COMP 512
Rice University
Spring 2015

Lessons from Fifteen Years of Adaptive Compilation

Keith Cooper, Tim Harvey, Devika Subramanian, and Linda Torczon, with Phil Schielke, Alex Grossman, Todd Waterman, and others

Funding from DOE, Microsoft, TI, and DARPA.

This lecture differs from the others given this semester in that it is explicitly a history of the work done at Rice between 1995 and 2010.

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 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

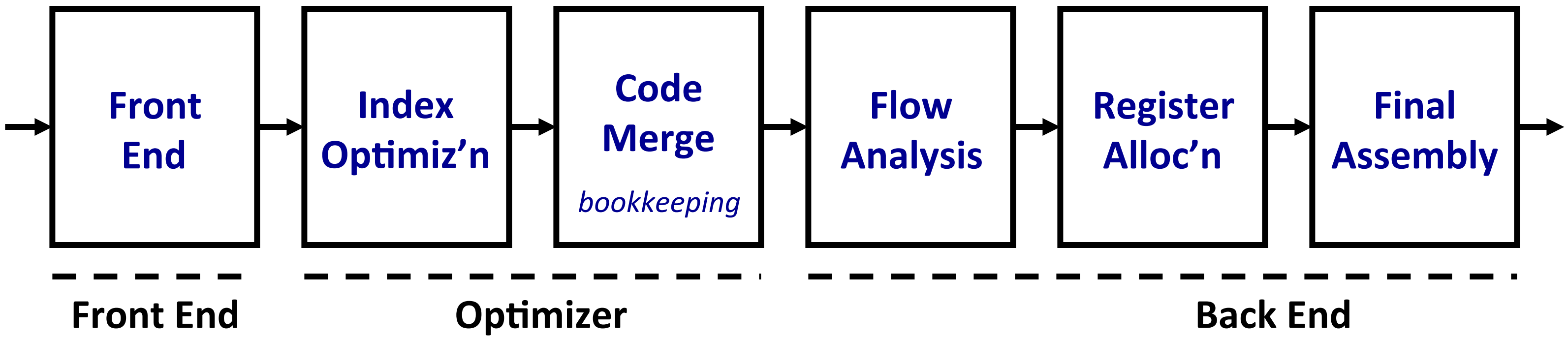
  $$\frac{\text{Retired ops}}{\text{second}}$$

  - ♦ 5 to 10% is 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

History

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

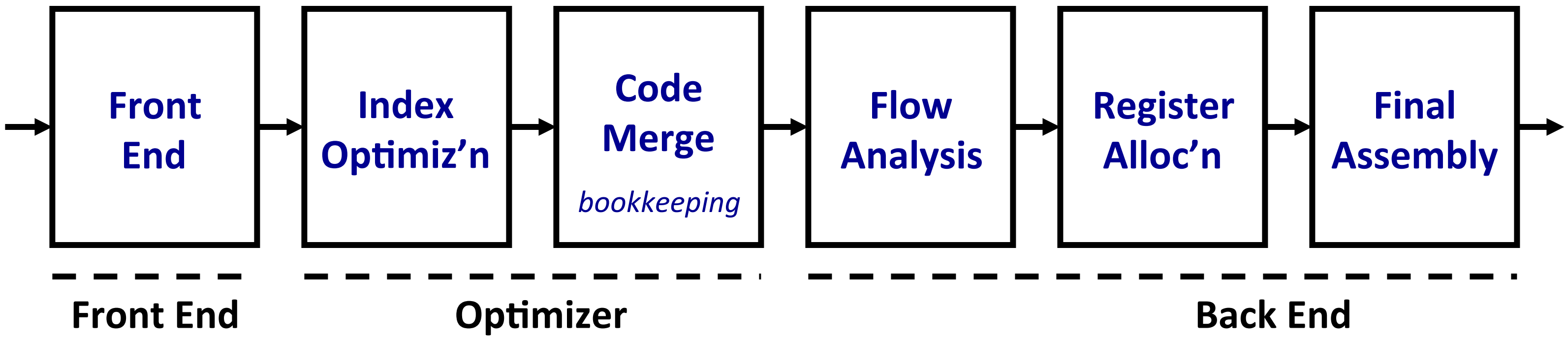


Fortran Automatic Coding System, IBM, 1957

- Compiler writers chose strategy when they designed the compiler
- Better compilers offered compile-time flags to modify behavior
- State of the art, 1957 to 1989

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?

> Modern compilers have the same basic structure …

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)

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

Roadmap

- Problems we have attacked
- Search space characterization
- Search algorithms
- Parameterization is important
- Lessons we have learned
- Future work

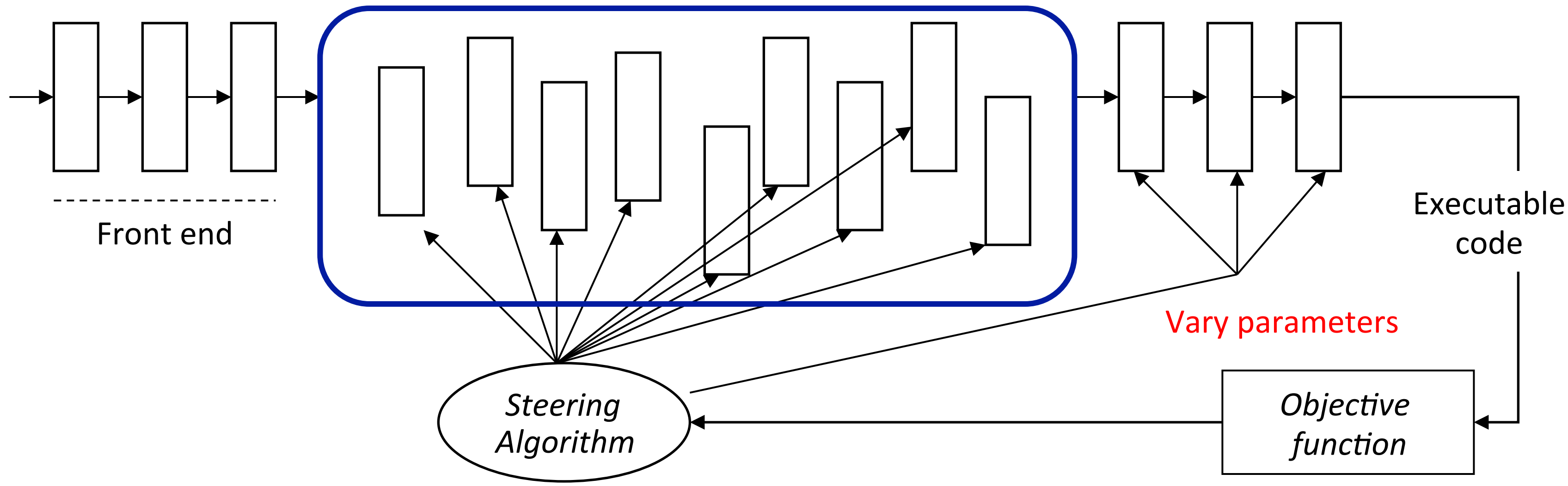
Some Sample Adaptive Compilation Problems

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 length 10 (1,099,511,627,776 strings)
  - ♦ Enumerated space of 5 opts, strings of length 10 (9,765,625 strings)
  - ♦ Compiled and ran some small codes with each sequence
- Use properties we discover to derive effective searches
- Validate the characterization by running the new search algorithms in the full space

