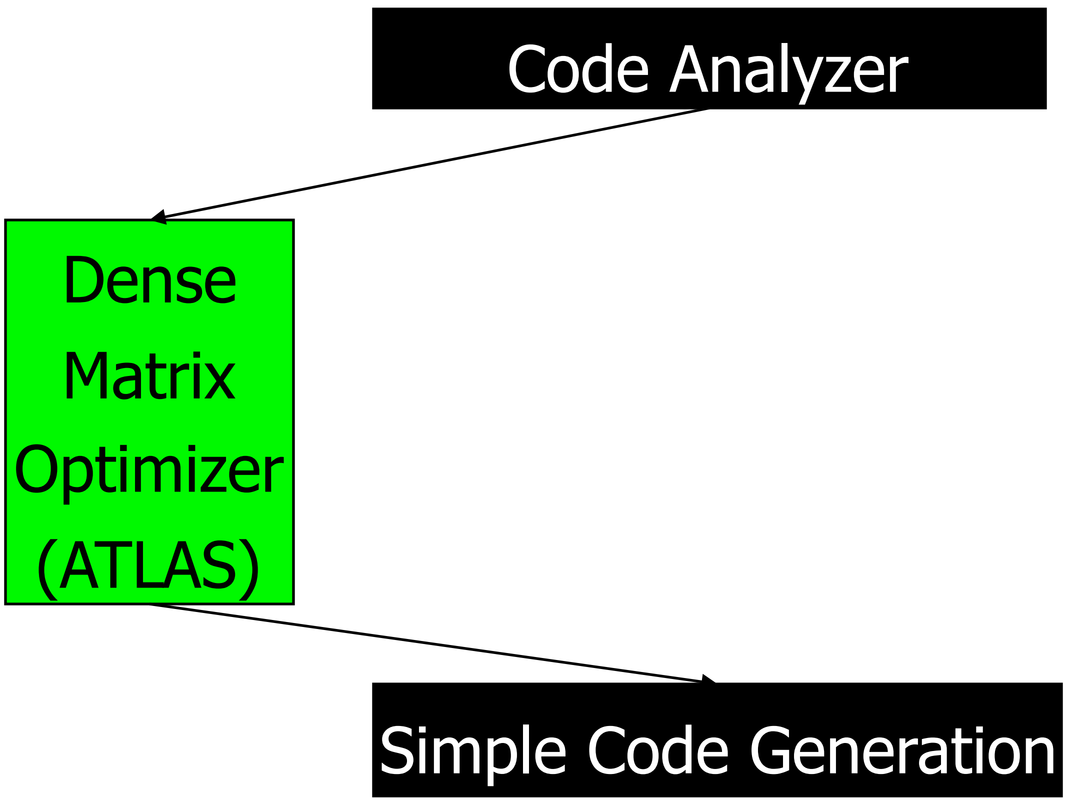
Intelligent Compilation

John Cavazos

Department of
Computer and Information Sciences
University of Delaware

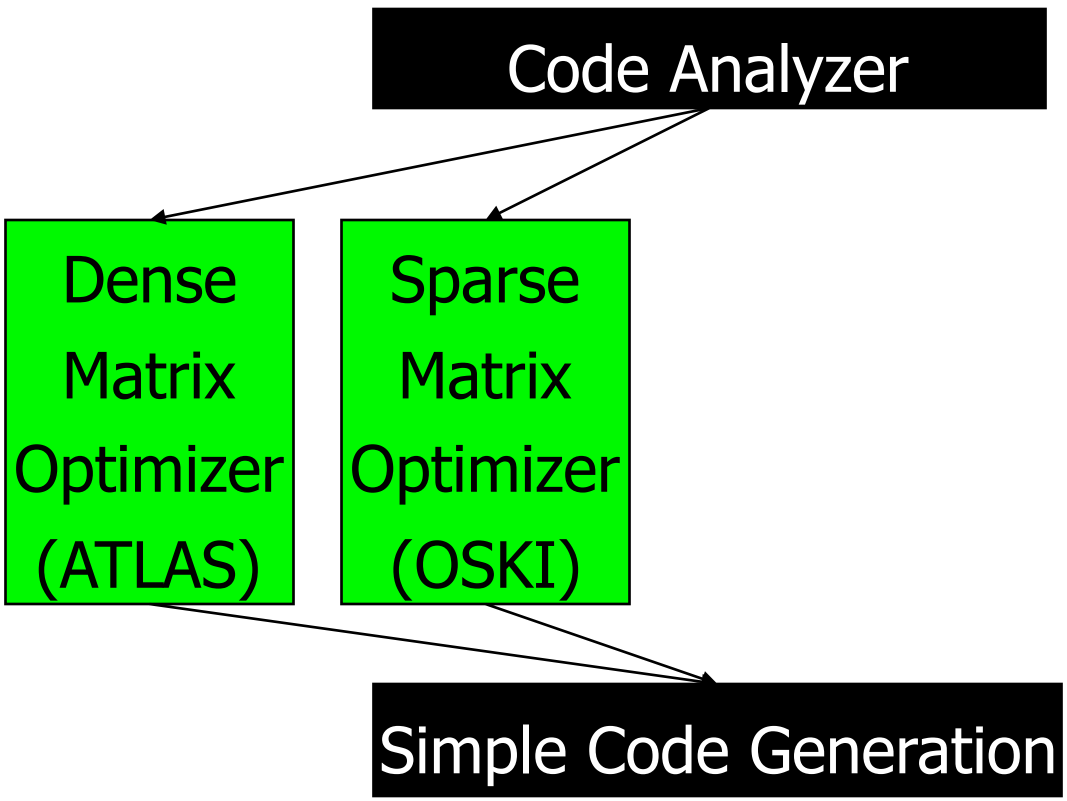
Autotuning and Compilers

- Proposition: Autotuning is a component of an Intelligent Compiler.



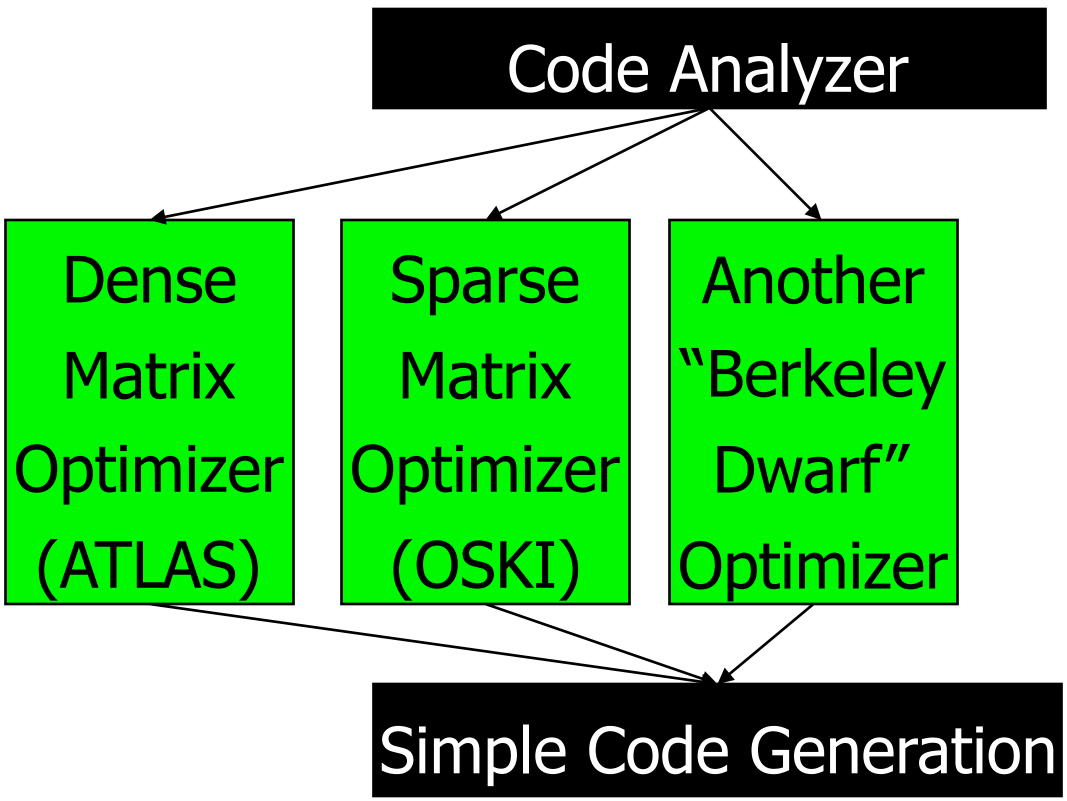
Autotuning and Compilers

- Proposition: Autotuning is a component of an Intelligent Compiler.



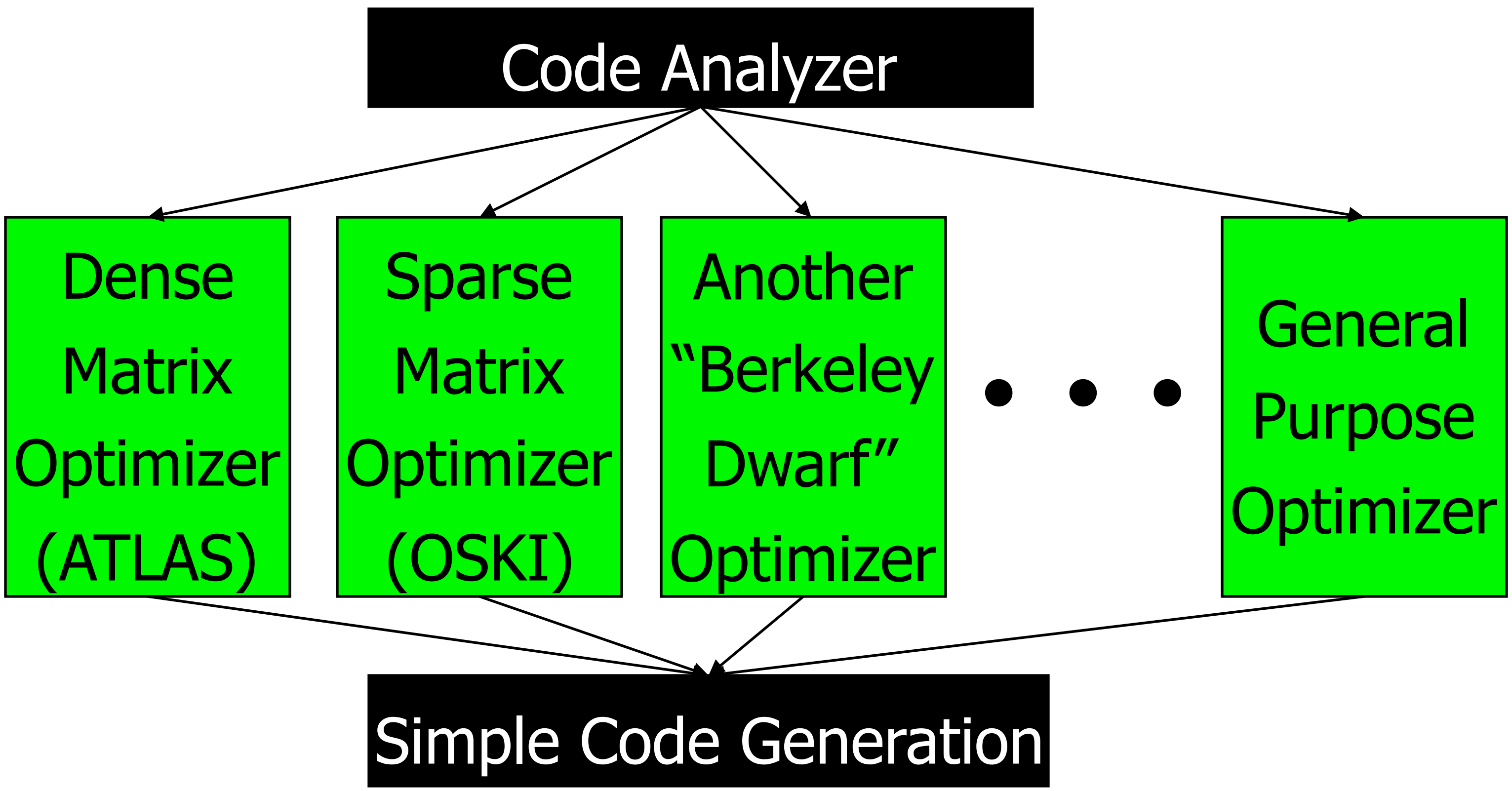
Autotuning and Compilers

- Proposition: Autotuning is a component of an Intelligent Compiler.



