Adaptive Optimizing Compilers for the 21st Century

Keith D. Cooper *Devika Subramanian* *Linda Torczon*

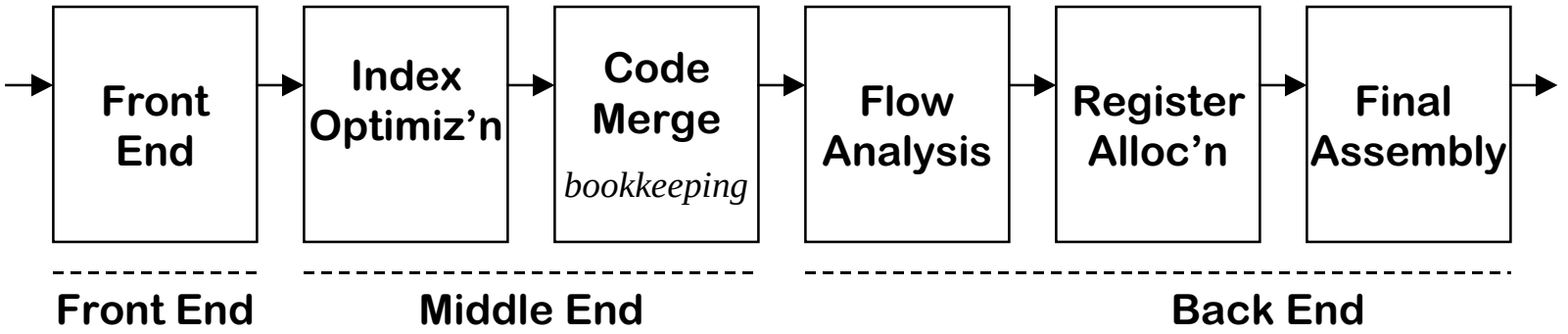
Department of Computer Science
Rice University
Houston, Texas, USA

Los Alamos Computer Science Institute
2001 Symposium

LACSI

The Big Picture

- For 45 years, compilers have followed the Fortran model

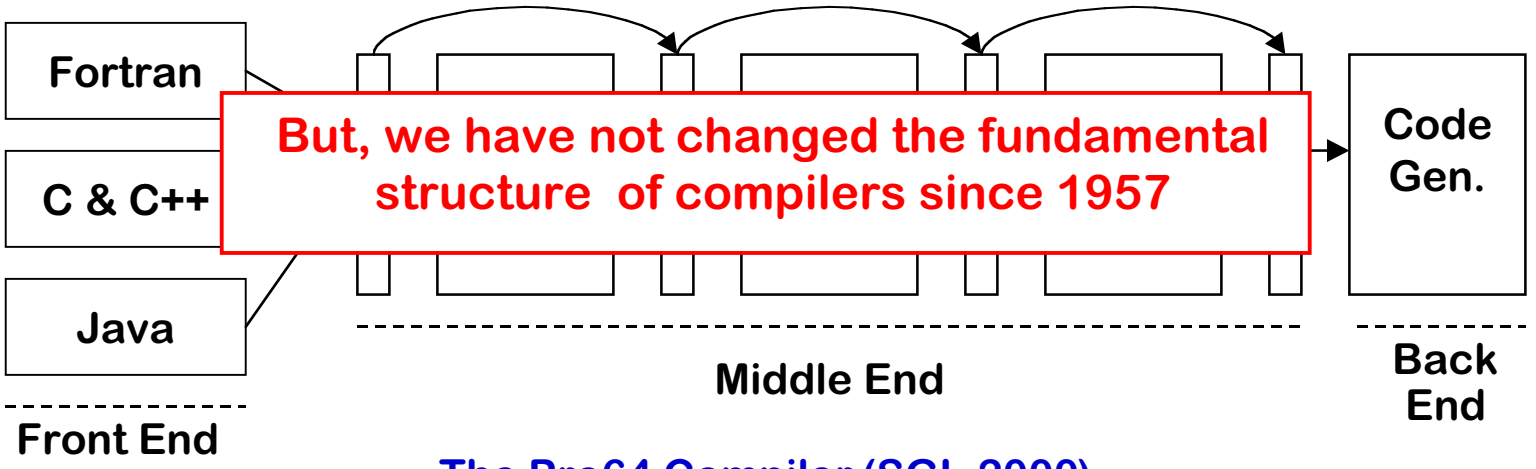


The FORTRAN Automatic Coding System (IBM, 1957)