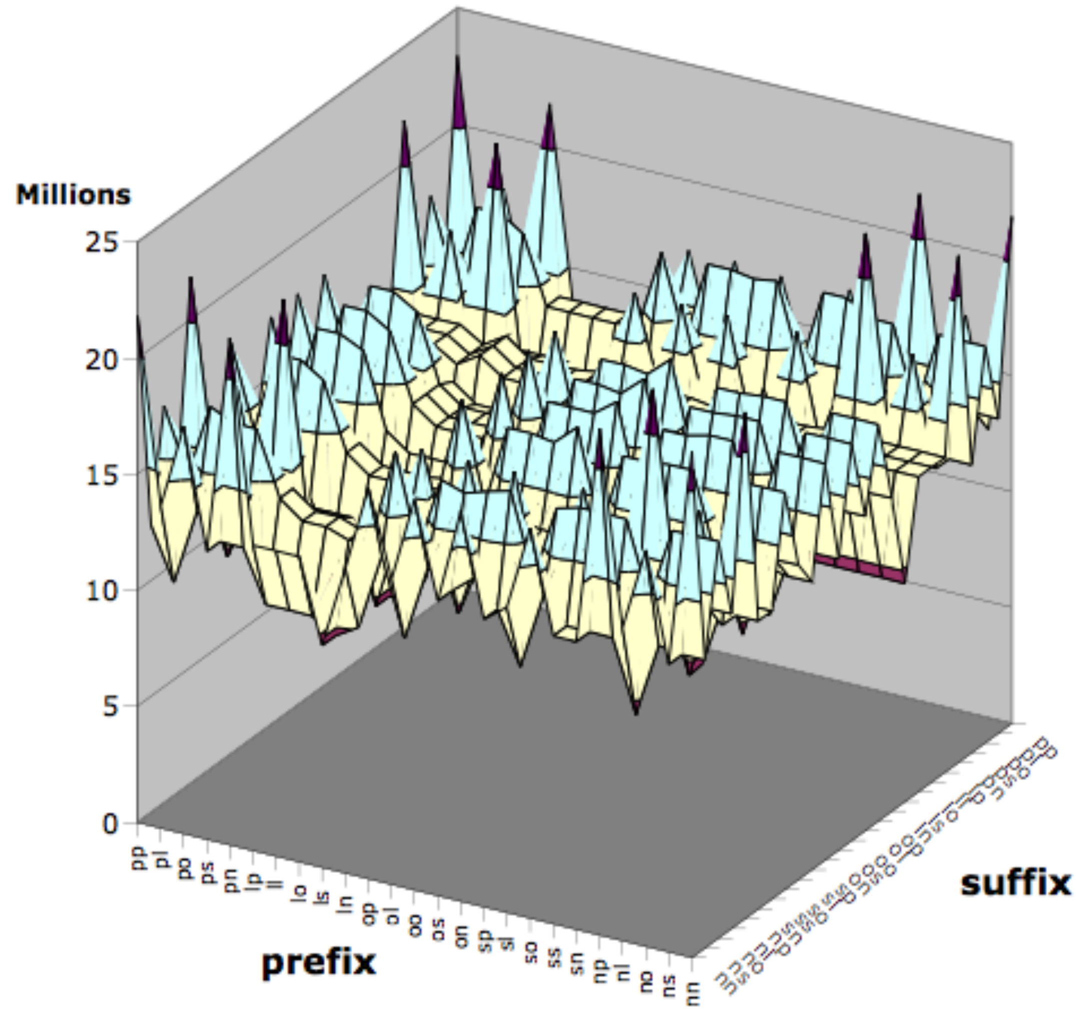
What Have We Learned About Search Spaces?

We confirmed some obvious points

These spaces are:

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

adpcm-coder, 5^4 space, plosn



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

What Have We Learned About Search Spaces?

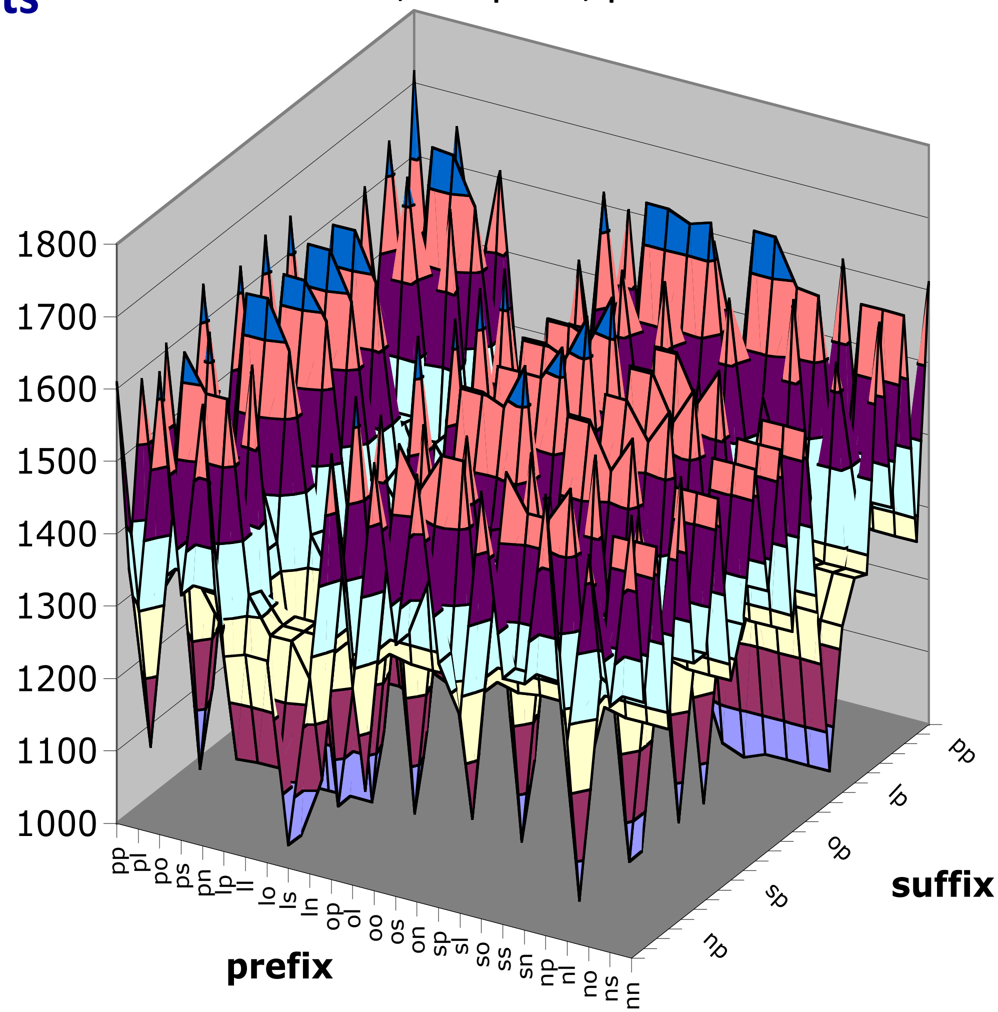
We confirmed some obvious points

These spaces are:

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

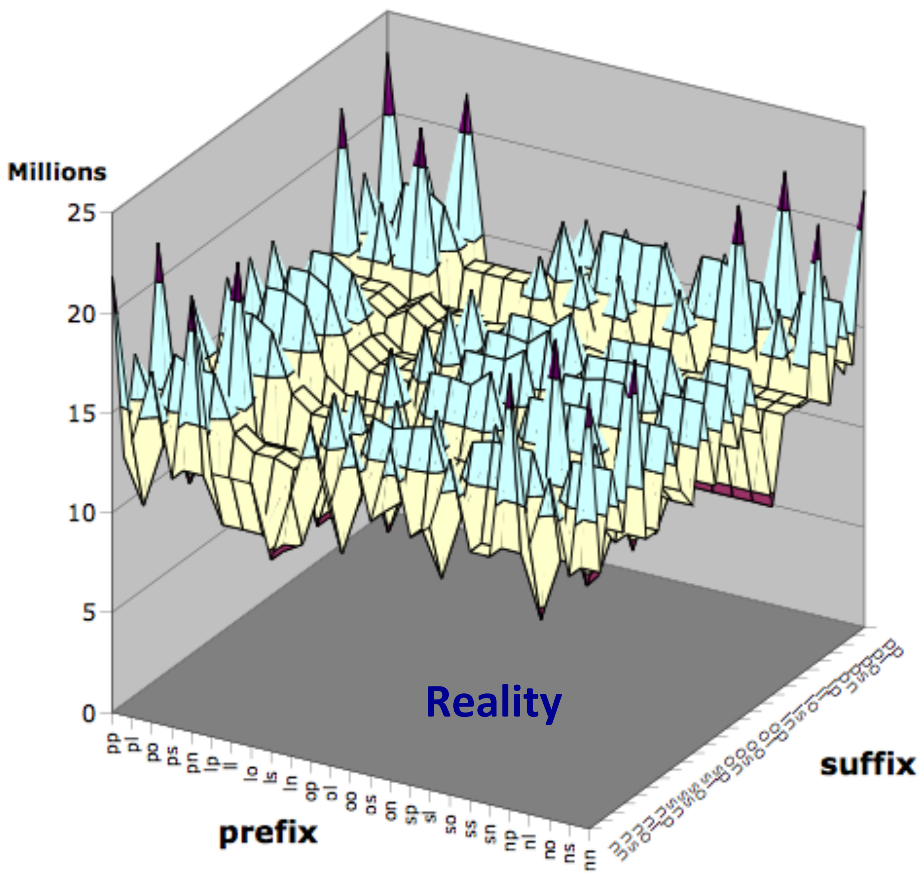
<u>p</u>: peeling
<u>l</u>: PRE
<u>o</u>: logical peephole
<u>s</u>: reg. coalescing
<u>n</u>: useless CF elimination

