Moving Adaptation into Industrial ~~Individual~~ Optimizations

Musings on a decade or more of work in adaptive compilation

Keith D. Cooper

Doerr Professor of Computational Engineering
Department of Computer Science
Rice University
Houston, Texas, USA

This talk presents work sponsored under a number of research contracts and grants over a 10-year period, including NSF grants CCR-0205303, CNS-0615180, and CCF-0541165, the DOE-funded Los Alamos Computer Science Institute, and the PACE Project, funded by DARPA through AFRL Contract FA8650-09-C-7915.

The opinions and findings in this talk do not necessarily reflect the views of either the United States Government or of Rice University.

Slides posted at http://www.cs.rice.edu/~keith/Smart2010

Posted version has a list of references at the end.

Road Map

- Background & Context
- Classification of Adaptive Schemes

- Brief review of adaptive behavior in compilers
- Barriers to adoption in “real” compilers
- Rough classification of adaptive schemes
  - Compile time adaptation, runtime parameterization, runtime recompilation
- Enticing problems?
- Barriers to adoption?

Brief History of Adaptive Behavior in Compilers

Prior to late 1980s, compilers operated on fixed strategies

- One-size fits all or a couple of command-line flags (-g, -O1, -O2, -O3, …)
- Those compiler writers may have been a lot smarter than we are

Early glimpses of adaptation

- Bernstein et al.'s "best of three spilling"
- Motwani et al.s' "alpha-beta tuning" for allocation & scheduling
- Granston & Holler's system to recommend compiler options

Bergner built on this idea, as did Simpson.

Adaptation became a minor industry this decade

- Many submissions, fewer published papers
- Hard to capture full set of data & experience in a conference paper

Background

The posted version of the talk includes a blbliography

Brief History of Adaptation in Compilers

After twenty years, where are we?

- Techniques are still (largely) confined to academic sandboxes
  - ◆ Influenced the design of major systems
    - → LLVM, Phoenix built with intent of allowing adaptive choices
  - ◆ Few deployed “real” compilers that use these ideas
- Much of the low-hanging fruit has been picked
  - ◆ Optimization choice, sequence ordering, command-line flags, …
  - ◆ Genetic algorithms, genetic programming, heuristic search, greedy algorithms

Two major questions for this talk

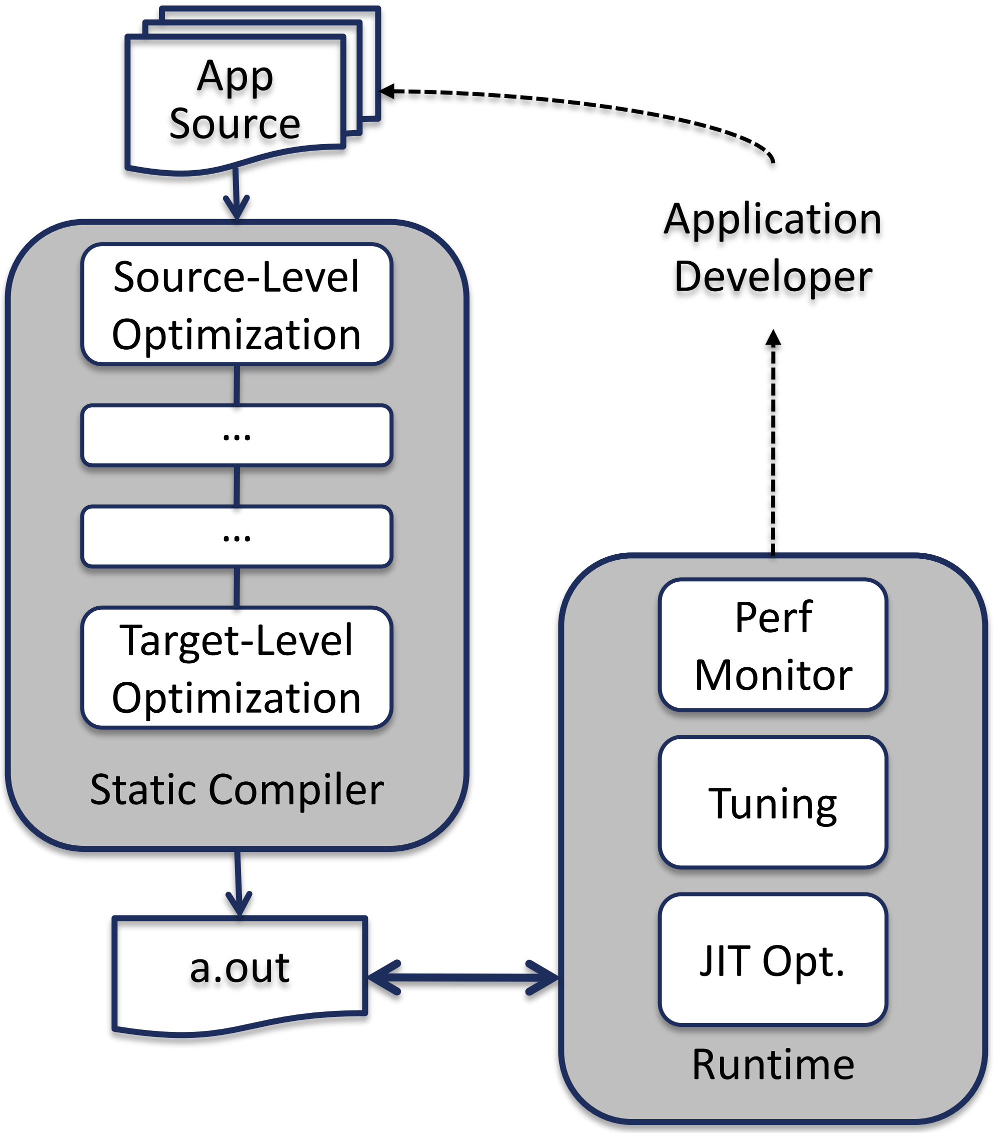
- How do we select problems for future work?
- How do we move these techniques into mainstream use?

The talk will take a somewhat circuitous path to reach these questions.