- Structure is ingrained in our heads, our books, & our tools
  - Fixed set of passes in a predetermined order
  - Clear division of labor between components
- This structure has let us make progress
  - New languages, new optimizations, new targets

The Big Picture

- RISC made compilers more responsible for performance
- Subsequent architectures rely more heavily on the compiler
  - In 1980, 85% or more of peak was typical
  - In 2001, often it is 5% to 10% of peak
- How has the compiler community responded?
  - Better analyses
  - More transformations

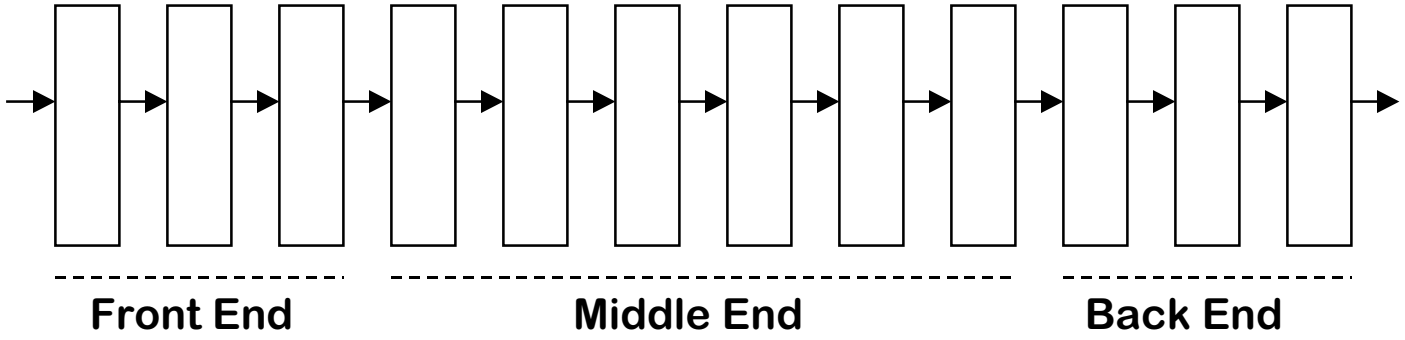


The Pro64 Compiler (SGI, 2000)

LACSI

The Big Picture

- The time has come to re-examine fundamental assumptions



- Compiler's organization is set long before it sees the user's code
  - Fixed set of transformations, fixed order
  - Same strategy for all programs and all targets
  - Same (implicit) objective function for all compilations
  - Compiler writer must accurately predict the user's needs
- Compiler writers are good, but not that good!
  - Need flexible structures that adapt to user's needs

