Automated Just-In-Time Compiler Tuning

Kenneth Hoste, Andy Georges and Lieven Eeckhout
Computer Systems Lab
Ghent University, Belgium
kenneth.hoste@elis.ugent.be

CGO 2010
April 26th 2010

Just-In-Time compilation
a (very) quick introduction

platform portability through dynamic optimization

- initially, code is interpreted or executed unoptimized
- hot code is recompiled on-the-fly with more optimization
- (re)compilation time is a part of the overall execution time

Just-In-Time compilation
a (very) quick introduction

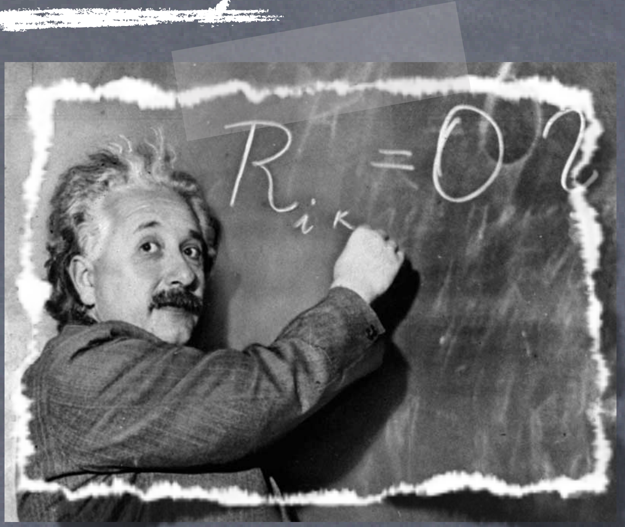
- a JIT compiler has multiple optimization levels (-O0, -O1, -O2, ...)
  - cost-benefit trade-off:
    required compilation time vs expected speedup
  - from cheap & low speedup
    to expensive & high speedup

- adaptive controller detects hot code and steers recompilation
  - based on sampled profiling of execution
  - exploits information on runtime behavior of application

- examples: Java, .NET, ...

JIT compiler tuning is complex

Currently, JIT compilers are tuned manually.



- very complex task, very time-consuming
  - large number of (interacting) optimizations
  - ⇒ huge design space for optimization levels
  - requires in-depth knowledge about optimizations
  - optimization levels need to offer suitable cost-benefit trade-offs
  - optimization levels interact with each other at run time
- retuning is required for different applications and platforms to obtain good performance
  - optimizations may yield different results
  - different cost-benefit trade-offs

Automated JIT compiler tuning

We propose:

a fully automated framework for tuning JIT compilers

- for a particular set of applications
- for a particular hardware platform
- uses an evolutionary algorithm which will gradually evolve better JIT compiler settings
- focuses both on startup and steady-state performance

Related work

- iterative compilation: targets just one single objective (e.g., speedup)
- COLE (CGO-2008): focuses on static compilers
- other work (Cavazos & O'Boyle): requires significant changes to the JIT compiler codebase

Prior work is insufficient for fully automated tuning of existing JIT compilers.

Prior work is insufficient

JIT compilers pose several new challenges compared to static compilers...

- multiple interacting optimization levels
- tunable adaptive controller that steers recompilation

Applying our COLE framework to a JIT compiler yields unsatisfactory results:

- representation of JIT compiler too complex for an evolutionary algorithm to handle
  - crossover? mutation?
- disappointing performance, excessively long exploration