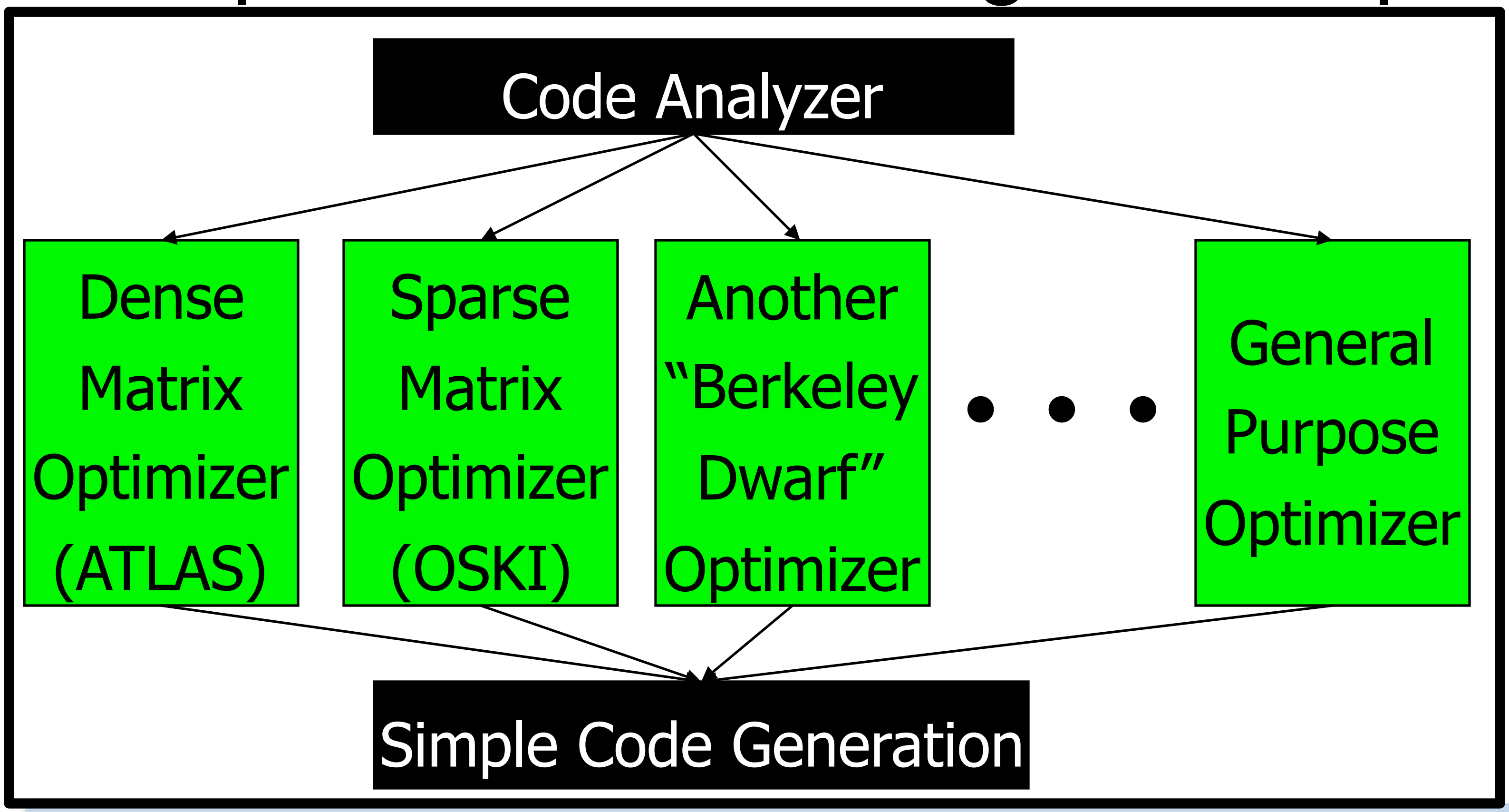
Autotuning and Compilers

- Proposition: Autotuning is a component of an Intelligent Compiler.



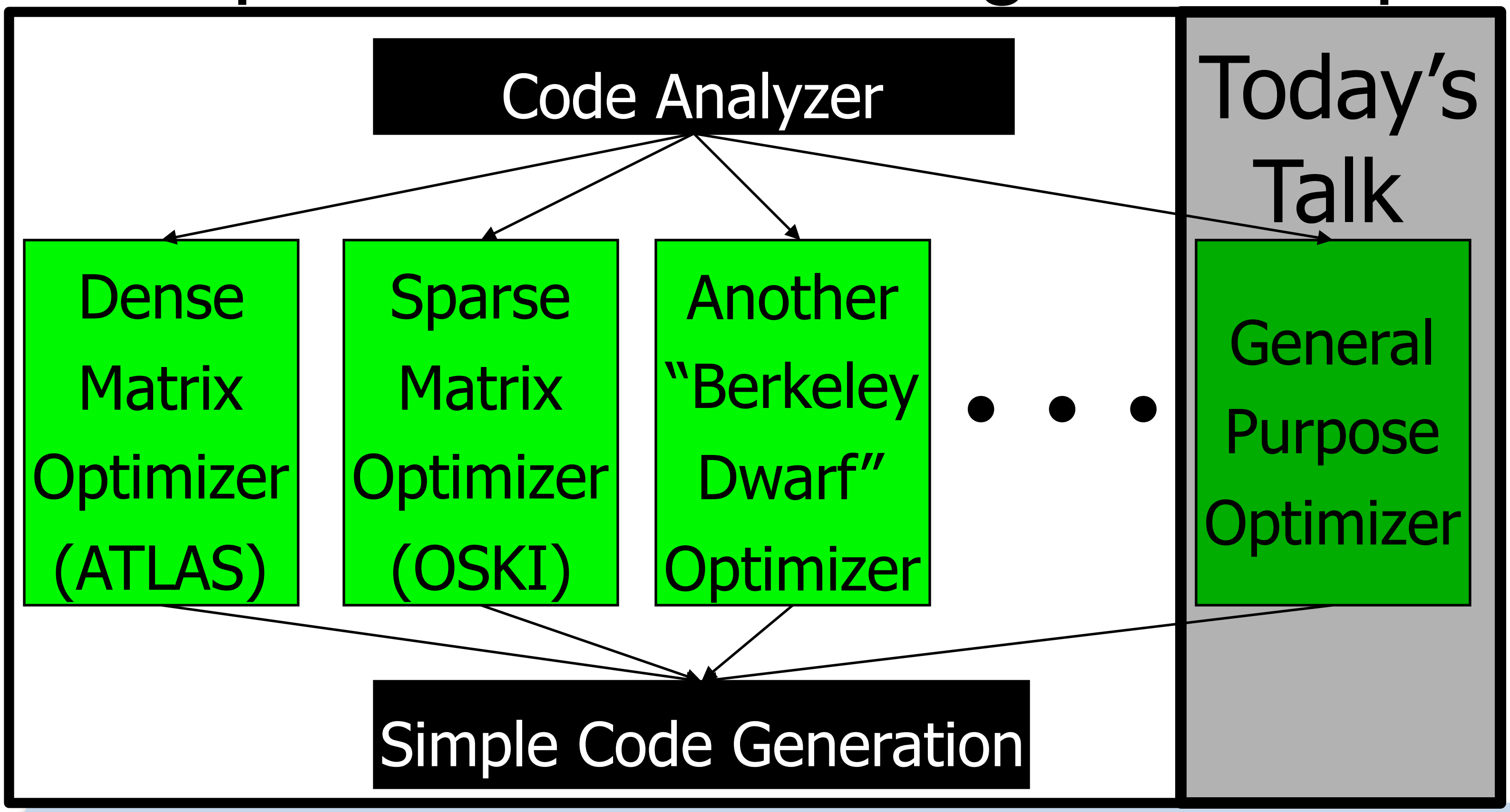
Autotuning and Compilers

- Proposition: Autotuning is a component of an Intelligent Compiler.



Autotuning and Compilers

- Proposition: Autotuning is a component of an Intelligent Compiler.



Traditional Compilers

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

Applications

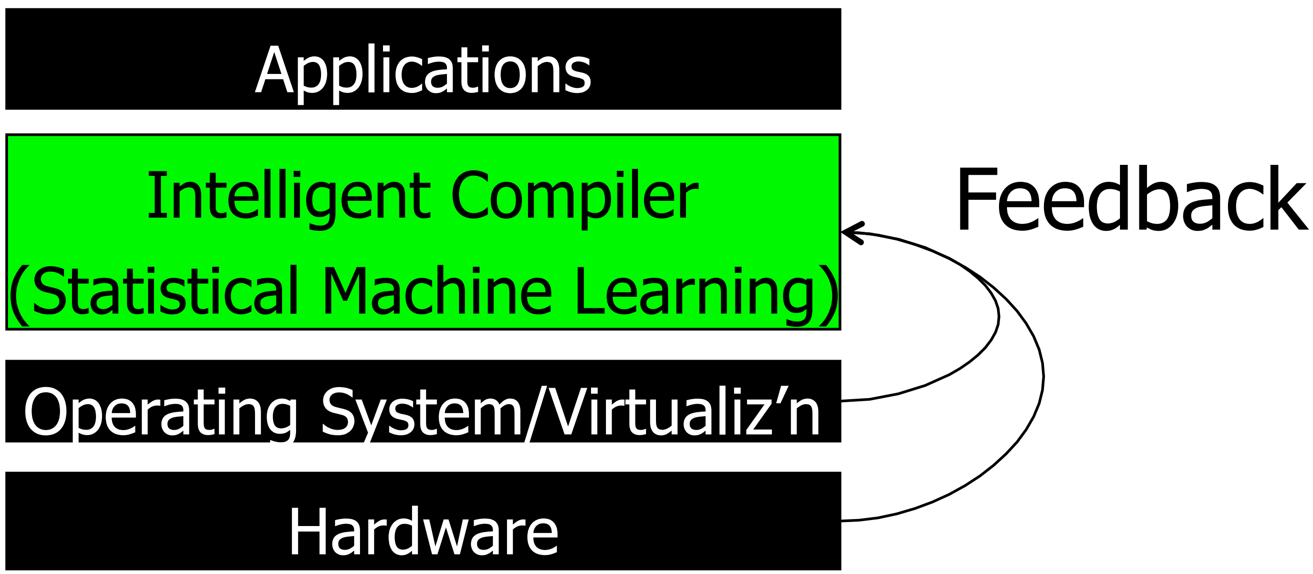
Compilers

Operating System/Virtualiz'n

Hardware

Proposed Solution

- *Intelligent* Compilers
  - Use machine learning
- Learn to optimize
  - Specialized to each Application/Data/Hardware



Building Intelligent Compilers

- We want intelligent, robust, adaptive behaviour in compilers.
- Often hand programming very difficult
- Get the compiler to program itself, by showing it examples of behaviour we want.
  - This is the machine learning approach!
- We write the structure of the compiler and it then tunes many internal parameters.

Intelligence in a compiler

- Individual optimization heuristic
  - Instruction scheduling [NIPS 1997, PLDI 2005]
- Whole-program optimizations [CGO ’06 / ’07]
- Individual methods [OOPSLA 2006]
- Individual loop bodies [PLDI 2008]

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

How to use Machine Learning

- Phrase as machine learning problem
- Determine inputs/outputs of ML model
  - Important characteristics of problem (features)
  - Target function
- Generate training data
- Train and test model
  - Learning algorithms may require “tweaking”

Train and Test Model

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

Case Studies

- **Whole Program Optimization**
- Individual Method Optimization

Putting Perf Counters to Use

- Model Input
  - Aspects of programs captured with perf. counters
- Model Output
  - Set of optimizations to apply
- Automatically construct model (Offline)
  - Map performance counters to good opts
- Model predicts optimizations to apply
  - Uses performance counter characterization