Background

The Compile / Execute Cycle

- End user compiles & runs application
- Runtime system works to measure & improve performance
- End user interprets feedback & changes code to improve it
- Compiler & runtime are opaque boxes to the application developer
- Compiler features multiple levels of "static" optimization

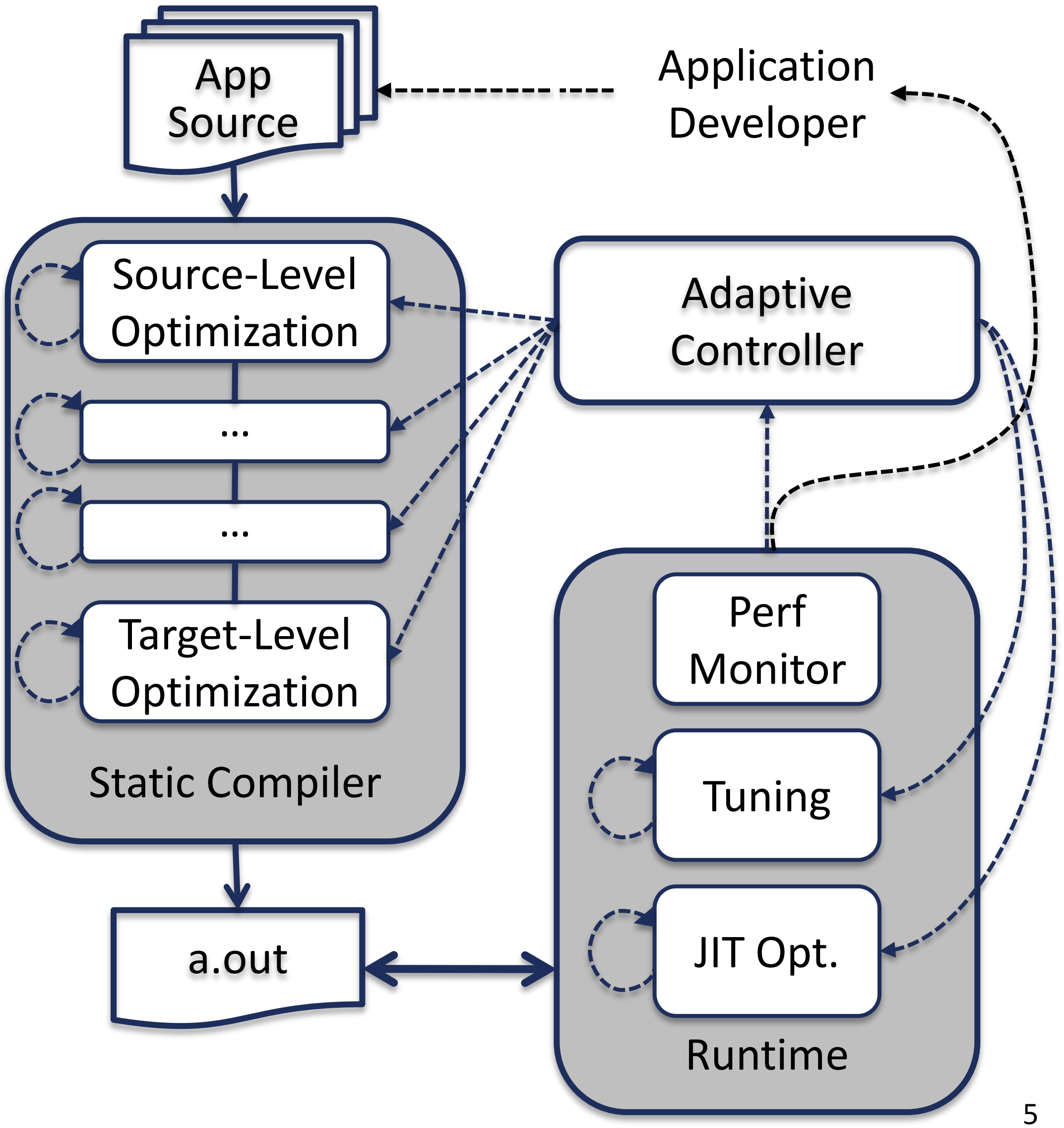


Context

Compile / Execute with Adaptive Techniques

- With adaptation, some change is automated
  - Both compile-time and runtime adaptation
  - Explicit control & coordination
- Feedback occurs in both large and small cycles
  - Several time scales for adaptation
- End user retains the ability to intervene

