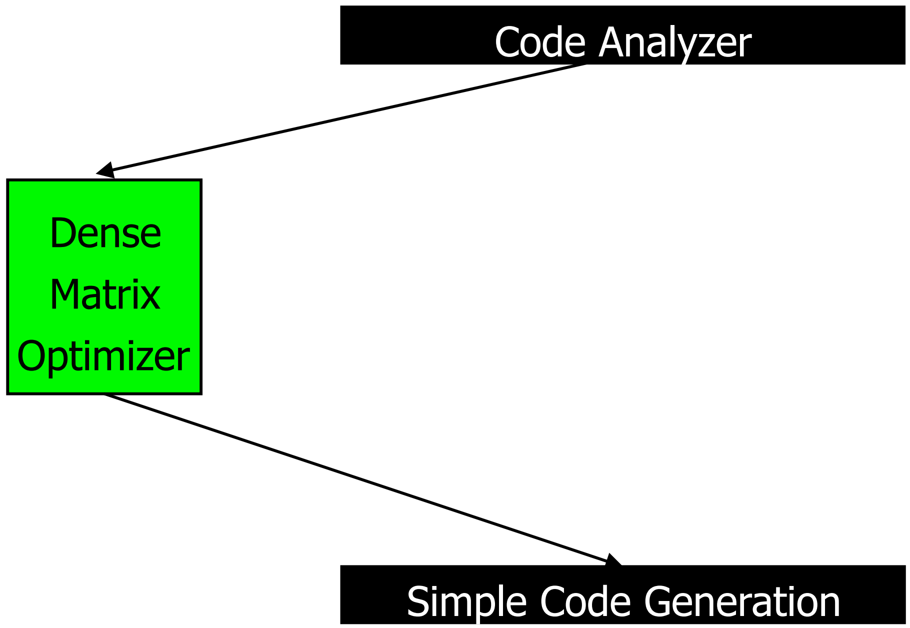
Intelligent Compilation

John Cavazos

Department of
Computer and Information Sciences
University of Delaware

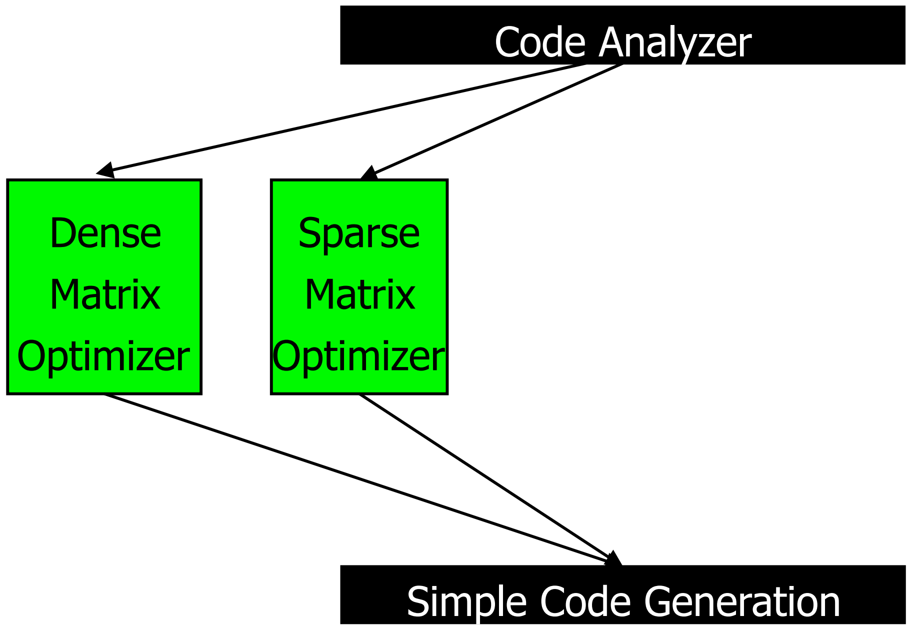
Proposition

- Proposition: “The focus on specialized tuning systems is too narrow …”
- Specialized tuning systems are compiler component



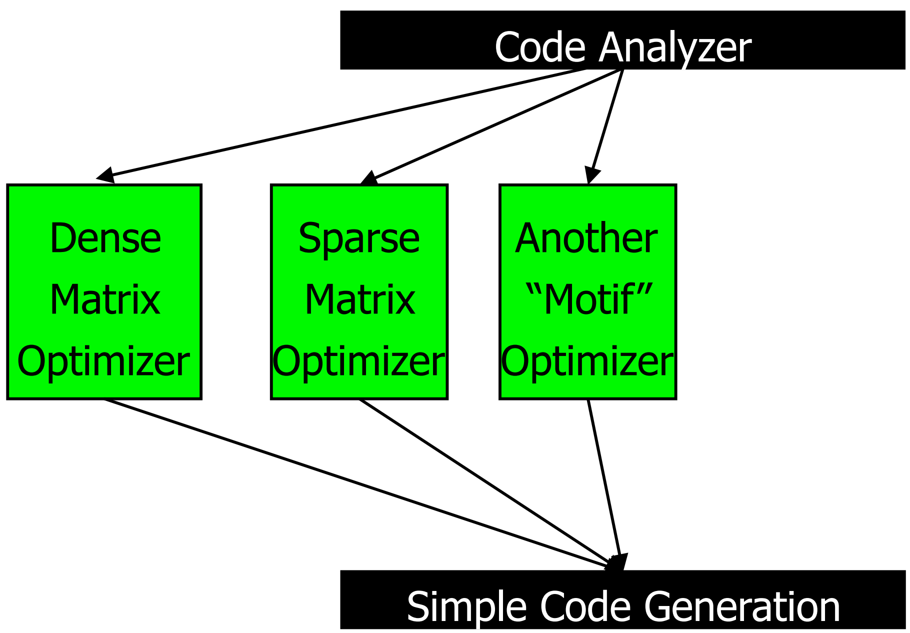
Proposition

- Proposition: “The focus on specialized tuning systems is too narrow …”
- Specialized tuning systems are compiler component



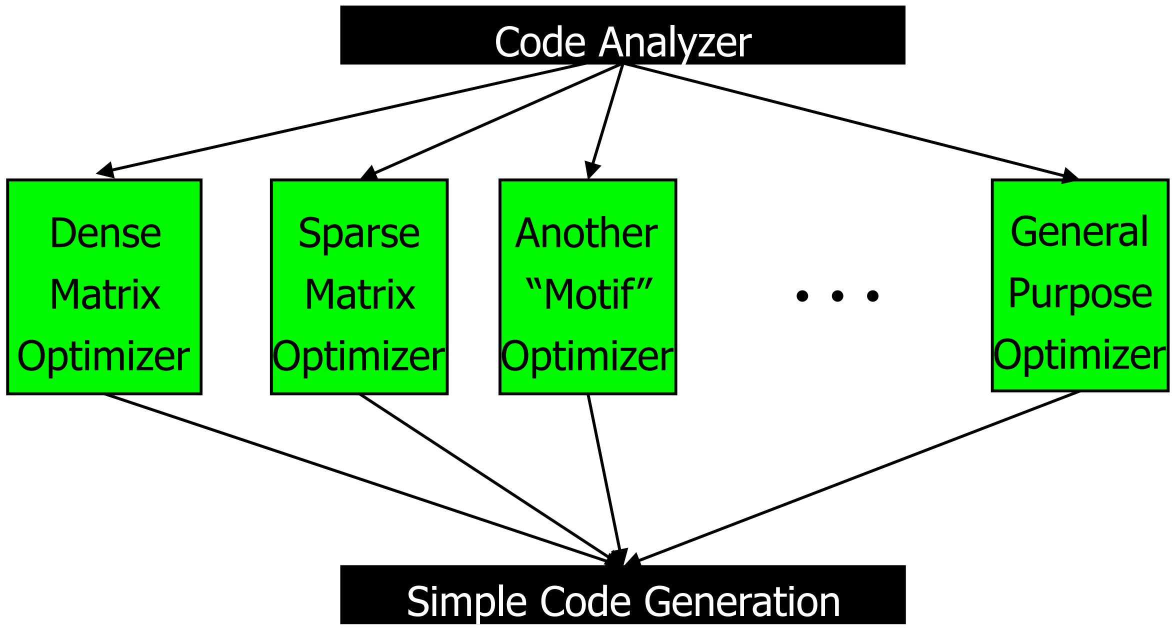
Proposition

- Proposition: “The focus on specialized tuning systems is too narrow …”
- Specialized tuning systems are compiler component



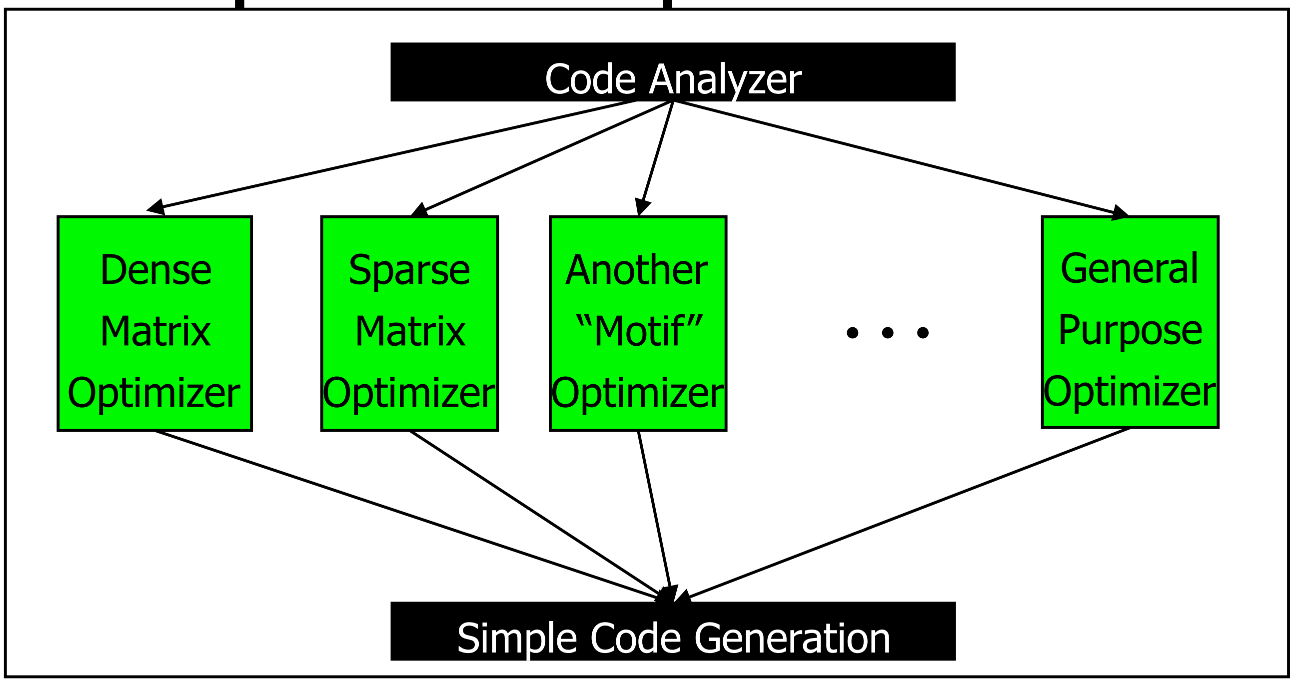
Proposition

- Proposition: “The focus on specialized tuning systems is too narrow …”
- Specialized tuning systems are compiler component



Proposition

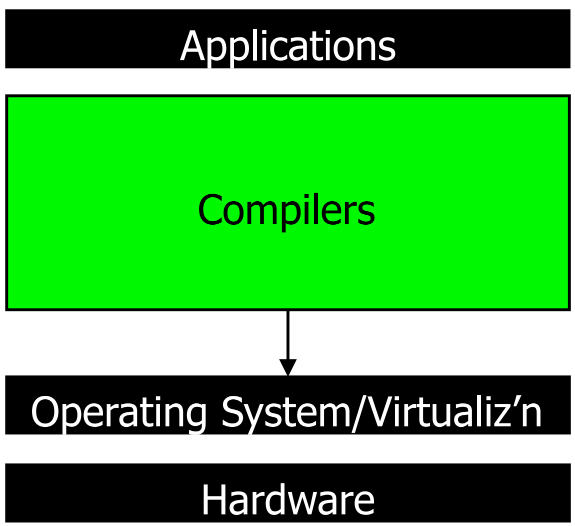
- Proposition: "The focus on specialized tuning systems is too narrow …"
- Specialized tuning systems are compiler component



