benchmarks

• Examined > 85,000 distinct synthetic blocks

Phil Schielke’s thesis
Iterative Repair Scheduling

What did we learn?

• List scheduling does well on codes & models we tested
  ♦ RBF does 5 forward & 5 backward passes, with randomized tie-breaking
  ♦ RBF found optimal schedules for 92% of blocks & 73% of EBBs
  ♦ RBF found optimal schedules for 80% of synthetic blocks

• IR Scheduling also finds good schedules
  ♦ Schedules that use fewer resources than RBF’s optimal one
  ♦ Optimal schedules for many where RBF fails

Parameter that predicts when to use IR

• Set of schedules where RBF is likely to find suboptimal answer and IR scheduler is likely to do well
Lessons from Iterative Repair Work

• Disappointment
  ♦ RBF does very well - conventional wisdom is right
  ♦ Used randomization, restart, & multiple trials (vs. tie breakers)

• Good understanding of the space of schedules
  ♦ Can find equivalent schedules that use fewer resources
  ♦ Can identify instances where IR is likely to beat RBF

• New model for our work
  ♦ Extensive, expensive exploration to understand problem space
  ♦ Development of effective & (reasonably) efficient techniques for the hard problems, using knowledge gained in explorations
Next Idea: Multiple Optimization Plans

Idea is simple: try several and keep the best result

- Front End
- Middle End
- Back End

Cost is roughly 4x the "old" way

Might produce better code

(Bernstein et al.)

Keep best code
Next Idea: Multiple Optimization Plans

Idea is simple: try several and keep the best result

Implementation leads immediately to some hard questions

• How do we pick good sequences?
• How do we implement a compiler that can handle multiple sequences?

This investigation produced a system that used a genetic algorithm to derive good program-specific sequences

• Improvements of (roughly) 20% in speed, 12% in space
• Paper in LCTES, summer 1999

Genetic algorithm led to “Evolving” in this talk’s title
Next Idea: Multiple Optimization Plans

This single idea hijacked our research agenda

• Questions inherent in this simple idea are quite difficult
• We saw no easy way to answer them
• Led to Schielke’s work on the impact of optimization order on code size & code speed
• Spawned a project to develop & engineer compilers that adapt their behavior to input program, objective function, and target machine

We did not know that it would become a ten-year odyssey
Evolving the Next Generation of Compilers

Tries to minimize an objective function using adaptive search

- Finds the “right” configuration for the compiler & input
  - Set of optimizations & an order to run them
- Uses multiple trials & feedback to explore solution space

Prototype Adaptive Compiler

- Front end
- Steering Algorithm
- Objective function
- Executable code
- Vary parameters

Evolving the Next Generation of Compilers
Finding “right configuration” is hard

- Multiple algorithms for each effect
- Different scopes, cases, domains, strengths, & weaknesses
- Overlap between effects complicates choices
Choosing Optimization Sequences

The Problem:

find the best sequence of transformations for your program

What’s hard

• 16 optimizations in our compiler (ignoring option flags)
• With 10 passes, that is $16^{10}$ possibilities
• Interactions are nonlinear, unpredictable, & overlapping
• Want to pick a minimizer for objective function quickly
Prototype Adaptive Compiler

- Based on MSCP compiler
- 16 transformations
  - Run in almost any order (not easy)
  - Many options & variants
- Search-based steering algorithms
  - Hill climber (valley descender?)
  - Variations on a genetic algorithm
  - Exhaustive enumeration
- Objective functions
  - Run-time speed
  - Code space
  - Dynamic bit-transitions
- Experimental tool
  - Exploring applications
  - Learning about search space
  - Designing better searches

An effective way to find some subtle optimization bugs
Early Experiments

• Genetic Algorithm (12 passes drawn from pool of 10)
  ♦ Evaluate each sequence
  ♦ Replace worst + 3 at each generation
  ♦ Generate new strings with crossover
  ♦ Apply mutation to all, save the best

• Optimizing for space then speed
  ♦ 13% smaller code than fixed sequence (0 to 41%)
  ♦ Code was often faster (26% to -25%; 5 of 14 slower)

• Optimizing for speed then space
  ♦ 20% faster code than fixed sequence (best was 26%)
  ♦ Code was generally smaller (0 to 5%)

• Found “best” sequence in 200 to 300 generations of size 20

Register-relative procedure abstraction gets 5% space, -2% speed

N.B.: 6,000 compilations
No measure of solution quality

This GA turns out to be fairly poor. Even so, it took many fewer probes to find “best” sequence than did random sampling.
Choosing Optimization Sequences