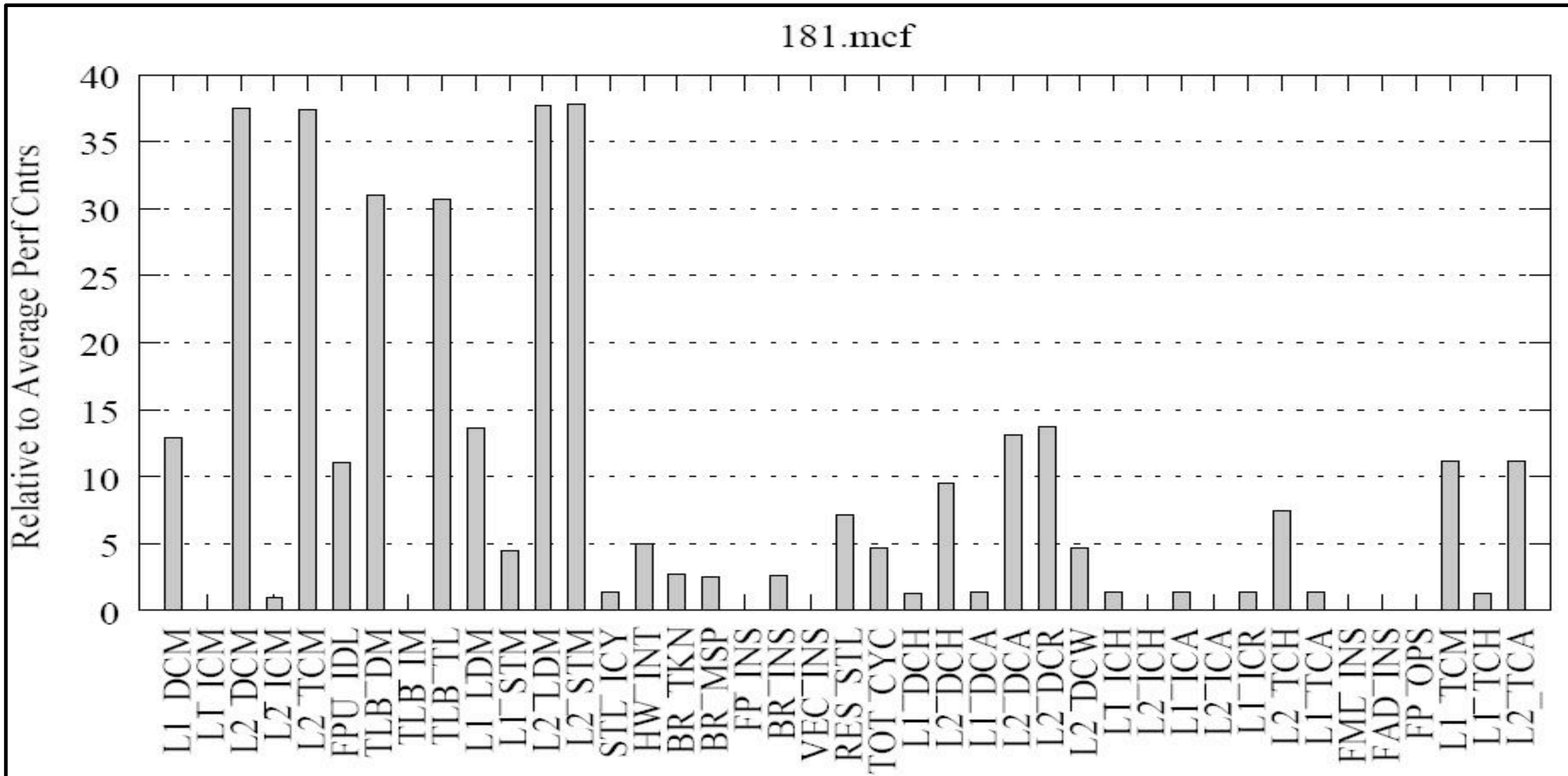
Performance Counters

- 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

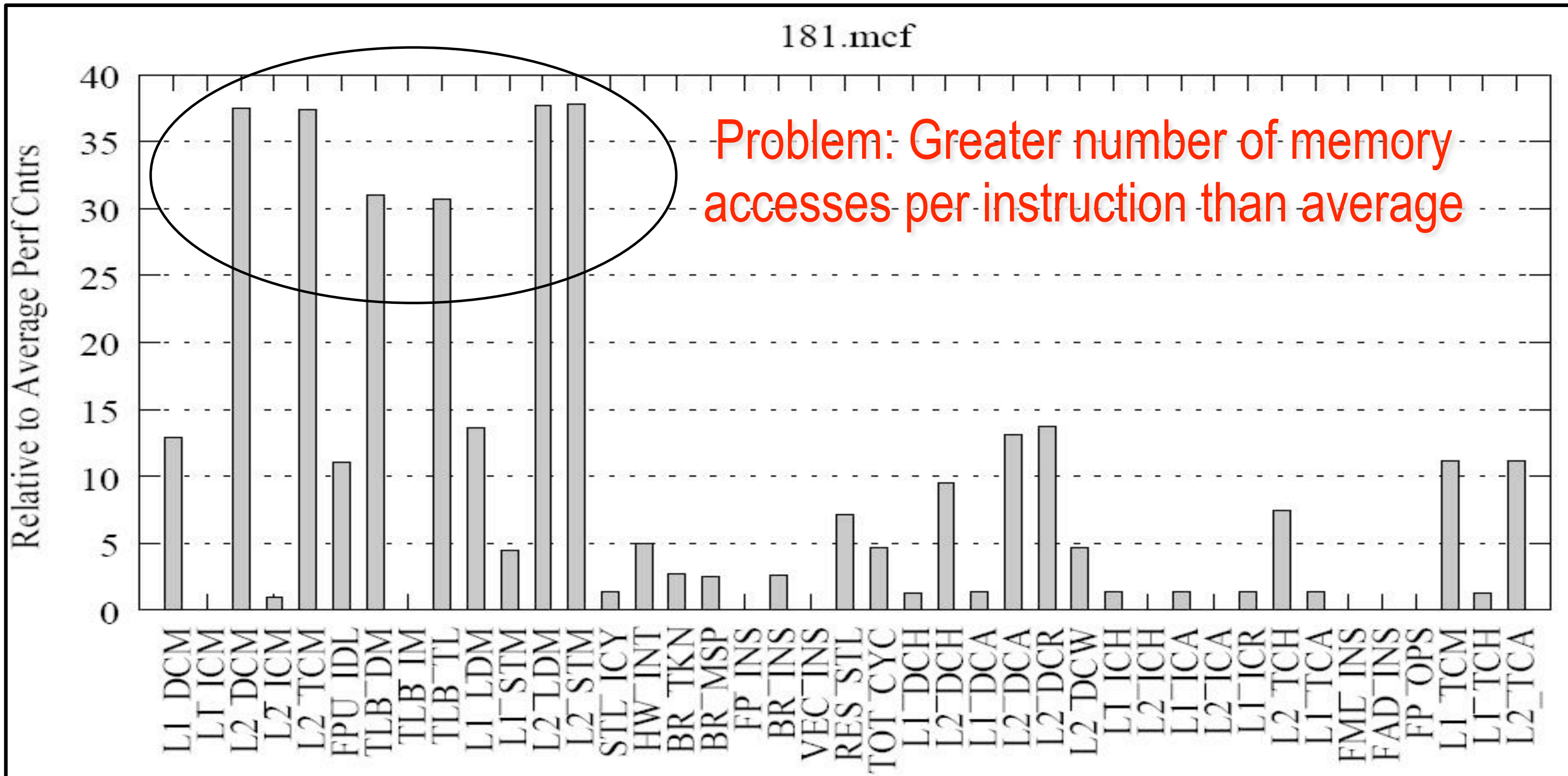
Characterization of 181.mcf

- Perf cntrs relative to several benchmarks

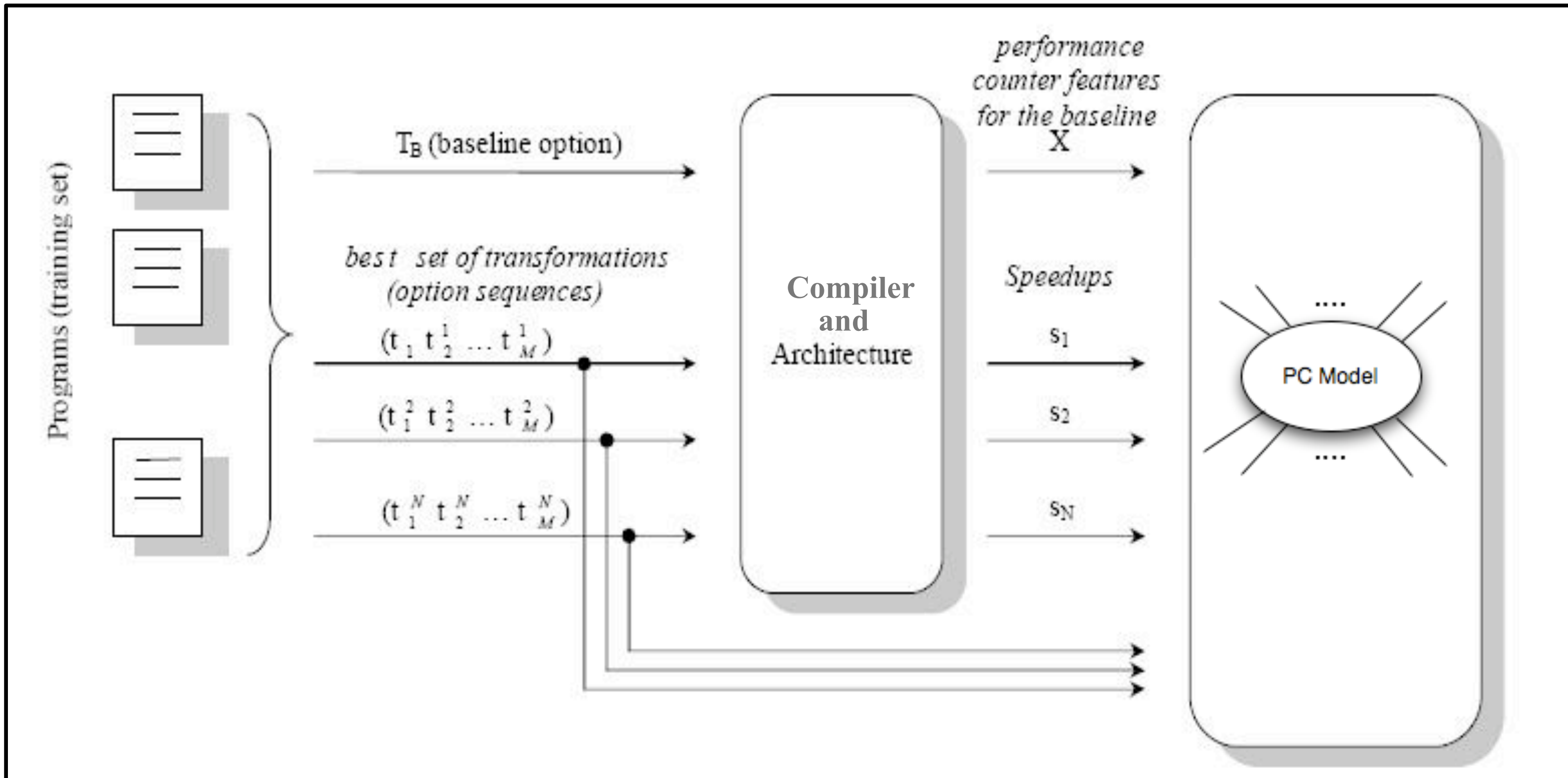


Characterization of 181.mcf

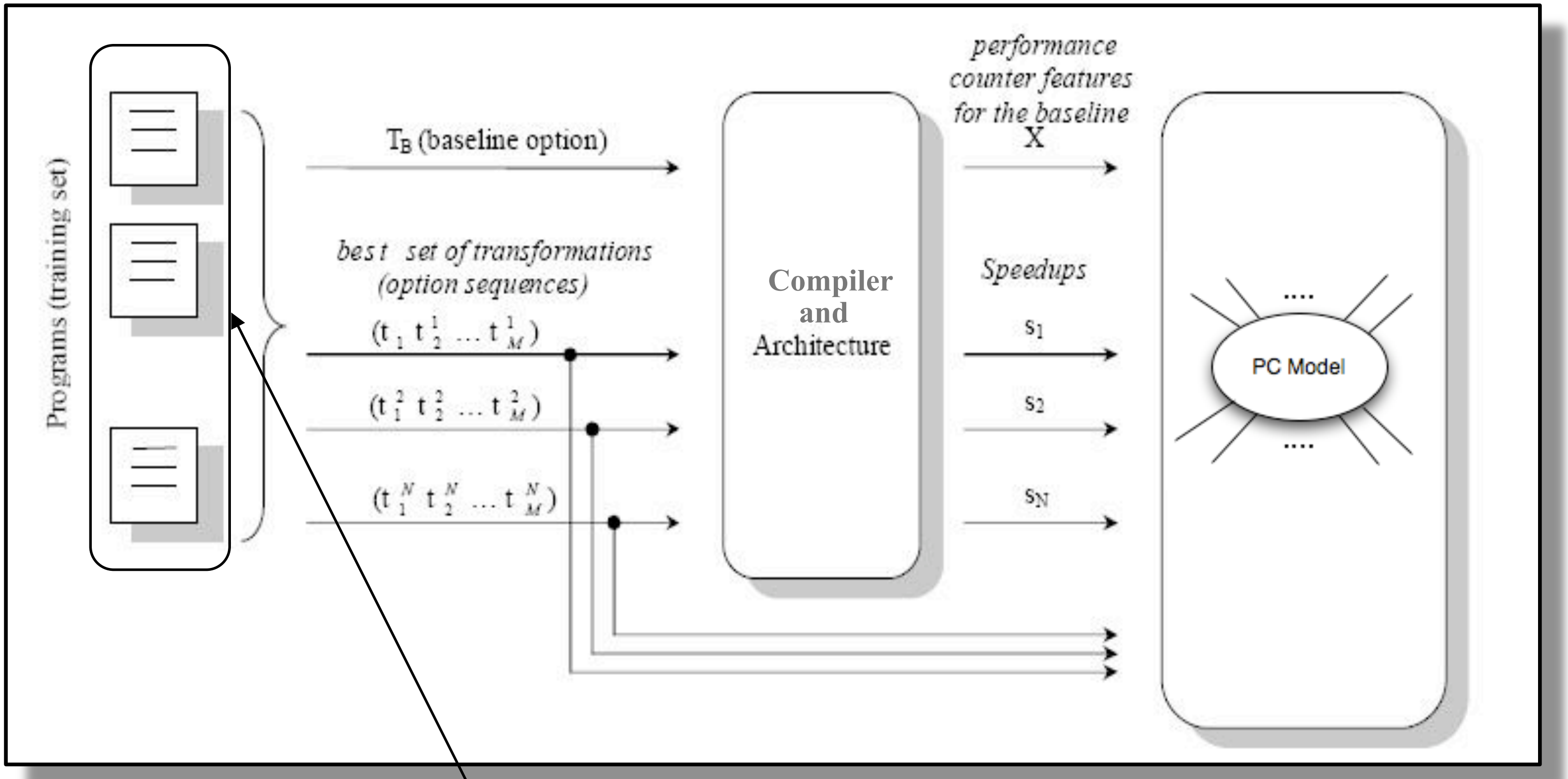
- Perf cntrs relative to several benchmarks



