Search-based Compilation:
Lessons from a Decade of Experiments

Keith D. Cooper

Department of Computer Science
Rice University
Houston, Texas

Many collaborators were involved this work, including Tim Harvey, Devika Subramanian, Linda Torczon, Phil Schielke, Lelac Almagor, Alex Grosul, Todd Waterman, Anshuman Das Gupta, Jeff Sandoval, Yi Guo, and Apan Qasem

The work has been funded over time by DARPA, DOE, NSF, and Texas Instruments. Any opinions expressed herein are those of the speaker, not the agencies. Obviously.

Roadmap

- The idea (old hat here)
- Sample problems in search-based compilation
- Experience
- Lessons
- Making It Practical

Big Picture

The goal of compiler-based autotuning work is to provide predictable good performance across a broad variety of applications, architectures, languages, and coding styles

- All applications should achieve "good" performance
  - On VAX 11/780, almost anyone could get 85% of peak
  - On modern processors, it requires heroic, processor specific tuning to get 85% of peak
- Small changes in the code (or machine) should not produce large changes in performance
- The user should not become an expert in the instruction dispatch discipline of the PowerPC G5 (or Opteron, or …)

Compilers offer the hope of achieving that goal

Hand tuning can achieve it for limited sets of applications

Big Picture (The Personal Story)

Original goal of our work was to pursue twin objectives

- Achieve "better" code quality
  - In context, that meant more compact code
    - → Has expanded to include speed, power efficiency, and other criteria
  - Work was part of a DARPA program on embedded systems
- Find productive ways to use more compilation time
  - Moore's law was making cycles exponentially cheaper
  - Compilers were not taking advantage of technological change
    - → What would we do with 10x cycles? 100x cycles?
    - → Most optimizers would declare victory and quit early

Goal was to return to the good old days when most codes attained good performance (VAX 11/780)

Search-based Compilation

Idea is seductively simple

- Optimization is complex *and* application dependent
  - Contrary to self image, compiler writers are not omniscient
  - Responsibility for compiler behavior should not fall to users
- Design & build self-tuning compilers
  - Palem, Motwani, Sarkar, & Reyen had taken 1st step
    - → α-β tuning for tradeoff between scheduling and allocation
    - → Derived new "heuristic" for problem in code generation
  - We wanted to spend more cycles than they had
    - → Application-specific solutions
    - → Try larger and harder optimization spaces

Many folks are working actively in this arena

- I will speak mostly from our experience

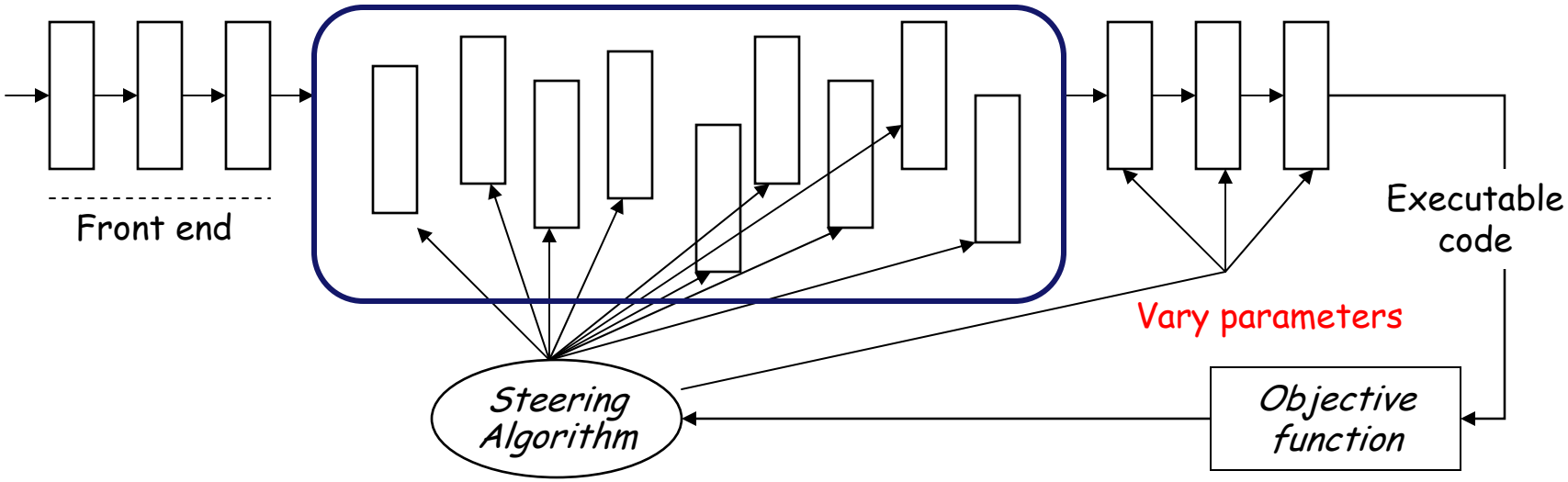
Sample Problems in Search-based Compilation

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
- Control of individual optimizations
  - Heuristics for inline substitution, coalescing register copies
- Controlling suites of transformation
  - Choosing loops to fuse and tiling them

Finding Optimization Sequences

Prototype adaptive compiler (1997 to present)



- 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 , …

Finding Good "Optimization" Parameters

- Corporate benchmarking compilers have myriad parameters
  - Experts find good settings for parameters
  - Use search to find parameter settings
    - → Work done by several groups
- Many transformations have specific control parameters
  - Block sizes, unroll factors, and so on
  - Use search to find parameter settings
    - → Techniques vary from brute force enumeration to search

Major lesson:

- Compilers are parameterized for convenience of compiler writers, not for quality of external control
- We can search to find settings for exposed knobs
  - Often, those knobs are the wrong knobs

Adaptive Control of Individual Transformations

Some transformations are controlled by complex heuristics

- Heuristics usually embody both experience and compromise
- Evidence suggests that one size does not fit allf

Example: Inline Substitution

- Transformation is easy
  - Replace call site with body of called procedure
  - Reduce call overhead & create opportunity for customization
- Control is complex
  - At each call site, decide to inline or not
  - Decisions are interrelated
    - → Inlining a into b may change decision on inlining b to c
    - → Require detailed knowledge of state of the code