- Compiler structure looks much the same as before



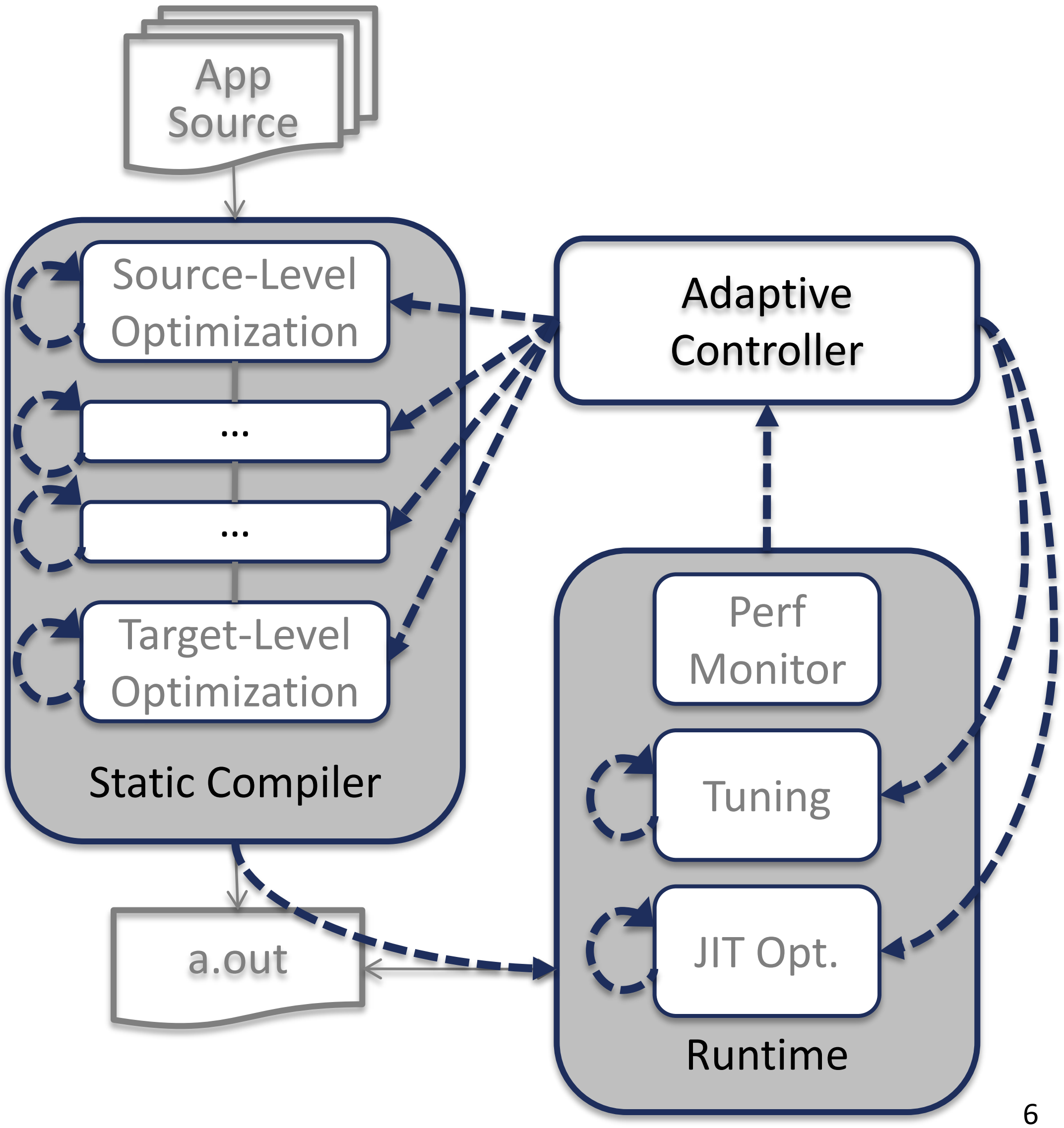
Context

Abstracting Away the Details

Ignore the flow of code through the system

- Feedback relationships become clear
- Many opportunities for feedback & adaptation

Good news: more theses are left in this area.



Context

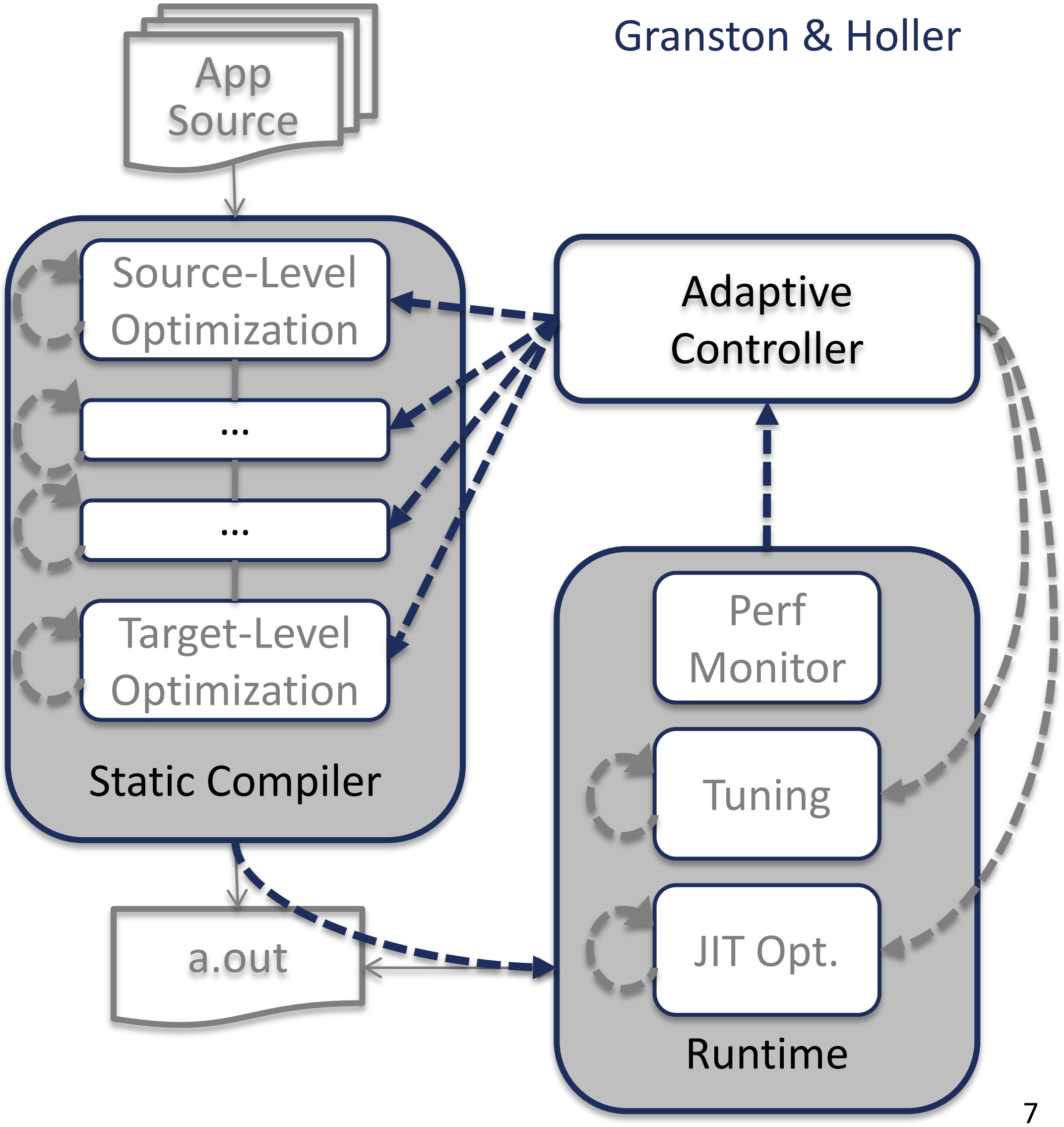
Abstracting Away the Details

Major adaptation cycle

- Compile, execute, monitor, analyze, and recompile
- Typical adaptations
  - Change compilation strategies
  - Change optimizations, sequences, or code shape (e.g., inlining)