Training PC Model

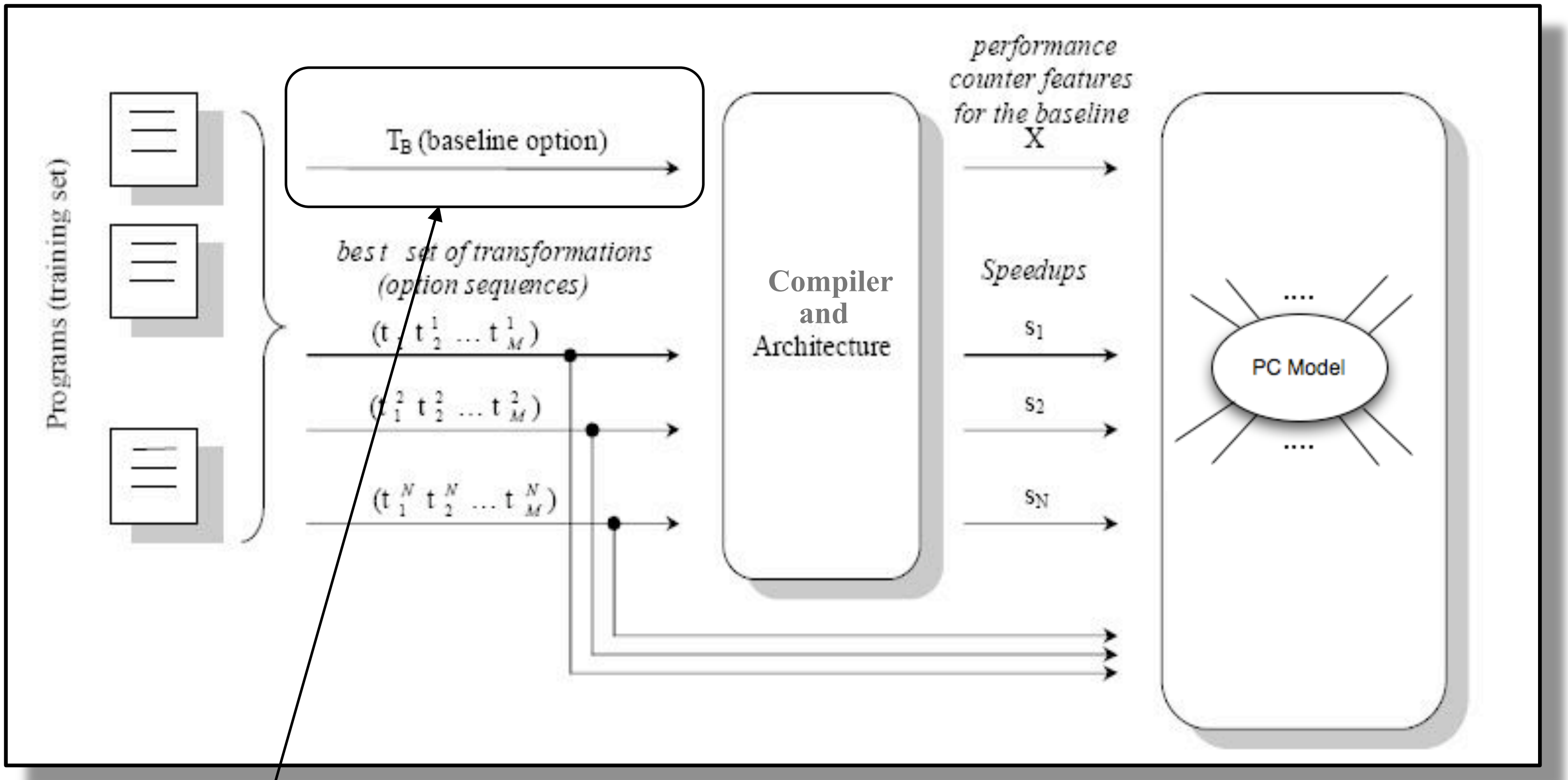


Training PC Model



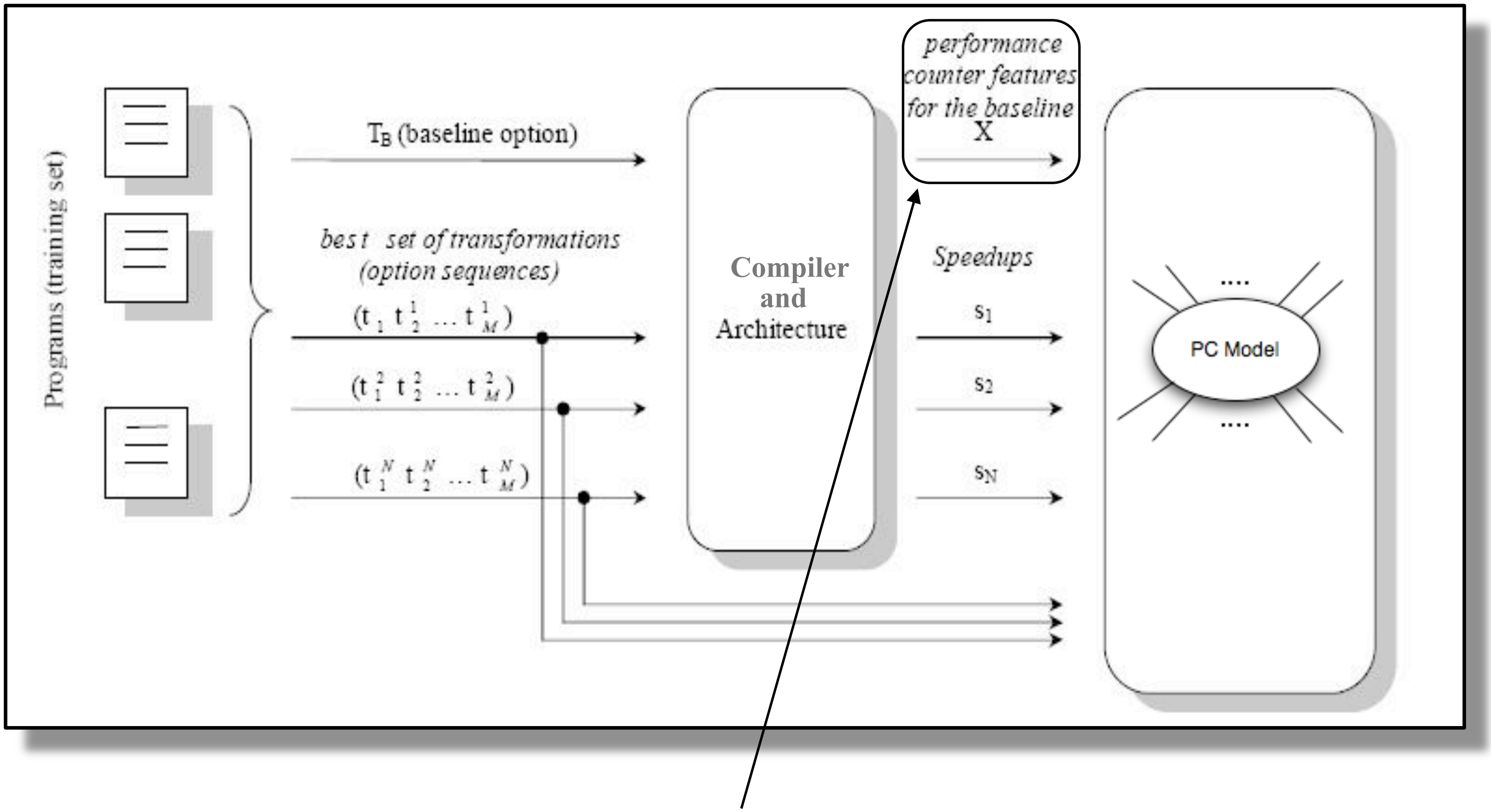
Programs to train model (different from test program).

Training PC Model



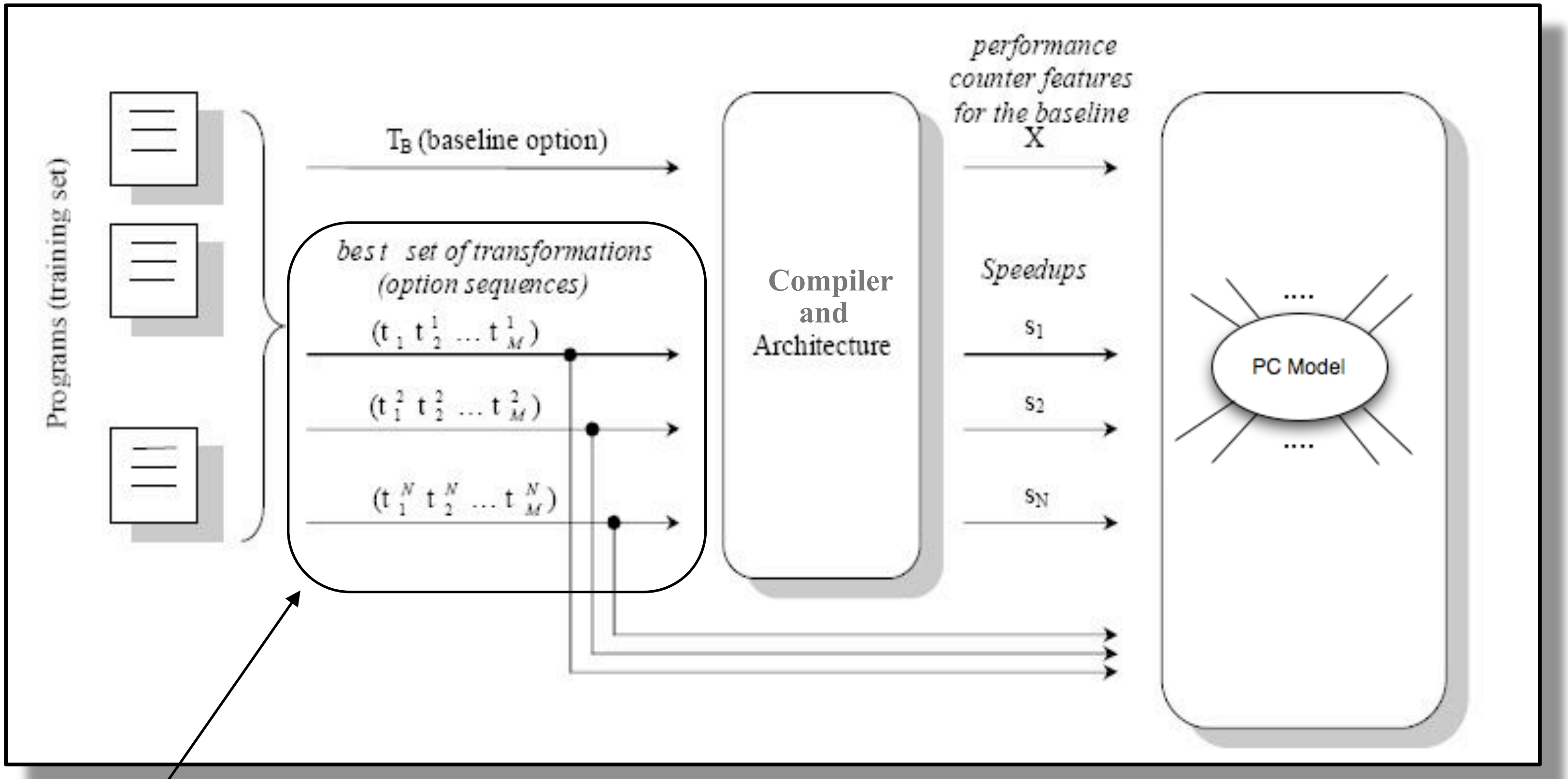
Baseline runs to capture performance counter values.

Training PC Model



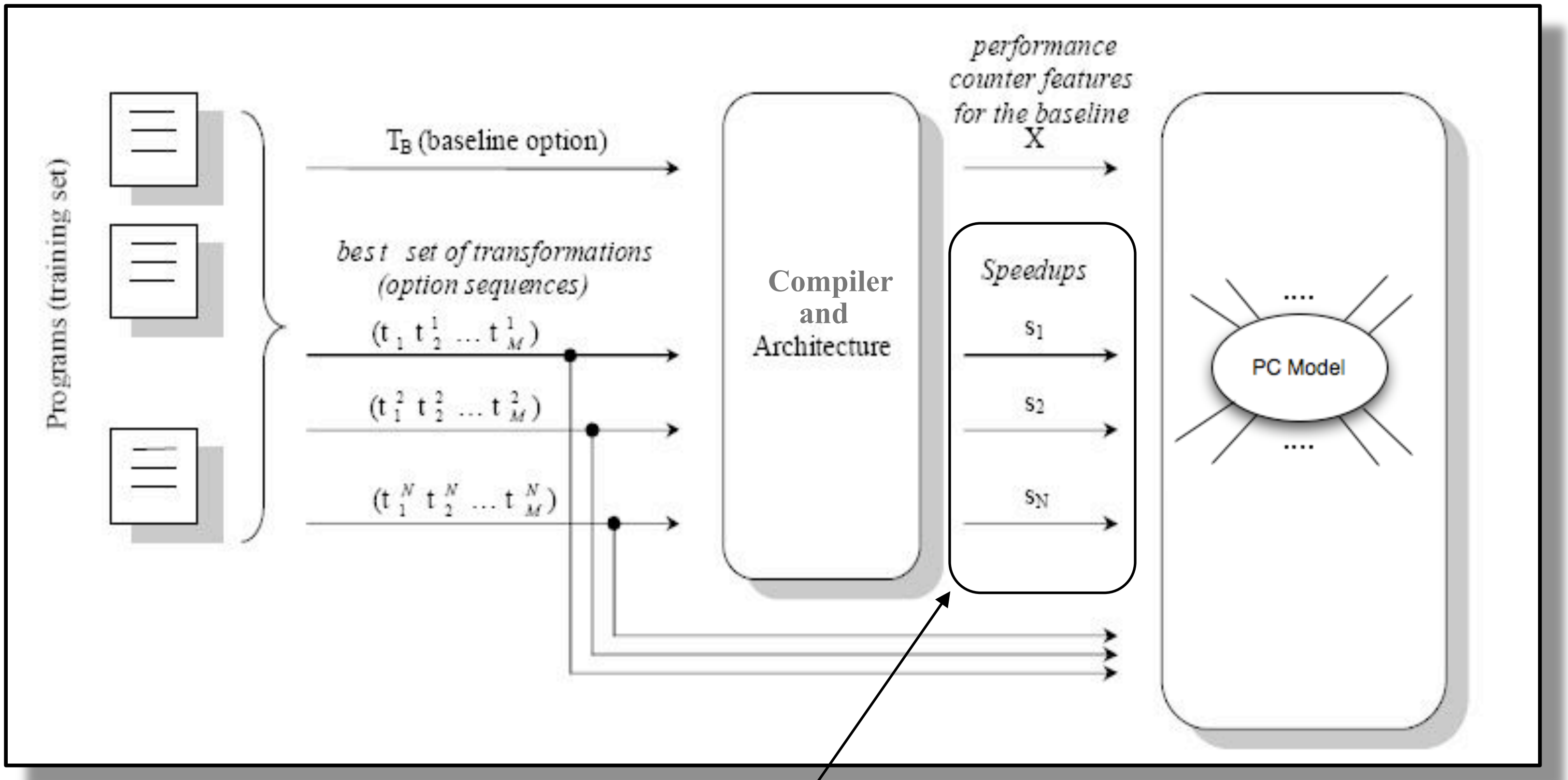
Obtain performance counter values for a benchmark.