      Sizes, frequencies, constants, opportunities, code growth, …

Adaptive Control of Suites of Transformations

Transformations interact with one another

- Loop fusion can increase reuse & decrease usable locality
- Loop tiling can increase reusable locality
- Good decisions on fusion & tiling go hand-in-hand

Joint adaptive control of fusion and tiling (+ array padding)

- Produces more consistent & predictable results than individual decision making
  - Solving combined problems can lead to novel solutions [Click]
- Model-based techniques give good results
  - Explore parameter space, pick good configuration (30-120 trials)
- Empirical exploration often finds better results
  - Accounts for actual ability of compiler to produce good code

Controller sees architecture through "lens" of compiler's capabilities

Experience: 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

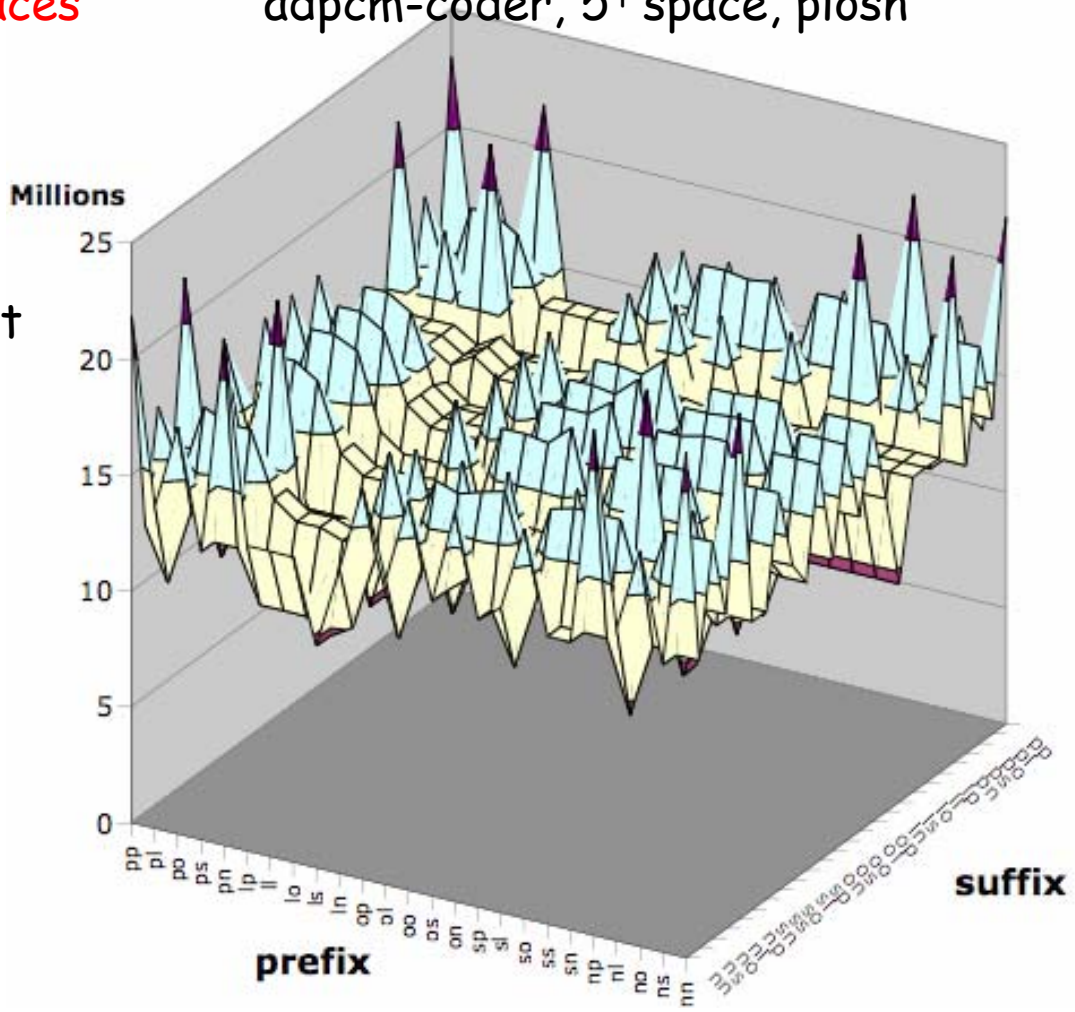
Experience: Sequence Finding

We learned about search spaces

These spaces are:

- not smooth, convex, 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

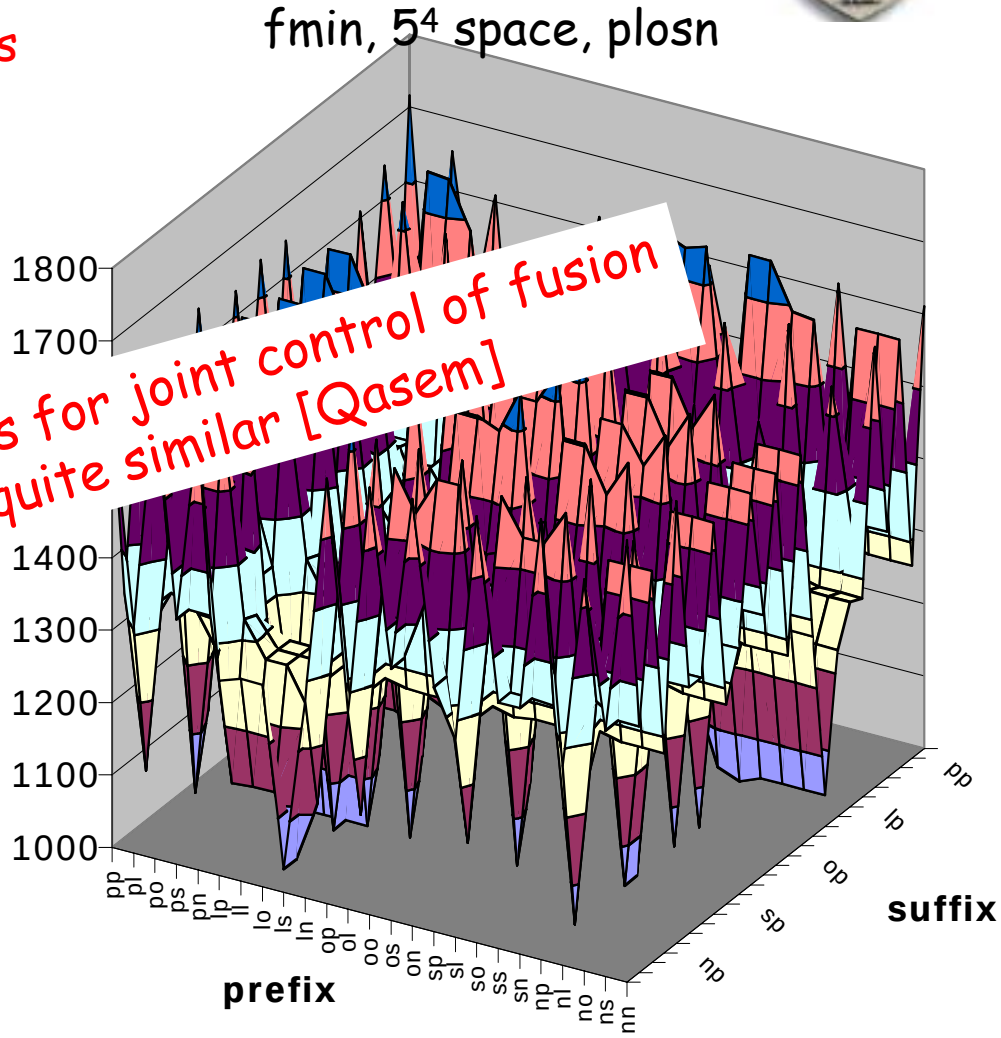
Experience: Search Spaces

We learned about search spaces

These spaces are:

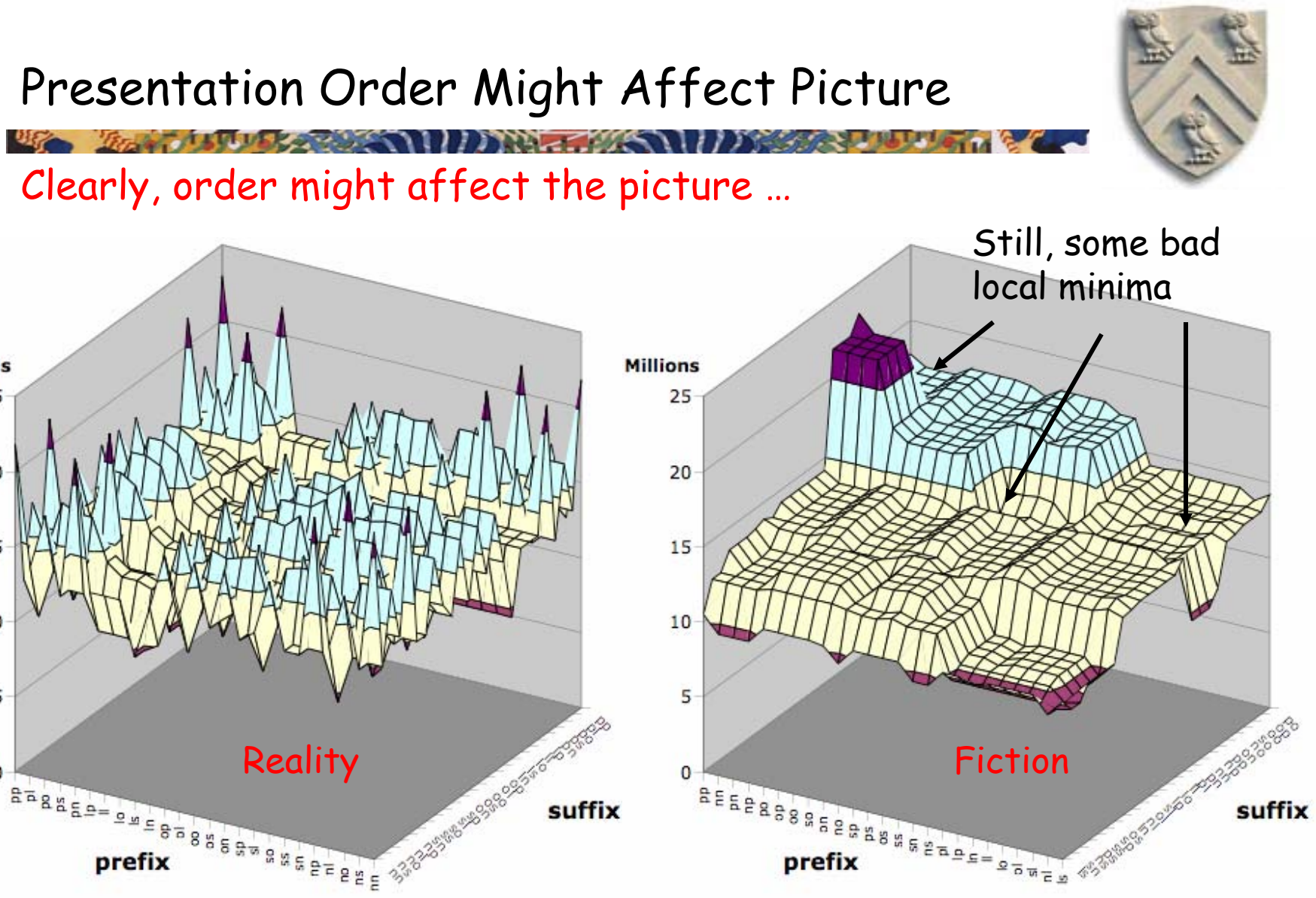
- not smooth, convex, or differentiable
- littered with local minima at different fitness values
- program deper

Search spaces for joint control of fusion & tiling look quite similar [Qasem]