Intelligent Compiler

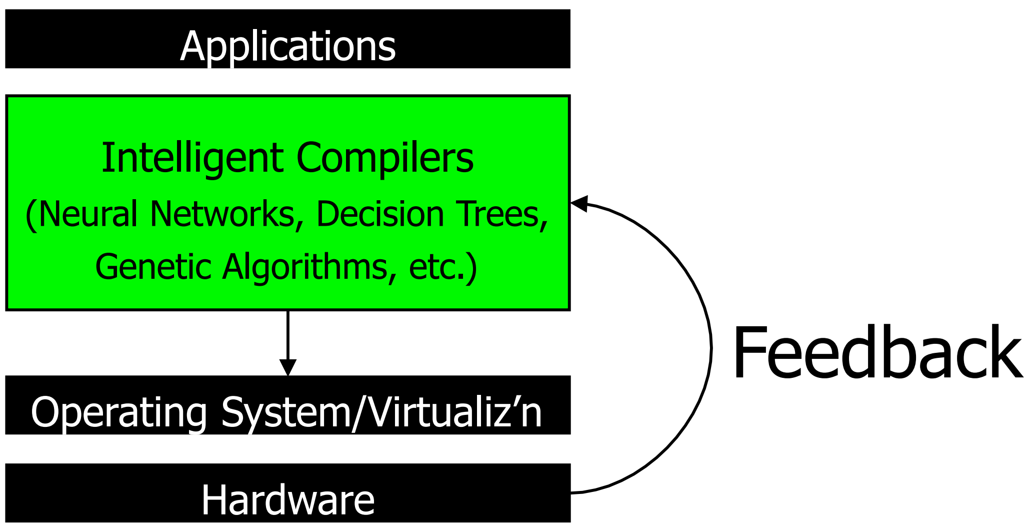
Traditional Compilers

- “One size fits all” approach
- Tuned for average performance
- Aggressive opts often turned **off**
- Target hard to model analytically



Proposed Solution

- *Intelligent* Compilers
  - Use Machine Learning
- Learn to optimize
  - Specialized to each Application/Data/Hardware



Intelligence in a compiler

- Global
  - Controlling compiler flags [CGO 2007]
- Local
  - Individual methods [OOPSLA 2006]
  - Individual loop bodies [PLDI 2008]
- Individual Optimization Heuristic
  - How and When to Perform Instruction Scheduling [NIPS 1997, PLDI 2005]

http://www.cis.udel.edu/~cavazos

Overall Approach

- Training of Model
  - Generate training data
  - Automatically construct a heuristic
  - Can be expensive, but can be done offline
- Testing of Model
  - During Compilation
    - Extract ***features***
    - Model outputs probability distribution
    - Generate optimizations from distribution
- Offline versus online learning

Outline

- **Using Performance Counters**
- Intelligent Polyhedral Search
- Method-Specific Compilation

Putting Perf Counters to Use

- Important aspects of
  pr
  o
  grams captured with performance counters
- Automatically construct model (Offline)
  - Map performance counters to good opts
- Model predicts optimizations to apply
  - Uses performance counter characterization

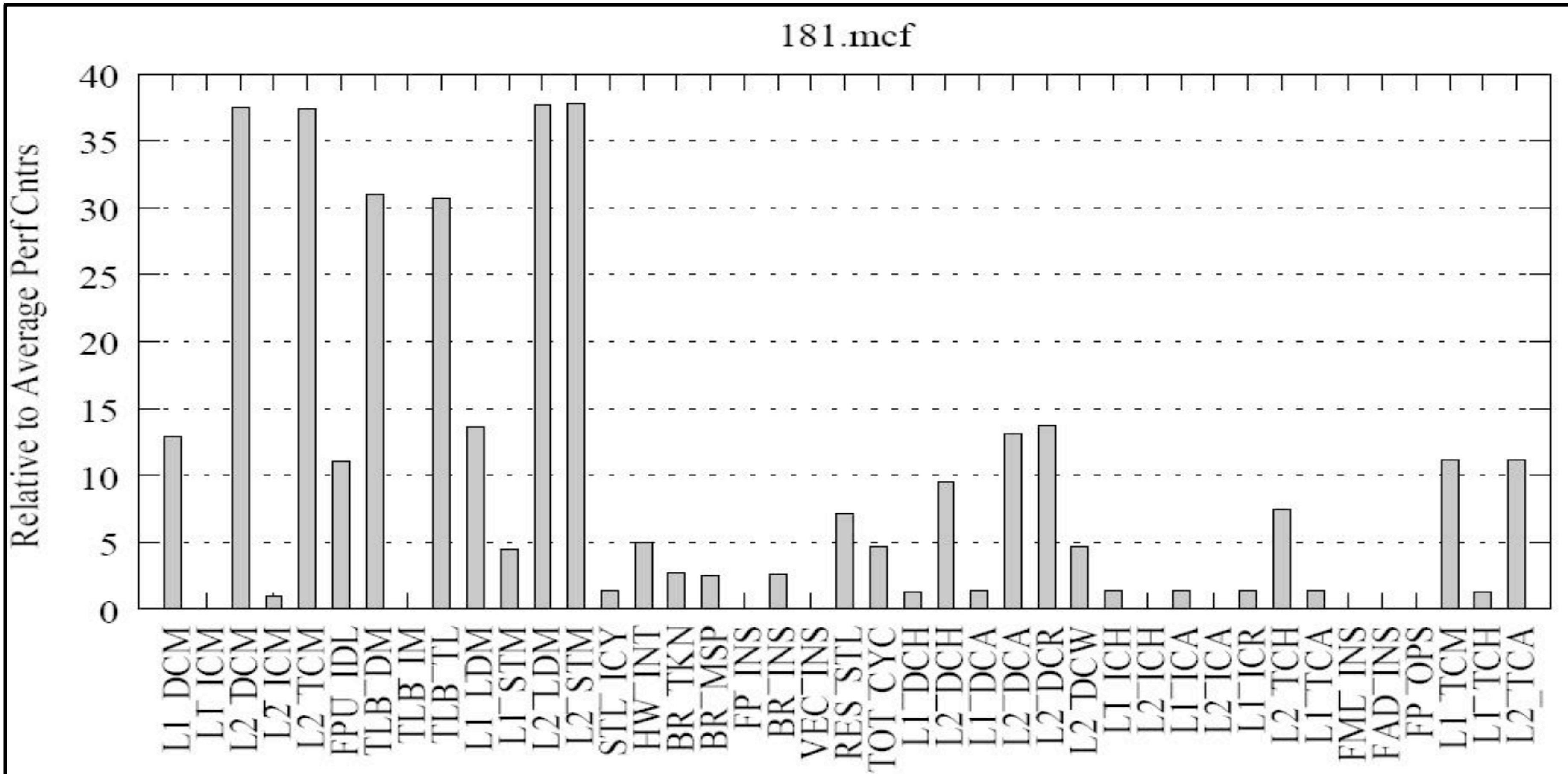
Performance Counters

- Many performance counters available
- Examples:

Mnemonic	Description	Avg Values
FPU_IDL	(Floating Unit Idle)	0.473
VEC_INS	(Vector Instructions)	0.017
BR_INS	(Branch Instructions)	0.047
L1_ICH	(L1 Icache Hits)	0.0006