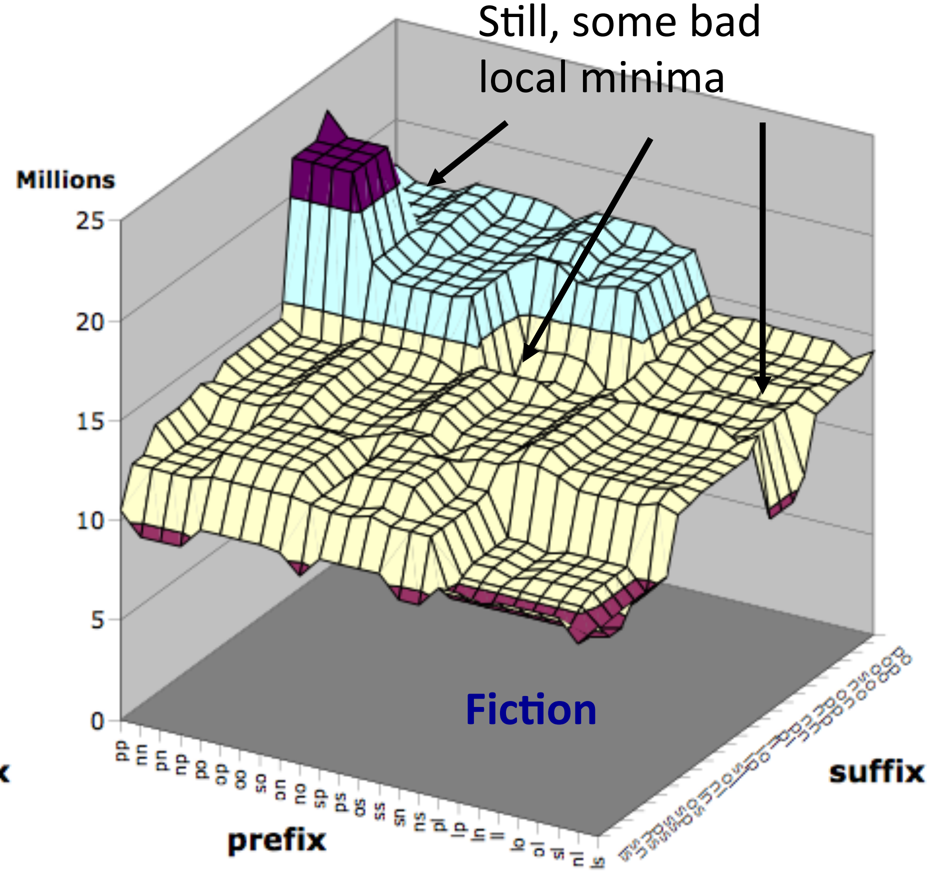
fmin, 5^4 space, plosn



What About Presentation Order?

Clearly, order might affect the picture …

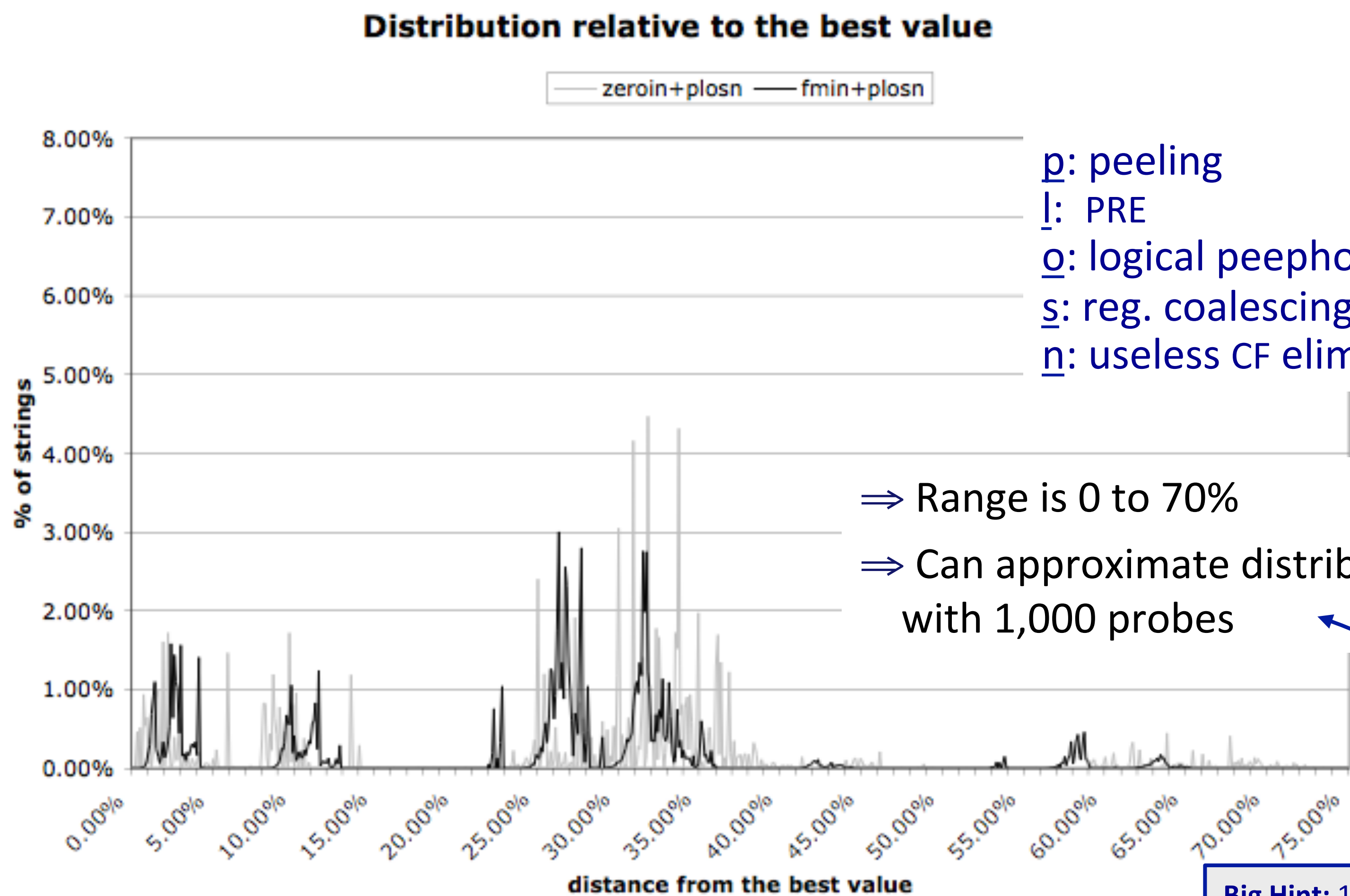




adpcm-coder, 5^4 space, plosn *

Both Programs & Optimizations Shape the Space

Two programs, same set of optimizations



<u>p</u>: peeling
<u>l</u>: PRE
<u>o</u>: logical peephole
<u>s</u>: reg. coalescing
<u>n</u>: useless CF elimination