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

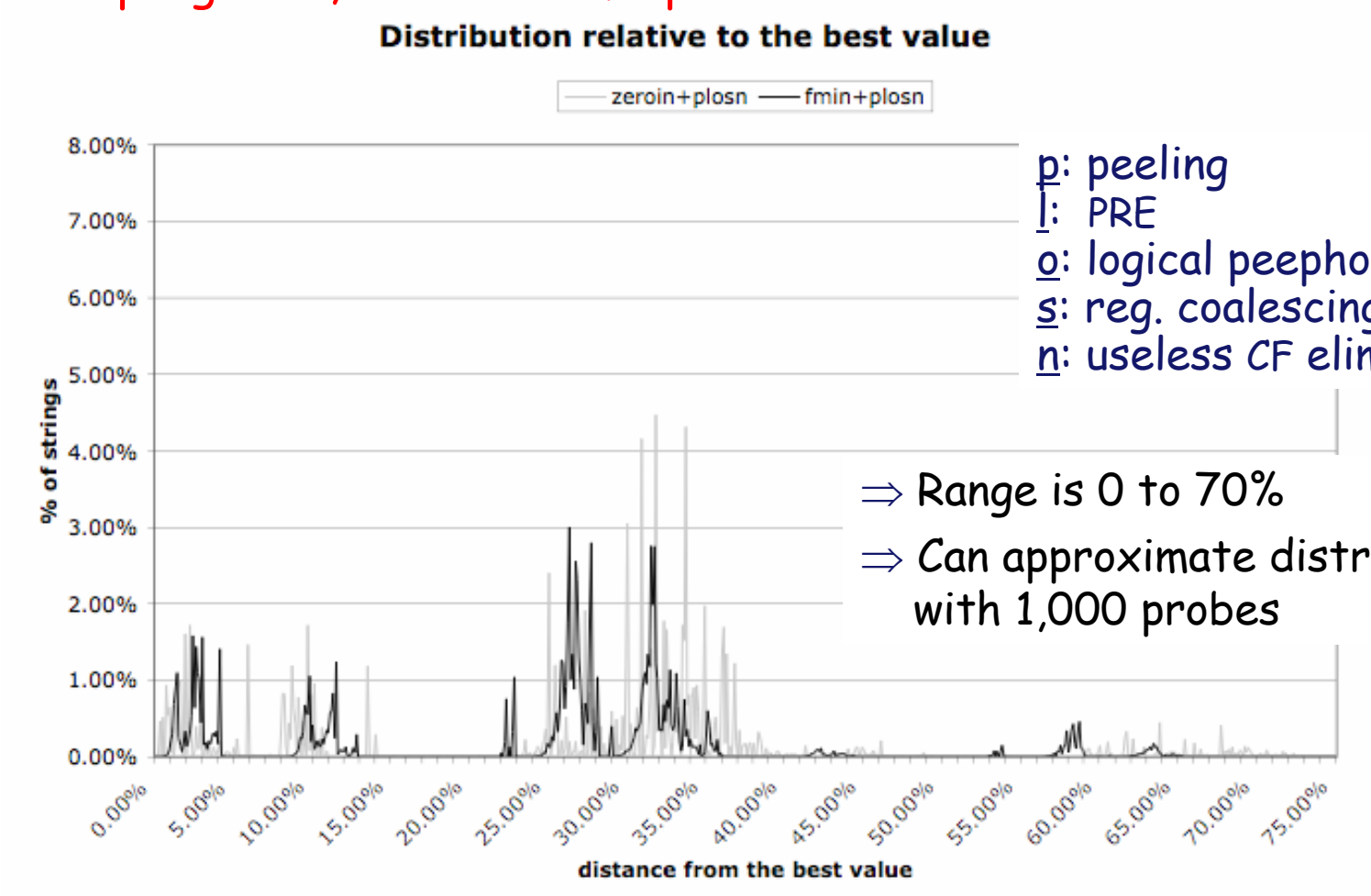
Presentation Order Might Affect Picture



adpcm-coder, 5^4 space, plosn

Experience: Search Spaces

Two programs, same set of optimizations



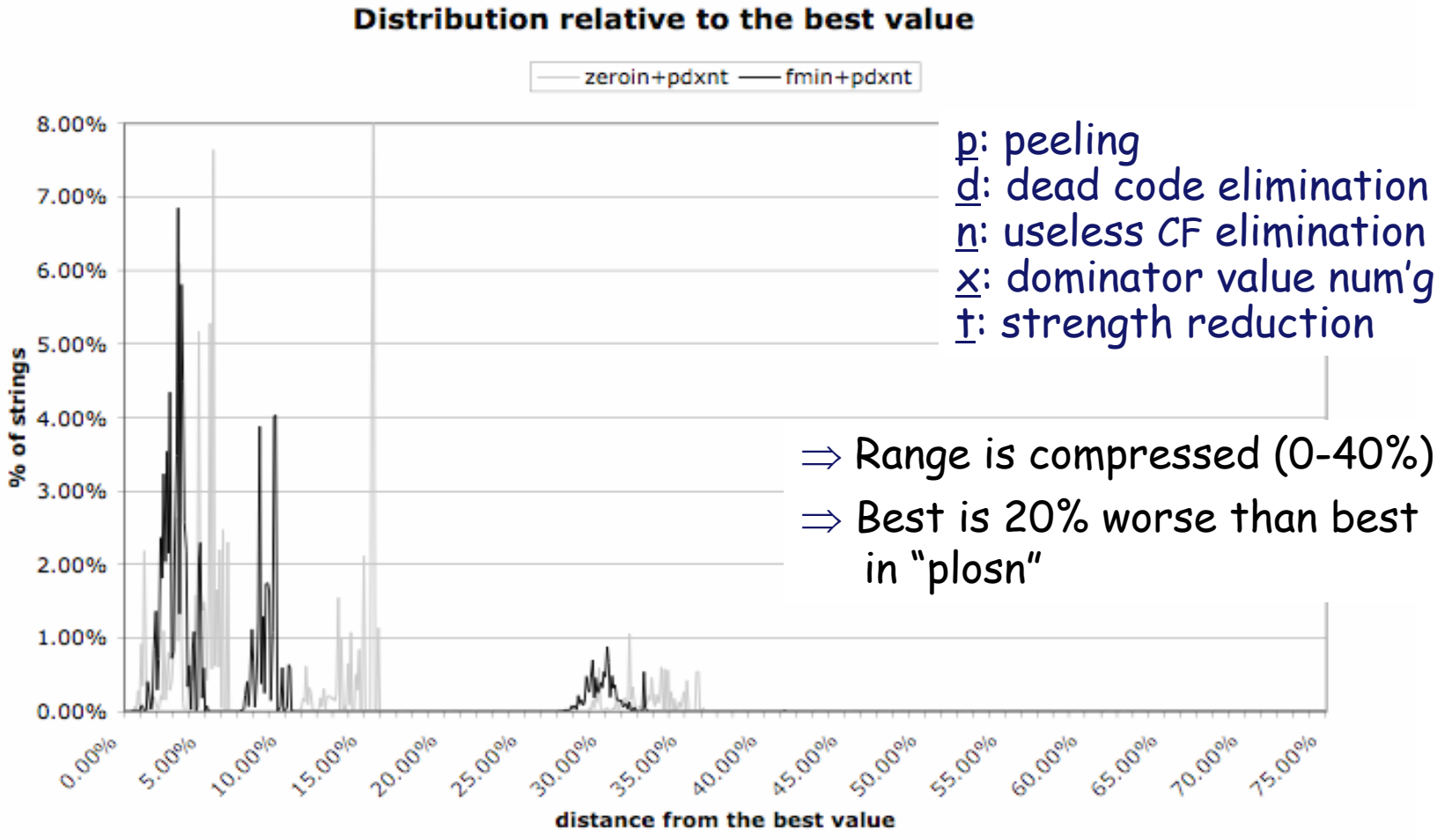
p: peeling
l: PRE
o: logical peephole
s: reg. coalescing
n: useless CF elimination