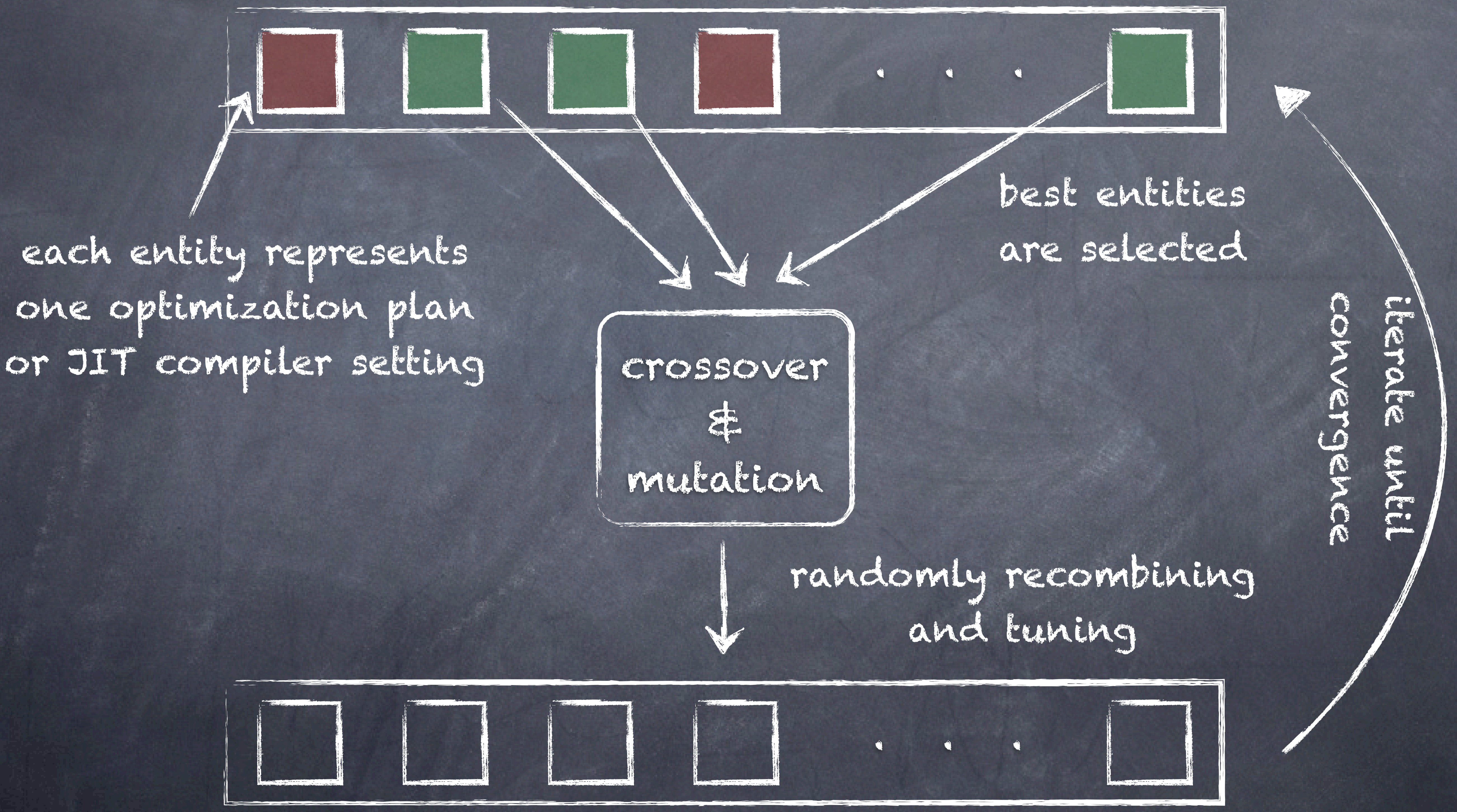
Our approach

split the tuning process into two steps

- step 1: optimization plans
- step 2: JIT compiler configurations

- optimization plan:
  set of optimizations and value parameters
- optimization level:
  optimization plan used in JIT compiler
- JIT compiler configuration:
  multiple optimization levels + tuned controller

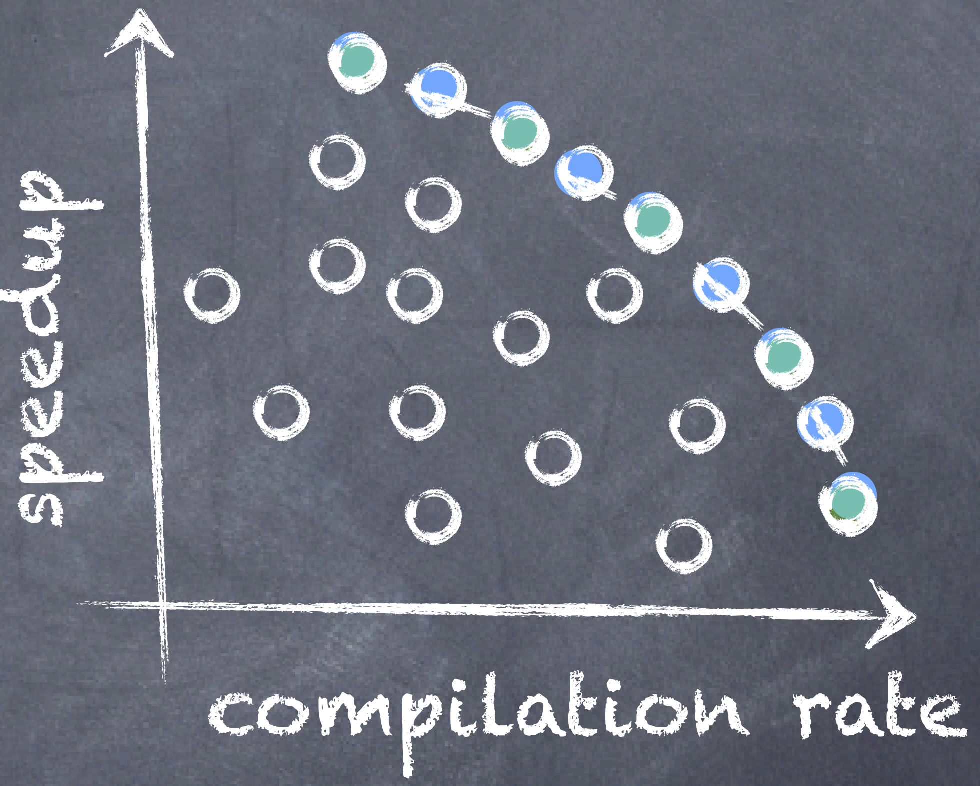
In short: evolutionary algorithms


each entity represents one optimization plan or JIT compiler setting
best entities are selected
crossover & mutation
randomly recombining and tuning
iterate until convergence