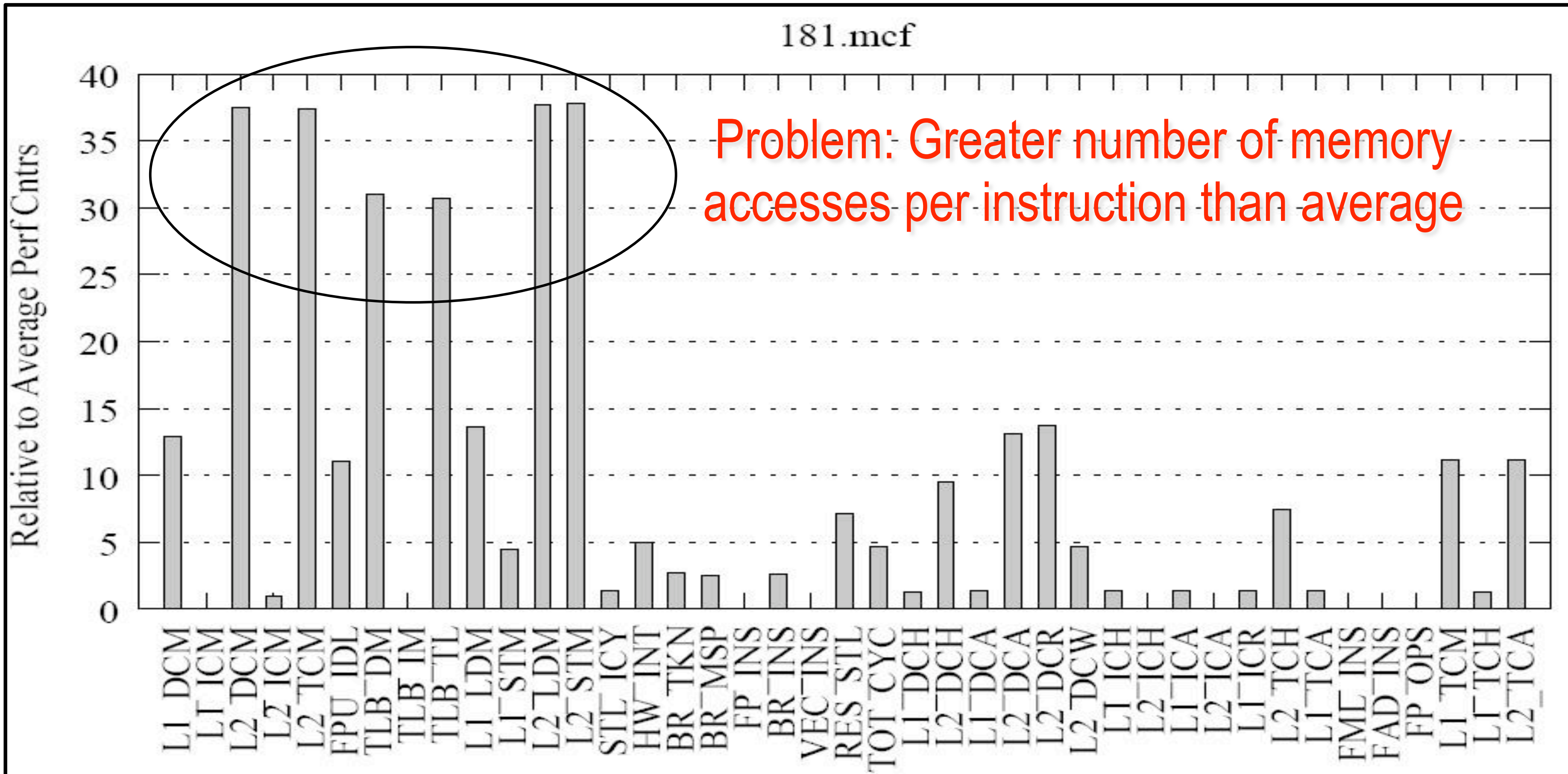
Characterization of 181.mcf

- Perf cntrs relative to 4 benchmark suites



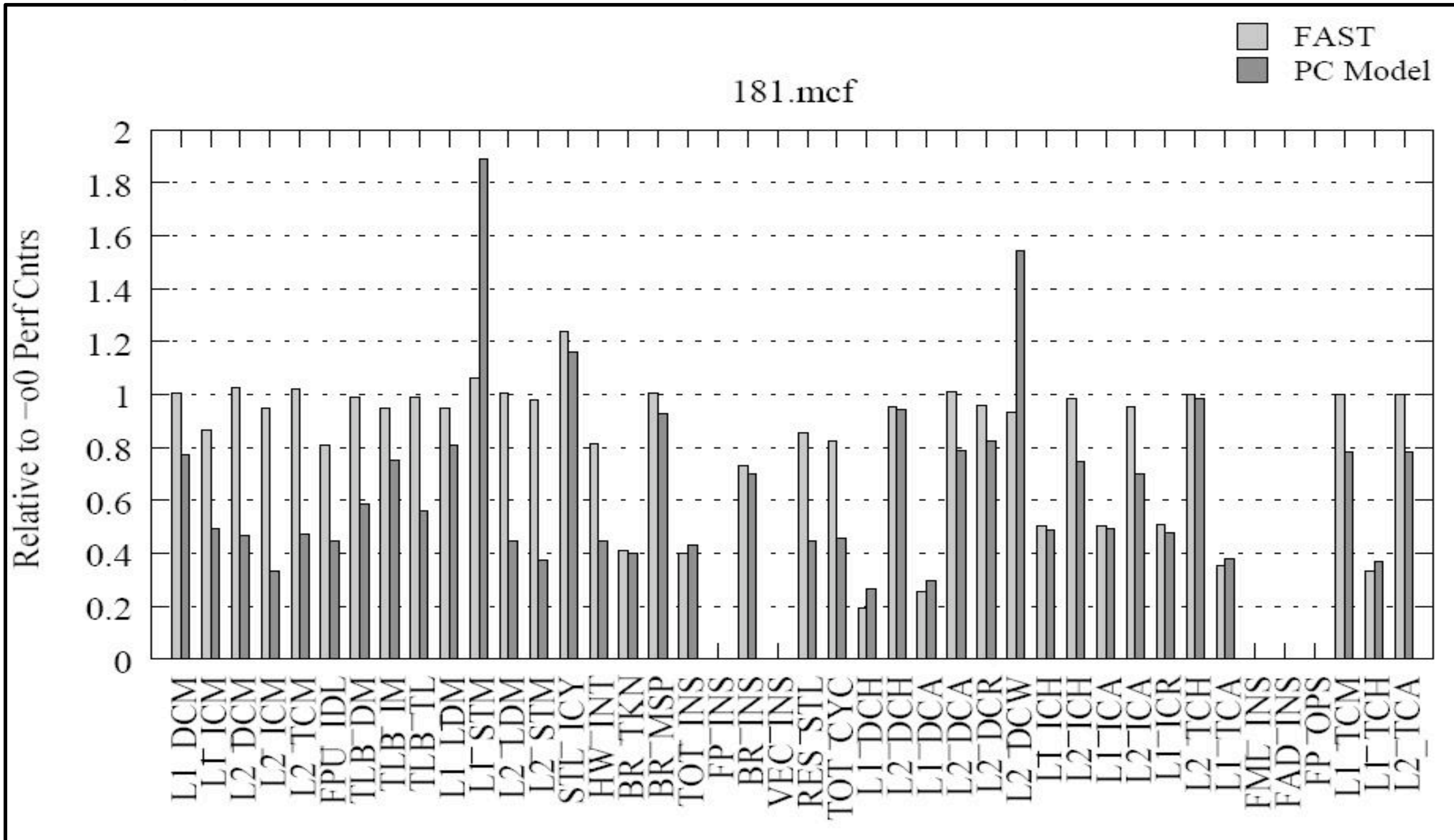
Characterization of 181.mcf

- Perf cntrs relative to 4 benchmark suites



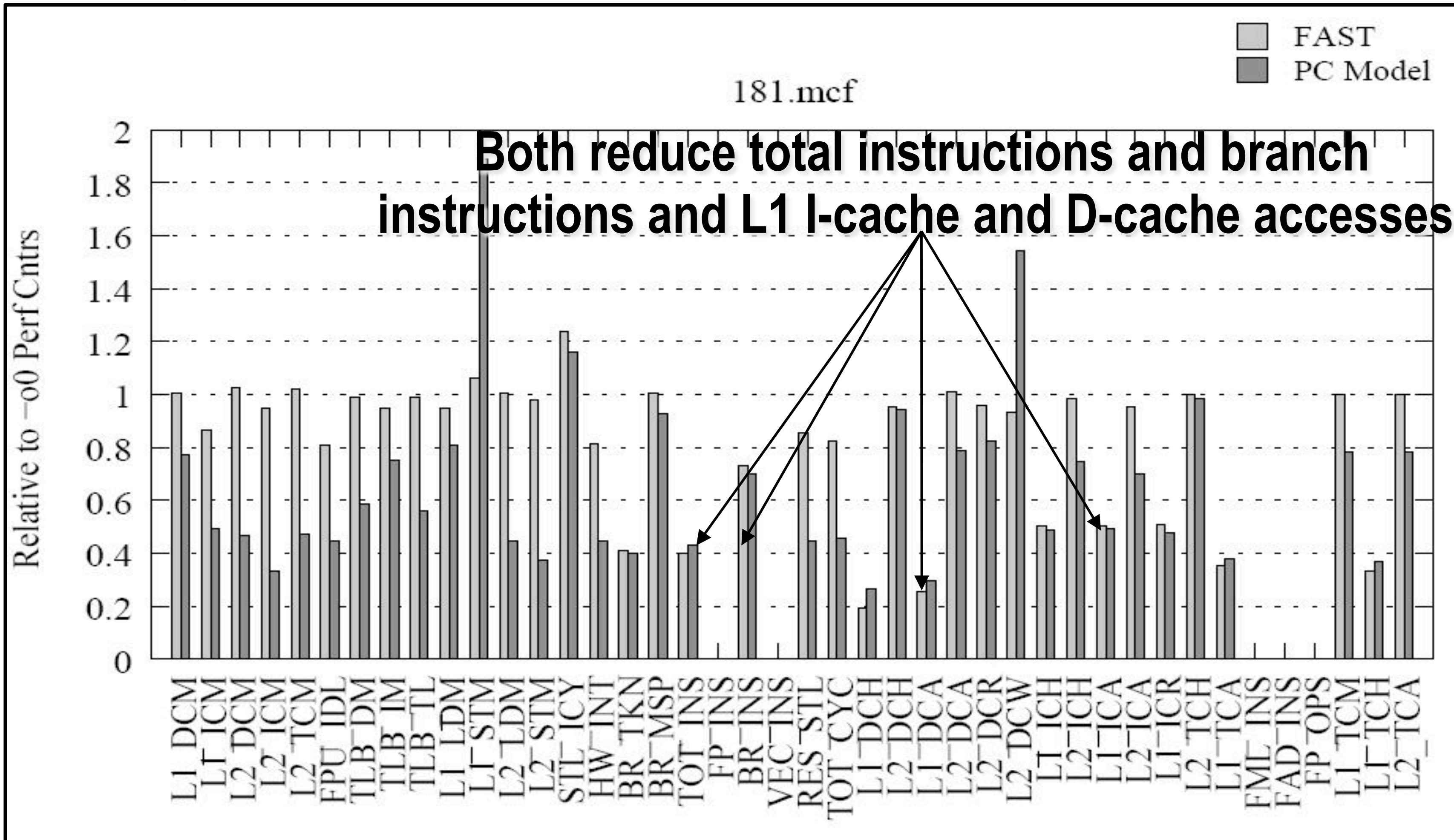
Characterization of 181.mcf

- Using -Ofast and search with a Model.



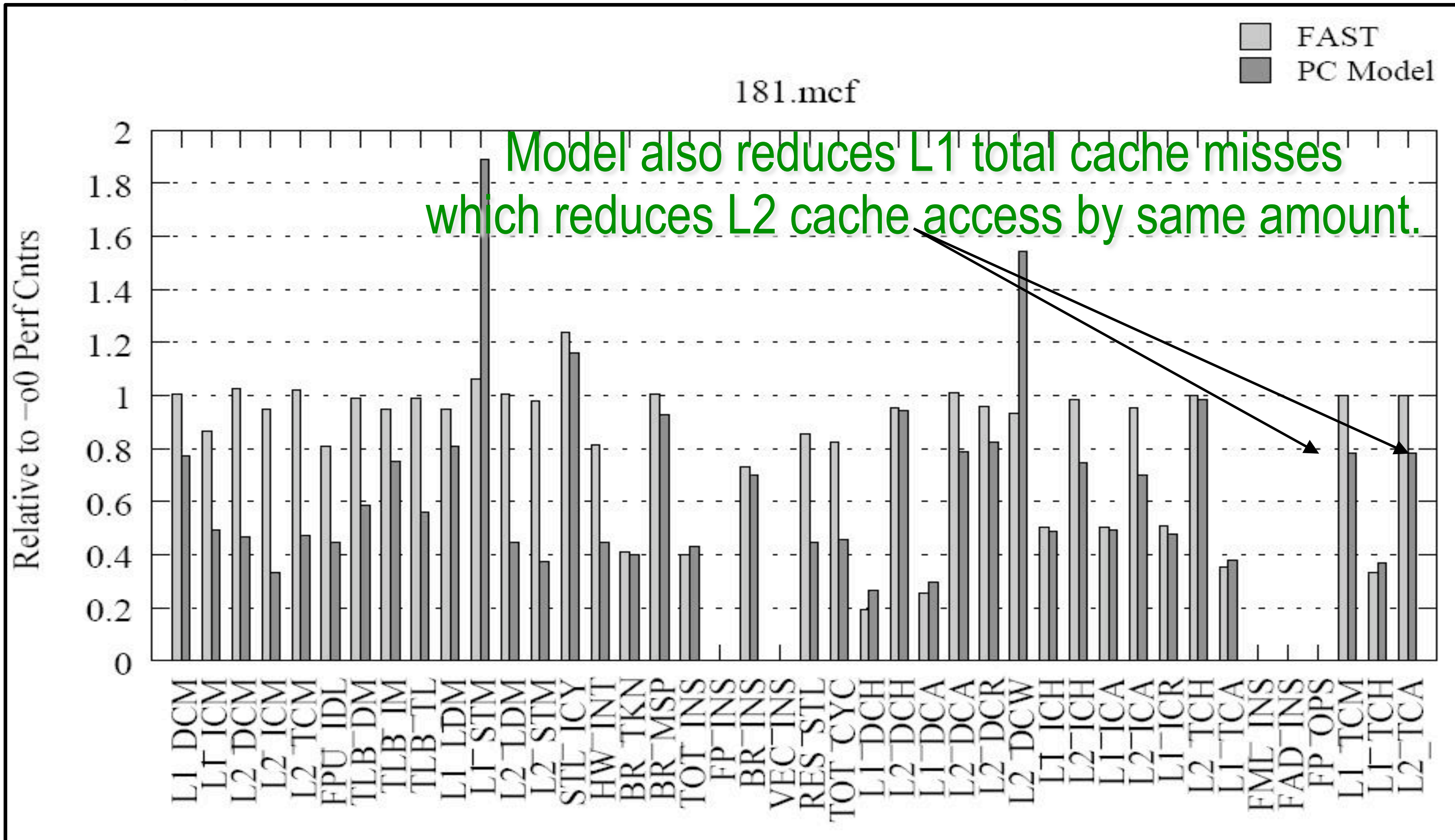
Characterization of 181.mcf

- Using -Ofast and search with a Model.