Training PC Model



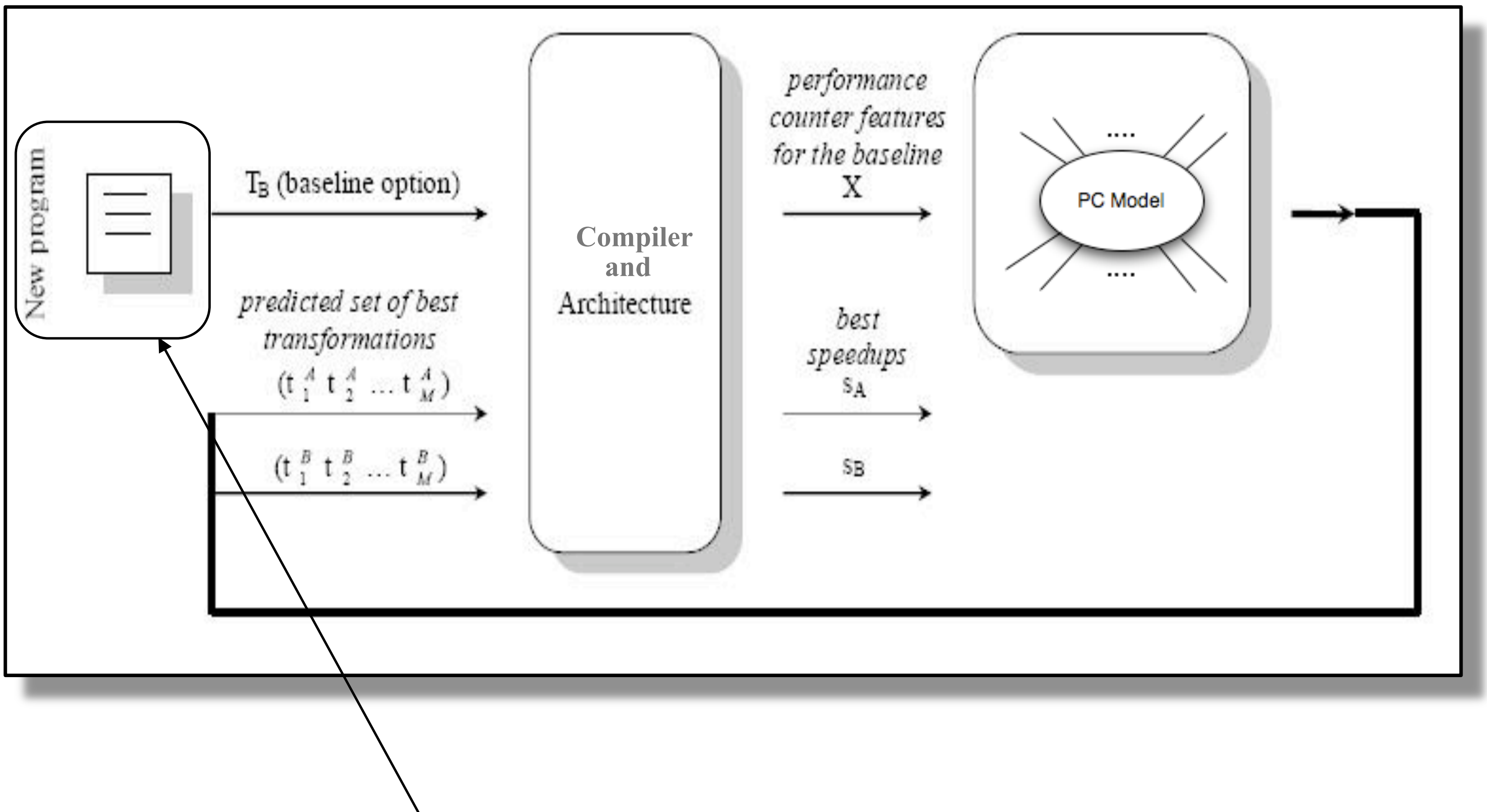
Best optimizations runs to get speedup values.

Training PC Model



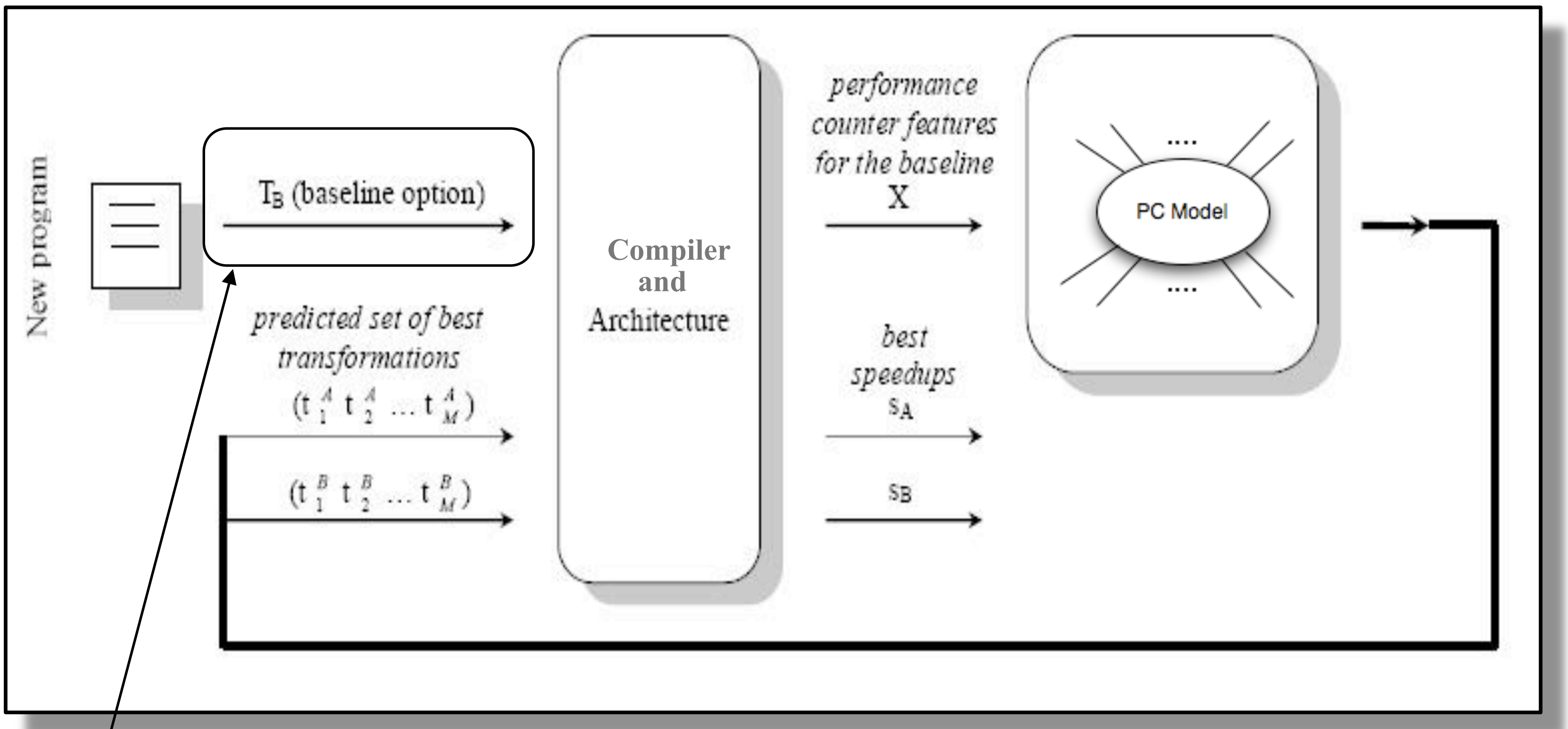
Best optimizations runs to get speedup values.

Using PC Model



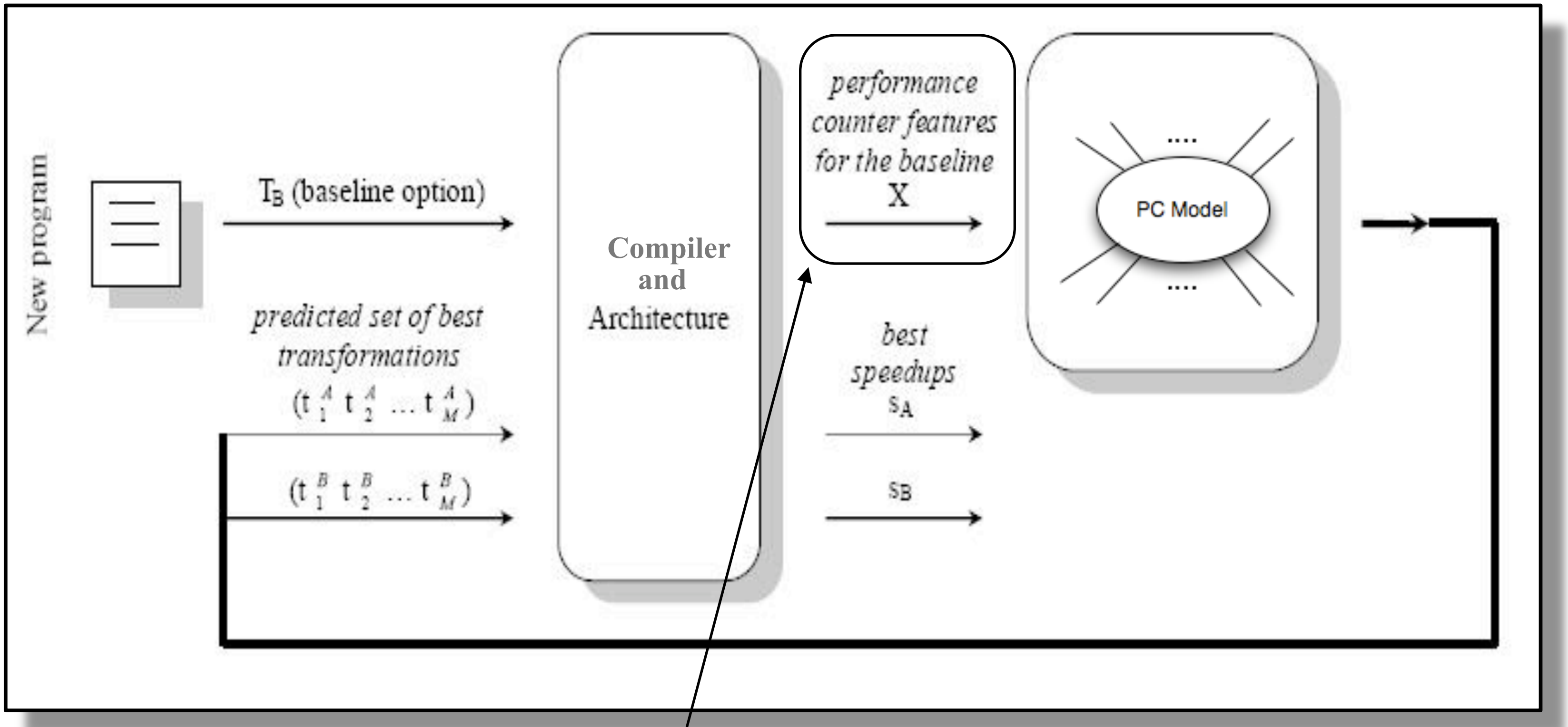
New program interested in obtaining good performance.