Trading off cost and benefit

Step I: Pareto-optimal optimization plans

- use COLE framework to find interesting optimization plans
  => trade off compilation rate and speedup

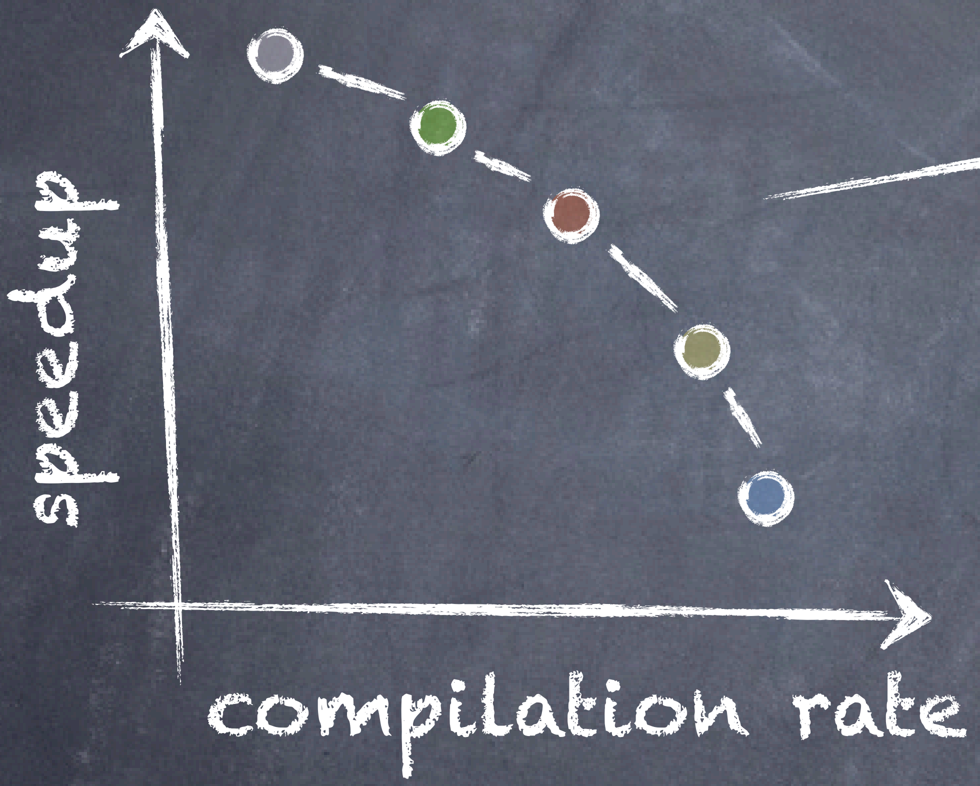


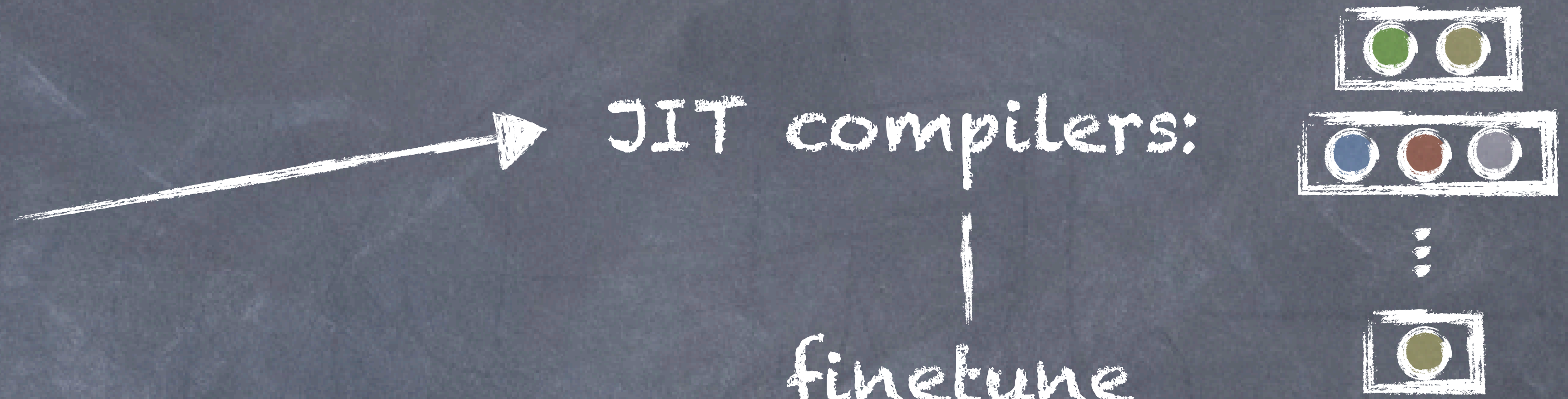
- a set of Pareto optimal optimization plans are evolved
  => complex interactions between plans are avoided (for now)