Classic optimization problem

- Compiler looks for minimizer in some discrete space
  - $16^{10}$ points for 10-pass optimization in prototype compiler
  - Can obtain function values for any point, at some cost

- Need to understand the properties of the search space
  - Depends on base optimizations & interactions between them
  - Depends on program being optimized
  - Depends on properties of the target machine

- Difficult and complex problem ...
  - But the genetic algorithm performs well in this space
Choosing Optimization Sequences

Work has two major thrusts

• Characterizing the search spaces
  ◆ Large-scale enumerations of small spaces to develop insights
  ◆ Small-scale experiments in large spaces to confirm insights

• Designing effective search algorithms
  ◆ Rapid offline experiments in enumerated spaces
  ◆ Confirming online experiments in large spaces

• Question: can we understand the space analytically?
  ◆ Models of optimizations & their combinations
  ◆ I don’t yet know enough about interactions & effects

Is it convex or differentiable?
Characterizing the Search Space

Enumeration Experiments

• Full search space is huge: \(1,099,511,627,776\) points in \(16^{10}\)

• Work with tractable subspaces: \(5^{10}\) has \(9,765,625\) points
  ♦ Work with small programs, of necessity
  ♦ Enumerate full \(5^{10}\) subspaces & analyze data offline
  ♦ First enumeration, FMIN, took 14 CPU-months
  ♦ Today, takes about 2.5 CPU-months
  ♦ We’ve done 6 or 7 full enumerations in \(5^{10}\)
    \[\geq 60,000,000\] compilations & evaluations

• Follow paradigm from iterative repair scheduling work
  ♦ Large-scale studies to gain insight, randomization & restart

Farm of Apple XServes and a couple of Suns

Evolving the Next Generation of Compilers

Characterizing the Spaces
What Have We Learned About Search Spaces?

We confirmed some obvious points:

These spaces are:

- not smooth, convex, or differentiable
- littered with local minima at different fitness values
- program dependent

\[ p: \text{peeling} \]
\[ l: \text{PRE} \]
\[ o: \text{logical peephole} \]
\[ s: \text{reg. coalescing} \]
\[ h: \text{useless CF elimination} \]
What Have We Learned About Search Spaces?

We confirmed some obvious points:

These spaces are:

- not smooth, convex, or differentiable
- littered with local minima at different fitness values
- program dependent

\[ \begin{align*}
  \text{p: peeling} \\
  \text{l: PRE} \\
  \text{o: logical peephole} \\
  \text{s: reg. coalescing} \\
  \text{n: useless CF elimination}
\end{align*} \]

Evolving the Next Generation of Compilers
What About Presentation Order?

Clearly, order might affect the picture ...

Evolving the Next Generation of Compilers

- adpcm-coder, $5^4$ space, plosn
What Have We Learned About Search Spaces?

Both programs and optimizations shape the space

- p: peeling
- l: PRE
- o: logical peephole
- s: reg. coalescing
- u: useless CF elimination

⇒ Range is 0 to 70%
⇒ Can approximate distribution with 1,000 probes
What Have We Learned About Search Spaces?

Both programs and optimizations shape the space

- peeling (p)
- dead code elimination (d)
- useless CF elimination (n)
- dominator value num’g (x)
- strength reduction (†)

⇒ Range is compressed (0-40%)
⇒ Best is 20% worse than best in “plosn”
What Have We Learned About Search Spaces?

Many local minima are “good”

Many local minima
258 strict
27,315 non-strict

Lots of chances for a search to get stuck in a local minima
What Have We Learned About Search Spaces?

Distance to a local minimum is small

Downhill walk halts quickly

Best-of-\(k\) walks should find a good minimum, for big enough \(k\)

Evolving the Next Generation of Compilers

Characterizing the Spaces
Search Algorithms

• Knowledge does not make the code run faster
• Need to use our knowledge to build better search techniques

Moving from curiosity-driven research to practical work
Search Algorithms: Genetic Algorithms

- Original work used a genetic algorithm (GA)
- Experimented with many variations on GA
- Current favorite is GA-50
  - Population of 50 sequences
  - 100 evolutionary steps
- At each step
  - Best 10% survive
  - Rest generated by crossover
    - Fitness-weighted reproductive selection
    - Single-point, random crossover
  - Mutate until unique