Using PC Model



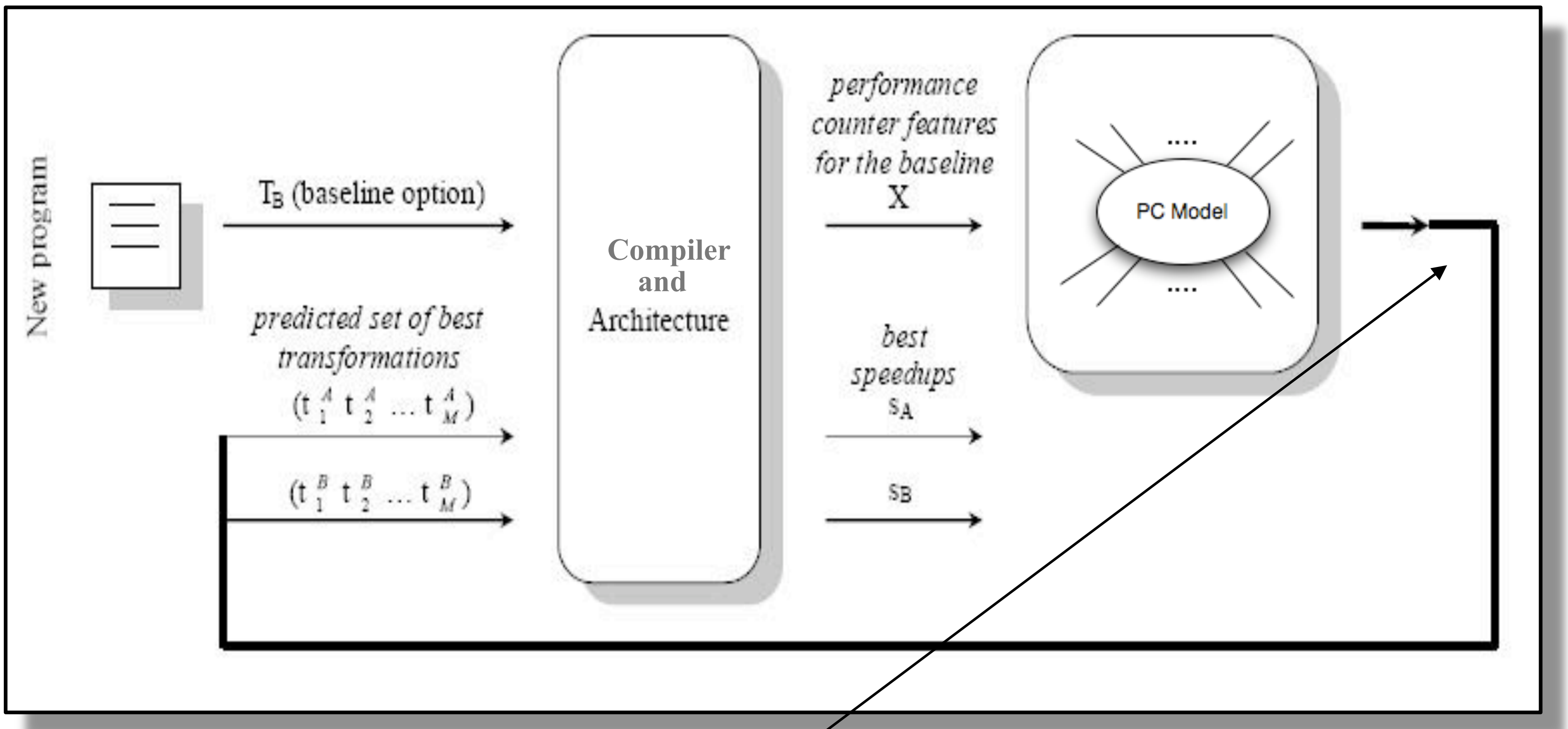
Baseline run to capture performance counter values.

Using PC Model



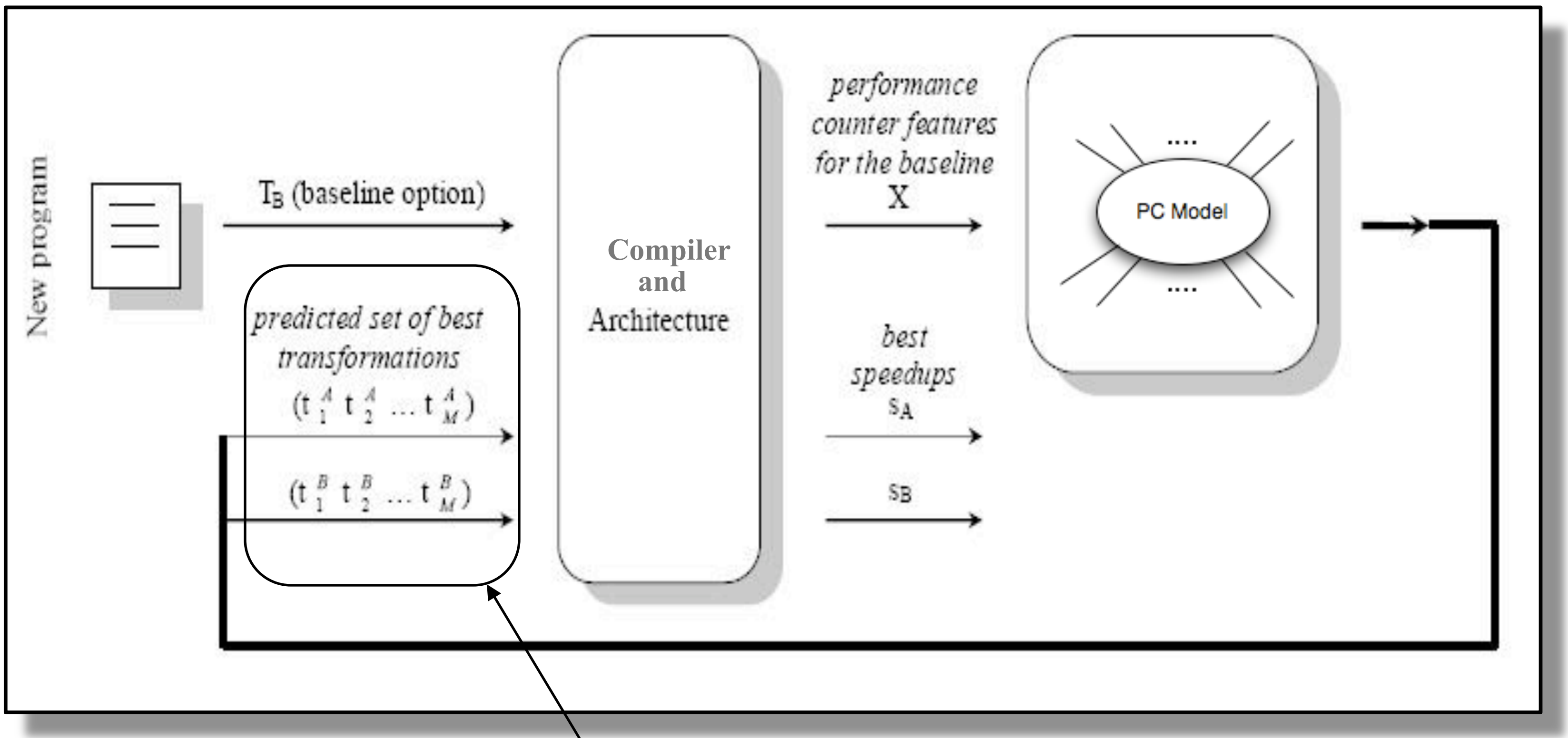
Feed performance counter values to model.

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

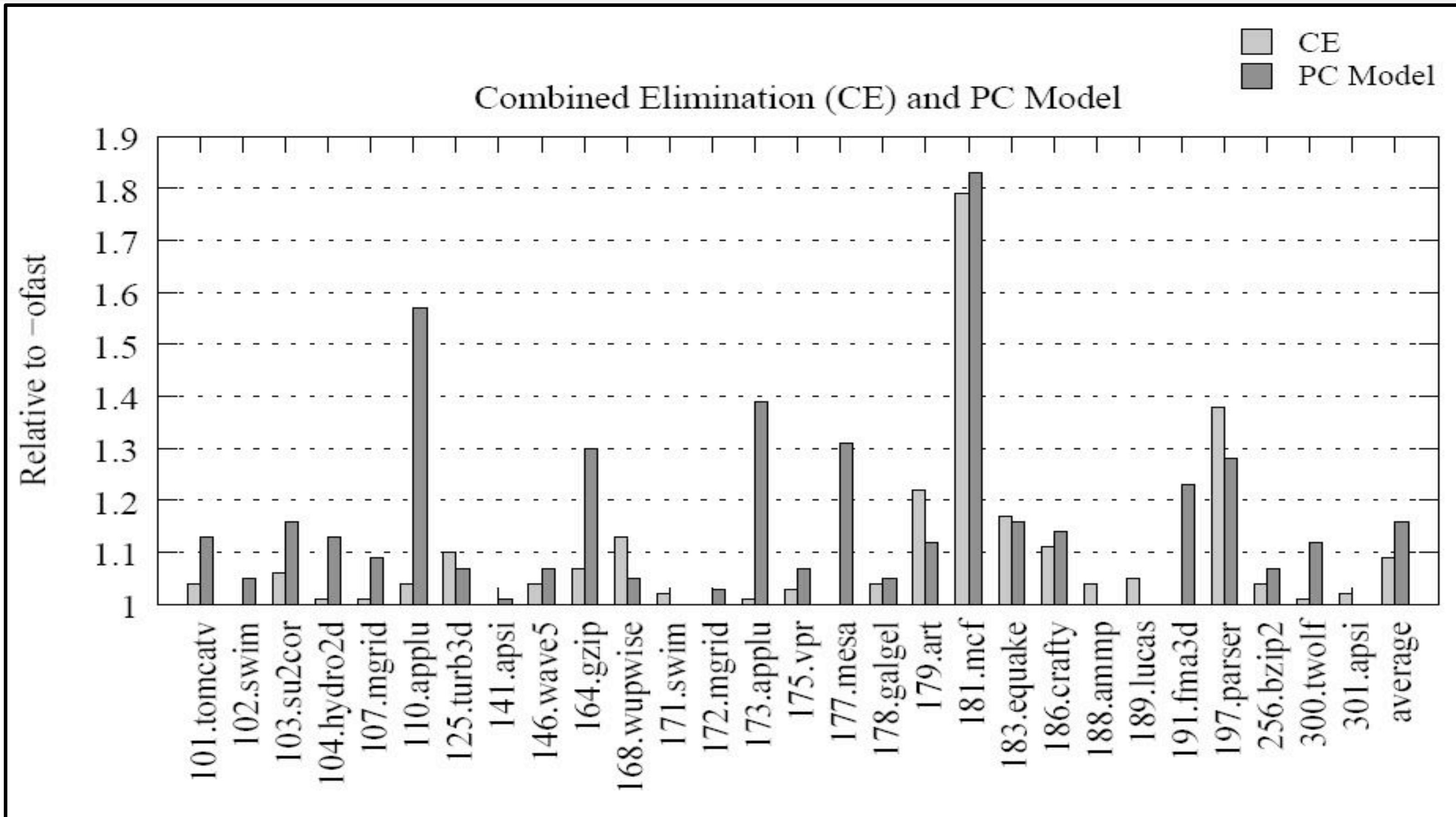
Experimental Methodology

- PathScale industrial-strength compiler
  - Compare to highest optimization level
  - Control 121 compiler flags
- AMD Athlon processor
  - ***Real*** machine; Not simulation
- 57 benchmarks

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.

Case Studies

- Whole Program Optimization
- **Individual Method Optimization**

Method-Specific Compilation

- Integrate machine learning into 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+

Overall Approach

- Phase 1: Training
  - Generate training data
  - Construct a heuristic
  - Expensive offline process
- Phase 2: Deployment
  - During Compilation
    - Extract code features
    - Heuristic predicts optimizations

Generate Training Data

- For each method
  - Evaluate many opt settings
  - Fine-grained timers
    - Record running time
    - Record compilation time
- For optimization level O2
  - Evaluate 1000 random settings
- One model for the optimization level

Training Data

- Training example for each method
  - Inputs - Features of method
  - Outputs - Good optimization setting

Training examples

methods	inputs	outputs
foo	108;25;0;0; ... ;.08;0;	1;0;1;1; ... 1;1;1;0
bar	93;21;0;1; ... :.50;0;	1;1;0;0; ... 1;0;0;0
...	.....	....
...	.....	....

Method Properties (inputs)

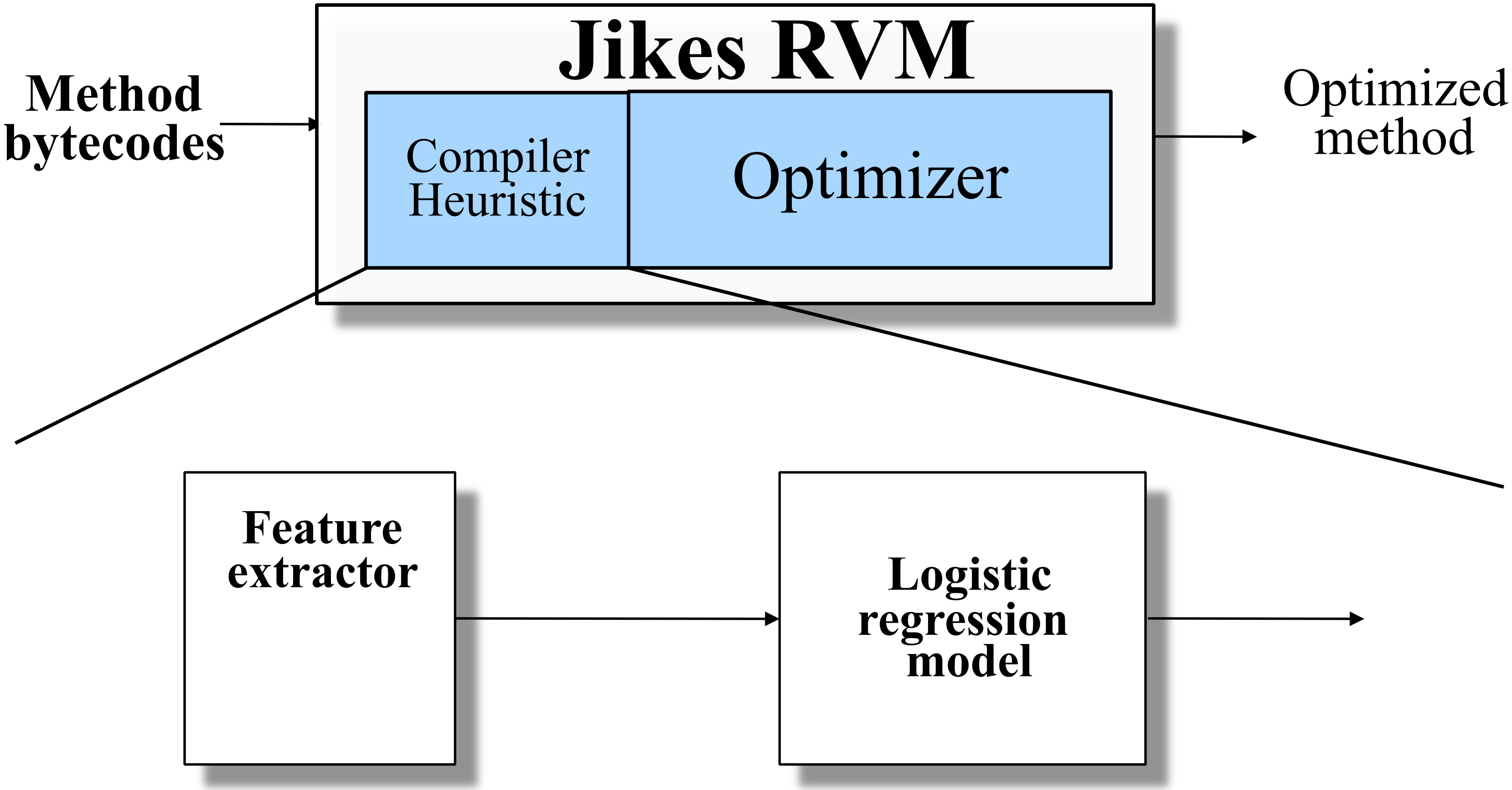
Method Features	Meaning
Size	Number of **bytecodes**
Locals Space	Words allocated for **locals space**
Characteristics	Is **syncronized**, has **exceptions,** is **leaf** method
Declaration	Is it declared **final, static, private**
Fraction of Bytecodes	Has array **loads** and **stores** primitive and long computations compares, **branches**, jsrs, switches, put, get, invoke, new, arraylength athrow, checkcast, monitor