- a limited number of Pareto optimal plans are selected for step II

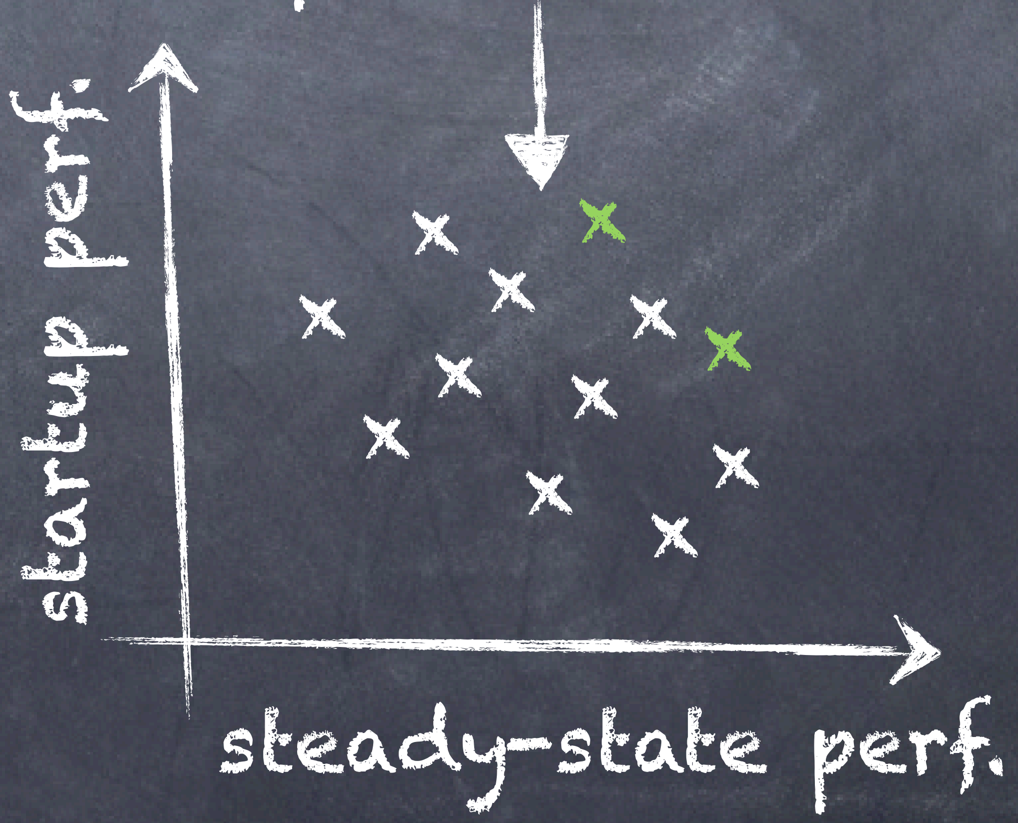
Combine and conquer

Step II: combine optimization plans and finetune





finetune parameters



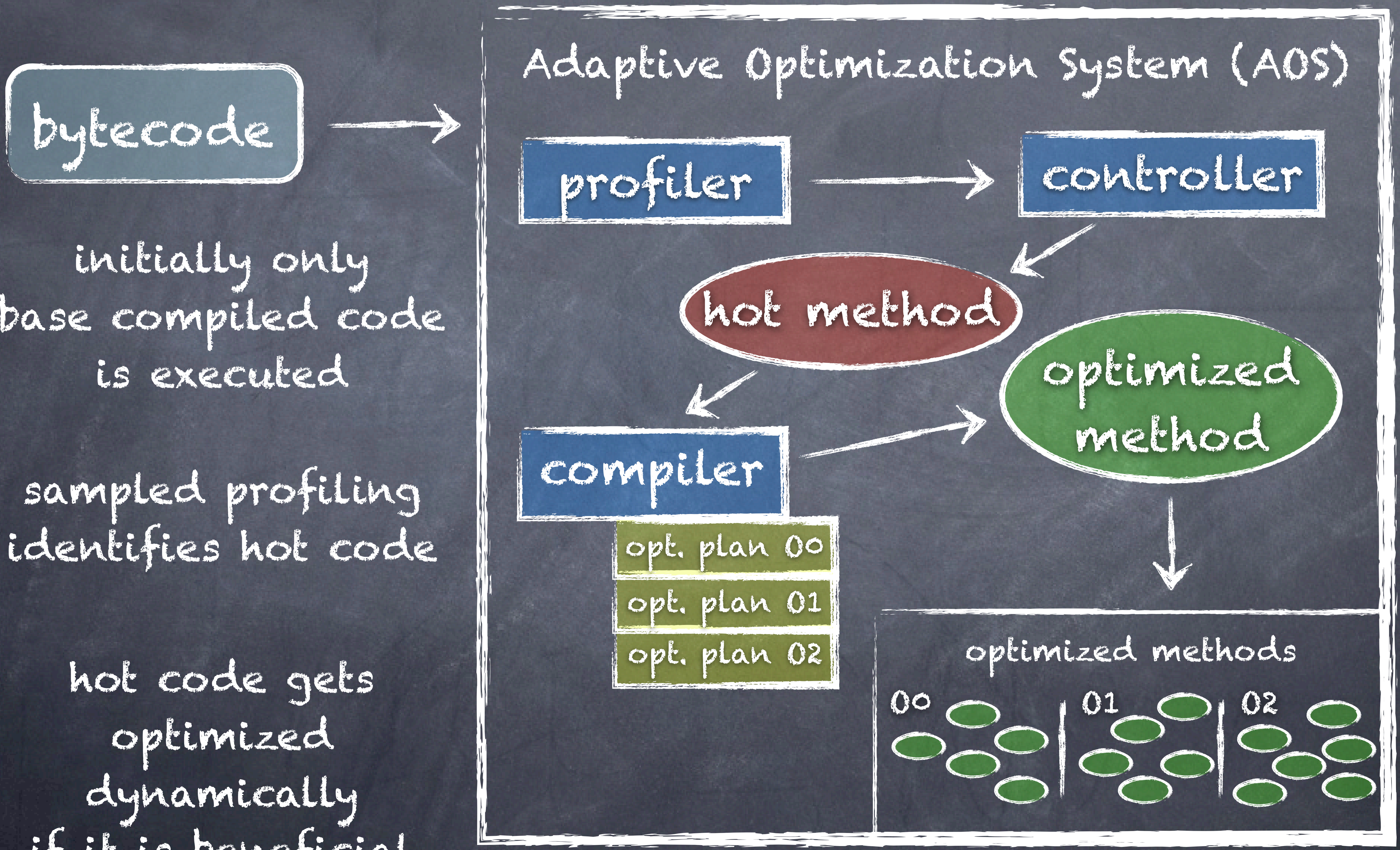
8 selected plans

=> 92 initial JIT compiler configurations

Experimental setup

- JikesRVM v3.0.1 (Java), 32-bit production build
- 16 benchmarks (SPECjvm98: 7, DaCapo 2006-10-MR2: 9)
- 4 different hardware platforms
  - AMD Opteron
  - Intel Pentium 4
  - Intel Core 2
  - Intel Core i7
- both steady-state and startup performance
- statistically rigorous performance analysis
- different heap sizes are considered (min. x2/x4/x8)