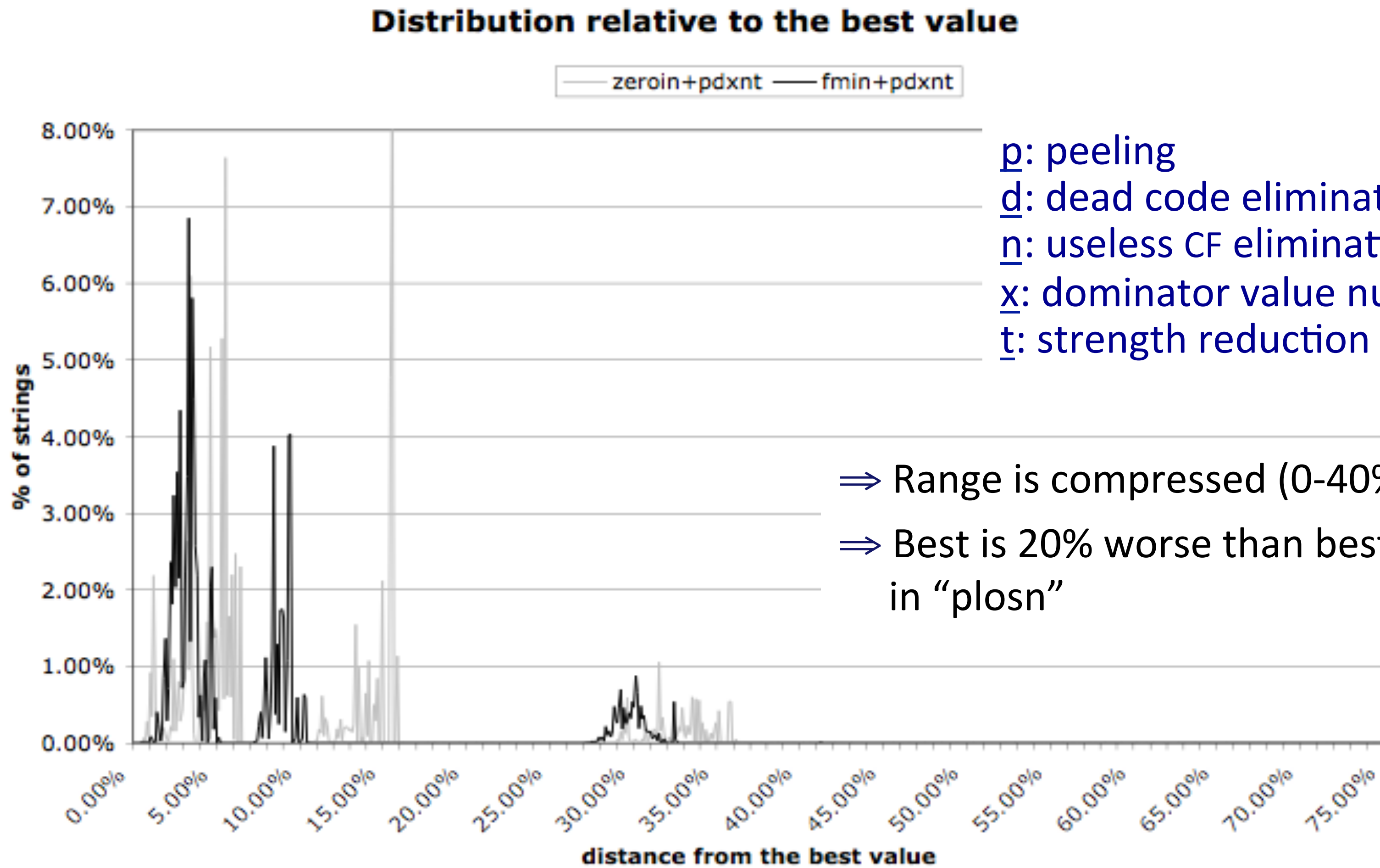
⇒ Range is 0 to 70%

⇒ Can approximate distribution with 1,000 probes

Big Hint: 1,000 probes should find a good solution

Both Programs & Optimizations Shape the Space

Same two programs, another set of optimizations



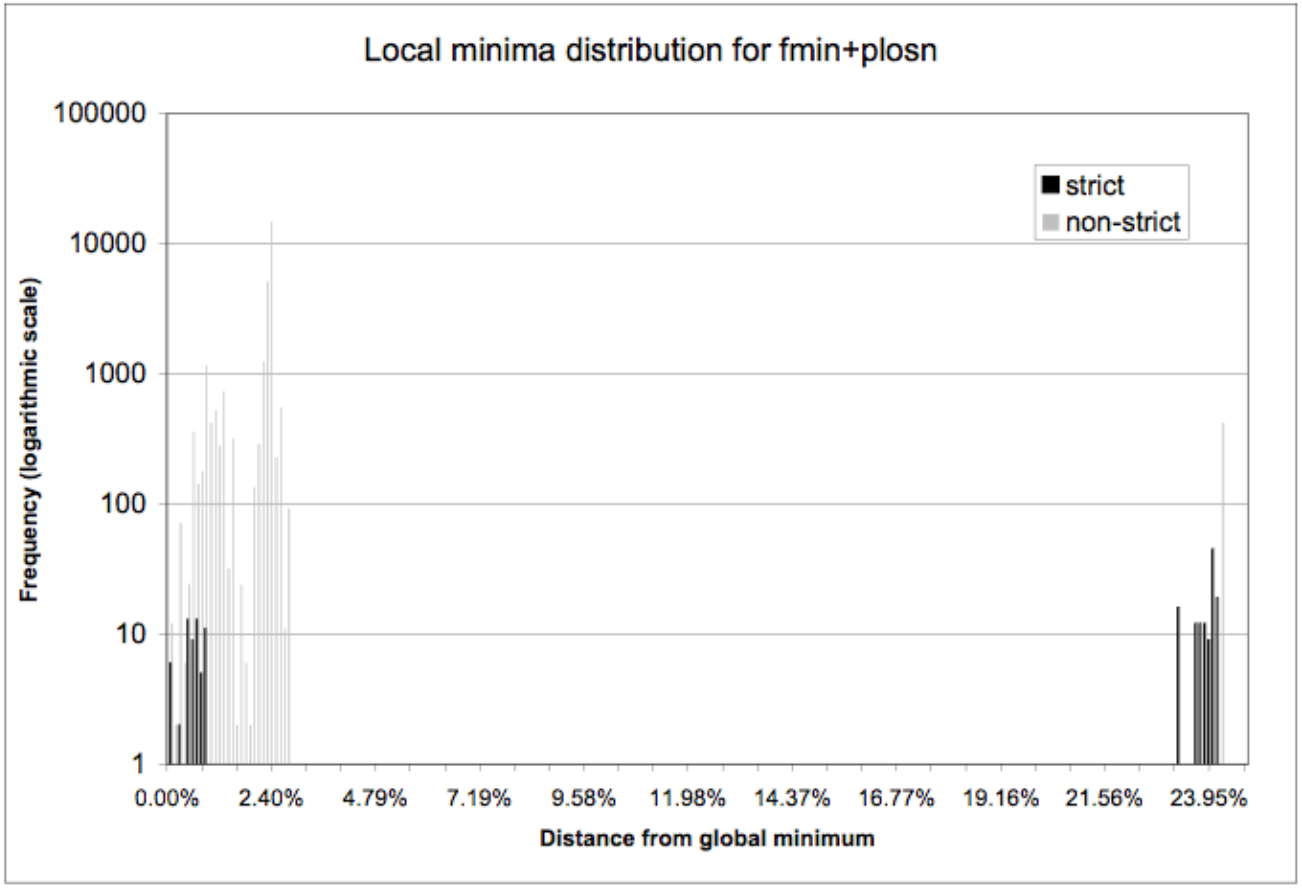
p: peeling
d: dead code elimination
n: useless CF elimination
x: dominator value num'g
t: strength reduction

⇒ 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

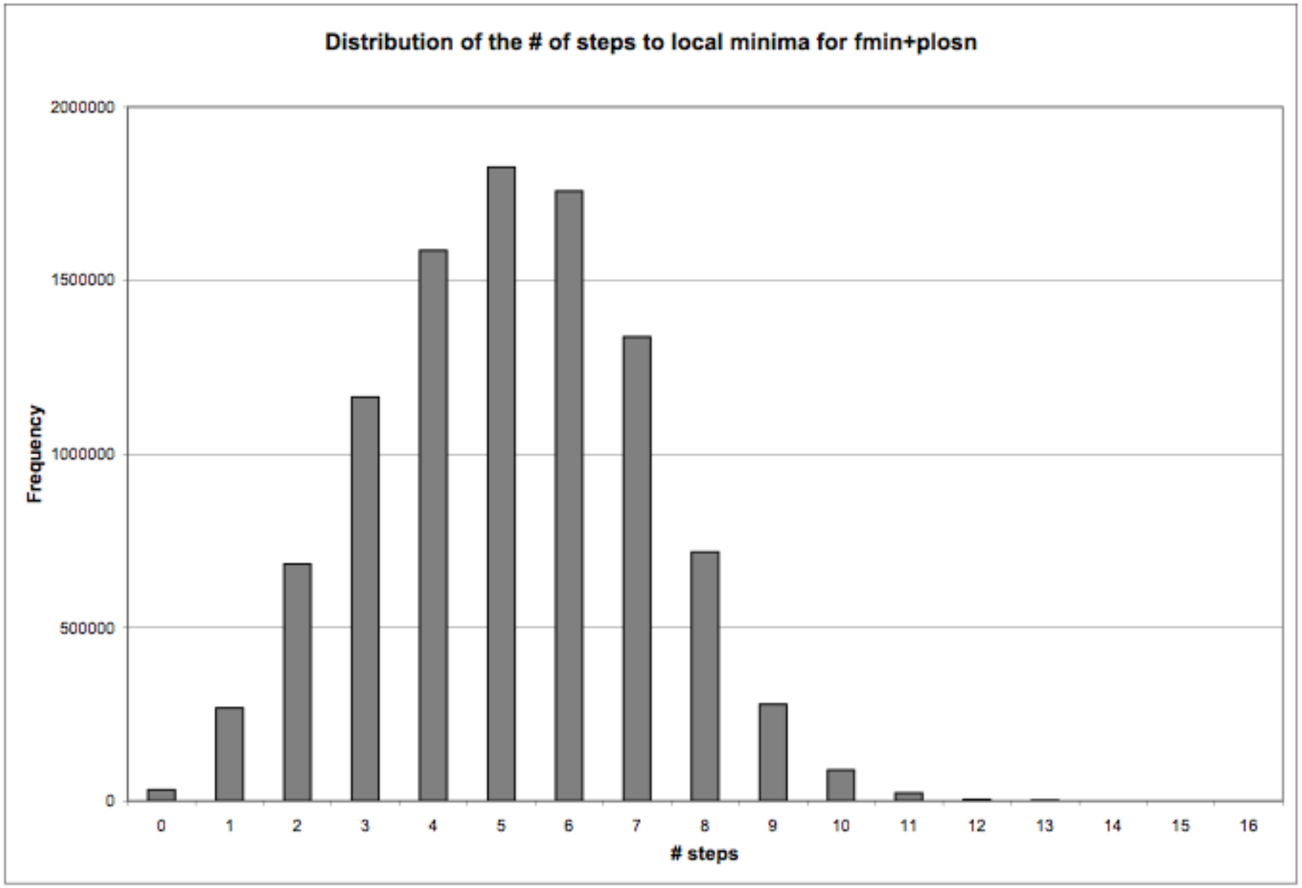
27,315 non-strict

(of 9,765,625)

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

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*

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**, better **GA**s, greedy algorithms, **GNE**
- Validate 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

10% for space
8% for bit transitions

Roadmap

- Problems we have attacked
- Search space characterization
- Search algorithms
- Parameterization is important
- Lessons we have learned
- Future work

Search Algorithms: Genetic Algorithms

Original work used a genetic algorithm (GA)

- Experimented with many variations on GA
- Favorite was 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

Makes it a search, rather than a simulation of evolution

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 …

Original GA ran 20,000 evaluations.

