Lessons from a Decade of Adaptive Compilation

Keith D. Cooper

Department of Computer Science
Rice University
Houston, Texas

Funding: DOE, NSF, Microsoft, TI

And many collaborators ...

This lecture differs from the others given this semester in that it is explicitly a history of the work on adaptive compilation done at Rice.

Thesis

Compilers that adapt their optimization strategies to new applications and new targets should produce better code than any single-strategy compiler

• This idea was novel when we first stated it in 1997
• It is now accepted as (almost dogmatically) true
  ♦ For scalar optimizations, difference is 20 to 40%

• We have spent a decade working on schemes that let the compiler adapt its behavior to specific applications
  ♦ Search space characterization & algorithm development
  ♦ Parameterization & control of optimizations

• This talk will try to distill some of that experience & insight
Let’s Make Optimization Cost Much More

We noticed that, paradoxically, (1) Moore’s law made cycles cheaper, and (2) compiler writers were focused on the asymptotic complexity of algorithms and compilers

- Given more cycles, compiler would declare victory & quit
- Fraction of peak performance was falling
  - 5 to 10% considered good on commodity processors
- In some contexts, customers will pay for performance
  - High-performance scientific computation (e.g., ATLAS)
  - Embedded systems
  - Reconfigurable devices & application-specific hardware

- The key is to spend those extra cycles profitably
  - Slower algorithms are obviously the wrong answer

Lessons from a Decade of Adaptive Compilation

History

In the beginning, compilers used a single, predetermined strategy to compile every application

Fortran Automatic Coding System, IBM, 1957

To recap:

Compiler designers decided how to optimize your application years before you wrote it!

Doesn’t that seem a bit like fortune telling?
History

First steps toward adaptive behavior in compilers

- Run multiple heuristics and keep the best result
  - Bernstein et al. with spill-choice heuristics (1989)
  - PGI i860 compiler ran forward & backward schedulers (1991)
  - Bergner, Simpson, & others followed …

- Randomization & restart
  - Briggs duplicated Bernstein’s results by renaming (1991)
  - Schielke studied instruction scheduling & allocation
    - Large scale studies with iterative repair (1995)
    - Grosul’s thesis has >200,000,000 runs behind it … (2005)

- Automatic derivation of compiler heuristics
  - Palem, Motwani, Sarkar, & Reyen used α-β tuning (1995)
  - Amarasinghe et al. used genetic programming (2003)
  - Waterman used search over space of heuristics (2005)

Lessons from a Decade of Adaptive Compilation

History

Our work to date

- Finding good application-specific optimization sequences
- Design & evaluation of search strategies
  - Large-scale studies of search-space structure & algorithm effectiveness (hundreds of thousands of trials)
  - Genetic algorithms, hill climbers, greedy constructive algorithm, GNE, pattern-based direct search
- Discovering optimization parameters for good performance
- Adaptation within transformations
  - Inline substitution, register coalescing
  - Loop fusion, tiling, unrolling
- Design of effective parameter schemes
  - Waterman’s work on inline substitution

Lessons from a Decade of Adaptive Compilation
Roadmap

- Problems we have attacked
- Search space characterization
- Search algorithms
- Lessons we have learned
  - Parameterization, Feedback metrics, ...
- Future work

Some Sample Adaptive Compilation Problems

We have worked on a number of problems in this area

- Finding good optimization sequences
  - Program-specific or procedure specific
- Finding good optimization parameters
  - Block sizes for tiling, loop unrolling factors
- Loop fusion & tiling
  - Choosing loops to fuse and tiling them
- Inline substitution
  - Deriving good program-specific inlining heuristics
- Adaptive coalescing of register-to-register copies
  - Unifying multiple heuristics in an adaptive framework
Finding Optimization Sequences

Prototype adaptive compiler (1997 to 2007)

- Treat set of optimizations as a pool
- Use feedback-driven search to choose a good sequence
- Performance-based feedback drives selection
  ♦ Performance might mean speed, space, energy, ...

Our Approach

We took an academic’s approach to the problem

- Experimental characterization of subset search spaces
- Use properties we discover to derive effective searches
- Validate the characterization by running the new search algorithms in the full space
Our Approach Applied to Sequence Finding

We took an academic's approach to the problem

- Experimental characterization of subset search spaces
  - Full space was 16 opts, strings of 10 \(1,099,511,627,776\) strings
  - Enumerated space of 5 opts, strings of 10 \(9,765,625\) strings
  - Compiled and ran code with each sequence

- Use properties we discover to derive effective searches
  - These search spaces are ugly
  - Many good solutions, steep downhill slopes
  - Derived impatient HC, GNE, better GAs

- Validate the characterization by running the new search algorithms in the full space
  - Large scale experiments reported in Grosul's thesis
  - Reduced 20,000 probes (1997) to a couple hundred (now)
  - 20% to 40% improvement in runtime speed

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

\(p\): peeling
\(f\): PRE
\(o\): logical peephole
\(s\): reg. coalescing
\(n\): 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

What About Presentation Order?