JIT compilation in Jikes RVM



initially only
base compiled code
is executed

sampled profiling
identifies hot code

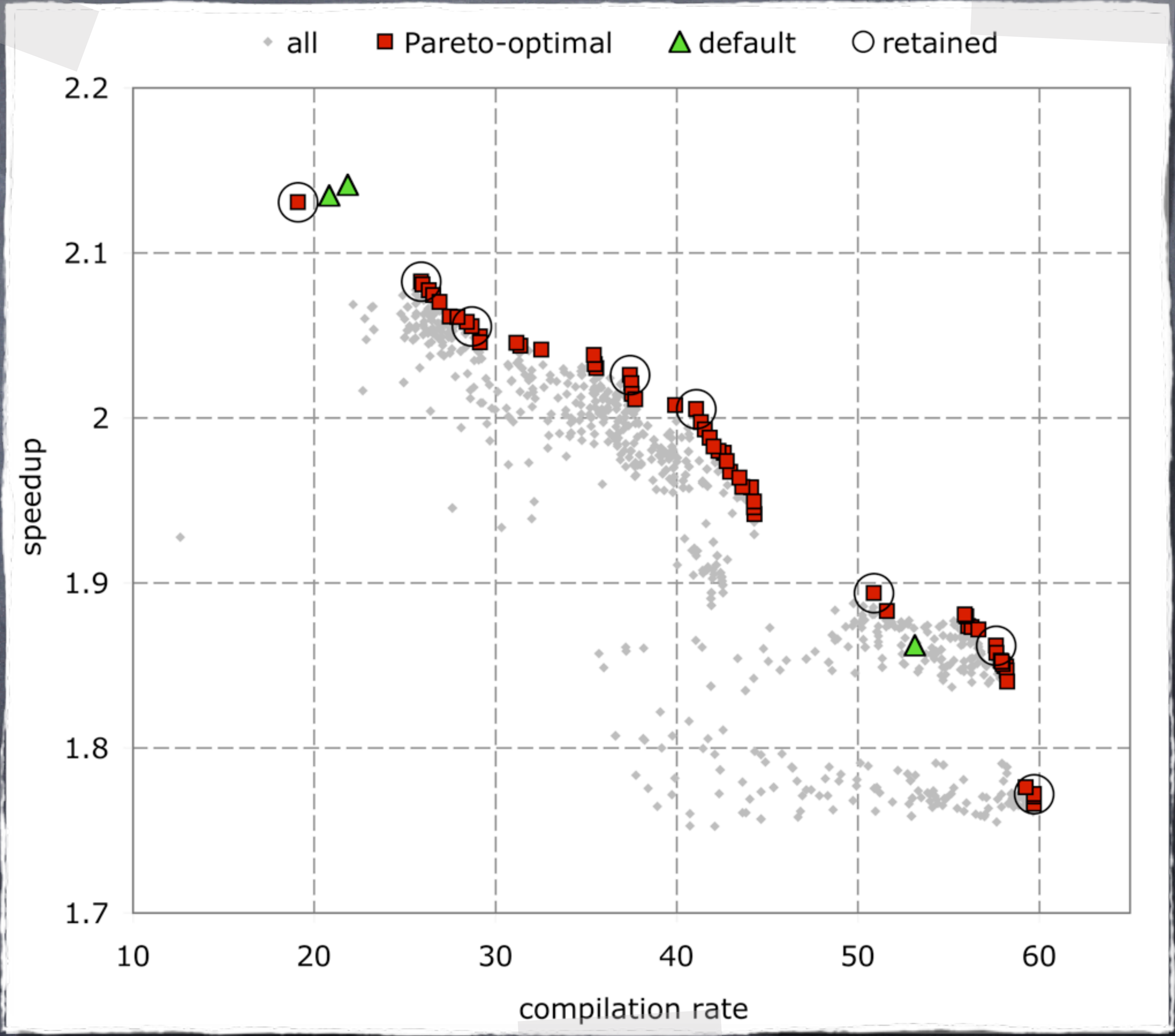
hot code gets
optimized
dynamically
if it is beneficial