Behavior evolves slowly over many cycles

Granston & Holler

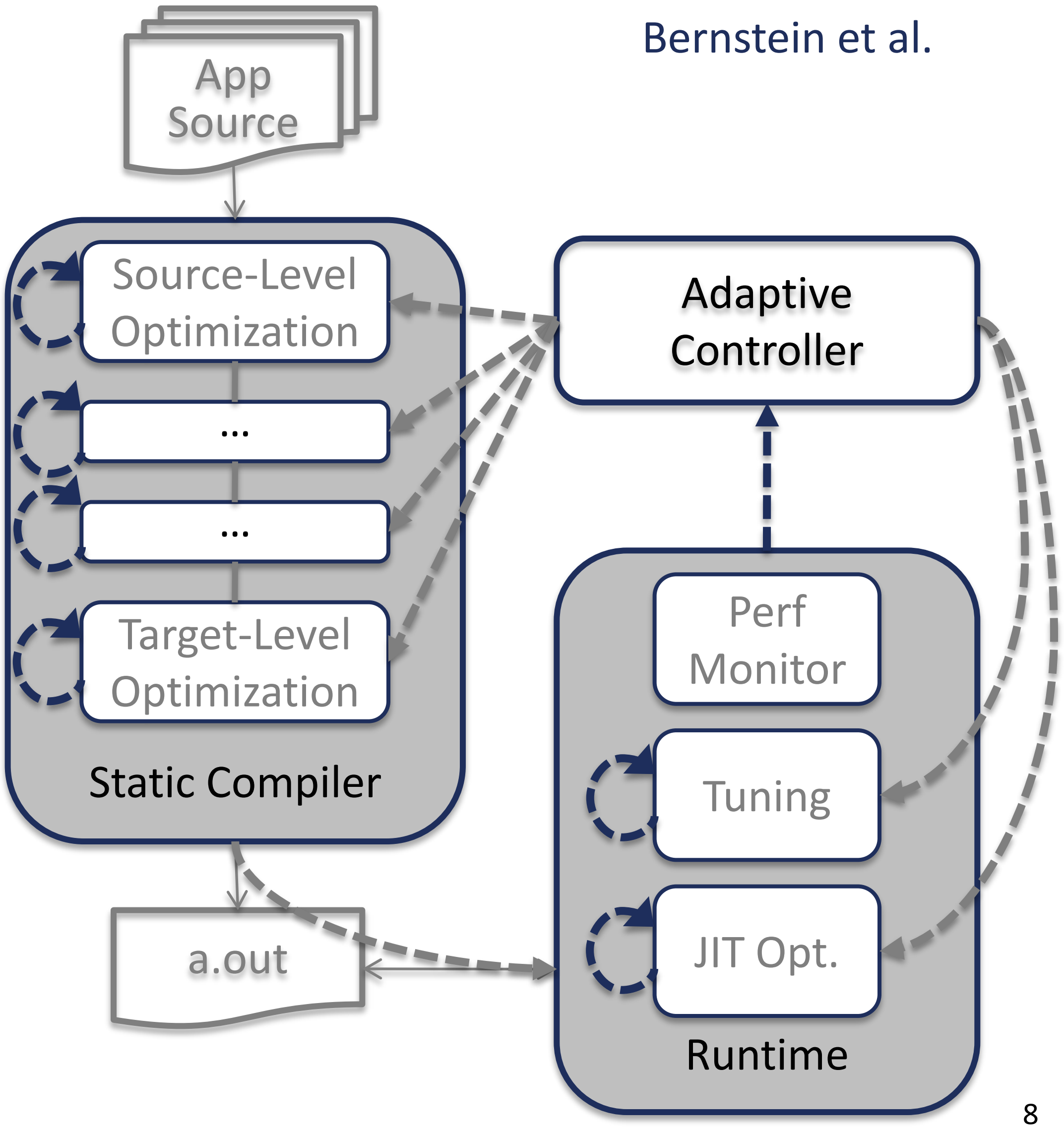


Context

Abstracting Away the Details

Minor adaptation cycles

- Repeated multiple times within a single major cycle
- Typical adaptations
  - Modify a heuristic
  - Modify some threshold
  - Try several approaches & keep best result



Behavior evolves rapidly; may repeat in major cycle

Context

Abstracting Away the Details

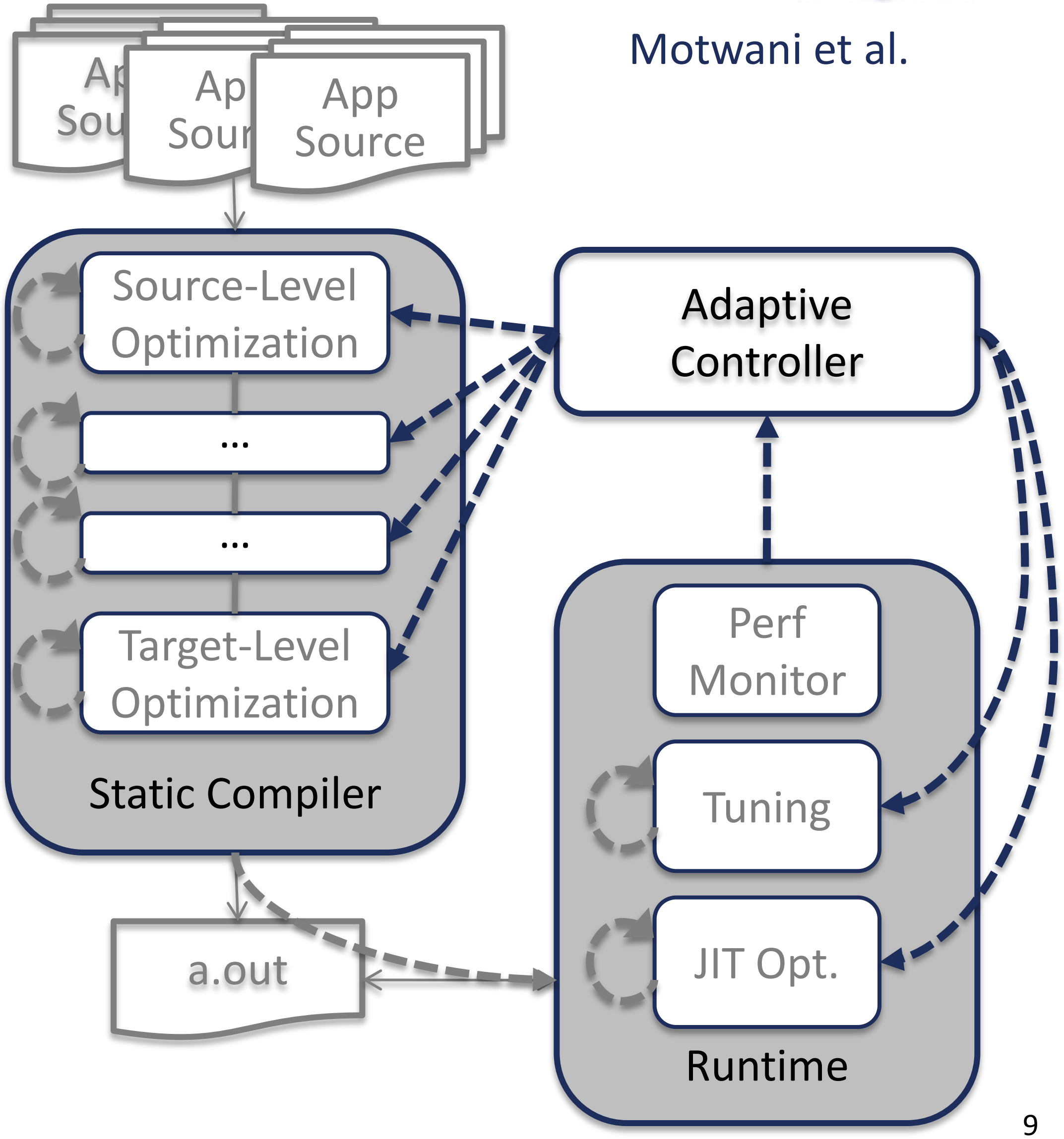
Motwani et al.