Clearly, order might affect the picture ...

Still, some bad local minima
Both Programs & Optimizations Shape the Space

Two programs, same set of optimizations

Distribution relative to the best value

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

⇒ Range is 0 to 70%
⇒ Can approximate distribution with 1,000 probes

1,000 probes should get us a good solution

Both Programs & Optimizations Shape the Space

Same two programs, another set of optimizations

Distribution relative to the best value

- p: peeling
- d: dead code elimination
- n: useless CF elimination
- x: dominator value num'g
- t: 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
(of 9,765,625)

Lots of chances for a search to get stuck in a local minima

Distance to a local minimum is small

Downhill walk halts quickly

Best-of-\(k\) walks should find a good minimum, for big enough \(k\)
Our Approach Applied to Sequence Finding

We took an academic’s approach to the problem

- Experimental characterization of subset search spaces
  - Full space was 16 opts, strings of 10 (1,099,511,627,776 strings)
  - Enumerated space of 5 opts, strings of 10 (9,765,625 strings)
  - Compiled and ran code with each sequence
- Use properties we discover to derive effective searches
  - These search spaces are ugly
  - Many good solutions, steep downhill slopes
  - Derived impatient HC, GNE, better GAs
- Validate the characterization by running the new search algorithms in the full space
  - Large scale experiments reported in Grosul's thesis
  - Reduced 20,000 probes (1997) to a couple hundred (now)
  - 20% to 40% improvement in runtime speed

Roadmap

- Problems we have attacked
- Search space characterization
- Search algorithms
- Lessons we have learned
  - Parameterization, feedback metrics, ...
- Future work

Lessons from a Decade of Adaptive Compilation
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 (4,550 trials)
- 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 (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 (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 create 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

95 evaluations for 10-of-5 space

Successive evaluations refine the string

<table>
<thead>
<tr>
<th>1st pass</th>
<th>2nd pass</th>
<th>3rd pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>sp</td>
<td>snp</td>
</tr>
<tr>
<td>l</td>
<td>ps</td>
<td>psn</td>
</tr>
<tr>
<td>o</td>
<td>sl</td>
<td>snl</td>
</tr>
<tr>
<td>s</td>
<td>ls</td>
<td>lsn</td>
</tr>
<tr>
<td>n</td>
<td>so</td>
<td>sno</td>
</tr>
<tr>
<td>s</td>
<td>os</td>
<td>osn</td>
</tr>
<tr>
<td></td>
<td>ss</td>
<td>sns</td>
</tr>
<tr>
<td></td>
<td>sn</td>
<td>ssn</td>
</tr>
<tr>
<td></td>
<td>ns</td>
<td>ssn</td>
</tr>
</tbody>
</table>

Lessons from a Decade of Adaptive Compilation

Search Algorithms: Greedy Constructive

Successive evaluations refine the string

<table>
<thead>
<tr>
<th>1st pass</th>
<th>2nd pass</th>
<th>3rd pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>sp</td>
<td>snp</td>
</tr>
<tr>
<td>l</td>
<td>ps</td>
<td>psn</td>
</tr>
<tr>
<td>o</td>
<td>sl</td>
<td>snl</td>
</tr>
<tr>
<td>s</td>
<td>ls</td>
<td>lsn</td>
</tr>
<tr>
<td>n</td>
<td>so</td>
<td>sno</td>
</tr>
<tr>
<td>s</td>
<td>os</td>
<td>osn</td>
</tr>
<tr>
<td></td>
<td>ss</td>
<td>sns</td>
</tr>
<tr>
<td></td>
<td>sn</td>
<td>ssn</td>
</tr>
<tr>
<td></td>
<td>ns</td>
<td>ssn</td>
</tr>
</tbody>
</table>

Lessons from a Decade of Adaptive Compilation
Search Algorithms: Greedy Constructive

Unfortunately, ties (equal-valued choices) pose a major problem

- 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-n breaks ties randomly and use n random starting points

<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>1.0</td>
<td>+0.003%</td>
<td>+2%</td>
<td></td>
</tr>
</tbody>
</table>

- Yi Guo developed GNE, a greedy variant that does a more careful search of local neighbors. In preliminary tests, it outperforms greed constructive.