Search Algorithms: Hill climbers

Many nearby local minima suggests descent algorithm

- Neighbor ⇒ Hamming-1 string (*differs in 1 position*)
- Evaluate neighbors and move downhill
- Repeat from multiple starting points

- Steepest descent ⇒ take best neighbor
- Random descent ⇒ take 1st downhill neighbor (*break ties randomly*)
- Impatient descent ⇒ 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 1^{st} element
3. for i = 2 to k
   try each pass as prefix and as suffix
   keep the best result

95 evaluations for 10-of-5 space

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

Successive evaluations refine the string

1st pass		2nd pass		3rd pass		
		sp		snp		
		ps		psn		
p		sl		snl		
l		ls		lsn		
o	s	so	sn	sno	snl	…
s		os		osn		
n	winner	ss	winner	sns	winner	
		sn		snn		
		ns		nsn		

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

adpcm-d	GC-exh	GC-bre	GC-50
Sequences checked	91,633	325	2,200
Code speed	1.0	+ 0.003%	+ 2%

- 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 $<p_1,p_2,p_3,\ldots,p_n>$, examines neighborhood of each p_i
    - → Systematically looks at $<p_1 \pm s, p_2 \pm s, p_3 \pm s,\ldots, p_n \pm s>$
    - → 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
- Parameterization is important
- Lessons we have learned
- Future work

Inline Substitution

The transformation is easy

- Rewrite the call site with the callee's body
- Rewrite formal parameter names with actual parameter names

Safety

- As long as the IR can express the result, it should be safe
- Semantics does not address the number of copies of a procedure in the executable code

Profitability

- The obvious profit comes from eliminating call overhead
- The complications arise from changes in how the code optimizes

Opportunity

- Most implementations traverse the (partial) call graph & look at each edge

Inline Substitution

The transformation is easy

- Rewrite the call site with the callee's body
- Rewrite formal parameter names with actual parameter names

The decision procedure is quite hard

- At a given call site, profitability depends on the extent to which the callee can be tailored to the specific context
  - ♦ Performance can improve or degrade
- Resource constraints limit the amount of inlining
  - ♦ Experience suggests register demand is important
  - ♦ Code size (whole program & current procedure) play a role
    - → Excessive code growth leads to excessive compilation time
- Each decision affects profitability & resource use of other call sites

Inline Substitution

Choosing which call sites to inline is hard

- Performance of transformed code is hard to predict an edge $E_i(p, q)$ (*i.e.* a call site in function p which calls function q in the call graph).[1]

$$temperature_{E_i(p,q)} = \frac{cycle\_ratio_{E_i(p,q)}}{size\_ratio_q} \quad (1)$$

where:

$$cycle\_ratio_{E_i(p,q)} = \frac{freq_{E_i(p,q)}}{freq_q} \times \frac{cycle\_count_q}{Total\_cycle\_count} \quad (2)$$

$freq_{E_i(p,q)}$ is the frequency of the edge $E_i(p, q)$ and $freq_q$ is the overall execution frequency of function q in the training execution.

$Total\_cycle\_count$ is the estimated total execution time of the application:

$$Total\_cycle\_count = \sum_{k \in PUset} cycle\_count_k \quad (3)$$

$PUset$ is the set of all program units (*i.e.* functions) in the program, $cycle\_count_q$ is the estimated number of cycles spent on function q.

$$cycle\_count_q = \sum_{i \in stmts_q} freq_i \quad (4)$$

where $stmts_q$ is the set of all statements of function q, $freq_i$ is the frequency of execution of statement i in the training run.

Furthermore, the overall frequency of execution of the callee q is computed by:

$$freq_q = \sum_{k \in callers_q} freq_{E_i(k,q)} \quad (5)$$

where $callers_q$ is the set of all functions that contain a call to q.

Essentially, $cycle\_ratio$ is the contribution of a call graph edge to the execution time of the whole application. A function's cycle count is the execution time spent in that function, including all its invocations. ($\frac{freq_{E_i(p,q)}}{freq_q} *$ $cycle\_count_q$) is the number of cycles contributed by the callee q invoked by the edge $E_i(p, q)$. Thus, $cycle\_ratio_{E_i(p,q)}$ is the contribution of the cycles resulting from the call site $E_i(p, q)$ to the application's total cycle count. The larger the $cycle\_ratio_{E_i(p,q)}$ is, the more important the call graph edge.

$$size\_ratio_q = \frac{size_q}{Total\_application\_size} \quad (6)$$

$Total\_application\_size$ is the estimated size of the application. It is the sum of the estimated sizes of all the functions in the application. $size_q$, the estimated size of the function q, is computed by:

[1] Because function p may call q at different call sites, the pair (p, q) does not define an unique call site. Thus, we add the subscript i to uniquely identify the i^{th} call site from p to q.

From "To Inline or Not to Inline? Enhanced Inlining Decisions" by Zhao & Amaral

Compute a "temperature" for each call site

- Complicated computation
- Single number to characterize each site
- Inline sites that are hotter than some threshold
- Tuning implies choosing the threshold

Explanation actually goes on for another half page

Inline Substitution

Choosing which call sites to inline is hard

- Performance of transformed code is hard to predict
- Decisions interact
  - ♦ Inlining A into B changes B's properties
  - ♦ Inlining A into B might make B a leaf
- Can't even name the call sites
  - ♦ Inlining destroys some & creates others
- Some decisions look easy, others look hard
  - ♦ Inline procedure smaller than linkage or called from one place
  - ♦ Don't inline large procedure or calls in critical loops

Existing compilers use heuristics, such as ORC's temperature

Inline Substitution

Benefits and Costs

- Inline substitution cures many of the inefficiencies that can arise at a call site
  - ♦ Eliminates overhead
  - ♦ Allows context-specific tailoring
  - ♦ Eliminates disruption to analysis in both caller and callee
- Inline substitution can cause its own problems
  - ♦ Unlimited compilation times (*ignoring the MIPS story*)
  - ♦ Performance degradation
  - ♦ Significant code growth
- There are other consequences of inline substitution …

Decision Procedures

Of course, the hard part is deciding what to do …

- Decision for one call affects behavior at other sites
- Difficult to predict effects
  - ♦ Demand for registers can cause increased spilling
  - ♦ Inlined code can have much larger name space (analysis)
  - ♦ Quality of global optimization may fall with procedure size
- MIPSPro computes a quantitative score
  - ♦ Gives a yes or no answer based on potential and size
- Some decisions are obvious
  - ♦ Inline small procedures (< linkage size)
  - ♦ Inline procedures called only once (leaf procedures)
- Still room for experimental work
  - ♦ See Cooper, Hall, & Torczon or Davidson & Holler or McKusick

See Waterman 2006

Inline Substitution

So, how should we determine a good inline decision heuristic?