Note: **26** features used to describe method

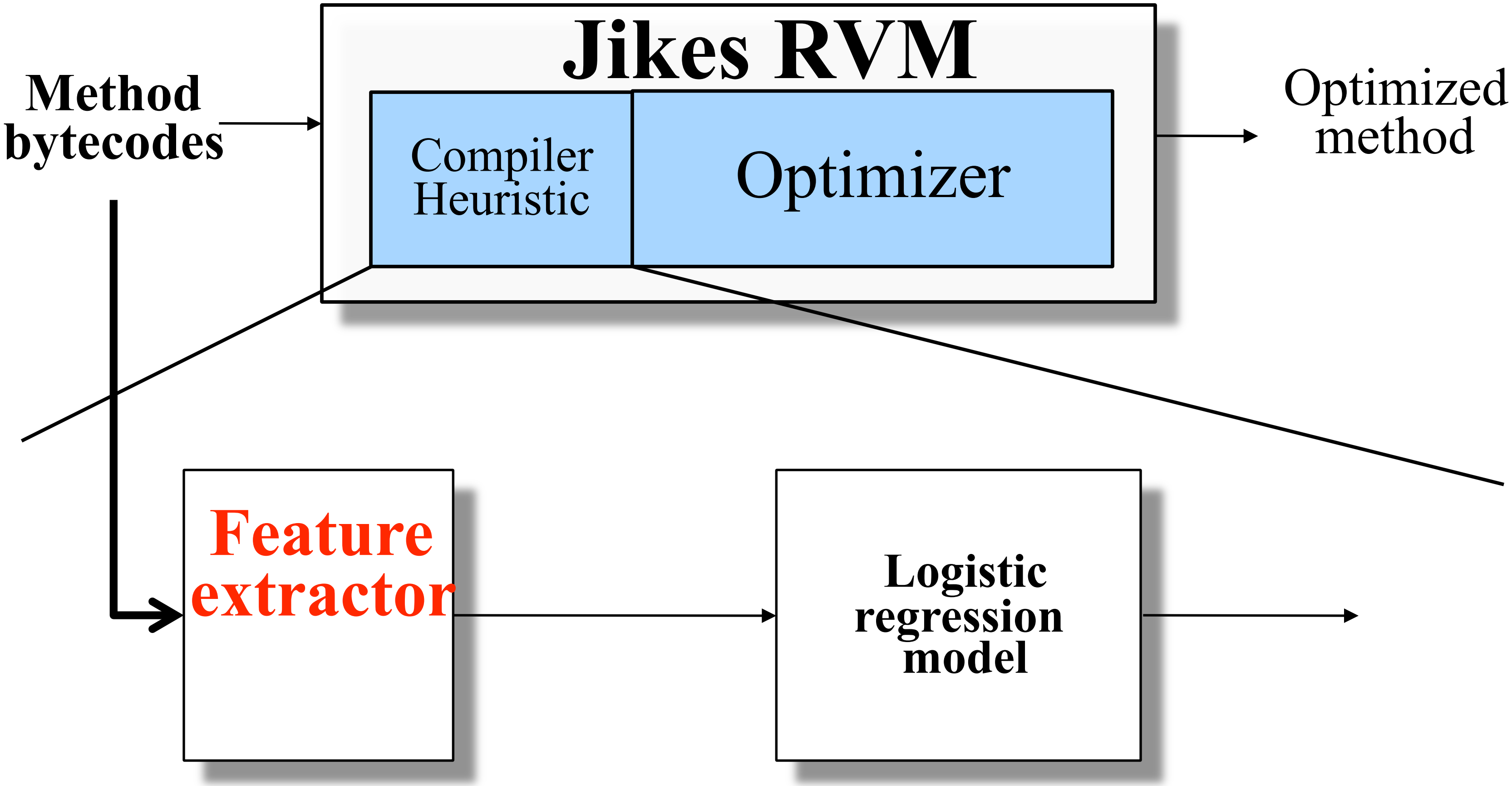
Optimizations (outputs)

Optimization Level	Optimizations Controlled
Opt Level O0	Branch Opts Low **Constant Prop** / Local CSE Reorder Code
Opt Level O1	**Copy Prop** / Tail Recursion Static Splitting / Branch Opt Med **Simple Opts Low**
Opt Level O2	While into Untils / **Loop Unroll** Branch Opt High / Redundant BR Simple Opts Med **/ Load Elim** Expression Fold / Coalesce Global Copy Prop / Global CSE **SSA**

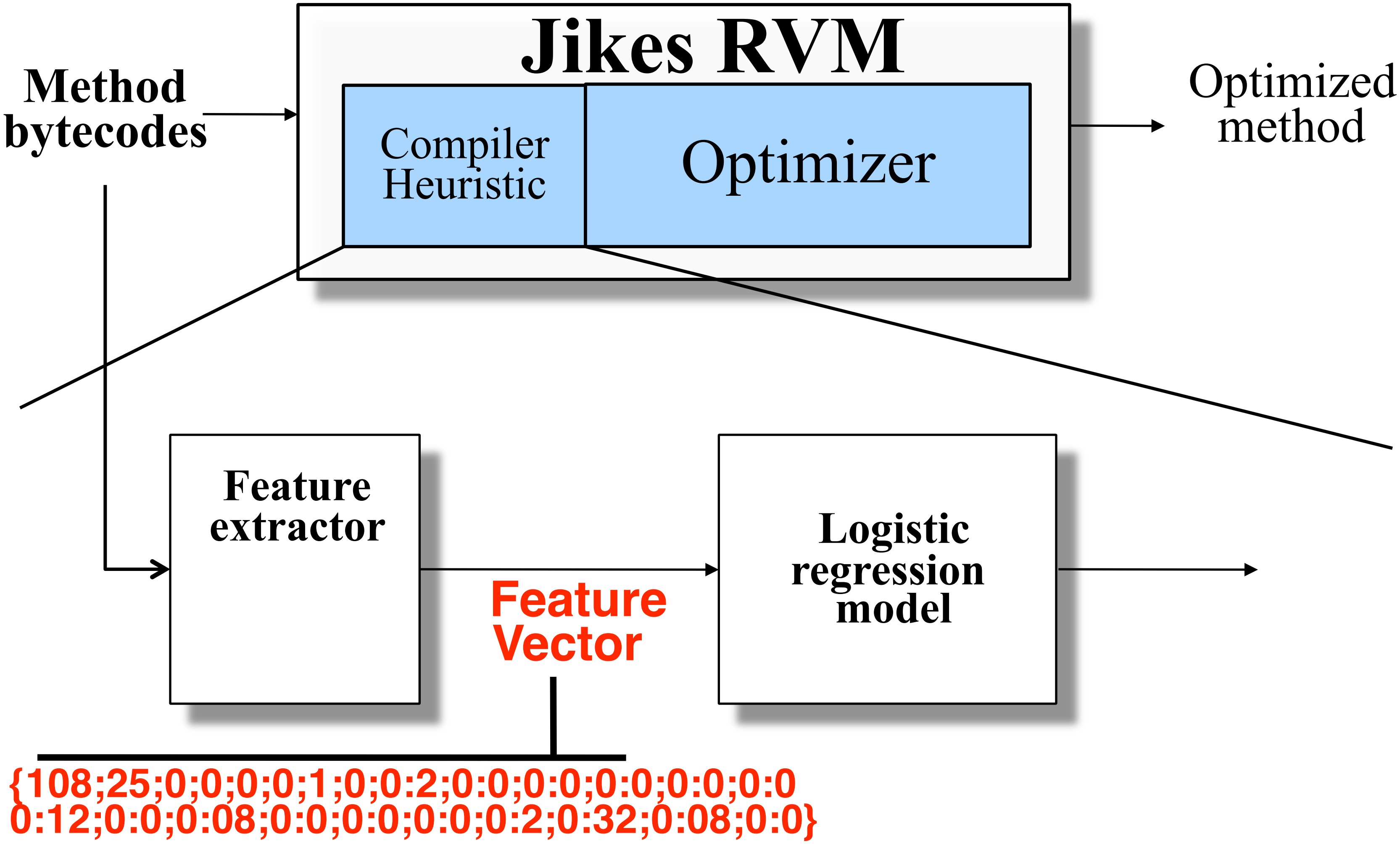
Compiler Heuristic (online)



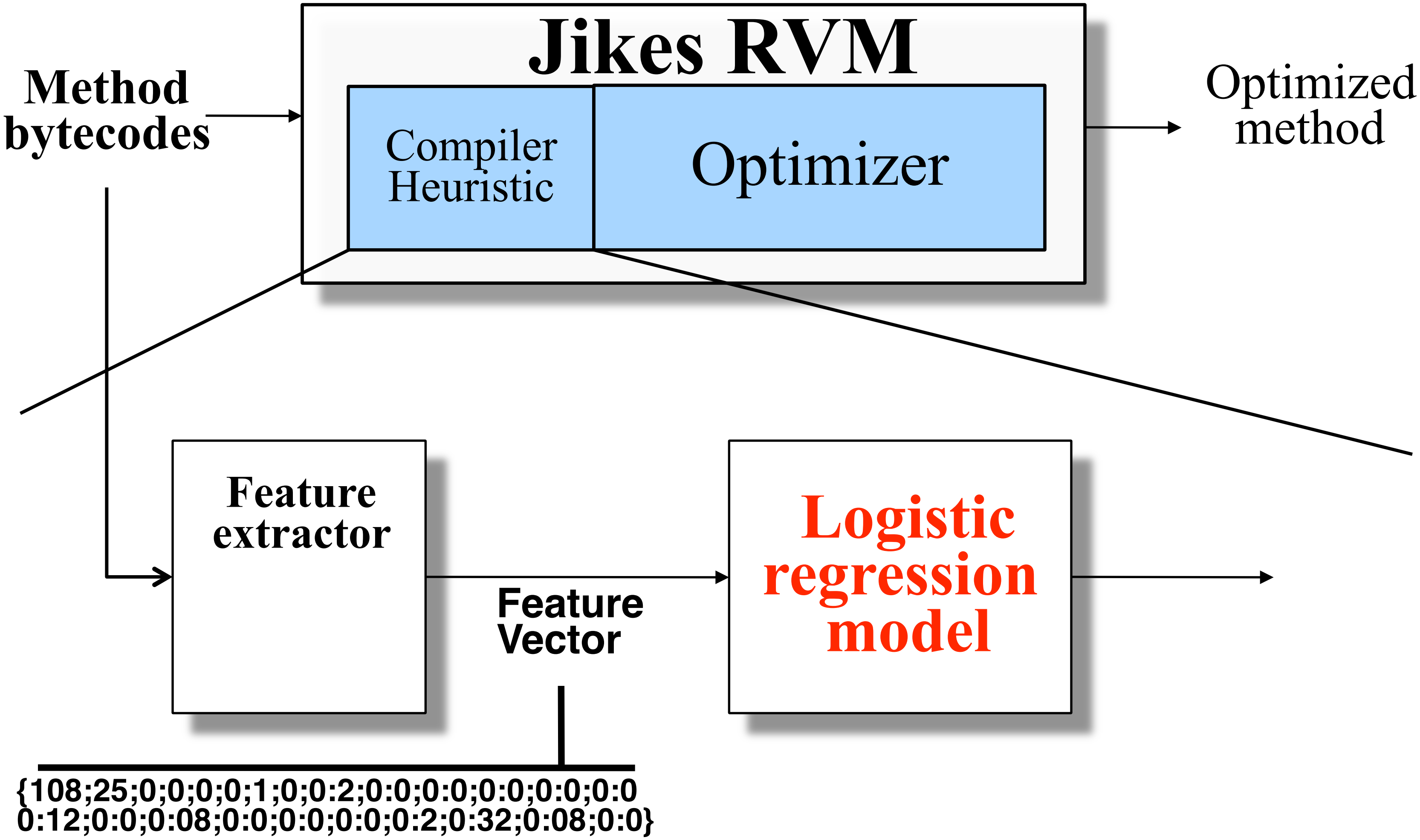
Compiler Heuristic (online)