Long term learning

- Involves multiple runs & multiple applications
- Typical adaptations
  - Refine models of system performance & optimization effects
  - Recommend (prescribe) optimization strategies & policies
- Good subject for a talk; I am the wrong speaker

Slow evolution over many compile/execute cycles

Context



Defining Internal Adaptation

Focus on adaptation that changes the behavior of one transformation

Ignore, for now, techniques that treat the compiler or its components as black boxes

- Optimization choice might pick one of LCM, DVNT, GVN
  - External or structural adaption
  - Not a problem for today's talk
- Internal adaptation might parameterize the placement algorithm in LCM
  - Modify the definition of earliest placement with a tuning parameter
  - Search for the parameter value that produces the best code

Less work has been done on this kind of adaptation

Defining Internal Adaptation

External adaptation changes the way that the compiler runs

- Connect the components in different ways
  - → Schielke, Kulkarni derive good optimization sequences for specific code
- Manipulate pre-planned options
  - → Triantafyllis chooses from a set of preplanned optimization sequences
  - → Granston & Holler, Cavazos find a set of appropriate compiler flags

Internal adaptation changes the behavior of a single pass in the compiler

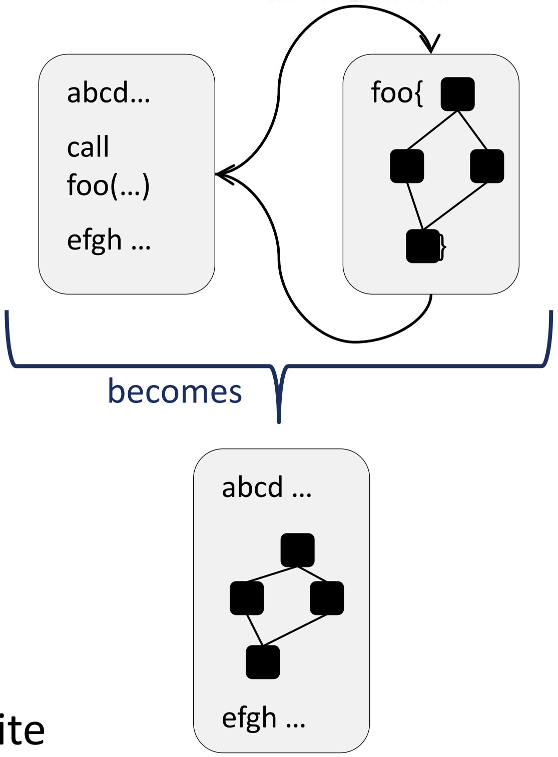
- Evolve a heuristic
  - → Motwani et al. derived new scheduling heuristics
- Choose among preplanned options
  - → Bernstein et al. choose best spill heuristic on a procedure-by-procedure basis
- Derive internal control parameters
  - → Liu's adaptive copy coalescing finds a threshold value to throttle coalescing

Let's examine a number of examples of internal adaptation and try to generalize from them.

Inline Substitution

Inline Substitution

- Simple transformation, complex decision problem
  - Choice at each call site, choices interact
  - Profitability depends on how the code optimizes
- Potential to improve or to degrade performance
  - Classic examples that produce integer factor speedups
  - Classic examples that produce significant slowdowns
- State of the art is one of two cases
  - Programmer makes the decision
  - Compiler applies complex heuristic formula at each call site