Search Algorithms: Pattern-based Direct Search

Qasem has shown that PBDS does well in the search spaces that arise in loop-fusion and tiling

- Deterministic algorithm that systematically explores a space
  - Needs no derivative information
  - Derived (via long trail) from Nelder-Meade simplex algorithm
  - For $p_1, p_2, p_3, ..., p_n$, examines neighborhood of each $p_i$
    - Systematically looks at $p_1 \pm s, p_2 \pm s, p_3 \pm s, ..., p_n \pm s$
    - Finds better values (if any) for each parameter, then uses them to compute a new point
    - When exploration yields no improvement, reduces $s$
  - For fusion and tiling, it outperforms window search, simulated annealing, & random search
    - Good solutions for fusion & tiling in 30 to 90 evaluations

Random does surprisingly well, suggesting that the space has many good points.
**Roadmap**

- Problems we have attacked
- Search space characterization
- Search algorithms
- Lessons we have learned
  - Parameterization, feedback metrics, ...
- Future work

**What Have We Learned?**

- Adaptation finds better solutions
  - Sequences, tiling, inlining, fusion & tiling, copy coalescing
- Search can navigate in these huge, ill-mannered spaces
  - Down from 20,000 trials to the range of 100 to 500 trials
  - In most spaces, can find reasonable improvements
- Specific parameterization is crucial
  - Must find effective parameterization
    - ORC's "temperature" heuristic vs. Waterman's CNF exprs
    - Sandoval added optimization that made space much larger, but produced faster search termination at better values
  - With PBDS, getting parameterization right is critical (Lewis)
What Have We Learned?

To make it practical, must combine lots of ideas

- Evaluation is expensive, so avoid it
  - Hash search points to avoid re-evaluation
  - Recognize identical results (same code, different point)
  - In many cases, simulated execution is good enough
    - Fall-back position when update fails? Run the code!
- Performance measures should be:
  - Stable (e.g., operation counts versus running time)
  - Introspective
    - Have allocator report amount of spilling
    - Look at the schedule for unused slots rather than execute
  - Directly related to solution quality (if possible)
    - Cache simulation for fusion & tiling

What Have We Learned?

Selecting optimizations where internal adaptation pays off

- Consider “decision complexity” of a transformation
  - LVN, SVN, LCM have $O(1)$ decision complexity
    - Each decision takes (small) constant time
  - Inlining, register coalescing have huge decision complexity
    - Making best decision is truly hard
  - Hypothesize that some transformations have low-order polynomial decision complexity
    - Block cloning with size constraint?
    - Loop unrolling?  
      (num regs is a practical constraint)
- Internal adaptation makes sense when complexity of making the best decision is high-order polynomial or worse
  - Have studies that show good results for inlining, coalescing, and combined loop optimization
Some of these passes have their own feedback-driven adaptive controls — memory hierarchy opts, inliner, allocator, scheduler.

The result is a compiler that uses (& manages) multiple levels of feedback-driven adaptation — From this structure, we have the platform to expand into managing other parameters that affect performance.
Lessons from a Decade of Adaptive Compilation

Multi-level Feedback-driven Adaptation

Source code → High-level opt’n → Adaptive Control
Machine Description → Low-level opt’n → Adaptive Control

Code generation → Adaptive Control

Analysis and/or Execution

Some passes should provide data directly to the adaptive controllers

Many open (research) questions

• Sequential approach to search
  ◦ Internal cycles run to completion
  ◦ Tune balance & ||'ism
  ◦ Fit code to architecture

• Solve joint search problem
  ◦ May reach solutions that cannot be reached separately
  ◦ Might be more chaotic

• What metrics best drive changes in machine description?

• Proxies for actual execution

• Efficacy of search in this context

• Replace search with learning
Multi-level Feedback-driven Adaptation

Many open (research) questions
• Impact of initial machine description on search results
• Quantization of machine parameters (num & ranges)
  ♦ May raise design questions
• Do we have the right knobs to turn? (choice & control)
• What useful metrics can the compiler expose to the process?
• Metrics other than speed
• Quantify the improvements?
Find the answers by experimentation

Lessons from a Decade of Adaptive Compilation

Multi-level Feedback-driven Adaptation

Long term questions
• Choice of source language
  ♦ How should we write applications?
  ♦ MATLAB? Mathematica?
• Heterogeneity in target?
  ♦ On-chip FPGA
  ♦ Multiple diverse cores
• Does available ||ism limit us?
Conclusions

Any conclusions would be premature at this point
- Need to start running experiments

We've come a long way since 1997
- From 10,000 to 20,000 evaluations down to hundreds
- Experience across a range of problems & search techniques
- Attracted many people to working on this kind of problem

Joint hardware/software evolution is an endpoint to our search

Search Spaces for Inline Substitution

Characterizing the search spaces with 2d parameter sweeps

Running times for inlined variants of Vortex
— as a function of inliner parameters

Space is simpler, but has unconstrained integer values
⇒ Parameterization is important
Search Spaces for Inline Substitution

Characterizing the search spaces with 2d parameter sweeps

Running times for inlined variants of bzip — as a function of inliner parameters

Space is simpler, but has unconstrained integer values

⇒ Parameterization is important

$SC$ for single-call procedures

$SC$ is inlined statement count

Lessons from a Decade of Adaptive Compilation