- Waterman proposed an adaptive approach
  - ♦ His system constructs a program-specific heuristic
  - ♦ Run once to find heuristic; use heuristic every time

Prior art

- Ad hoc heuristics based on program properties
  - ♦ Inline leaf procedures of less than *k* lines
  - ♦ Inline by call frequency until code grows by *x* percent
  - ♦ Inline calls with more than one constant parameter
- Combine ad hoc heuristics into a single test applied at each call site – applied in a fixed order based on original call graph

Adaptive Inline Substitution

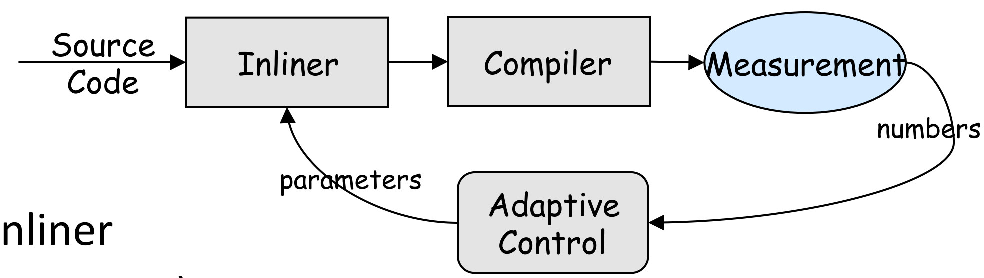
Waterman 2006

Inline substitution is a natural application for adaptive behavior

- Built a demonstration system for **ANSI C** programs
  - ♦ Analyzes whole program and collects data on program properties
    - → Nesting depth, code size, constants at call, call frequency, etc.
    - → Experimented with 12 properties in Waterman's thesis
  - ♦ Apply tunable heuristic at each call site
    - → Compare actual values against parameter values
    - → Use search to select best parameter values

Order based on original call graph

  - ♦ Produce transformed source
  - ♦ Compile, run, evaluate



  - ♦ Improvements of 20% over static inliner and 30% over original (PowerPC & Pentium)
  - ♦ Heuristics vary by application and by target architecture

Adaptive Inline Substitution

Waterman 2006

Key design issues

- Finding a good way to parameterize the problem & the software
  - ♦ Takes a "*condition string*" in **CNF** where each clause is a program property and a constant, e.g.,

    inliner -c "sc < 25 | lnd > 0, sc < 100" foo.c

  - ♦ Search produces a condition string that can be used repeatedly
- Search space is huge
  - ♦ Range of values depends on input program
    - → Estimate the range & discretize it into 20 intervals
  - ♦ Condition string syntax admits too many choices
  - ♦ Designed a single format for condition strings in our experiments

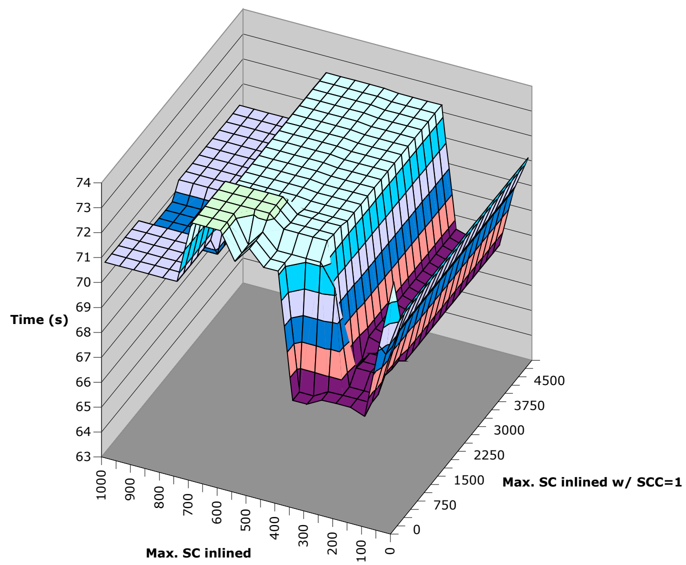
    sc < A | sc < B, lnd > 0 | sc < C, scc = 1 |
    clc < D | cpc > E, sc < F | dcc > G

    Fixes the search space's "shape"

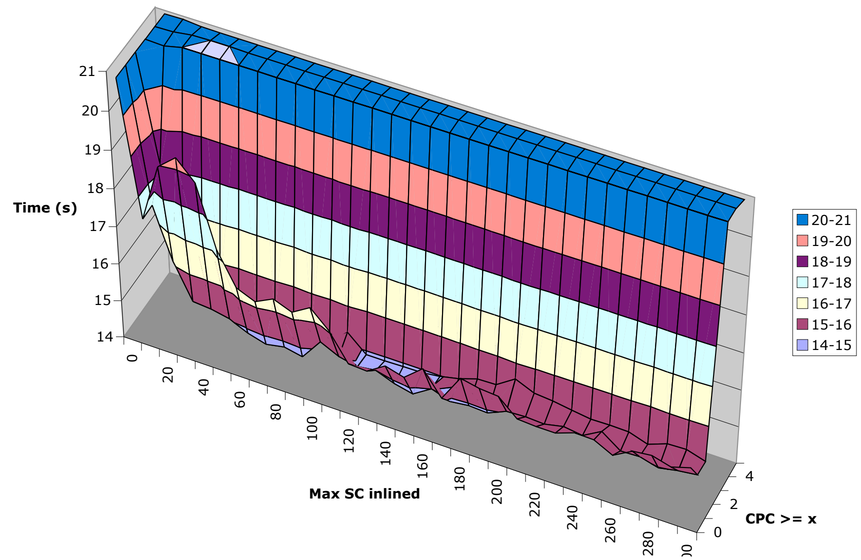
Adaptive Inline Substitution

Waterman 2006

Search spaces are much smoother than in sequence finding problem



bzip, varying sc and sc for single-call procedures



vortex, varying sc and constants per call

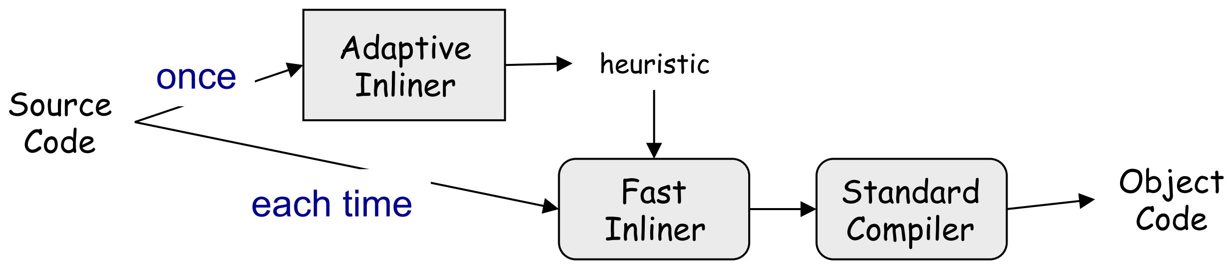
- Designed search techniques for these spaces
  - ♦ Impatient hill-climber and random restart
- And validated them experimentally

Adaptive Inline Substitution

Waterman 2006

How might we deploy these results?

- Source-to-source inliner
  - ♦ Runs for a while and produces a **CNF** expression that describes a program-specific heuristic
  - ♦ Use the inliner on subsequent compilations with that heuristic
    - → If code properties change "enough", re-run the search



- Tools
  - ♦ Current implementation is an ad hoc **C** program
  - ♦ Should reimplement it in Rose or something similar

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 adaptive compilation practical, must combine lots of ideas

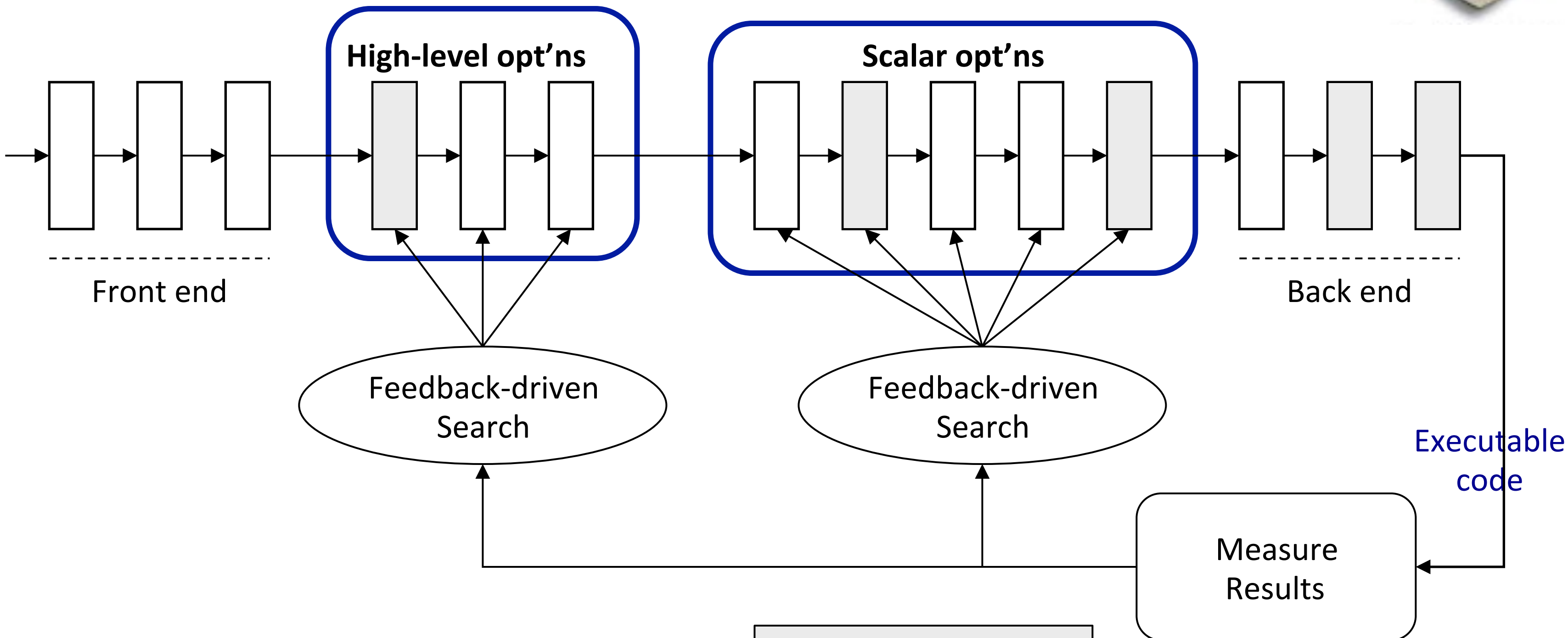
- Evaluation is expensive, so avoid it
  - ♦ Hash search points to avoid re-evaluation
  - ♦ Recognize identical results (same code, different point)
  - ♦ In many cases, simulated execution is good enough
    - → Fall-back position when update fails? Run the code !
- Performance measures should be:
  - ♦ 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?