The Big Picture

Maybe we should learn from other fields …

In numerical analysis, optimization means

- **Finding a (local) minimizer for some objective function**

In compilation, optimization has meant

- **Running a fixed series of passes & declaring victory**

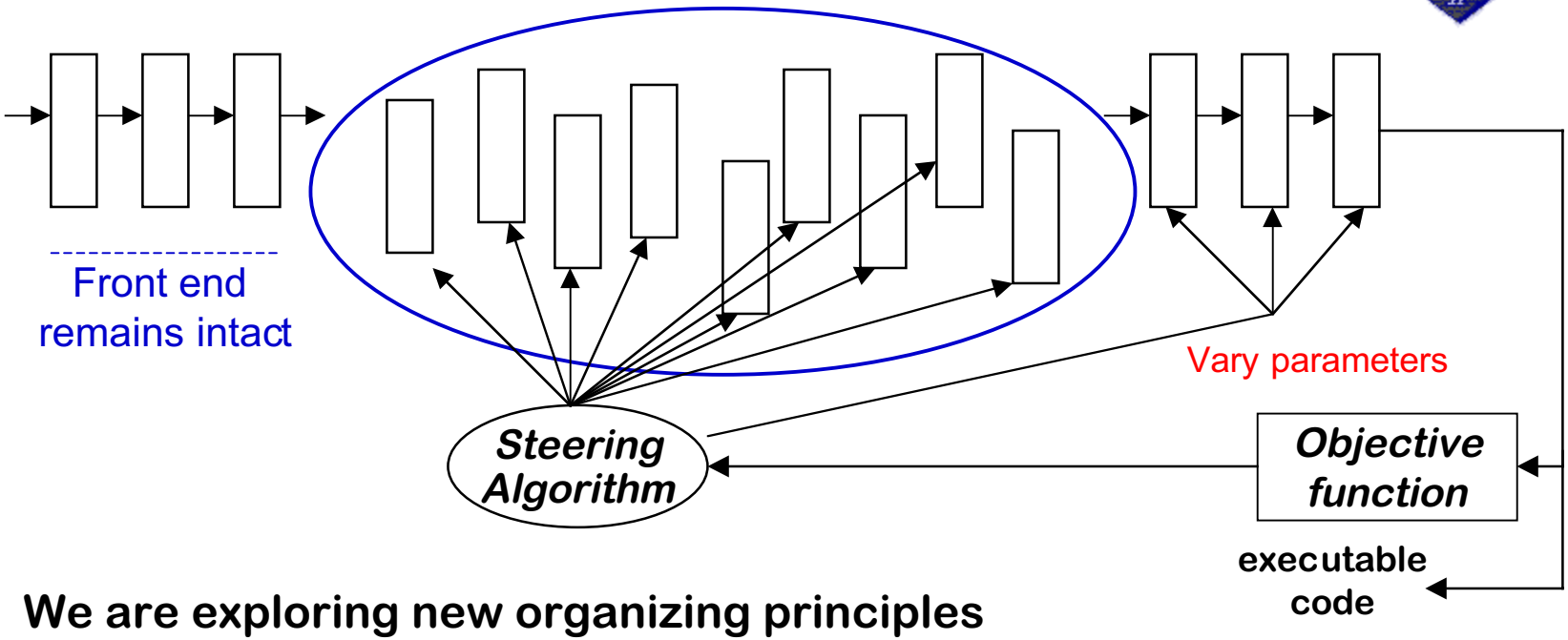
In benchmarking (where performance counts)

- **Many flags control the compiler's behavior**
- **Experts profile code and choose settings for flags**