⇒ Range is 0 to 70%

⇒ Can approximate distribution with 1,000 probes

Experience: Search Spaces

Same two programs, another set of optimizations



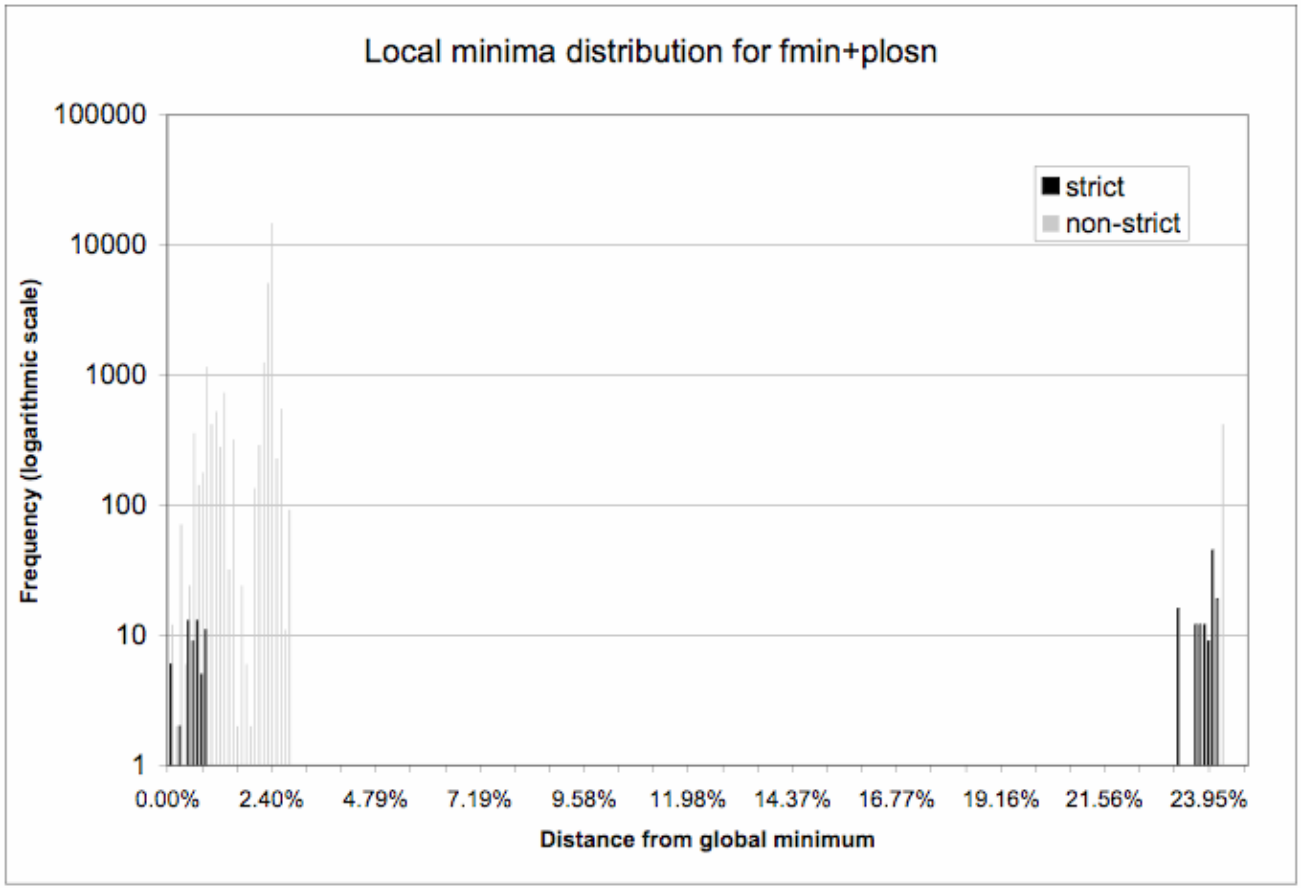
p: peeling
d: dead code elimination
n: useless CF elimination
x: dominator value num'g
t: strength reduction

⇒ Range is compressed (0-40%)

⇒ Best is 20% worse than best in "plosn"