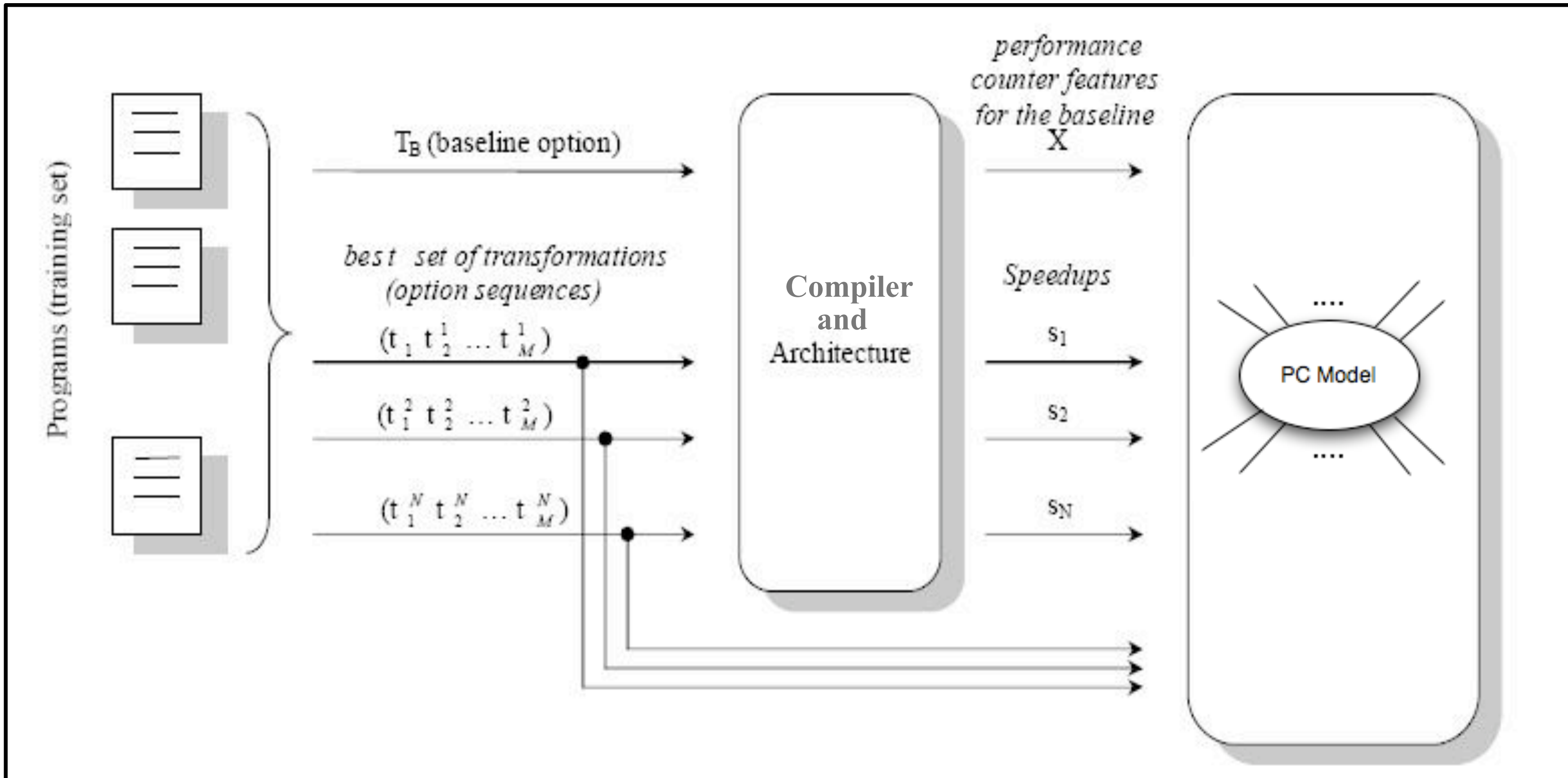


Characterization of 181.mcf

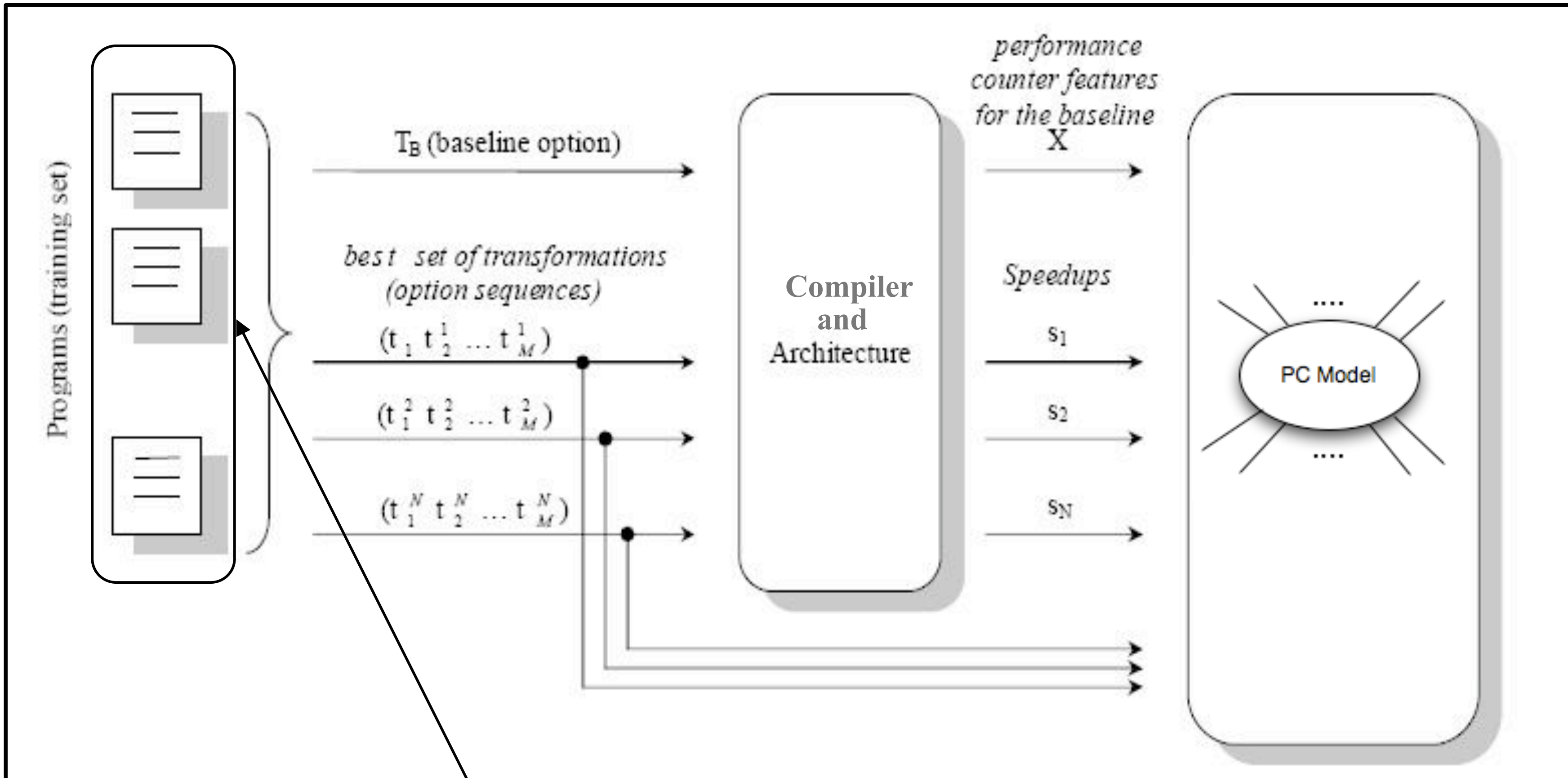
- Using -Ofast and search with a Model.



Training PC Model

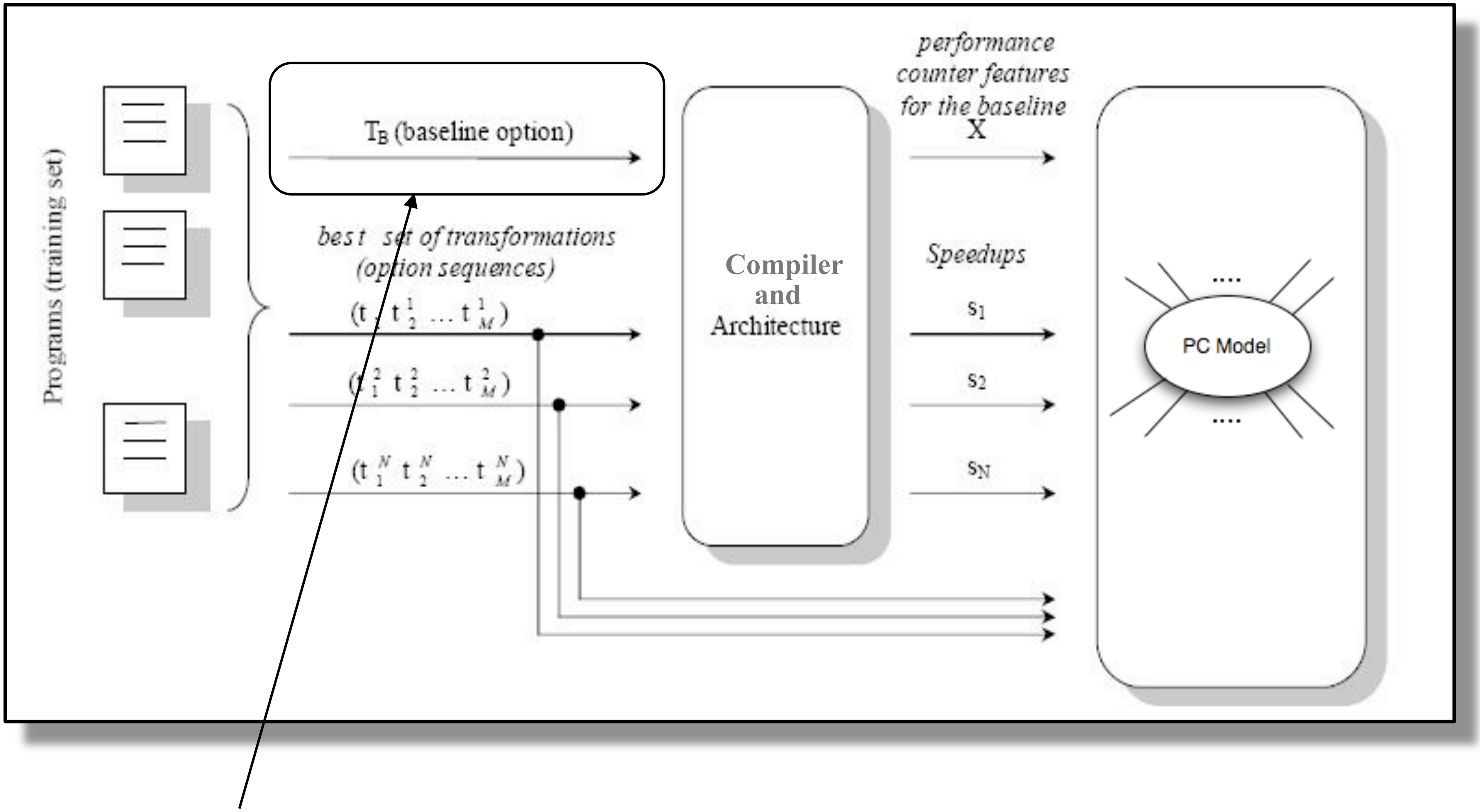


Training PC Model



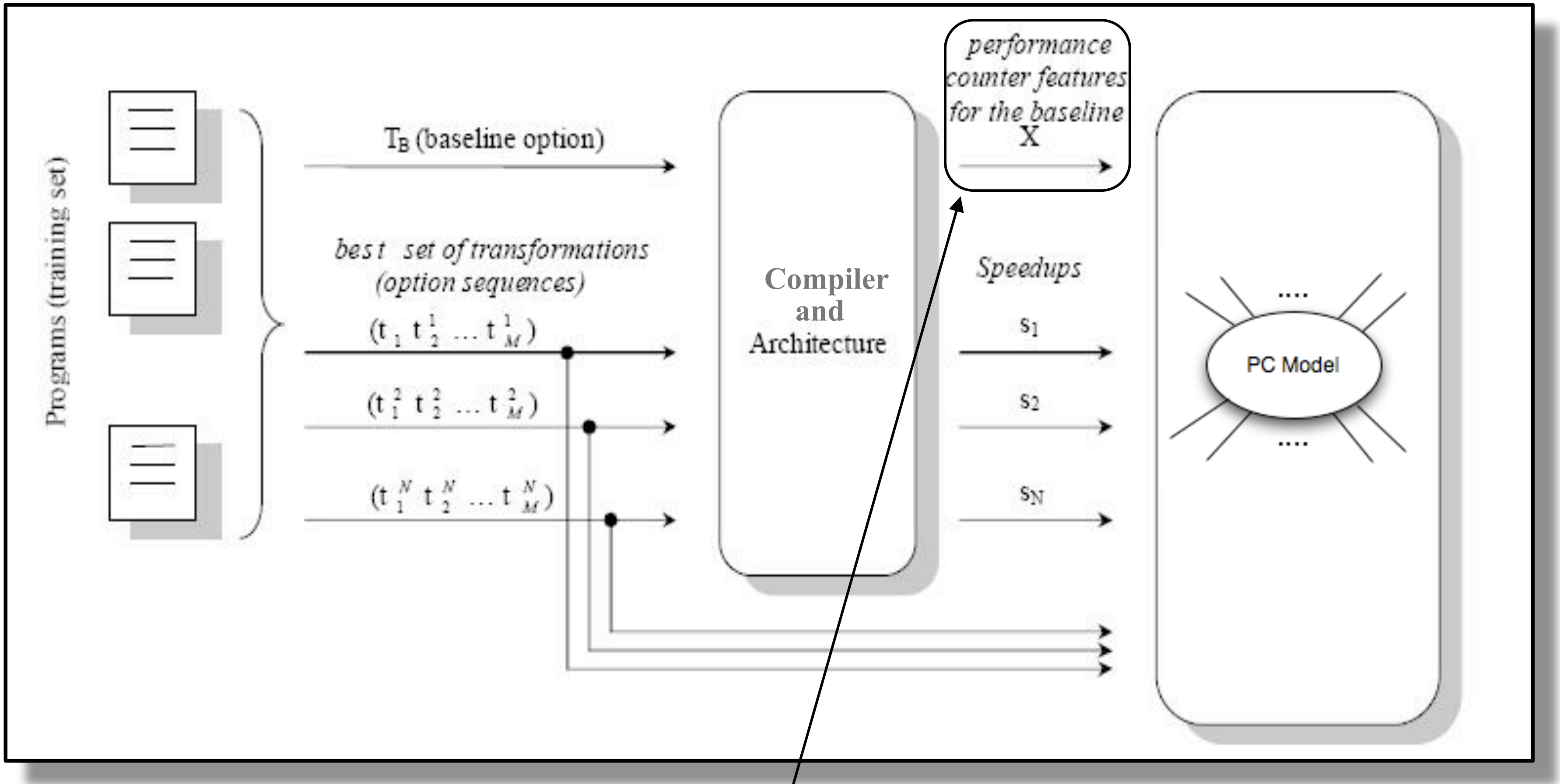
Programs to train model (different from test program).