We can build compilers that automate this kind of tuning

It takes multiple rounds of trial and evaluation to find an answer

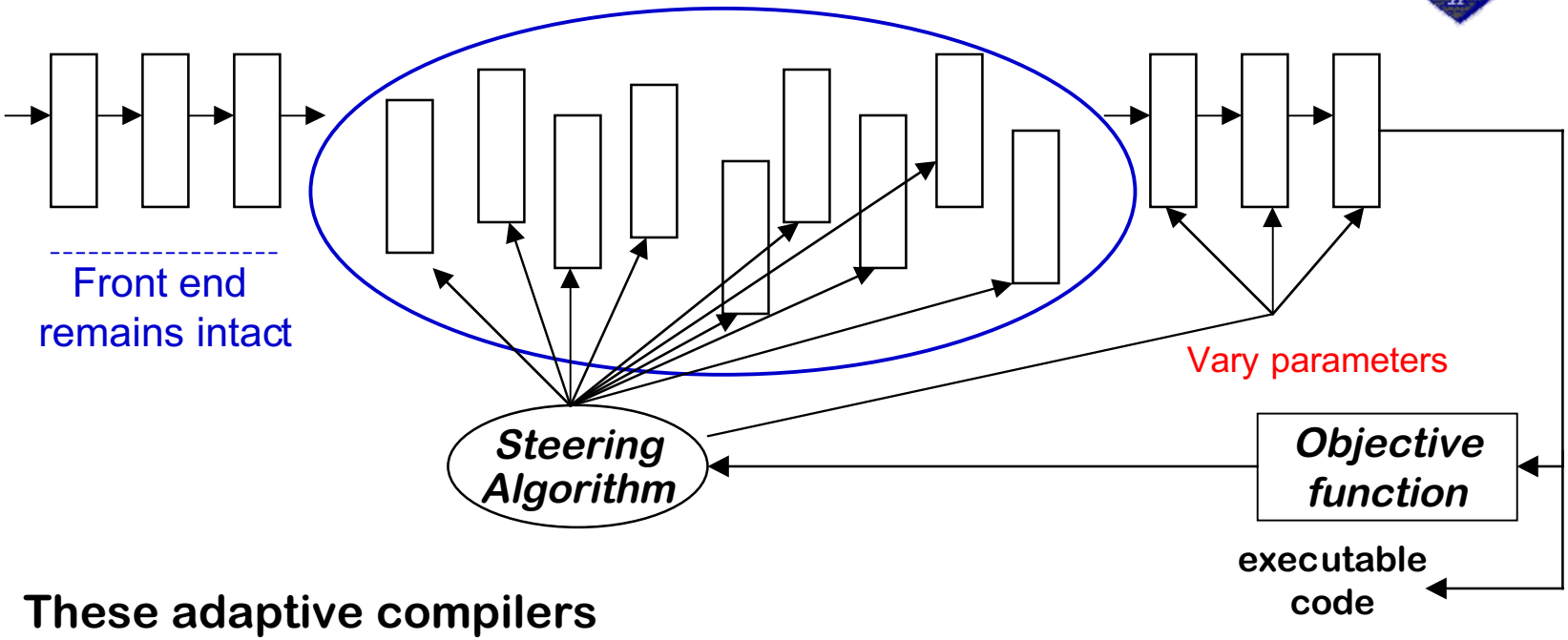
Adaptive Compilers



We are exploring new organizing principles

- **Explicit objective function** (***chosen by the user***)
- **Steering algorithm controls optimizer and back end**
  - **Picks & orders methods, chooses parameters**
  - **Use multiple trials to explore the solution space**
  - **Finds a configuration to minimize objective function**

Adaptive Compilers



These adaptive compilers

- **Adapt to the program, the target, & the objective function**
- **Use power of modern computers to improve code quality**
- **Produce better results than their fixed-sequence siblings**
- **Are much more expensive (today) than fixed-sequence compilers**

What's the Point?

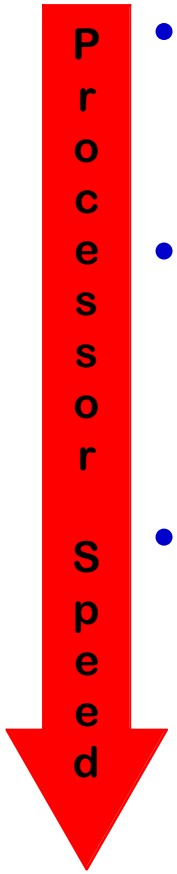
These compiler configurations matter …

- **Choice of transformations**
  - **Significant overlap in coverage**
  - **Each code has different opportunities**
  - **Best choice depends on input, target, & objective function**
- **Order of transformation**
  - **They create & destroy opportunities for each other**
  - **We see 2x variations up and down on simple codes**
  - **Best choice depends on input, target, & objective function**

- **We do not know enough, today, to predict a good sequence**
- **Our prototype systems search the space of sequences**