Experience: Search Spaces

Good local minima are plentiful



Many local minima

258 strict

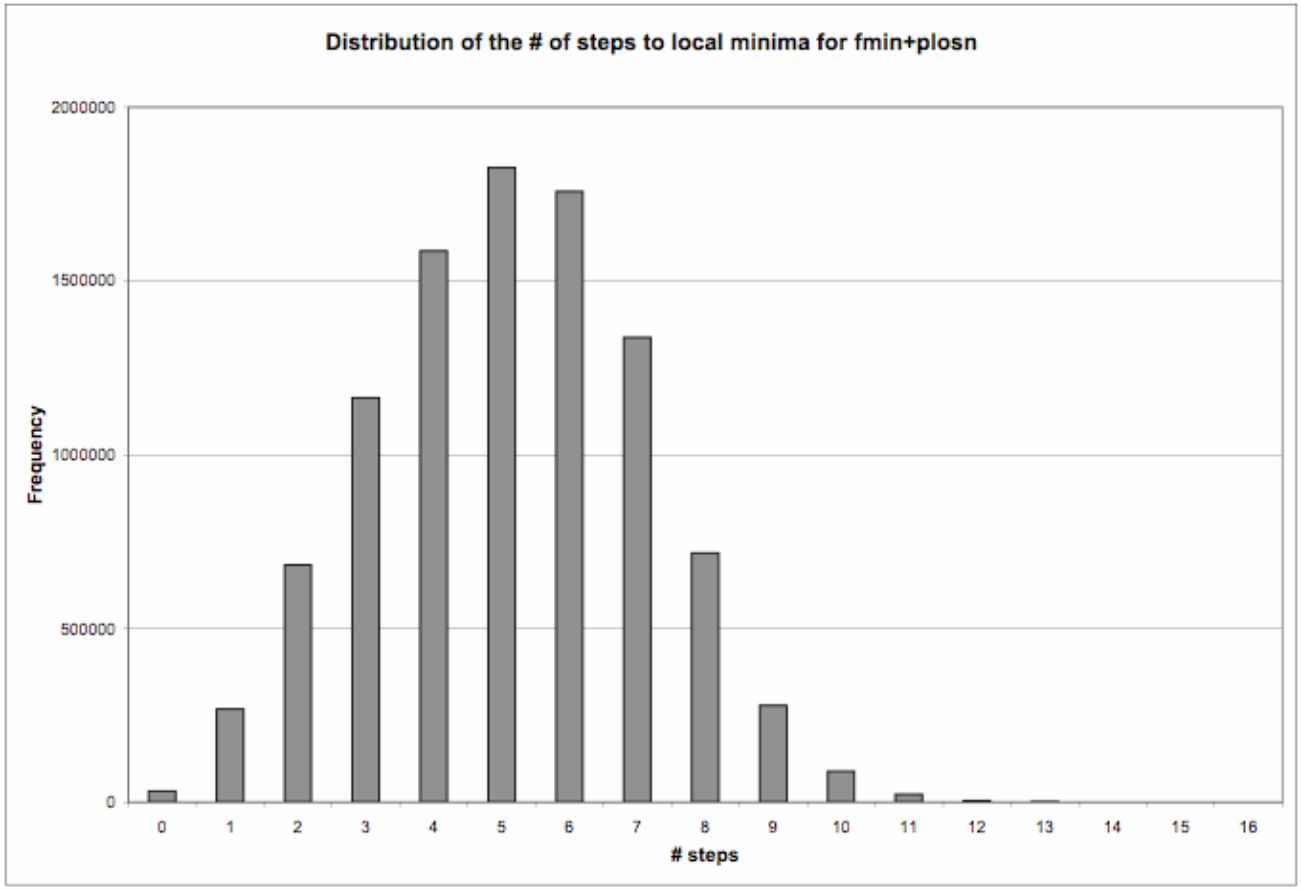
27,315 non-strict

(of 9,765,625)

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

Experience: Search Spaces

Distance to a local minimum is small



Downhill walk halts quickly

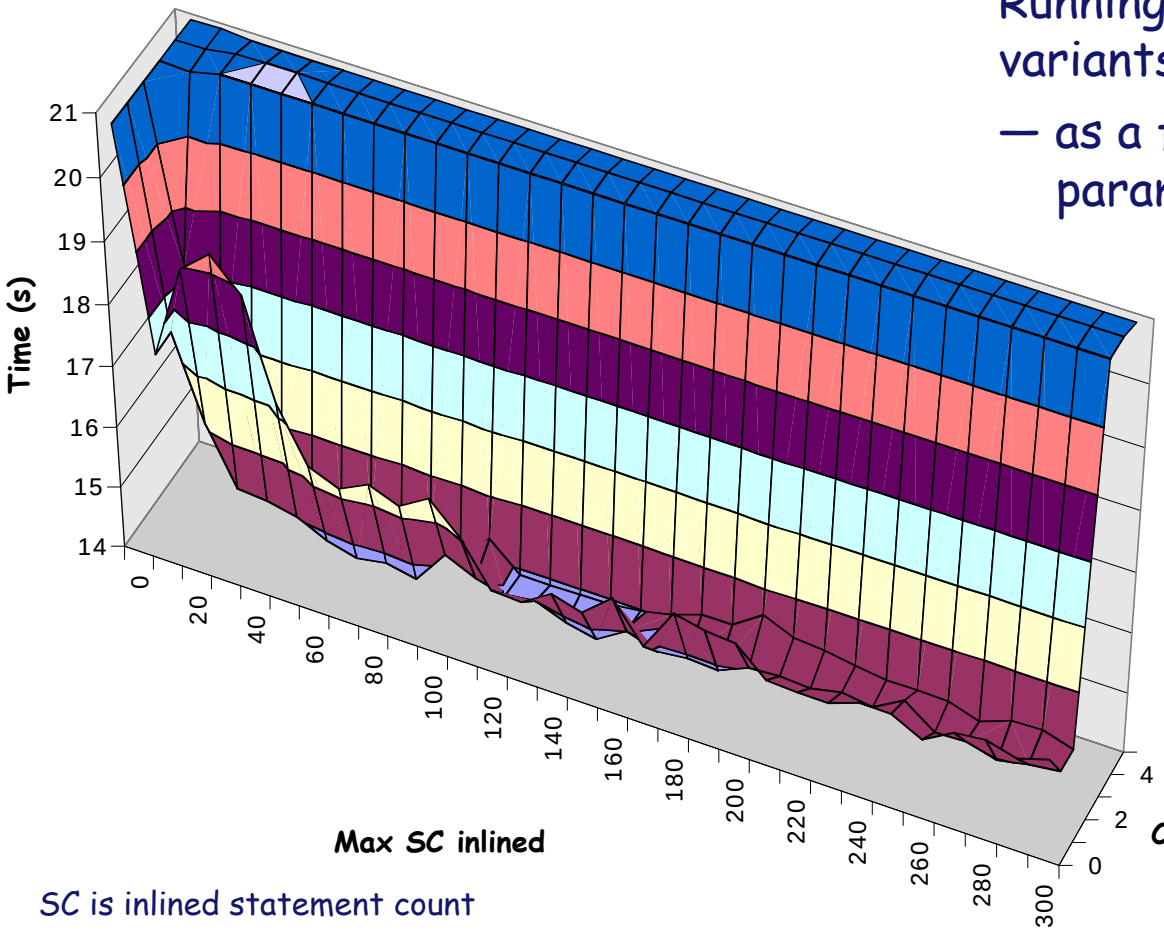
Best-of-*k* walks should find a good minimum, for big enough *k*