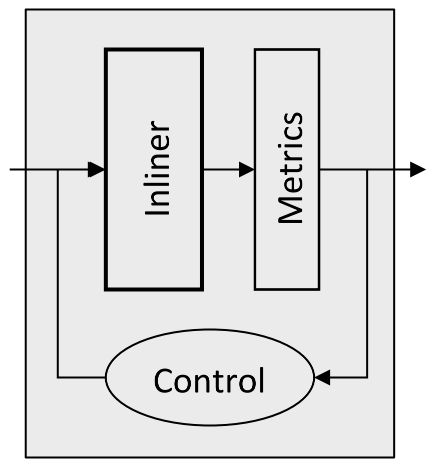
Selecting optimizations where internal adaptation pays off

- 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

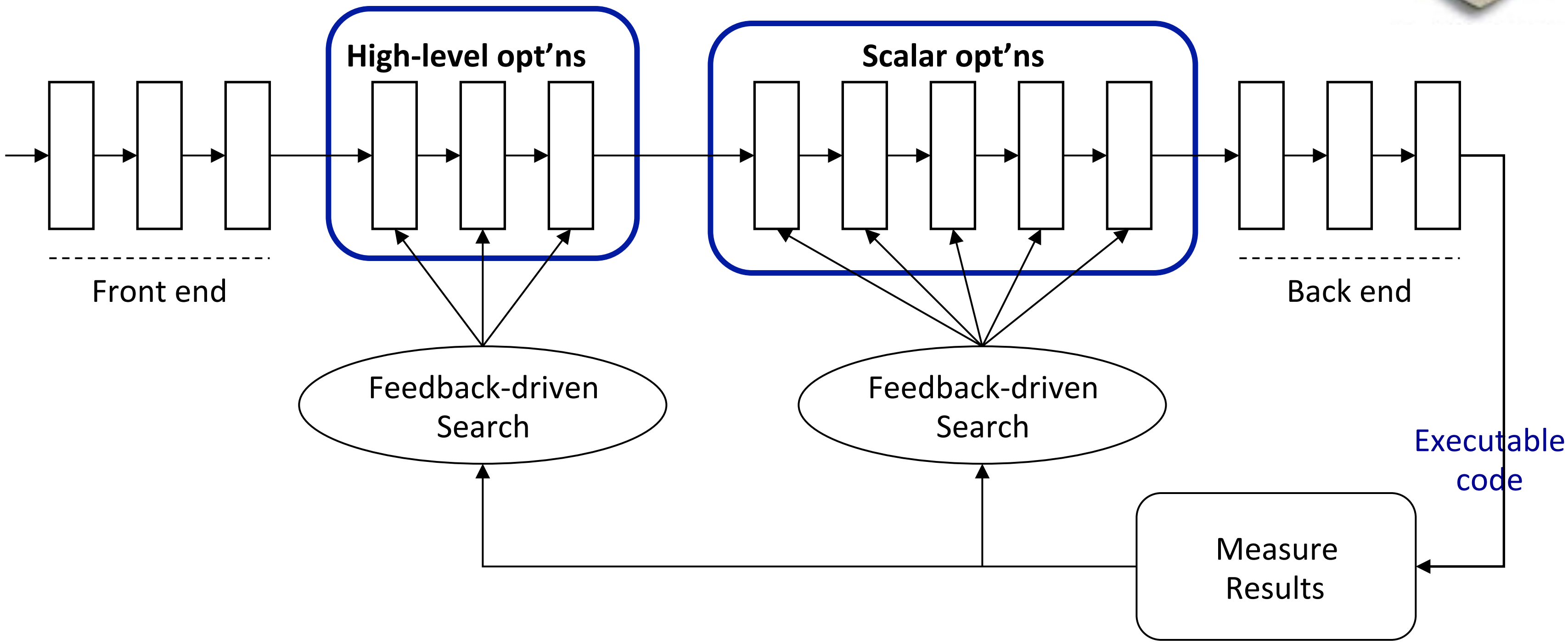
Future Compiler Structure



- **Some of these passes have their own feedback-driven adaptive controls**
  - **— memory hierarchy opts, inliner, allocator, scheduler**



Future Compiler Structure

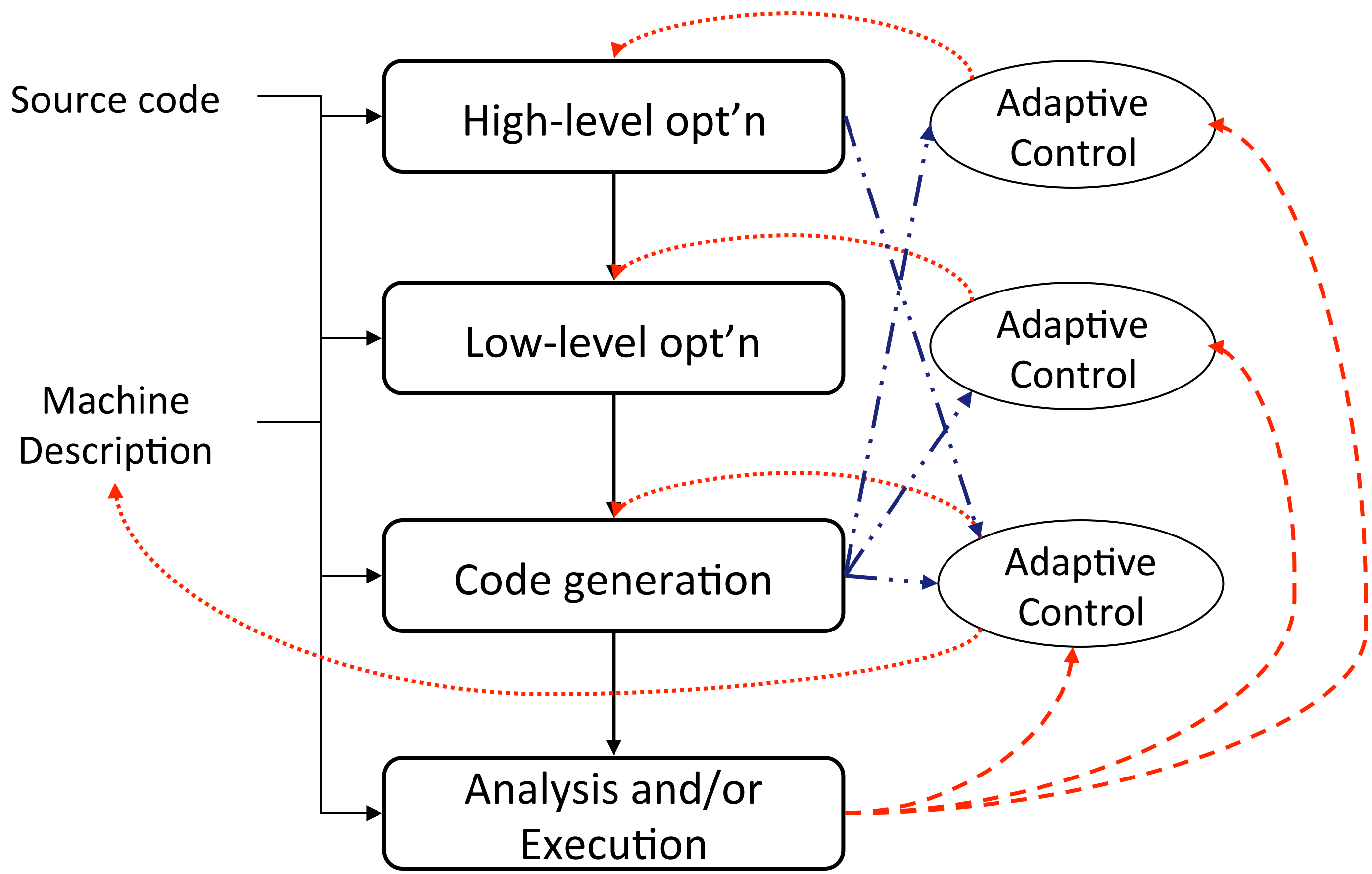


The result is a compiler that uses (& manages) multiple levels of feedback-driven adaptation