Global tuning: optimization plans

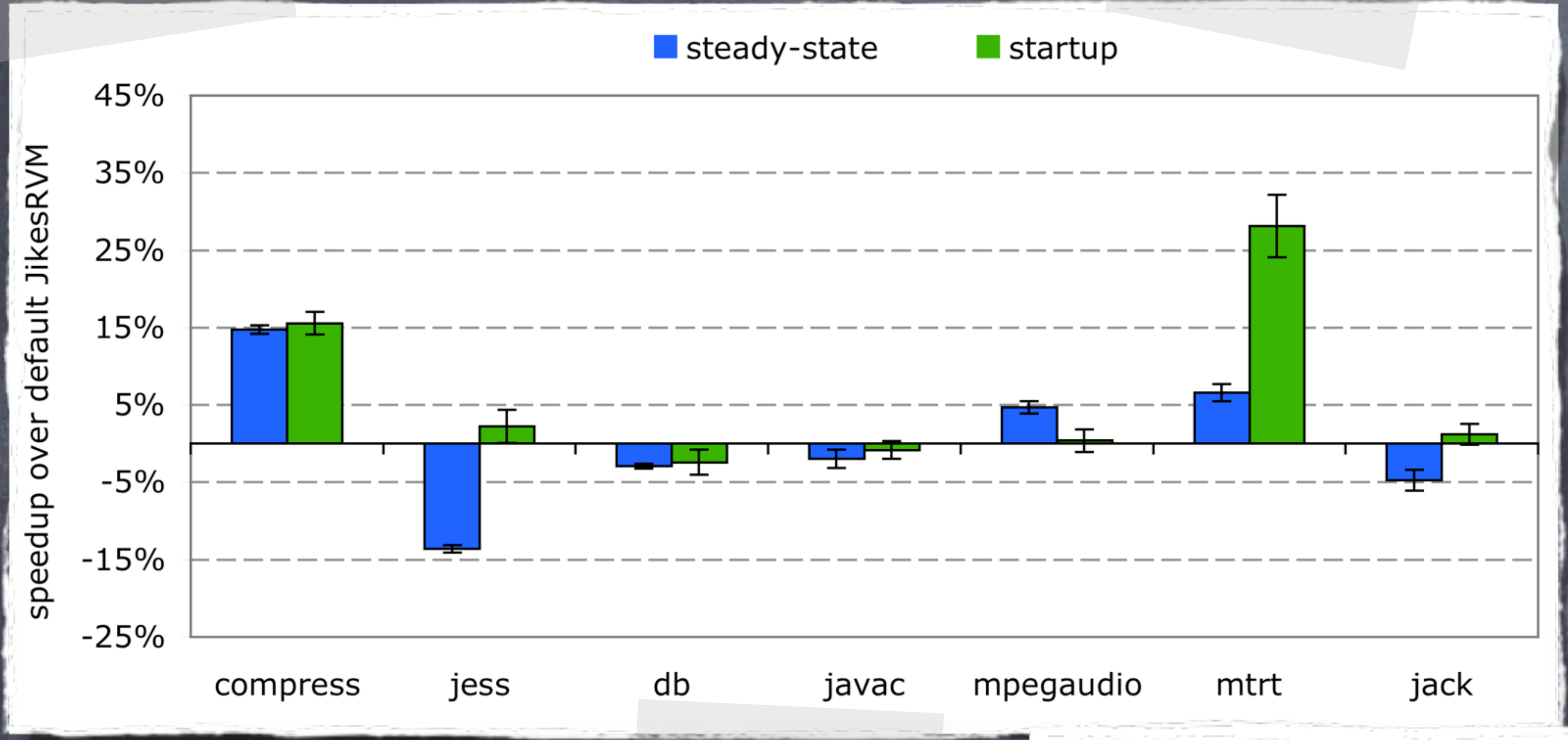
Pareto optimal optimization plans

=> competitive with manually tuned optimization plans

=> too many, so pick a selected subset with a good spread along Pareto-curve

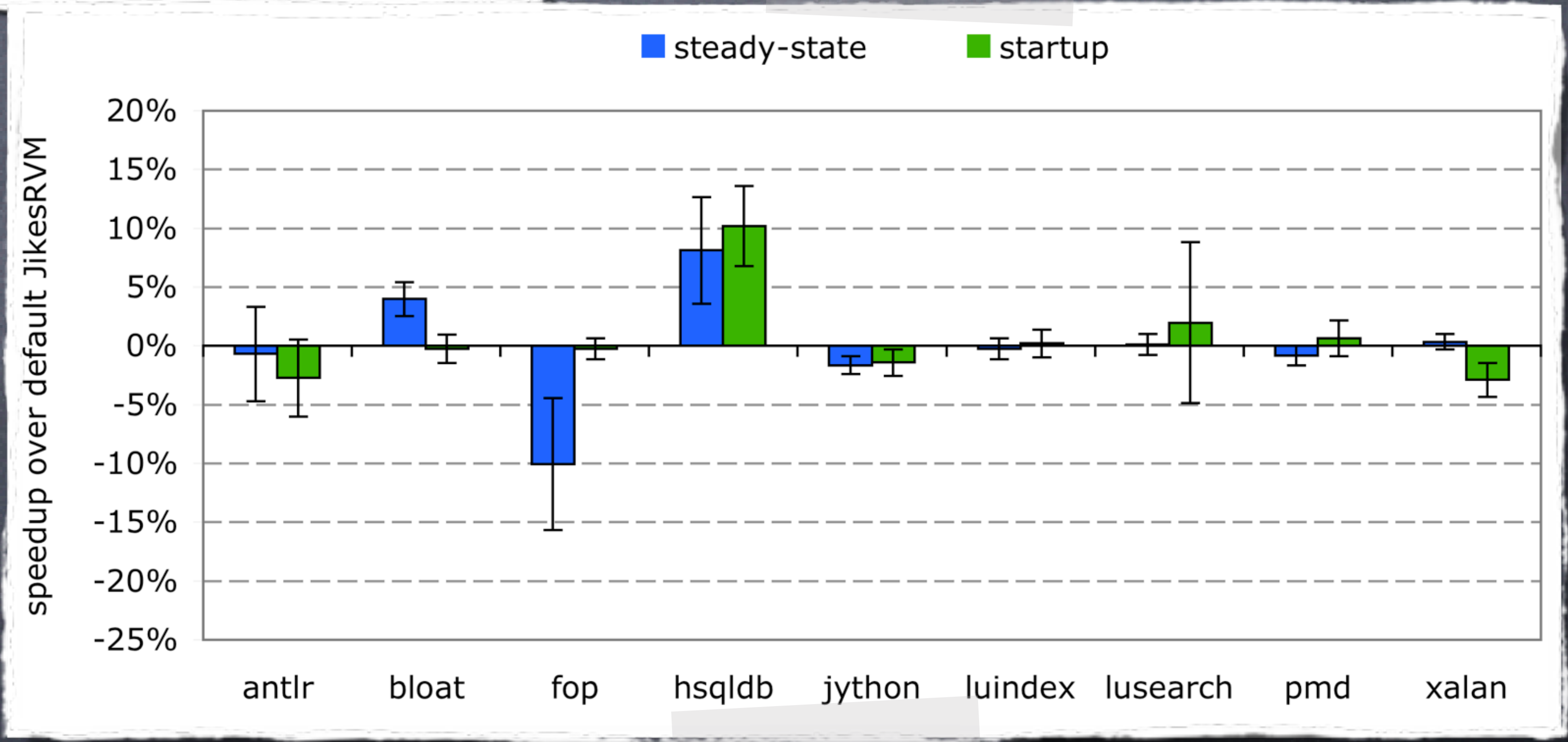