Adaptive Compilation

Our goals

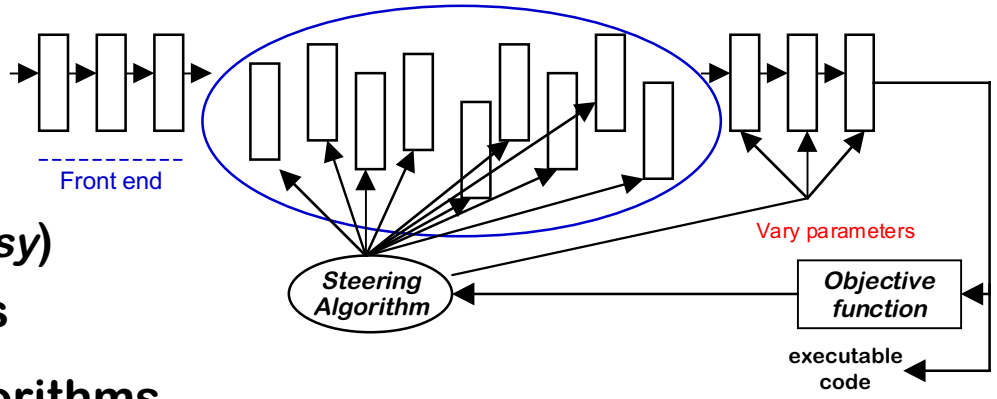


- **Short term**
  - > **Characterize the problems, the potential, & the search space**
  - > **Learn to find outstanding sequences quickly (*search*)**
- **Medium term**
  - > **Develop practical adaptive compilers for scalar optimization**
  - > **Develop a framework for self-tuning optimizers**
  - > **Develop proxies and estimators for performance (*speed*)**
- **Long term**
  - > **Apply these techniques to harder problems**
    - → **Data distribution, parallelization schemes on real codes**
    - → **Compiling for complex environments, like the Grid**
  - > **Build systems that use knowledge from search to make adaptive compilation both practical & routine**

Adaptive Compilation

Research prototype

- Based on MSCP compiler
- 20 transformations
  - > Run in any order (*not easy*)
  - > Many options & variants
- Search-based steering algorithms
  - > Hill-climber
  - > 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



Experimental Results

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

Early Experiments

- Space then speed (10 transformation subset)
  - 13% smaller code than fixed sequence (0 to 41%)
  - Code was generally faster (26% to -25%; 5 of 14 slower)
- Speed then space
  - 20% faster code than fixed sequence (best was 26%)
  - Code was generally smaller (0 to 5%)
- Genetic Algorithm
  - Evaluate each sequence
  - Replace worst + 3 at each generation
  - Generate new strings with crossover
  - Apply mutation to all, save the best
- Found "best" sequence in 200 to 300 generations of size 20

GA took many fewer probes to find "best" sequence than did random sampling.

Experimental Results

Improving the Genetic Algorithm

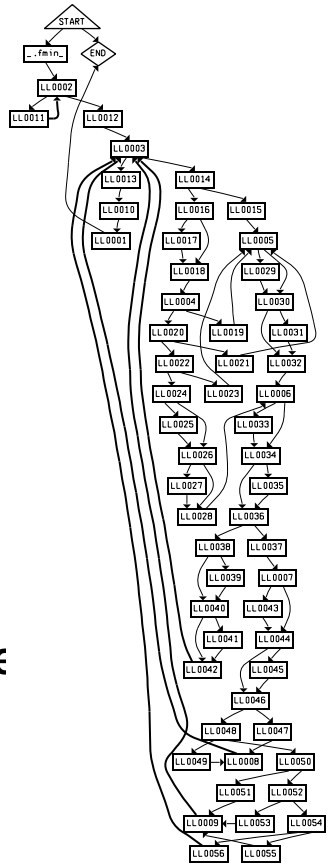
- Experiments aimed at understanding & improving convergence
- Larger population helps
  - Tried pools from 50 to 1000, 100 to 300 is about right
- Use weighted selection in reproductive choice (*vs. random*)
  - Fitness scaling to exaggerate late, small improvements
- Crossover
  - 2 point, random crossover
  - Apply "mutate until unique" to each new string
- Variable length chromosomes
  - With fixed string, GA discovers NOPs
  - Varying string rarely has useless pass

GA now dominates both the hill climber and random probing of the search space, in results/work.

Experimental Results

Characterizing the Search Space

- Took one application, FMIN
  - 150 lines of Fortran, 44 basic blocks
  - Exhibited complex behavior in other experiments
- Ran hill-climber from many random starting points
  - Picked the 5 most used passes from winners
- Generating all strings of length 10 from those 5
  - About 10,000,000 combinations
  - 34,000 to 37,000 experiments/machine/day
  - Produces sequence & number of cycles to execute
  - Finished 2.8 million to date
- We're learning from the results



FMIN

Experimental Results

After 2,800,099 sequences

- Best sequence is 1002 cycles
- Worst sequence is 3316
- Unoptimized code takes 1765
- Range of - 43% to + 88%