Experience: Inline Substitution

Characterizing the search spaces with 2d parameter sweeps

Running times for inlined variants of Vortex

— as a function of inliner parameters



SC is inlined statement count

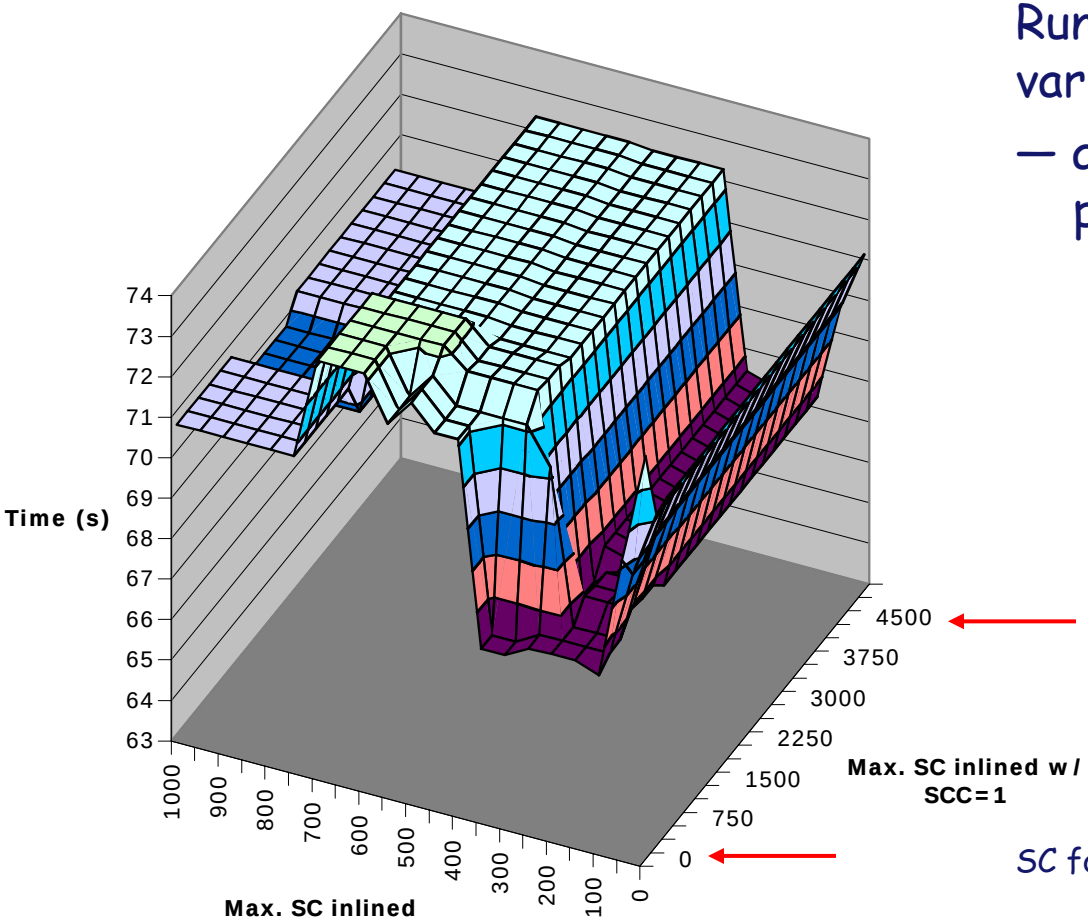
Space is simpler, but has unconstrained integer values

⇒ Parameterization is important

CPC is constants per call site

Experience: 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

Designing Search Algorithms

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

10% for space
8% for bit transitions

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 ⇒ 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 (random order)
- 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 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