GA-50 finds best sequence within 30 to 50 generations
Difference between GA-50 and GA-100 is typically < 0.1%
This talk shows best sequence after 100 generations ...
Search Algorithms: Hill climbers

Many nearby local minima suggests descent algorithm

• Neighbor $\Rightarrow$ Hamming-1 string \hspace{1cm} (differs in 1 position)
• Evaluate neighbors and move downhill
• Repeat from multiple starting points

• Steepest descent $\Rightarrow$ take best neighbor
• Random descent $\Rightarrow$ take 1st downhill neighbor \hspace{1cm} (random order)
• Impatient descent $\Rightarrow$ random descent, limited local search
  ♦ HC algorithms examine at most 10% of neighbors
  ♦ HC-10 uses 10 random starting points, HC-50 uses 50
Search Algorithms: Greedy Constructive

Greedy algorithms work well on many complex problems.

How do we do a greedy search?

1. start with empty string
2. pick best optimization as 1st element
3. for i = 2 to k
   try each pass as prefix and as suffix
   keep the best result

Algorithm takes $k \cdot (2n - 1)$ evaluations for a string of length $k$.

Takes locally optimal steps.

Early exit for strings with no improvement.

Local minimum under a different notion of neighbor.
Search Algorithms: Greedy Constructive

One major complication: equal-valued extensions, or ties

- Ties can take GC to wildly different places
- Have experimented with three GC algorithms
  - GC-exh explores pursues all equal-valued options
  - GC-bre does a breadth-first rather than depth-first search
  - GC-10 & GC-50 break ties randomly and use restart to explore

<table>
<thead>
<tr>
<th></th>
<th>adpcm-d</th>
<th>GC-exh</th>
<th>GC-bre</th>
<th>GC-50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequences checked</td>
<td>91,633</td>
<td>325</td>
<td>2,200</td>
<td></td>
</tr>
<tr>
<td>Code speed</td>
<td>—</td>
<td>+ 0.003%</td>
<td>+ 2%</td>
<td></td>
</tr>
</tbody>
</table>

Experiments use GC-10 & GC-50
Search Algorithm Results

Variety of codes
5 searches
+ training/testing
All do pretty well

Operations executed relative to rvzco
dtvzcod
Simulated RISC machine

Evolving the Next Generation of Compilers
Variety of codes
5 searches
+ training/testing
All do pretty well

Greedy has some problems
- fmin, tomcatv
- price to pay?

Operations executed relative to rvzcodtvzcod
Simulated RISC machine

Evolving the Next Generation of Compilers
Variety of codes
5 searches
+ training/testing
All do pretty well

Training/testing data shows small variation
- no systematic bias from training data
Surprisingly fast
- old GA took 6,000
- several < 1,000

GC can explode
- zeroin, nsieve

50 generations of
GA-50 does almost as well as 100
Evolving the Next Generation of Compilers

Focusing on the Cheap Techniques

HC-10:
- 10x faster than old GA
- 7x faster than GA-50

GC-10
- does well, on average
- ties cause problems
- fmin & tomcatv had slow code; does not show up in costs...

Sequences evaluated by search algorithm
Search Algorithms

So, where are we?

• Find good sequences in 200 to 600 probes
  ♦ Good means “competitive with GA-50 for 100 generations”

• How close are we to global minimum?
  ♦ Cannot know without enumeration
  ♦ Enumeration is hard to justify in $16^{10}$ (& harder to perform)
  ♦ Current experiment: HC-100000 on several codes
    (on a 264-processor IA-64 machine)

• Next major step is bringing code features into the picture ...
Designing Practical Adaptive Compilers

User may not want to pay 300x for compilation

⇒ Moore’s law will help ...

Engineering approaches

• Make the search incremental across many compilations
• Develop faster techniques to evaluate sequences on codes
• Use parallelism
• And, of course, make the compiler fast
Speeding up Evaluation

Compile-evaluate cycle takes most of the time

- Faster evaluation methods
  - Low-level performance models (Mellor-Crummey et al.)
  - Analytical models (Soffa et al.)