Training PC Model



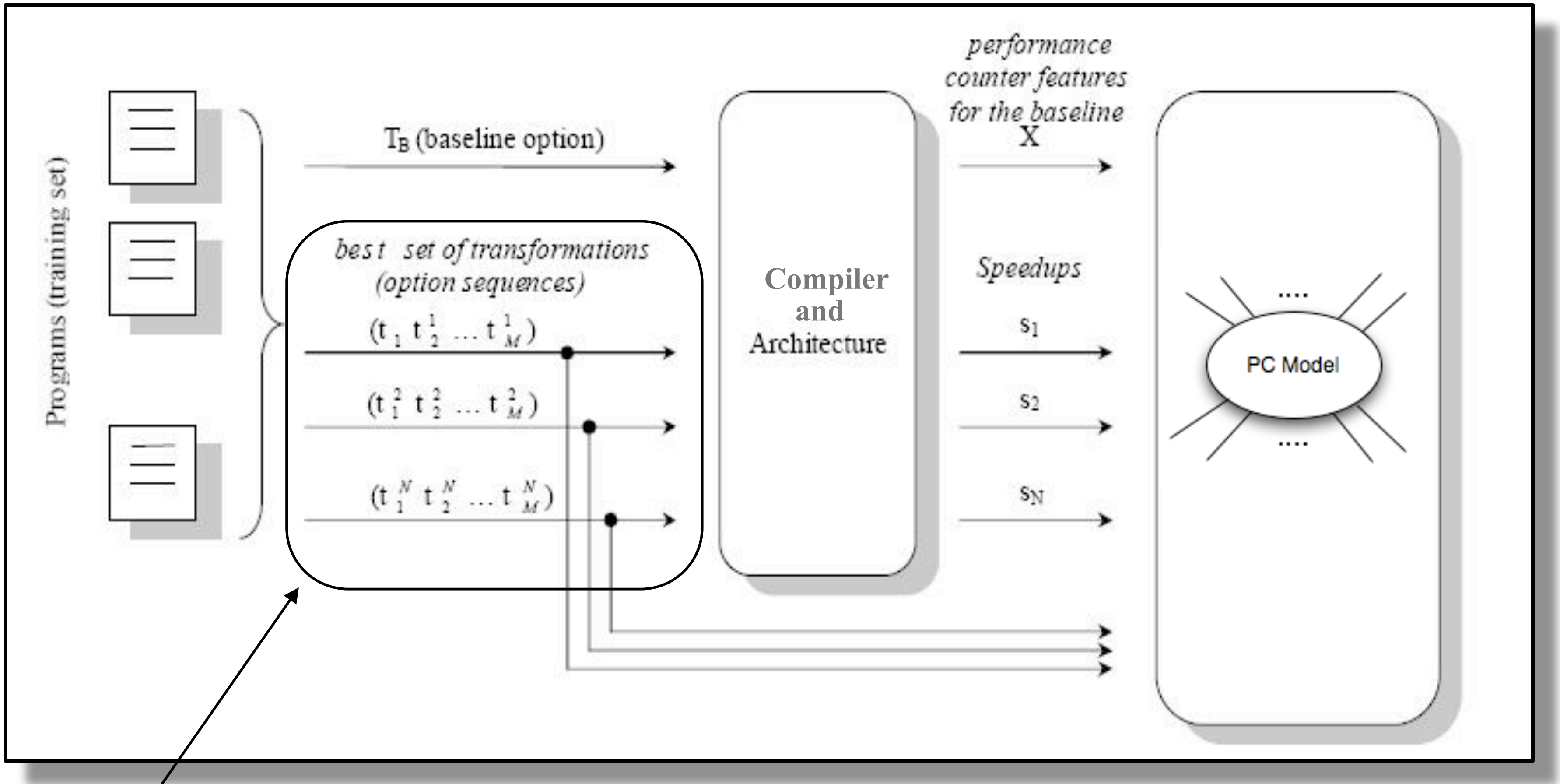
Baseline runs to capture performance counter values.

Training PC Model



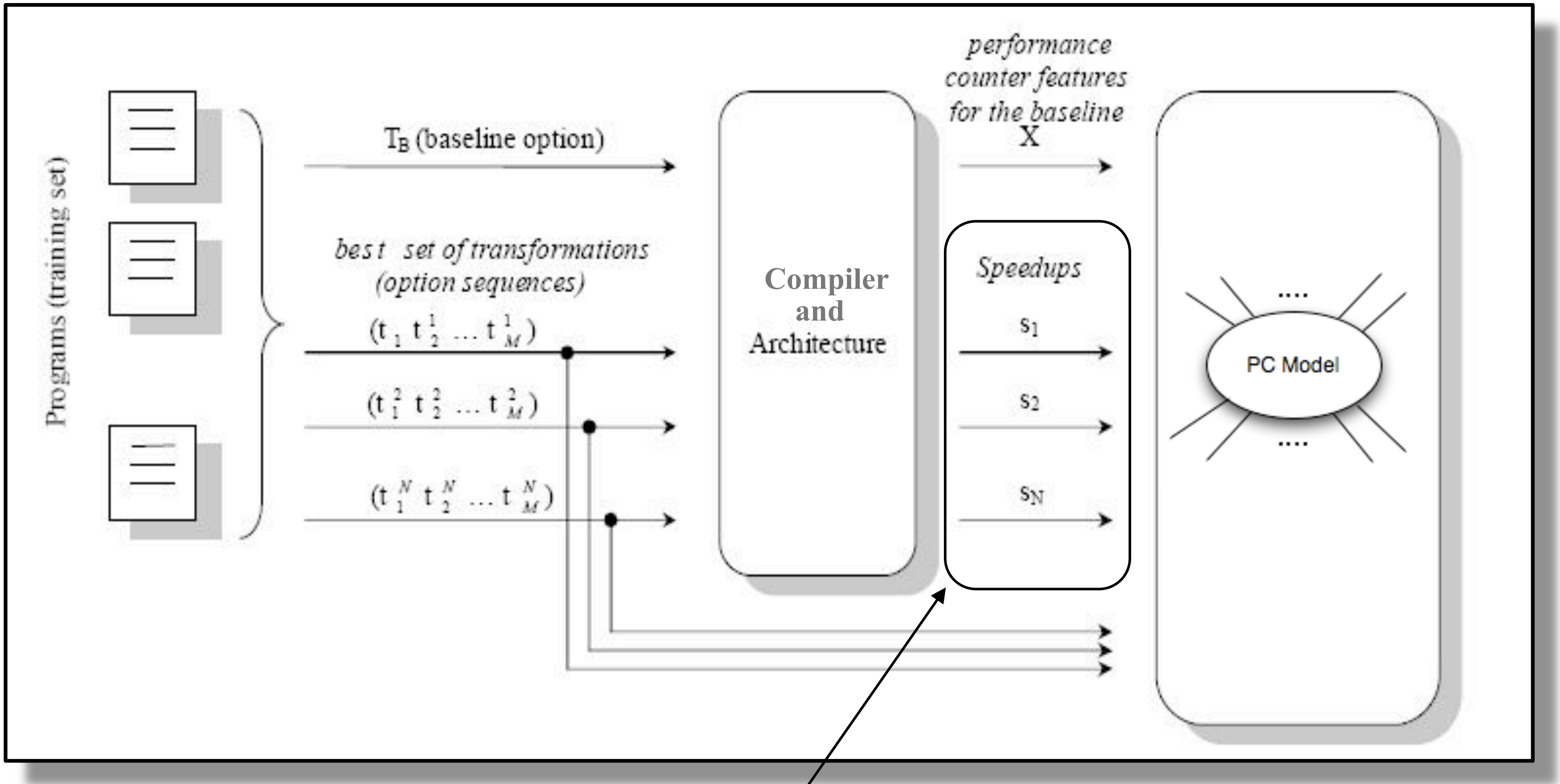
Obtain performance counter values for a benchmark.

Training PC Model



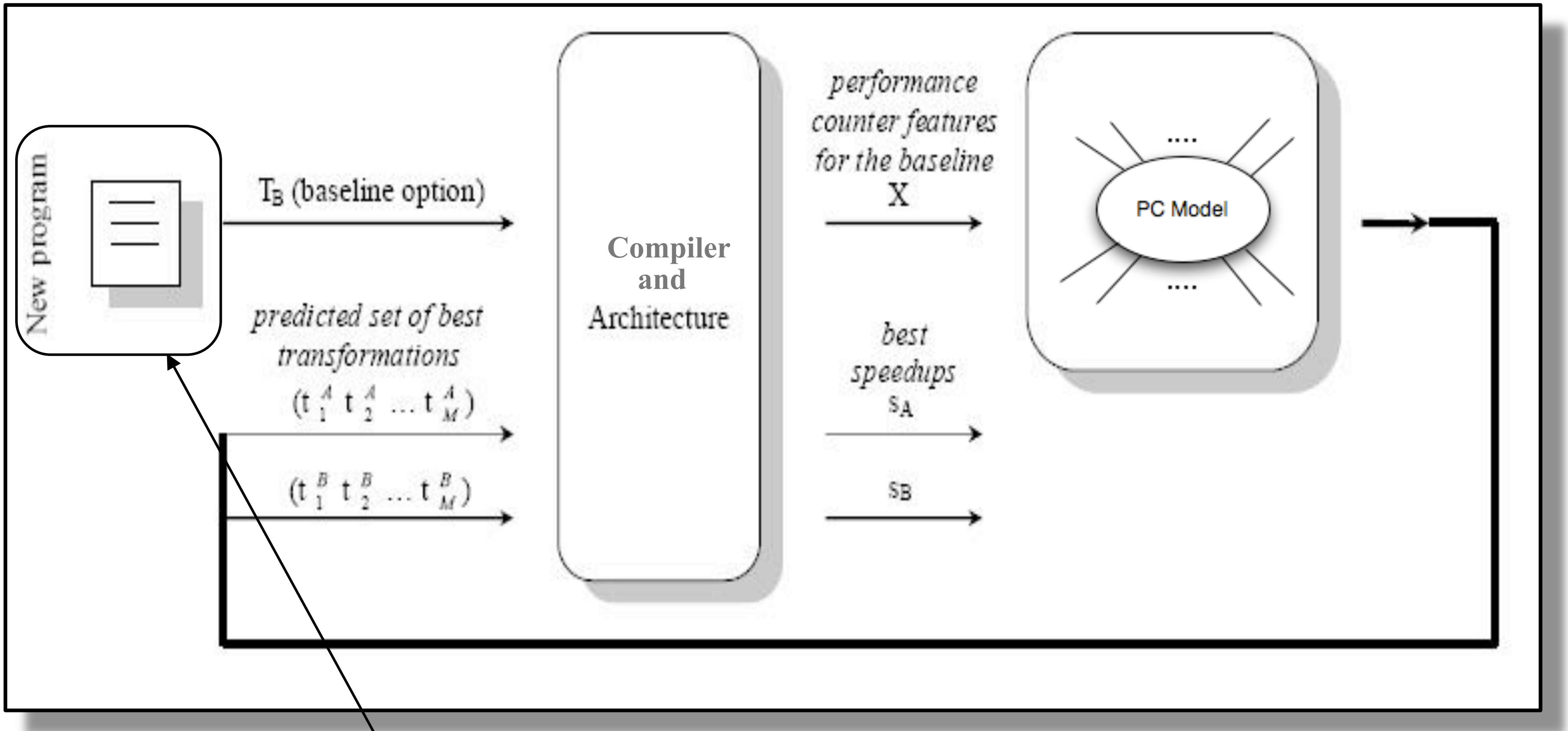
Best optimizations runs to get speedup values.