Textbook example of adaptation within a single optimization

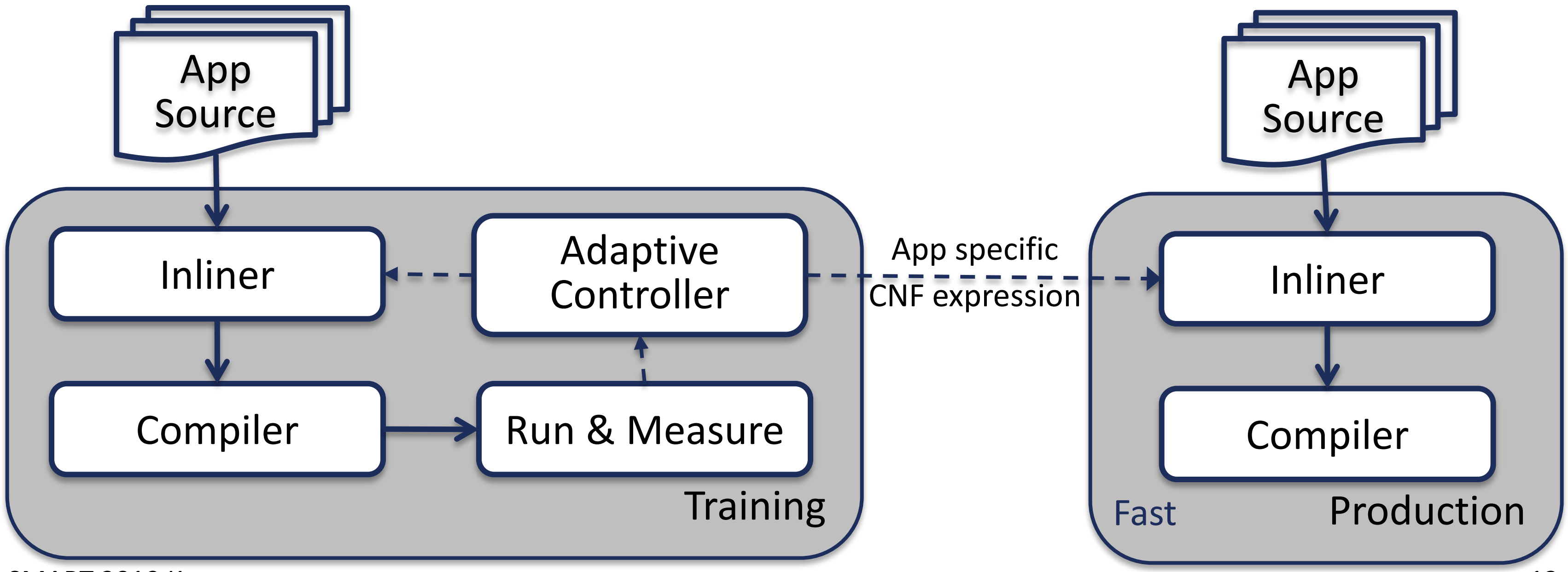
Weakness as an example is the slowness of evaluation

Examples

Inline Substitution

Waterman's solution

- Fast inline substitution pass controlled by a heuristic
- Feedback-driven system to derive program-specific heuristics
  - ◆ Used a classic compile-run feedback loop to search a huge parameter space
  - ◆ Derive a program-specific heuristic & then use it for subsequent compiles
- Consistently beat one-size fits all heuristics



Choosing Tile Sizes in the MIPS Compiler

The MIPS compiler did a poor job of choosing tile sizes on some codes

- Command-line parameter to override its choice
- Dasgupta showed that a simple binary search on tile size often beat MIPS
  - ◆ Tile size was used on all loops in the compilation unit (*poor parameterization*)

Similar Setup to Inline Substitution

- Need full compile and execute cycle to evaluate each choice
  - ◆ Treated compiler as a black box, so no other choice
  - ◆ Loop-by-loop tile sizes would be better, but would still need the major cycle
- Derived compilation-unit specific tile size

Examples

Lazy Code Motion *(negative example)*

LCM combines code motion & redundancy elimination

- Effective technique that improves most programs
- Complex transformation, but an easy decision procedure

Adaptive LCM?

- One downside of LCM is that it can increase register pressure
  - Introducing spills degrades performance
  - Might build an adaptive LCM moderated by register pressure $\left\{ \begin{array}{l} \text{Limit motion} \\ \text{Limit replacement} \end{array} \right.$
- Information on register pressure is not available when LCM runs
- Ratio of $\frac{\text{profit}}{\text{cost}}$ is too low to justify repeated compilations

Many other optimizations are equally unsuited to adaptation

⇒ Strength reduction, constant propagation, dead code elimination, …

Examples

Adaptive Register Coalescing

The Problem

- Eliminate all unneeded register-to-register copies (e.g, r_i□r_j)
- Copies occupy time and code space, also slow down compiler
- Simple optimization, simple decision procedure

Competing ideas about how to make the decision

In the context of a Chaitin-style graph-coloring register allocator:

- If r_i□r_j and r_i does not interfere with r_j, then r_i and r_j can be coalesced
- Combining them can increase register pressure (& lead to more spills)
- Popular technique is *conservative coalescing*
  - Only combine r_i and r_j if result has $< k$ neighbors of high degree
  - Conservative coalescing is safe, but leaves too many unneeded copies

Coalescing looks like a good candidate for internal adaptation

In allocation & coalescing, k is the number of registers. A node has "high degree" if it has $\geq k$ neighbors.

Examples

Adaptive Register Coalescing

Liu's Solution

- Perform conservative coalescing, but adaptively vary *k*
  - His allocator finds a *k* such that minimizes copies without adding spill code
  - Distinct *k* values for integer & floating-point registers
- Range of values for *k* is limited (*# registers ≤ k ≤ max degree in graph*)
- Feedback is quick, accurate, and *introspective*
  - The allocator measures its own behavior & uses that metric for feedback

Results

- Fewer copies, no more spills, relatively inexpensive at compile time
- On x86 in LLVM, best value of *k* is typically 5 or 10 more than # registers

Examples

Problems Similar to Adaptive Coalescing

Adaptive Coloring (for allocation)

- On x86, overlapping register classes make graph coloring allocators less effective
- Liu made Briggs adaptive by varying *k* in the coloring phase
  - Separate *k* for floating-point and integer (*as well as for coalescing*)
  - Introspective evaluation based on estimated spill cost
- Preliminary numbers suggest 15 to 20% speedup on x86 in LLVM

Software Pipelining (a la Lam)

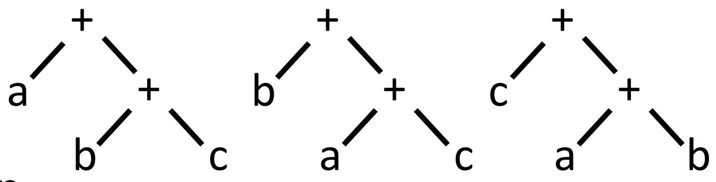
- Uses a simple parameter sweep to find the initiation interval
  - Bounds the space with lower-bound estimates
- Introspective measure of success – modulo scheduler fails or succeeds
- Widely used technique

Examples

Adaptive Reassociation *(speculation)*

The compiler can use commutativity, associativity, and distributivity to reorder expressions so that they optimize in different ways

- Expose different sets of expressions
- Optimizations need different orders
- Choosing a good order for an expression tree depends on the optimizations & the specific performance problems
- Daunting number of ways to rearrange an expression (*not yes/no decision*)