— **From this structure, we have the platform to expand into managing other parameters that affect performance**

Multi-level Feedback-driven Adaptation

The PACE Compiler

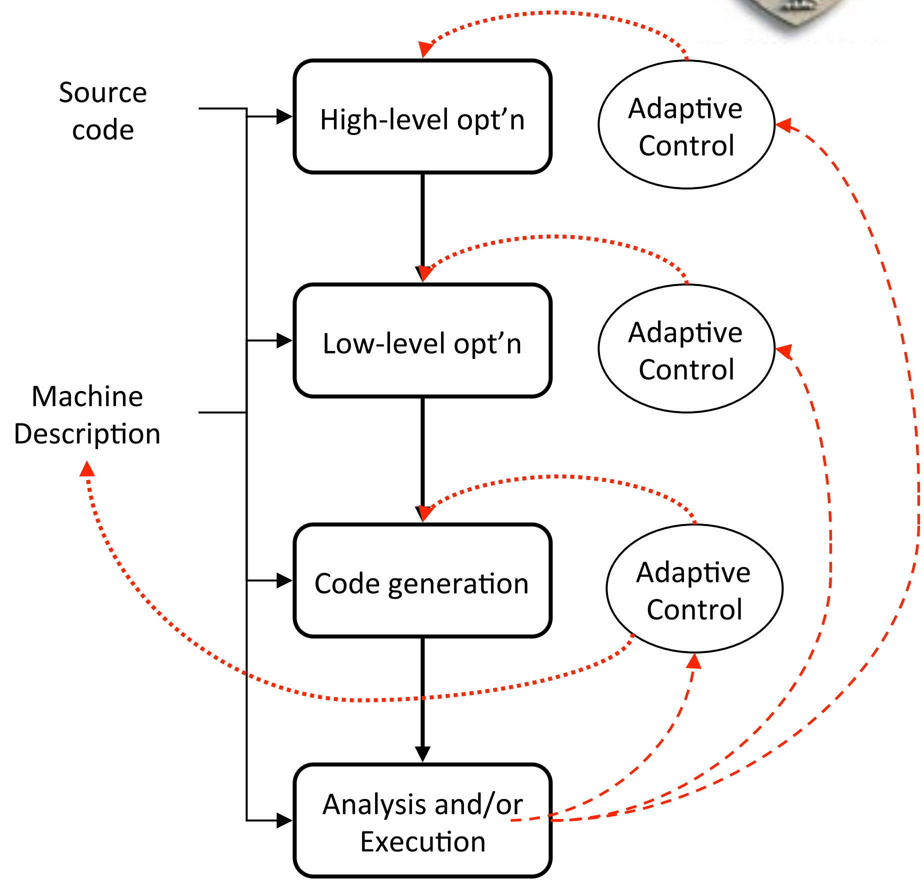


Some passes should provide data directly to the adaptive controllers

Multi-level Feedback-driven Adaptation

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***

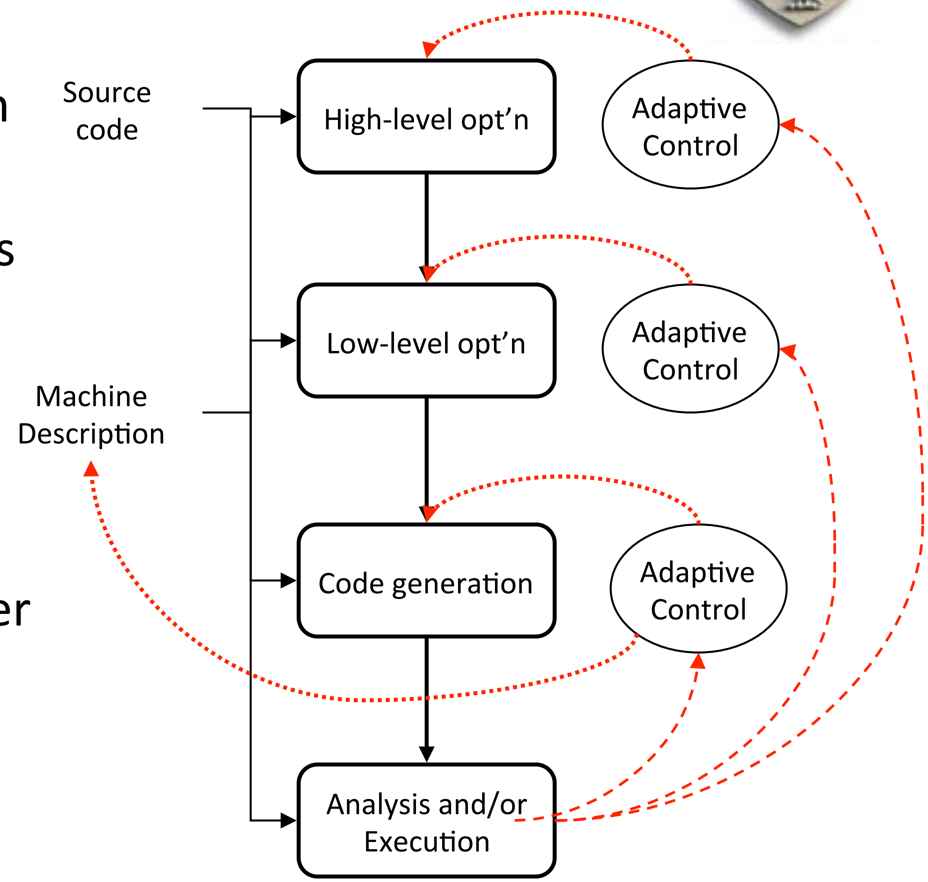


Multi-level Feedback-driven Adaptation

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?

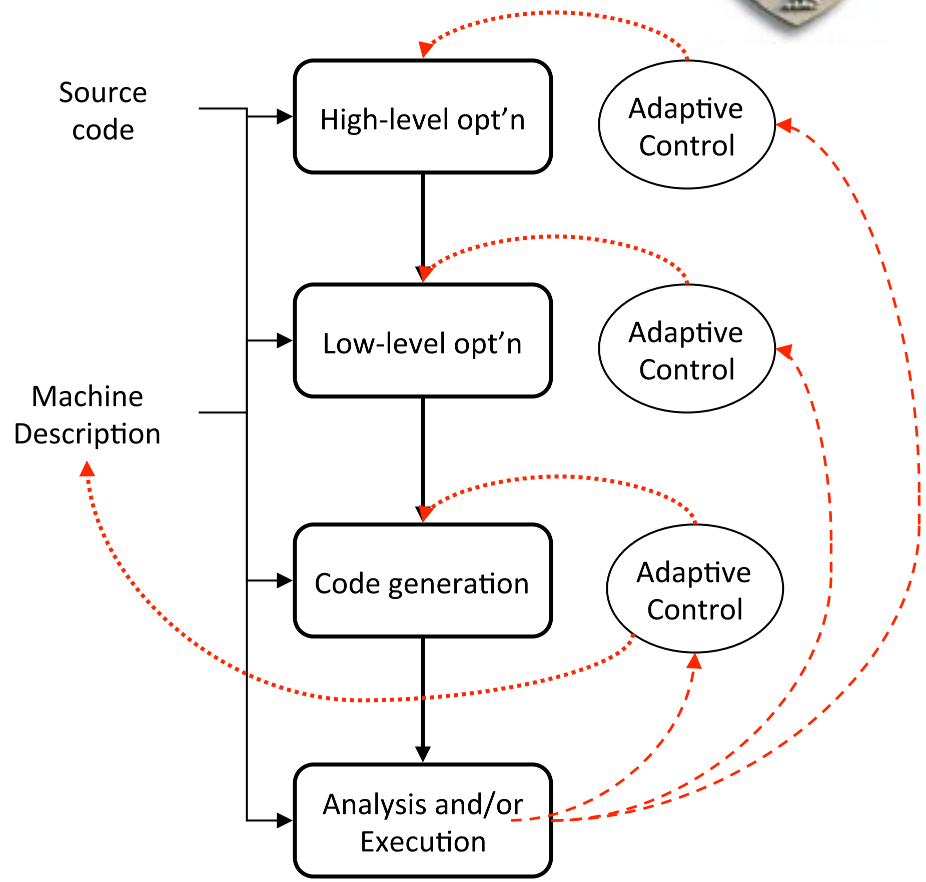
Find the answers by experimentation



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

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