Training PC Model



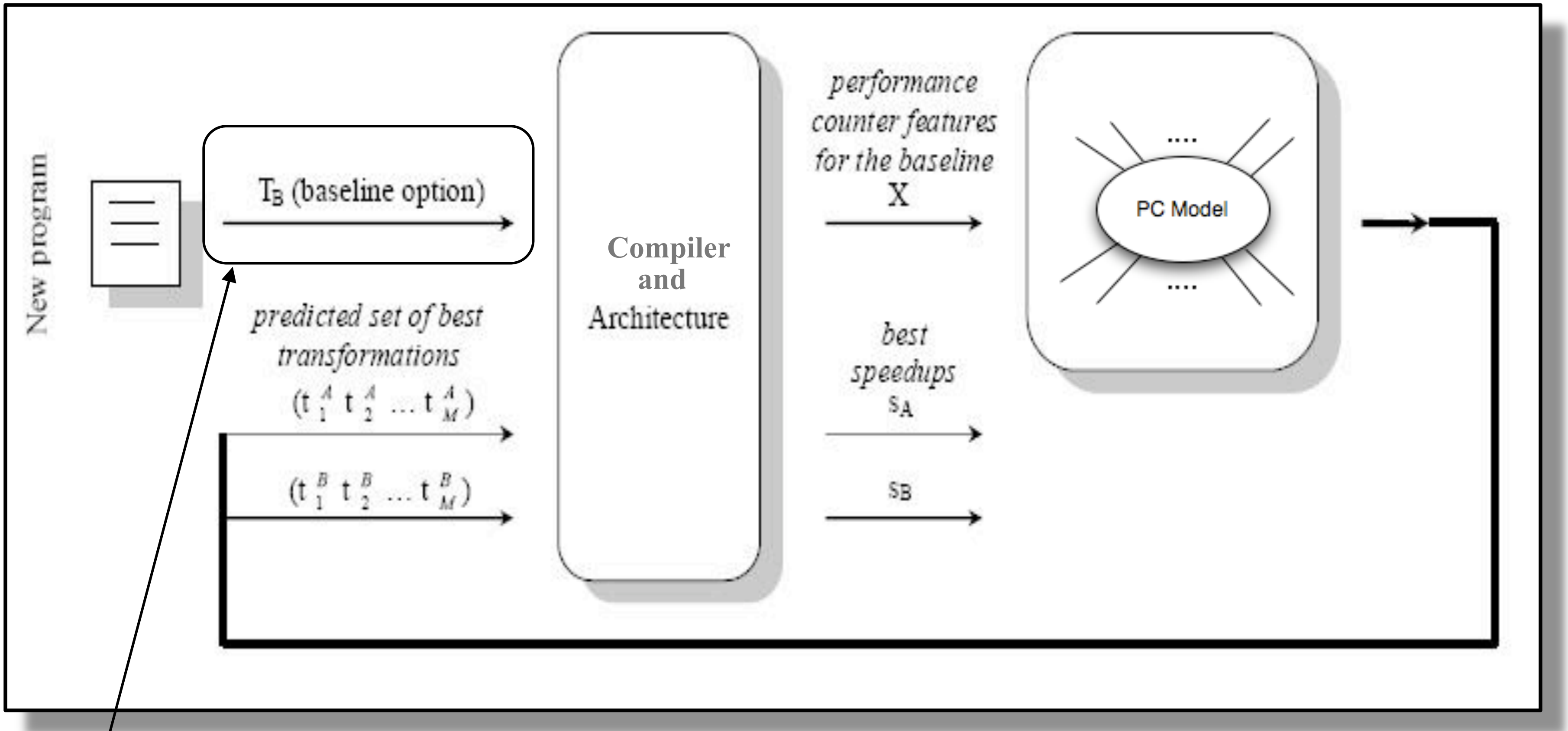
Best optimizations runs to get speedup values.

Using PC Model



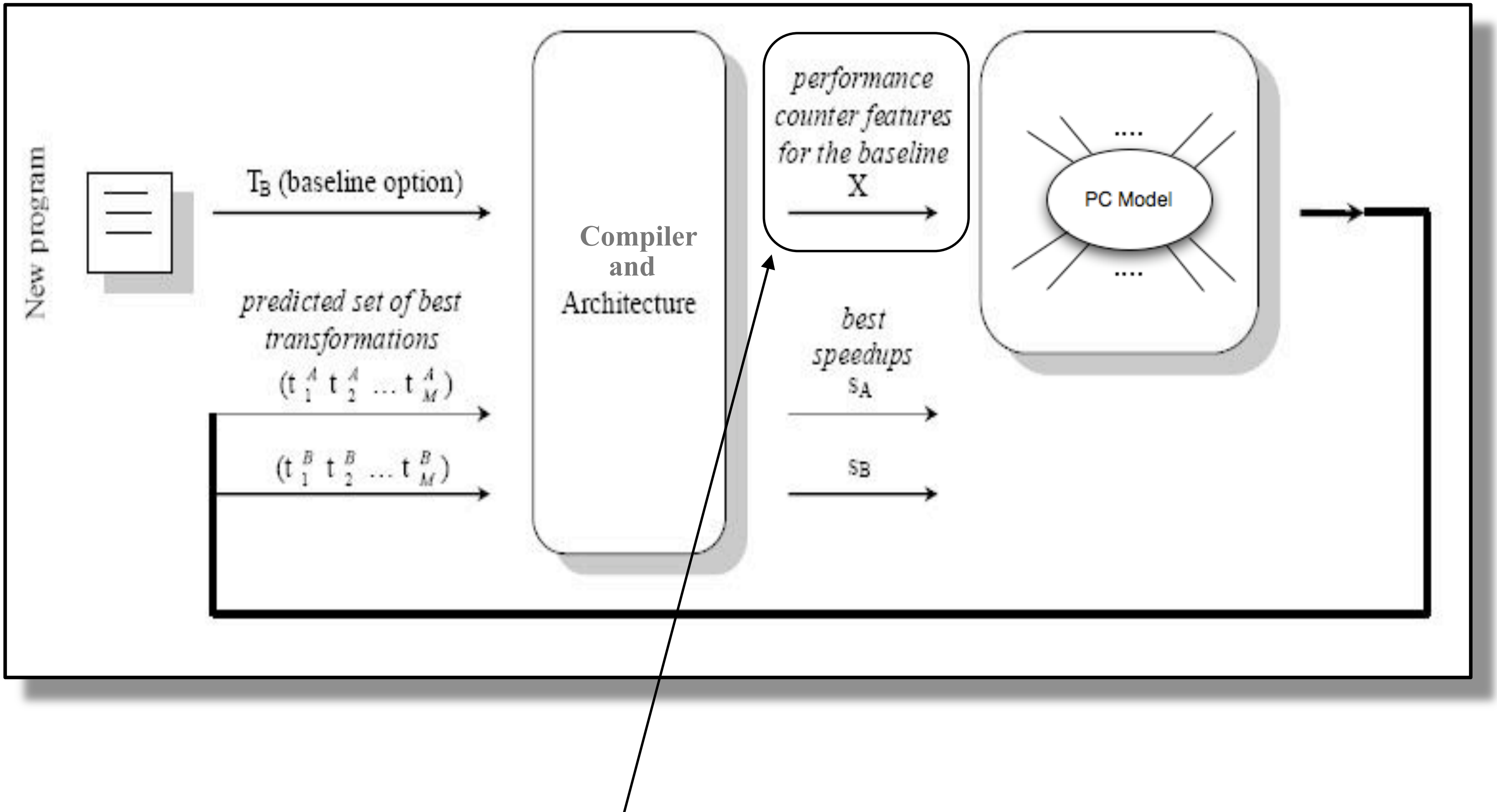
New program interested in obtaining good performance.

Using PC Model



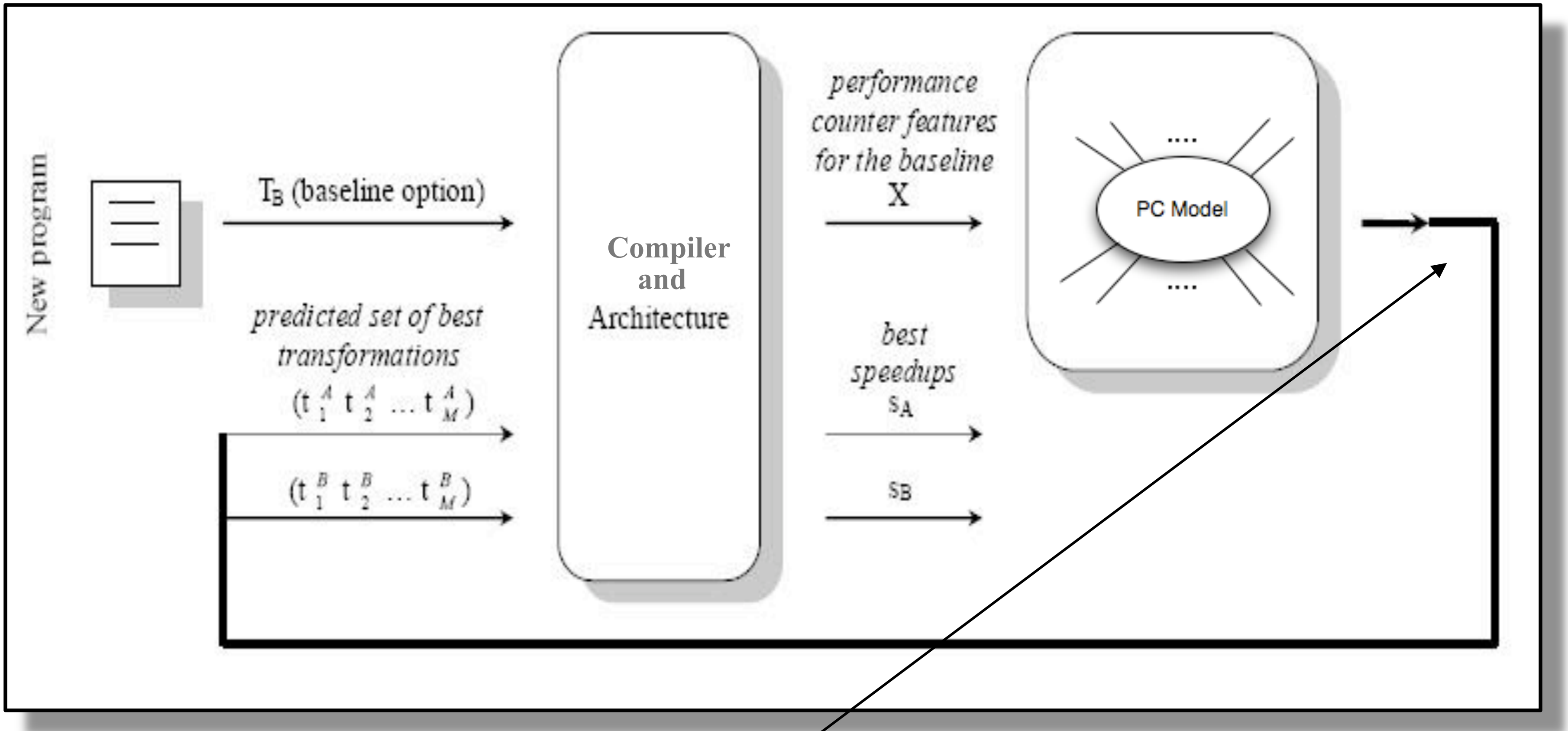
Baseline run to capture performance counter values.

Using PC Model



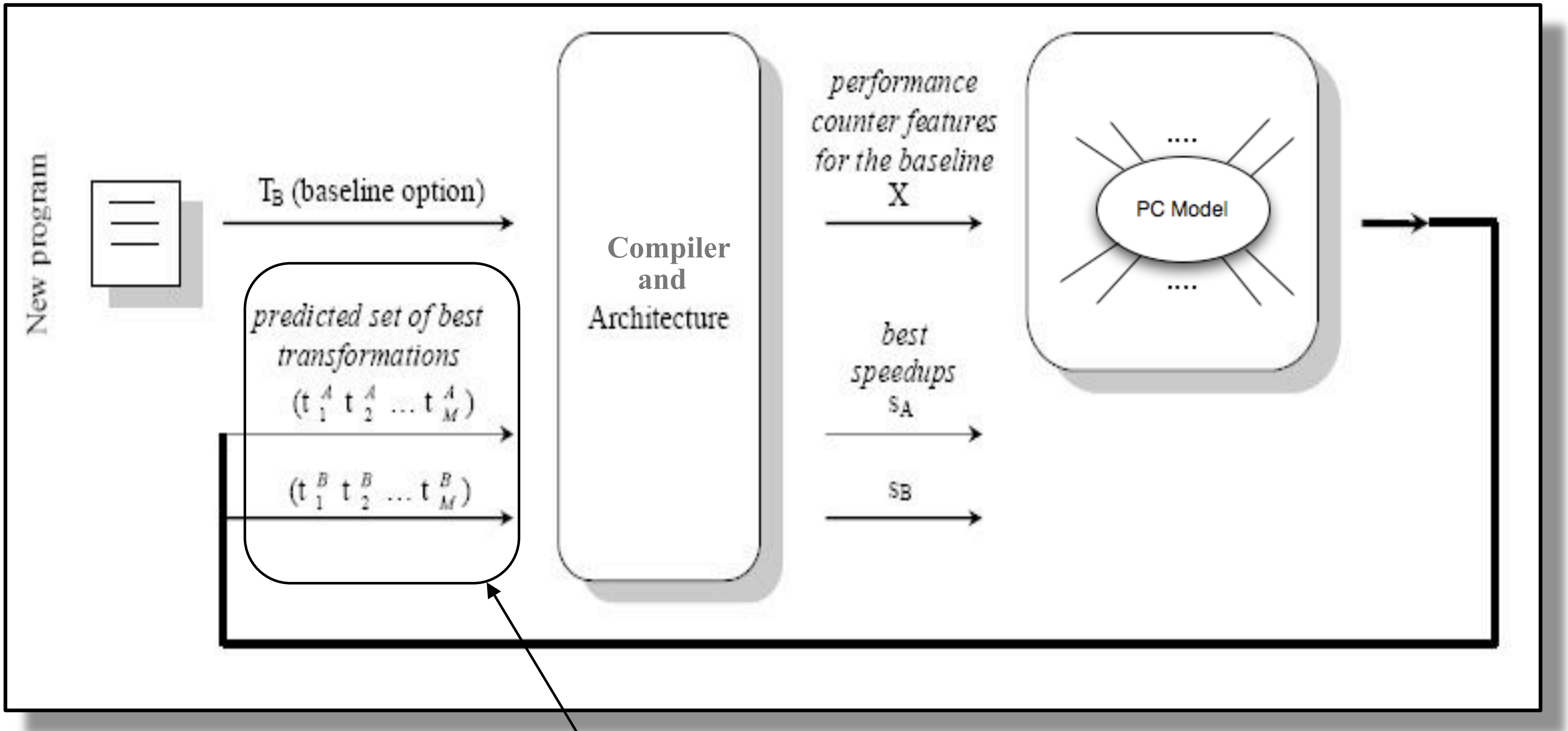
Feed performance counter values to model.