Properties & Strategy

- Needs major cycle to assess impact on other optimizations
- Can rely on introspective evaluation to decrease the cost

Examples

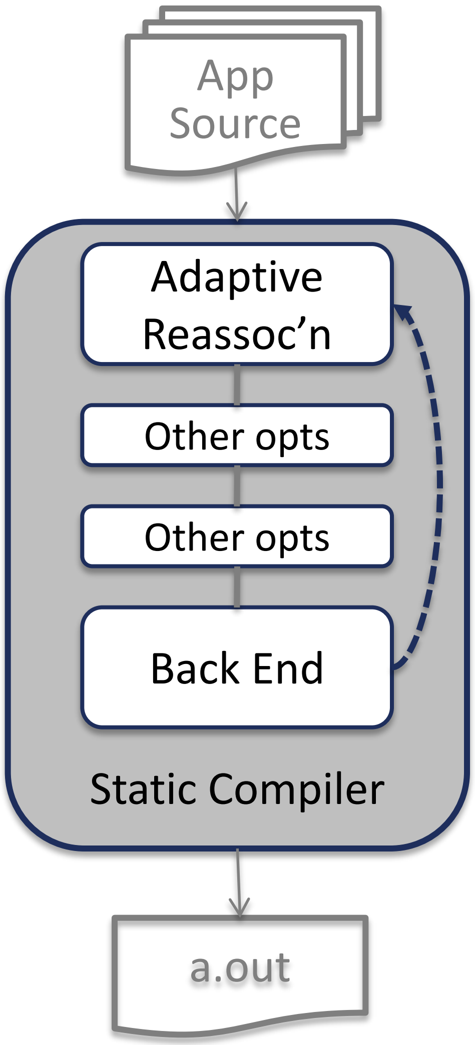
Adaptive Reassociation *(speculation)*

Strategy

- Perform reassociation early
  - For each expression, look @ context & transform
- Use a closed-form representation of heuristic & evolve parameters on loop-by-loop basis
- Measure code quality in the back end
  - Operation counts in loops, spills, schedule density

Goal

- Quick feedback cycle for rapid evolution
- Application-specific decision making



Examples

Summarizing the Examples

Optimization	Feedback Cycle	Feedback Metric	Costs / Benefits
Inline substitution	major	Execution time	High cost Moderate benefits
Choosing tile sizes	major	Execution time	High cost High benefits
Coalescing Copies	minor	Spill costs	Low cost Moderate benefit
Allocation	minor	Spill costs	Low cost Moderate benefit
Software Pipelining	minor	Success/Failure	Low cost Moderate benefit
Lazy Code Motion	major	Spill costs	Low cost Low benefit
Reassociation	major	Op counts, spills, schedule density	Moderate cost Large benefit

Summarizing the Examples

Optimization	Feedback Cycle	Feedback Metric	Costs / Benefits
Inline substitution	**major**	**Execution time**	**High cost** **Moderate benefits**
Choosing tile sizes	**major**	**Execution time**	**High cost** **High benefits**
Coalescing Copies	minor	Spill costs	Low cost Moderate benefit
Allocation			enefit
Software Pipelining	minor	Success/Failure	Moderate benefit
Lazy Code Motion	major	Spill costs	Low cost Low benefit
Reassociation	major	Op counts, spills, schedule density	Moderate cost Large benefit

Major feedback cycle using execution time metric
⇒ slow evolution of solution (1 step per compile)
⇒ clever heavy weight searches to minimize cost

Summarizing the Examples

Optimization	Feedback Cycle	Feedback Metric	Costs / Benefits
Inline substitution	major	Execution time	High cost Moderate benefits
Choosing tile sizes	major	Execution time	High cost High benefits
Coalescing Copies	**minor**	**Spill costs**	**Low cost** **Moderate benefit**
Allocation	**minor**	**Spill costs**	**Low cost** **Moderate benefit**
Software Pipelining	**minor**	**Success/Failure**	**Low cost** **Moderate benefit**
Lazy Code Motio			
Reassociation		schedule density	Large benefit

Minor feedback cycle using introspective metrics
⇒ rapid evolution of solution (many steps per compile)
⇒ simple lightweight searches, such as parameter sweeps

Summarizing the Examples

Optimization	Feedback Cycle	Feedback Metric	Costs / Benefits
Inline substitution	major	Execution time	High cost Moderate benefits
Choosing tile sizes	major	Execution time	High cost High benefits
Coalescing Copies	minor	Spill costs	Low cost Moderate benefit
Allocation			enefit
Software Pipelining	minor	Success/Failure	Low cost Moderate benefit
Lazy Code Motion	major	Spill costs	Low cost Low benefit
Reassociation	major	Op counts, spills, schedule density	Moderate cost Large benefit

Major feedback cycle using execution time metric
⇒ slow evolution of solution (1 step per compile)
⇒ cost / benefit ratio is too high

LCM does well enough on its own.

Summarizing the Examples

Optimization	Feedback Cycle	Feedback Metric	Costs / Benefits
Inline substitution	major	Execution time	High cost Moderate benefits
Choosing tile sizes	major	Execution time	High cost High benefits
Coalescing Copies	minor	Spill costs	Low cost enefit
Allocation			enefit
Software Pipelining	minor	Success/Failure	Low cost Moderate benefit
Lazy Code Motion	major	Spill costs	Low cost Low benefit
Reassociation	major	Op counts, spills, schedule density	Moderate cost Large benefit

Major feedback cycle using introspective metrics
⇒ choose loop-nest specific solutions
⇒ large expected benefit

Looking for Candidates for Internal Adaptation

For success, I think that we need to find optimizations that have:

- Complex, inter-related decision problems
  - If each decision is **O**(1), the problem may be too easy
  - If context plays a role in the good decision, that's a good sign
- Algorithm that can be parameterized in a clean fashion
  - Parameter should be tunable, preferably over a small range
  - We have ignored the issue of parameter choice for too long
- Simple, well-defined metrics
  - Must relate performance to the parameter space
  - Ideally, should be obtainable with introspection (*avoid execution to lower costs*)

Barriers to Adoption of Adaptive Techniques

Usability Issues

- Where does the system store all those annotations and history?
- How does the user specify her objective function? (*speed, space, energy, … ?*)

Mitigating and Explaining the Long Compile Times

- These techniques are expensive—especially major cycle examples
- Can we incrementalize the search? Work 3 or 4 examples each compile?
  - Again, where do we store the results & the experience?