- Machine-learning to predict sequences
  - Probabilistic models may reveal consistent relationships
  - Want to relate sequences to source-code features

Success in any or all of these approaches could reduce the cost of evaluating each sequence
But, All These Experiments Used Our Compiler

We have tried a number of other compilers, with no success

- Try to reorder a pass in GCC
- Hit problems in GCC, SUIF-1, ORC, ...
- Look forward to using LLVM & Phoenix

Our platform is reconfigurable by accident of design

- Have run > 100,000,000 configurations in our system
- One unavoidable phase-order bug

We have used MIPSPro in another series of experiments
Our goals

- **Short term (now)***
  - Characterize the problems, the potential, & the search space
  - Learn to find good sequences quickly (*search*)

- **Medium term (3 to 5 years)**
  - Develop proxies and estimators for performance (*speed*)
  - Demonstrate practical applications for adaptive scalar optimization
  - Understanding interface between adaptive controller & compiler

- **Long term (5 to 10 years)**
  - Apply these techniques to harder problems
    - Data distribution, parallelization schemes on real codes
    - Compiling for complex environments, like the Grid
  - Develop a set of design & engineering principles for adaptive compilers
Where Does This Research Lead?

Practical systems within ten years

How will they work? (Frankly, we don’t yet know)

- Efficient searches that capitalize on properties of the space
- Incremental searches distributed over program development
- Predictive techniques that use program properties to choose good starting points
- Compiler structures & parameterizations that fit adaptation

In the meantime, we have a lot of work to do

And machines will keep getting faster …
Support Slides Begin Here
Characterizing FMIN

- One application, FMIN
  - 150 lines of Fortran, 44 basic blocks
  - Exhibited complex behavior in other experiments
- Ran hill-climber from 100 random starting points
  - Picked the 5 most used passes from winners
- Evaluated all strings of length 10 from those 5
  - 9,765,625 distinct combinations (for speed)
  - 34,000 to 37,000 experiments/machine/day
  - Produces sequence & number of cycles to execute
  - First subspace took 14 CPU-months to complete
- We’re learning from the results
  - Can run experiments against the data offline
Results from FMIN-PLOSN

In the initial subspace - plosn (9,765,625 sequences)

- Best sequence is 986 cycles
- Seven at 987
- Worst sequence is 1716
- Unoptimized code takes 1765
- Range is 3% to 43% improvement

Local minima

- 258 strict local minima  \( x < \) all Hamming-1 points
- 27,315 non-strict local minima  \( x \leq \) all Hamming-1 points
- All local minima are improvements
- They come in many shapes

With full set of transformations:

- GA: 822 in 184 generations of 100
  825 in 152 generations of 100
- Hill: 833 at round 2232
  830 at round 750
Results from FMIN-PLOSN

Strict local minima have many shapes

132 of these spikes, from loop peeling

Magnitude is wall height (≥ 1)

Hamming-1 points

Landscape looks like a pock-marked asteroid, in 10 dimensions

Evolving the Next Generation of Compilers
Applying IR Scheduling

1. Measure average ready queue length in list scheduler

2. Check resulting code
   - No holes in schedule ⇒ take the existing schedule
   - Holes in the schedule + schedule longer than critical path
     ⇒ invoke IR scheduler to look for better schedule

This regime invokes IR when it might pay off

- Infrequent use
- Reasonable likelihood of profit
Iterative Repair Scheduling (& Int. Lin. Prog.)

- We also tried integer linear programming
- Produced improvements that were similar to IR scheduling
- CPLEX ran for a long time on simple cases
  - Too many choices that lead to good schedules
  - May be an artifact of our formulation of the problem
  - From this perspective, Sebastian Winkel appears to have a better formulation than ours
Support: Choosing Optimization Sequences

Consider redundancy elimination

- **Dominator Value Numbering (DVNT)**
  - Extends local value numbering to dominator regions
- **Alpern-Wegman-Zadeck’s partitioning method (AWZ)**
  - Uses Hopcroft’s DFA minimizer to prove equivalence
  - Second pass performs replacement
- **GCSE using Available Expressions**
  - Global data-flow analysis to find redundant expressions
  - Second pass performs replacement
- **Lazy Code Motion (LCM)**
  - Extends GCSE to handle partial redundancy
  - Computes careful placement for inserted expressions
Support: Choosing Optimization Sequences

How do we choose among DVNT, AWZ, GCSE, & LCM?

- Only DVNT finds $x + x = 2 \times x$
- Only DVNT & AWZ can prove $x \times y = x \times z$ when $y = z$
- DVNT cannot find redundancy across a loop-closing branch
- Only AWZ is optimistic; it finds loop-based redundancies that the others miss
- Only LCM finds partial redundancies
- Only DVNT folds constants
- Only LCM moves loop-invariant code

The BEST choice depends on the input code, the opportunities that it presents, and the tradeoffs that it embodies.