Using PC Model



Model outputs a distribution that is use to generate sequences

Using PC Model



Optimization sequences drawn from distribution.

PC Model

- Trained on data from Random Search
  - 500 evaluations for each benchmark
- Leave-one-out cross validation
  - Training on N-1 benchmarks
  - Test on Nth benchmark
- Logistic Regression

Logistic Regression

- Variation of ordinary regression
- Inputs
  - Continuous, discrete, or a mix
  - 60 performance counters
    - All normalized to cycles executed
- Ouputs
  - Restricted to two values **(0,1)**
  - Probability an optimization is beneficial

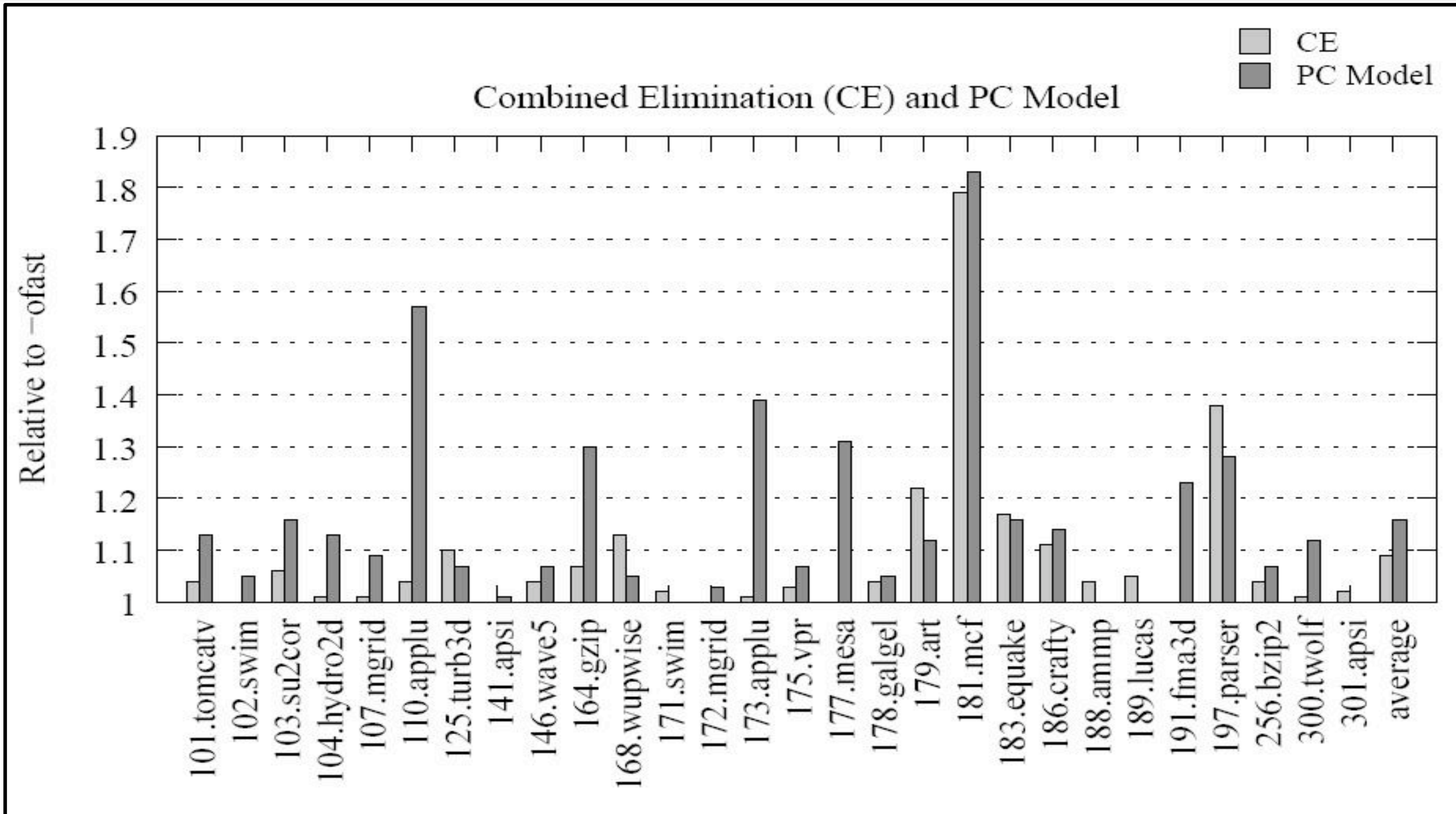
Experimental Methodology

- PathScale compiler
  - Compare to highest optimization level
  - 121 compiler flags
- AMD Athlon processor
  - ***Real*** machine; Not simulation
- 57 benchmarks
  - SPEC (INT 95, 2000), MiBench, Polyhedral

Evaluated Search Strategies

- Combined Elimination [CGO 2006]
  - Pure search technique
    - Evaluate optimizations one at a time
    - Eliminate negative optimizations in one go
  - Out-performed other pure search techniques
- PC Model

PCModel/CE *(SPEC INT 95/SPEC 2000)*

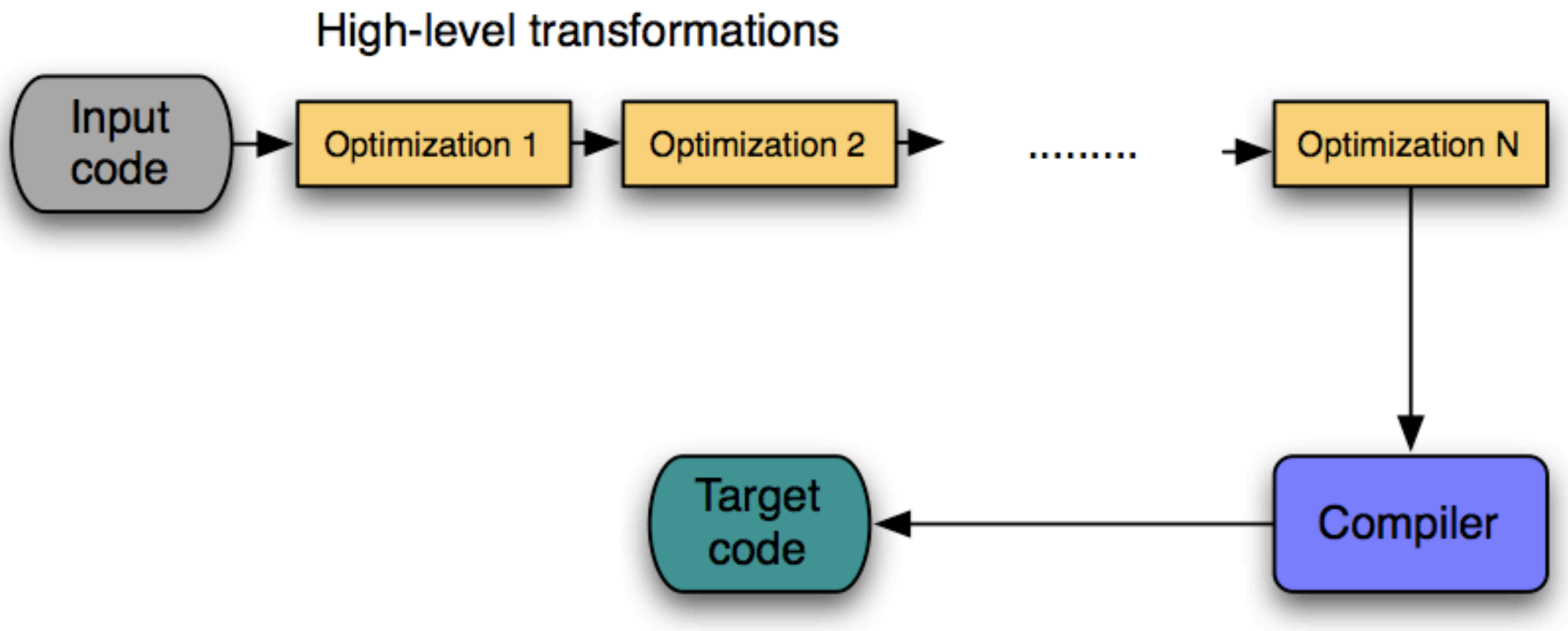


Obtained > 25% on 7 benchmarks and 17% over highest opt.

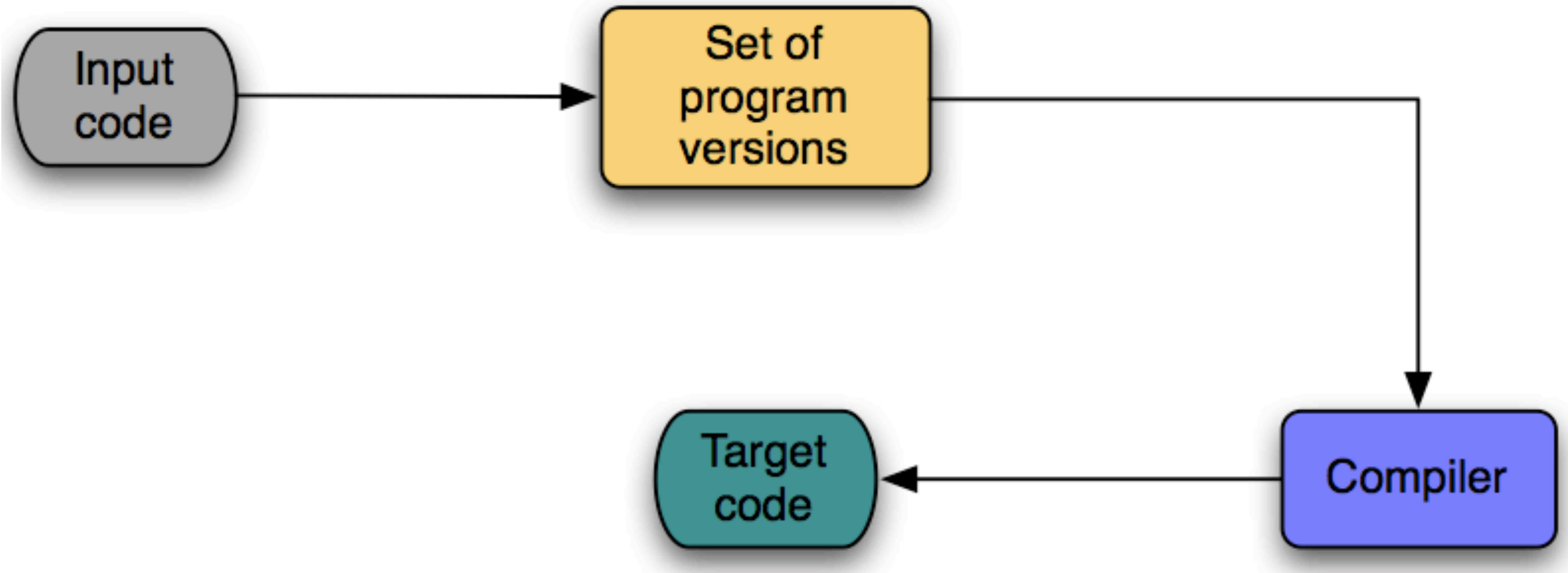
Two Additional Approaches

- Intelligent Polyhedral Search [PLDI 2008]
- Method-Specific Compilation [OOPSLA 2006]