Implementation complexity

- Try making all your optimizations run correctly in any arbitrary order
  - Good debugging technique for optimizations, bad plan for meeting deadlines
- Large & complex systems have issues with reproducibility & publication
- Implementation hassle outweighs benefits

Industrialization

Parameterization is Crucial

We have ignored the issue of how to parameterize optimizations

Lesson of the VCR

- Proliferation of command-line flags
  - Magic number 7 +/- 2 applies
  - Parameters are chosen to aid compiler writer, not compiler user
- Lack of effective control over behavior
  - What parameter can you set to throttle inline substitution for i-cache size?
  - Can you specify different tile sizes for different loop nests?
- Good parameterization makes adaptive control natural
  - Same properties that help an adaptive implementation may help the user
  - Want a clear relationship between parameter and the outcomes

This problem may be a "full professor problem".

STO
P

nal Points

Introspection Provides a Different Perspective

Execution time is a bad metric for adaptation

- Open ended scale with no way to declare victory
- Change in execution time may be obscured by unchangeable base number
  - → It is okay to pursue the small improvement while fixing the large problem
- Hard to differentiate a long-running code from a poorly compiled code

Introspection looks at what is wrong with the code

- Optimizer can only fix problems
  - ◆ Optimizers do not invent new algorithms or shrink problem sizes
- Introspective measures show the magnitude of the problems
  - ◆ Number of spills, slack cycles in the schedule, …

Introspection may provide a better metric for recognizing & declaring victory

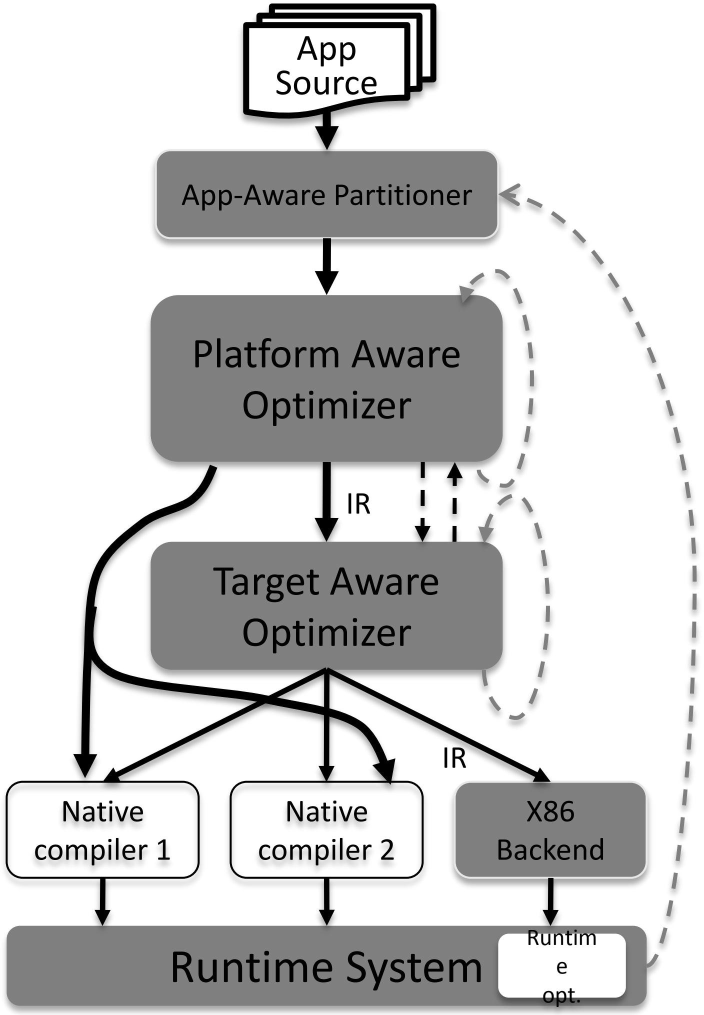
Final Points

Introspection in the PACE Compiler

The PACE Compiler carries introspection to an extreme

- Use instrumented version of LLVM to provide feedback to high-level opts
- Small query language to specify what should be measured
- LLVM compiles, measures, & reports

- PACE Runtime provides region-by-region performance metrics to guide optimization, as well
- Feedback cycles to include both introspection & execution



Cooper, Mellor-Crummey, Palem, Sarkar, & Torczon

References for Work Mentioned in the Talk

1. Bergner et al., Spill Code Minimization Via Interference Region Spilling, PLDI 97
2. Bernstein et al., Spill Code Minimization Techniques for Optimizing Compilers, PLDI 89
3. Cavazos et al., "Rapidly Selecting Good Compiler Optimizations using Performance Counters", CGO 07
4. Eckhardt et al., "Redundancy Elimination Revisited", PACT 08, October 2008. (*for rematerialization*)
5. Granston & Holler, "Automatic Recommendation of Compiler Options," 4th International Workshop on Feedback and Data-Driven Optimization, 2001.
6. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines", PLDI 98
7. Liu, "Parameterization and Adaptive Search for Graph Coloring Register Allocation", MS Thesis, Rice University, 2009.
8. Kulkarni et al., "Fast and efficient searches for optimization-phase sequences", ACM TACO, June 2005.
9. Motwani et al., Combining Register Allocation And Instruction Scheduling, NYU Courant Institute TR 689, July 19955
10. Schielke et al., "Optimizing for Reduced Code Space with Genetic Algorithms", LCTES 99
11. Simpson, "Live Range Splitting in a Graph Coloring Register Allocator", CC 1998
12. Stephenson, et al., "Meta-optimization: improving compiler heuristics with machine learning," PLDI 03
13. Triantafyllis et al. "Compiler Optimization Space Exploration", CGO 03
14. Waterman et al., "An Adaptive Strategy for Inline Substitution", CC 2008
15. Waterman et al. "Investigating Adaptive Compilation with the MIPSPro Compiler," LACSI Symposium, *Journal of High Performance Computing Applications* 19(4), 2005.

These are minimalist citations, but they should provide enough information to find the paper in a good search engine.