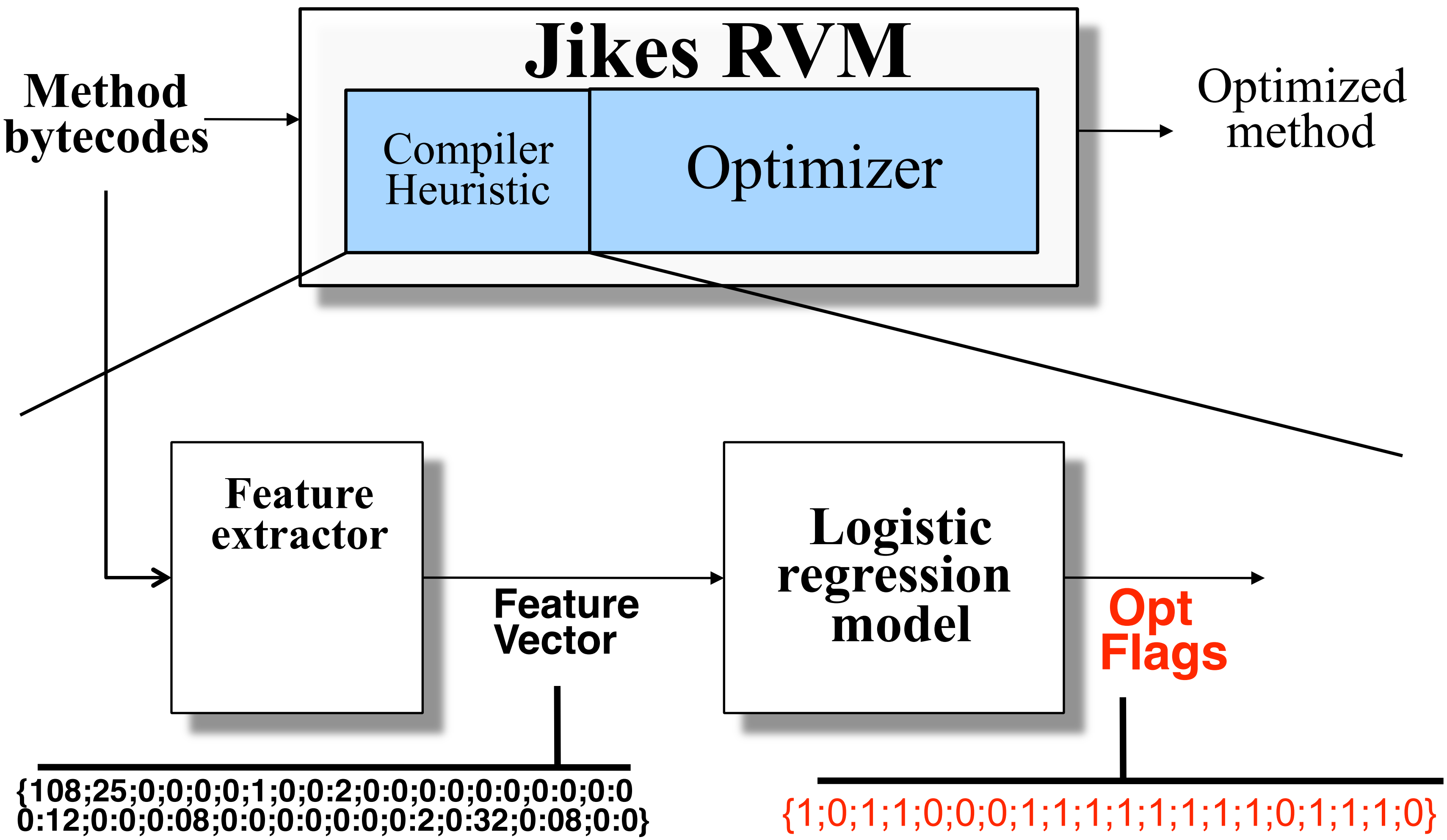
Compiler Heuristic (online)



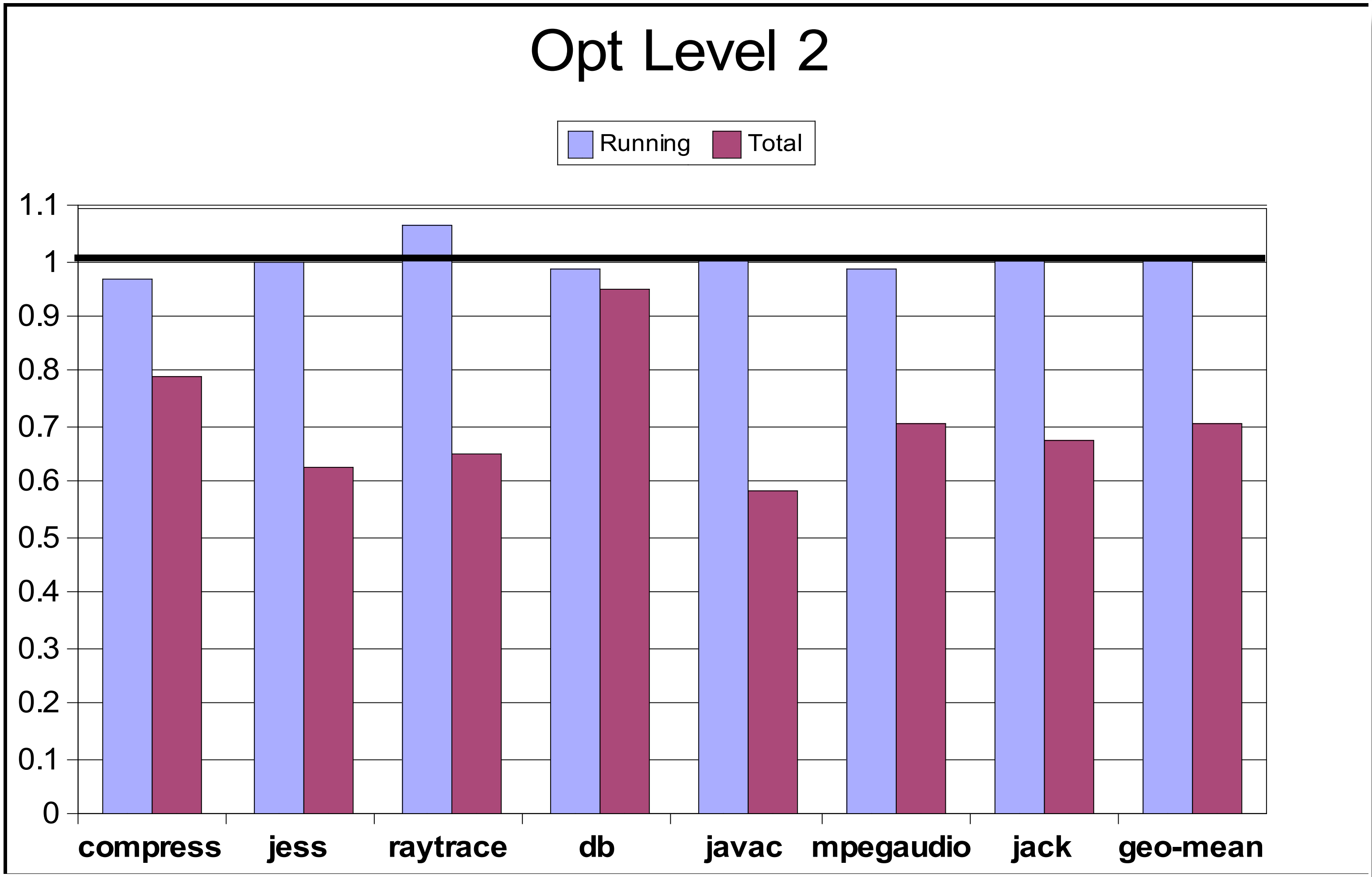
Compiler Heuristic (online)



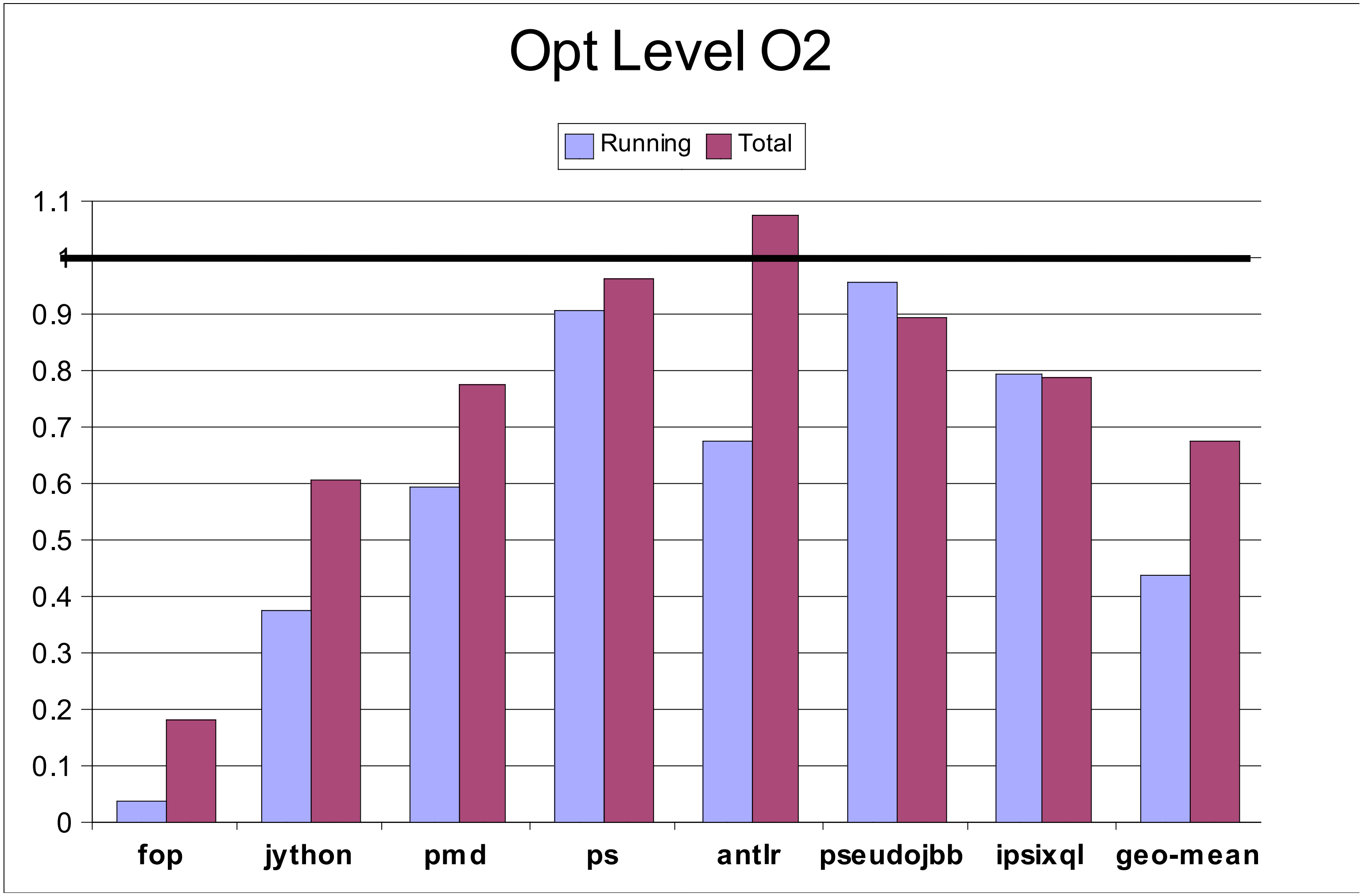
Compiler Heuristic (online)



SPECJVM (Highest Opt Level)



DaCapo+ (Highest Opt Level)



Challenges Remaining

- Single-core optimizations still important
  - Optimization phase-ordering
  - Optimization for program phases
  - Speculative optimizations
- Parallel optimizations
  - Task partitioning
  - Communication/computation overlap
  - Task scheduling/migration
  - Data placement/migration/replication

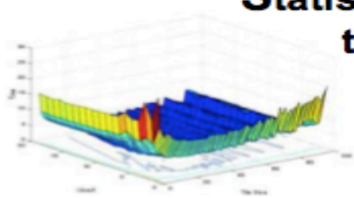
Conclusions

- Using **machine learning successful**
  - Out-performs production compiler in few evaluations
- Using **perf. counters/code characteristics**
  - Determines automatically what characteristics are important
  - Optimizations applied only when beneficial

SMART Workshop

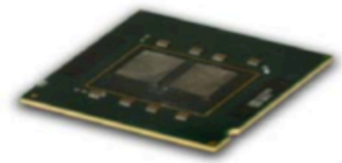
http://www.hipeac.net/smart-workshop.html

3rd Workshop on
Statistical and Machine learning approaches to ARchitectures and compilaTion (SMART '09)



January 25, 2009, Paphos, Cyprus

(co-located with HiPEAC 2009 Conference)



Sponsored by:

MILEPOST

Program Chair:

David Padua
University of Illinois at Urbana-Champaign, USA

Organizers:

Grigori Fursin
INRIA Saclay, France

John Cavazos
University of Delaware, USA

The rapid rate of architectural change and the large diversity of architecture features has made it increasingly difficult for compiler writers to keep pace with microprocessor evolution. This problem has been compounded by the introduction of multicores. Thus, compiler writers have an intractably complex problem to solve. A similar situation arises in processor design where new approaches are needed to help computer architects make the best use of new underlying technologies and to design systems well adapted to futureapplication domains.

Recent studies have shown the great potential of statistical machine learning and search strategies for compilation and machine design. The purpose of this workshop is to help consolidate and advance the state of the art in this emerging area of research. The workshop is a forum for the presentation of recent developments in compiler techniques and machine design methodologies based on space exploration and statistical machine learning approaches with the objective of improving performance, parallelism, scalability, and adaptability.

Topics of interest include (but are not limited to):

Machine Learning, Statistical Approaches, or Search applied to