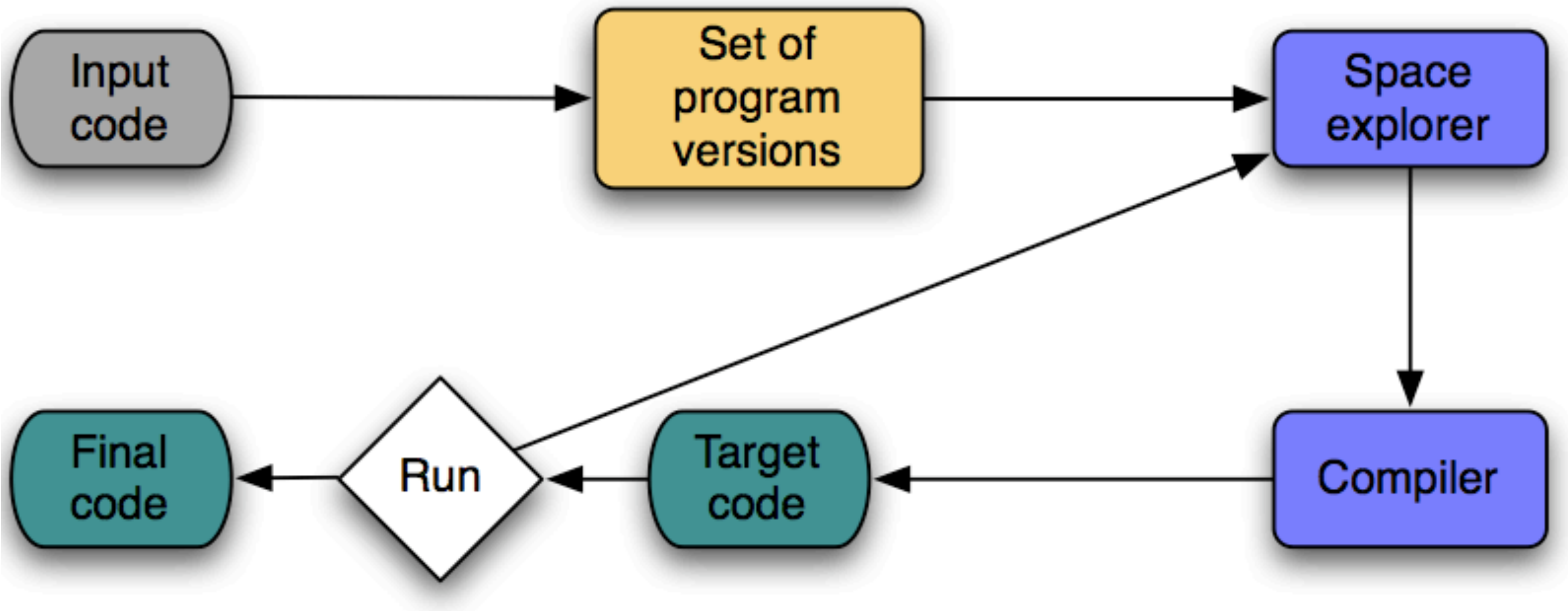
Intelligent Polyhedral Search



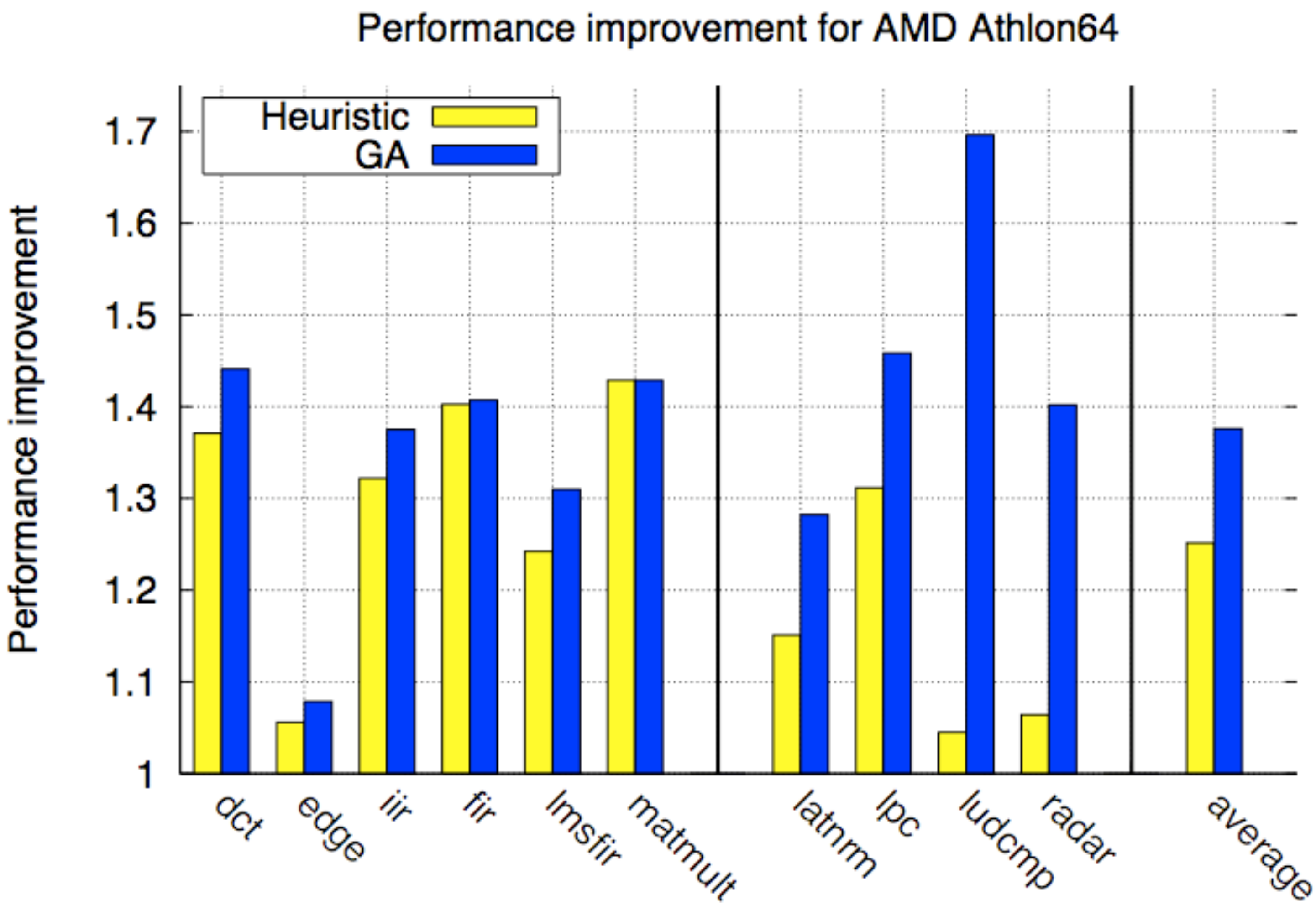
Intelligent Polyhedral Search



Intelligent Polyhedral Search



Intelligent Polyhedral Results



Relative to gcc -O3 -ftree-vectorize -msse2

Method-Specific Compilation

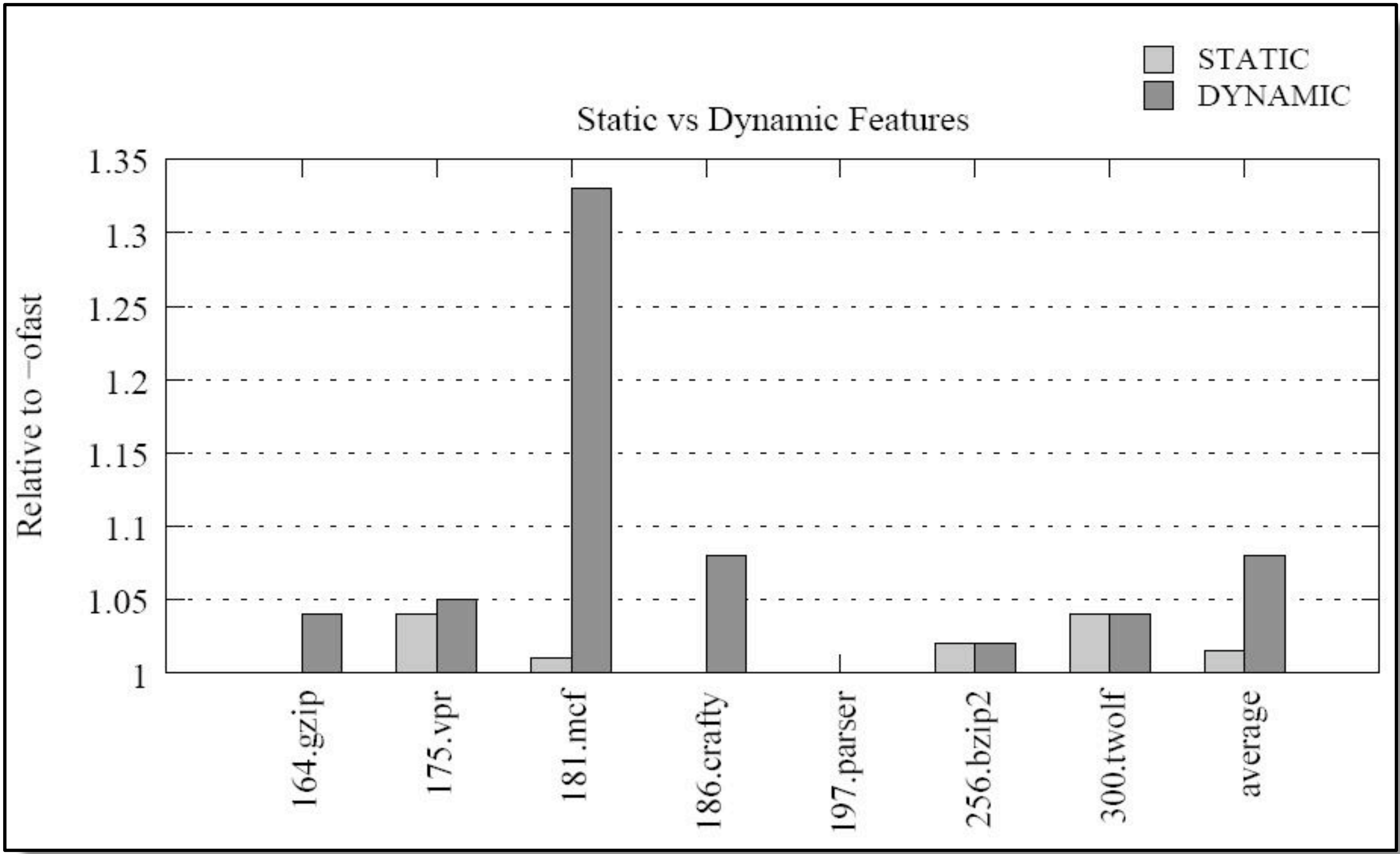
- Integrate Machine Learning into a Java JIT compiler
- Use simple code properties
  - Extracted from one linear pass of bytecodes
- Model controls up to 20 optimizations
- Outperforms hand-tuned heuristic
  - Up to 29% SPEC JVM98
  - Up to 33% DaCapo+

Conclusions

- Using **performance counters**
  - Out-performs production compiler in few evaluations
- **Intelligently traverses** Polyhedral Space
- Using **code characteristics**
  - Can outperform hand-tuned heuristic
  - Opts applied only when beneficial

Backup Slides

Static vs Dynamic Features



Most Informative Features

Most Informative Performance Counters	
1. L1 Cache Accesses	9. L2 Dcache Hits
2. L1 Dcache Hits	10. L2 Cache Accesses
3. TLB Data Misses	11. L1 Dcache Accesses
4. Branch Instructions	12. Hardware Interrupts
5. Resource Stalls	13. L2 Cache Hits
6. Total Cycles	14. L1 Cache Hits
7. L2 Icache Hits	15. Branch Misses
8. Vector Instructions	

Why is CE worse than RAND?

- Combined Elimination
  - Dependent on dimensions of space
  - Easily stuck in local minima
- RAND
  - Probabilistic technique
  - Depends on distribution of good points
  - Not susceptible to local minima

Note: CE may improve in space with many bad opts.

Program Characterization

- Characterizing large programs hard
- Performance counters effectively summarize program's dynamic behavior
- Previously* used static features [CGO 2006]
  - Does not work for whole program characterization