95 evaluations for 10-of-5 space

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

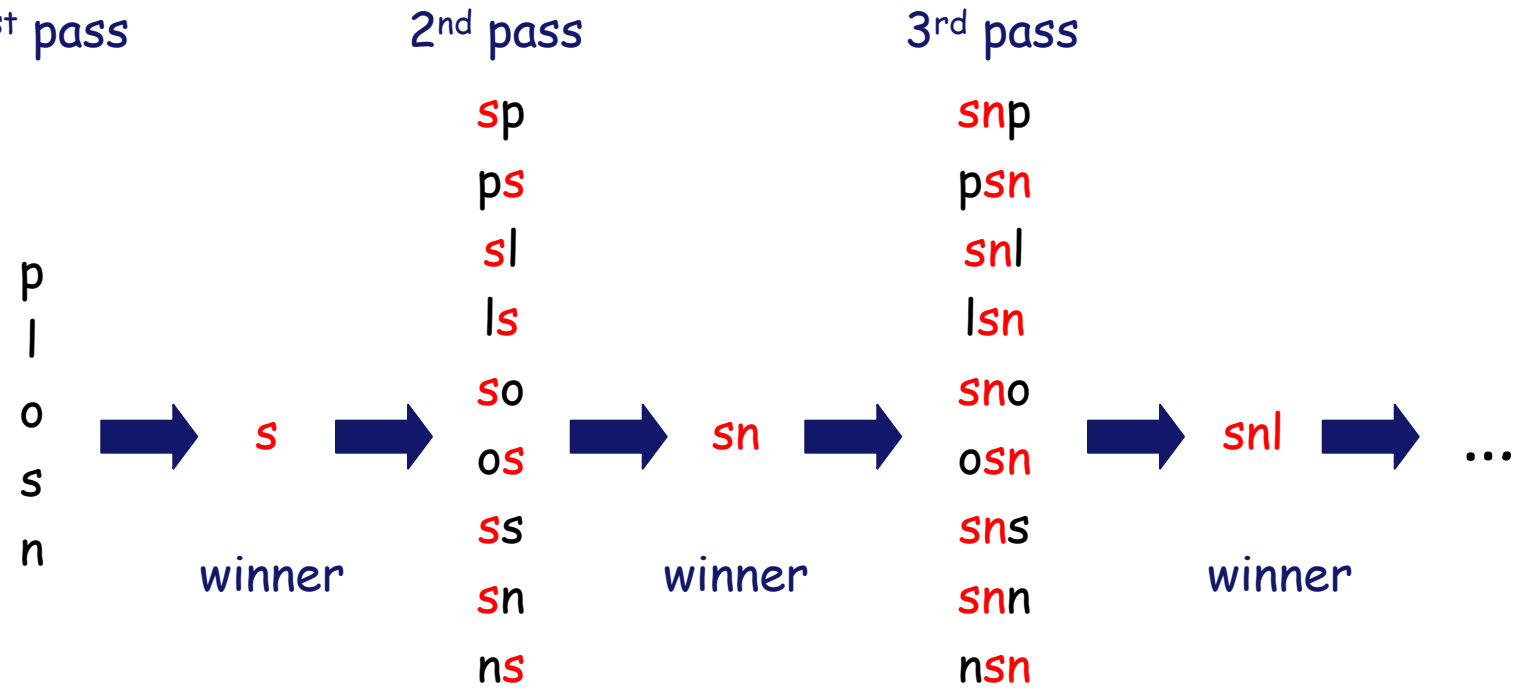
Takes locally optimal steps

Early exit for strings with no improvement

Local minimum under a different notion of neighbor

Search Algorithms: Greedy Constructive

Successive evaluations refine the string



Search Algorithms: Greedy Constructive

Two problems with Greedy Constructive

- Ties in objective function take GC to wildly different places
- GC ignores coordinated effects of multiple transformations

Recent work includes other greedy approaches

- Yi Guo developed GNE, a greedy variant that does a more careful search of local neighbors. In preliminary tests, it beats GC in efficiency (trials) & effectiveness (quality)

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 $\langle p_1, p_2, p_3, \ldots, p_n \rangle$, examines neighborhood of each p_i
    - → Systematically looks at $\langle p_1 \pm s, p_2 \pm s, p_3 \pm s, \ldots, p_n \pm s \rangle$
    - → 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

Lessons

The Big Picture

- Compilers that adapt their behavior produce better code than "one-approach-fits-all" compilers
  - Your mileage will vary, but performance will improve
- Attention to search techniques improves efficiency
  - Sequence finding: from 20,000 trials to 100s
  - Fusion & tiling: need 30 to 120 trials for "good" results
- Choice of playing field is important
  - Search can blindly find good combinations
  - Careful design of space & algorithm can improve efficiency (trials) and effectiveness (quality)

Lessons

Parameterization is critical

- Software engineering: compilers expose the wrong parameters
  - Waterman's inliner took 1 parameter that encoded heuristic
    - → *CNF expression over program properties & constants*
    - → Controller evolves a (reusable) program-specific heuristic
  - Qasem showed the need for loop-by-loop control
  - Different issues for large numerical ranges & small ones
    - → SLOCs, unroll factors, encoded fusion chocies
- Search: parameterization shapes search space
  - Radical effects on search efficiency & effectiveness
  - Waterman used a canonical CNF & varied constants
  - Qasem contrasted search over architectural parameters against transformation decisions
  - Sandoval showed that larger spaces are sometimes easier

Lessons

Models versus measurement

- Issue has generated as much heat as light
  - Models radically reduce evaluation cost
  - Measurement captures {all,most} of the actual effects
- Neither side has an exclusive lock on the truth
  - Both models & measurement will play important roles in practical systems (*hybrid evaluation strategies*)

What I can say:

- Stability is important
  - Deterministic measures (Misses, ops retired) versus time
- Speed is important
  - Evaluation cost dominates overall search time

Lessons

Highly personal complaint

Theoretical characterization for publication may have no relationship to actual difficulty of finding good solution

Example [*from Qasem's thesis but widely true*]

- Optimal choice for fusion is NP-hard
- Solution space for fusion & tiling is "large" (100,000 points)
- These facts do not mean that finding good solutions is hard
- Qasem finds "good" solutions in 30 to 120 trials
- Random probing comes within a couple of percent of PBDS

Making Adaptive Compilation Practical

Critical constraint is cost of finding right configuration

- Good parameterizations
- Good search algorithms

Likely solutions

- Incremental search
  - Distribute cost over multiple compilation cycles
  - Want to implement this feature for fault tolerance
- Hybrid evaluation strategies
  - Combine model-based approaches with empirical evaluation
  - Recognize equivalent results

Making Adaptive Compilation Practical

Need to provide user control over process

- User should specify granularity
  - Whole program, partial programs (libraries), procedures, methods, loop nests, individual blocks
  - Compiler writers don't worry about user interfaces
- User should specify objective function
  - Can look for speed, code size, energy, ILP, locality, …
  - Need to express tradeoffs in human-comprehensible way
    - → Again, need a user interface
- User should be shielded from the mess
  - Running 100 compile/execute/measure steps is a pain
  - User should hit a button and get (eventually) a result

Making Adaptive Compilation Practical

Retrofitting these ideas into 1980 compiler is hard

- Detailed design issues make a difference
  - How do you run *GCC* passes in another order?
  - Exposing the right set of controls (-O3)
- Need to perform adaptation at "right" level of abstraction
  - Unroll, unroll & jam are difficult at near-machine level
  - Different levels of abstraction needed at various points
- Need the best back-end algorithms
  - See Clint Whaley's talk, slide 8
  - Militates against back-end portability

Open-source works well for widely understood problems

May not be practical (yet) in this arena

Conclusions

From 1956 to 1996, one-size-fits-all compilers ruled the roost

Today

- We know that application-specific behavior is better (duh)
- We are beginning to understand how to build these systems
  - Growing body of knowledge on adaptation
  - Growing set of ideas on single xforms & sets of xforms
- We are beginning to see these ideas emerge in commercial compilers
  - Large effort, but beginning to see results
- We expect these technologies to take us back to an era of robust, compiler-produced performance (VAX 11/780)
  - Not a substitute for better algorithms & applications