Score	Sequences	% of Solutions
Best	1	0.000036%
1%	14,968	0.53%
2%	84,926	3%
5%	213,674	8%
10%	299,582	11%
15%	354,762	13%
20%	354,762	13%
25%	372,413	13%
Worst	3,874	0.14%

- GA found better sequences (*using full set of transformations*)
  - GA found 822 in 184 generations of 100
  - Another run found 825 in 152 generations of 100
- Hill climber finds local minima fairly quickly
  - 833 at round 2232
  - 830 at round 750

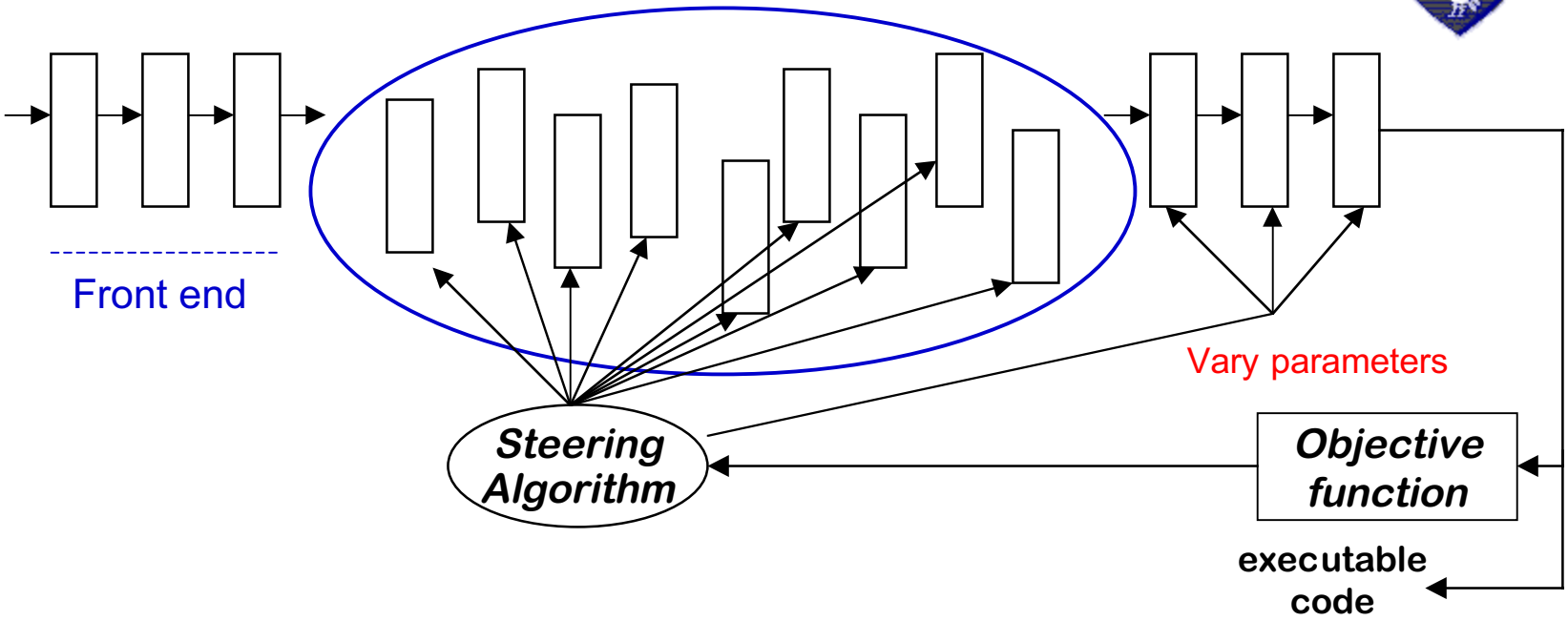
} Sensitive to starting point (best runs)

Adaptive Optimizing Compilers

What problems remain?

- **Convergence**
  - **Better steering algorithms**
  - **Good starting points**
  - **Stopping criteria**
- **Speed**
  - **Estimators for speed & other performance criteria**
  - **Shrinking the search space** (***one result of FMIN***)
- **Pragmatism**
  - **Engineering systems that trade compile time against results**
  - **Designing & engineering components that can be re-ordered**

The 21st Century Optimizing Compiler



We are building a new generation of flexible, self-tuning optimizers that reorganize to match the source, target, & objective

- **Harness code quality to Moore's law**
- **Automatically adapts behavior to changing situation**
- **Puts control in the user's hands ♦**

EXTRA SLIDES START HERE

Experimental Results

Optimizing for new criteria

- **Bit-transitions between successive operations**
  - **Represents one component of processor power consumption**
  - **May be significant on DSP chips**

- **Built an objective function that measures transitions & turned the GA loose on the problem**
  - **Saw 6 to 7% reductions from fixed sequence**
  - **Using vendor's simulator to estimate actual power savings**

- **The GA is optimizing over compiler's pseudo-random behavior**

Details on FMIN

The transformations

- **Logical peephole optimization**
- **partial redundancy elimination**
- **dead code elimination**
- **copy coalescing**
- **loop peeling**

Adaptive Optimizing Compilers

Making it practical

- **Algorithm needs good stopping criteria**
  - **Stop when it ceases to make progress, or Δ is tiny**
  - **Stop after a specified amount of time**
  - **Stop when it reaches some actual goal** **(*k* bytes of code)**

Other options for mitigating the cost

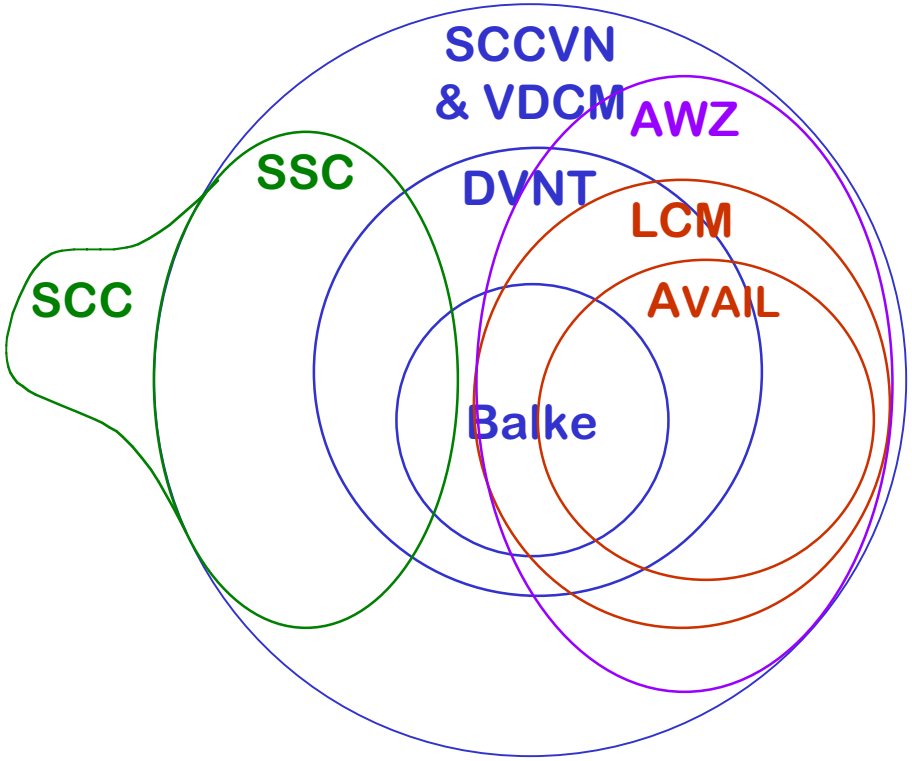
- **Use *k* best sequences for routine compilation**
- **Find program-specific sequence, then reuse it**
- **Distribute the search over many compiles**
  - **Start from previous best sequences & improve**

This should be fertile ground for research & experimentation

Why is this Problem Hard?

Consider one optimization - eliminating redundancy

- **Many techniques have been proposed**
- **Each catches a different set of cases**



Balke - Value Numbering

DVNT - Dominator VN

SCCVN & VDCM - Global VN

AVAIL - Classic CSE

LCM - Lazy Code Motion

AWZ - Partitioning algorithm

SSC - Sparse Simple Constant

SCC - Sparse Cond. Constant

And, there are many others …