Global tuning: JIT compiler settings



tuning for SPECjvm98

tuning for DaCapo

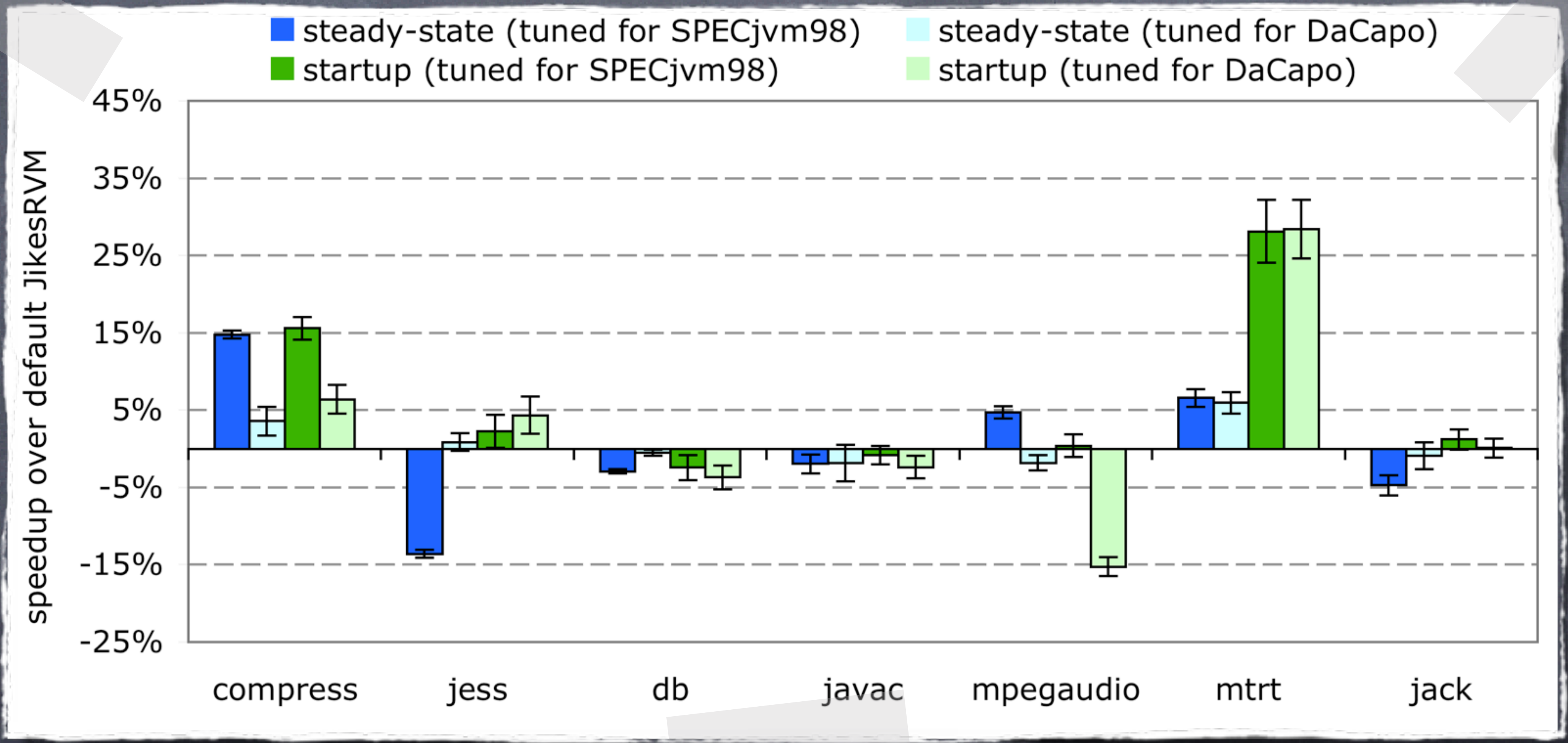


point of reference:
manually tuned
default Jikes RVM

roughly same steady-state performance as manually tuned default, slightly better startup performance

Cross-validation

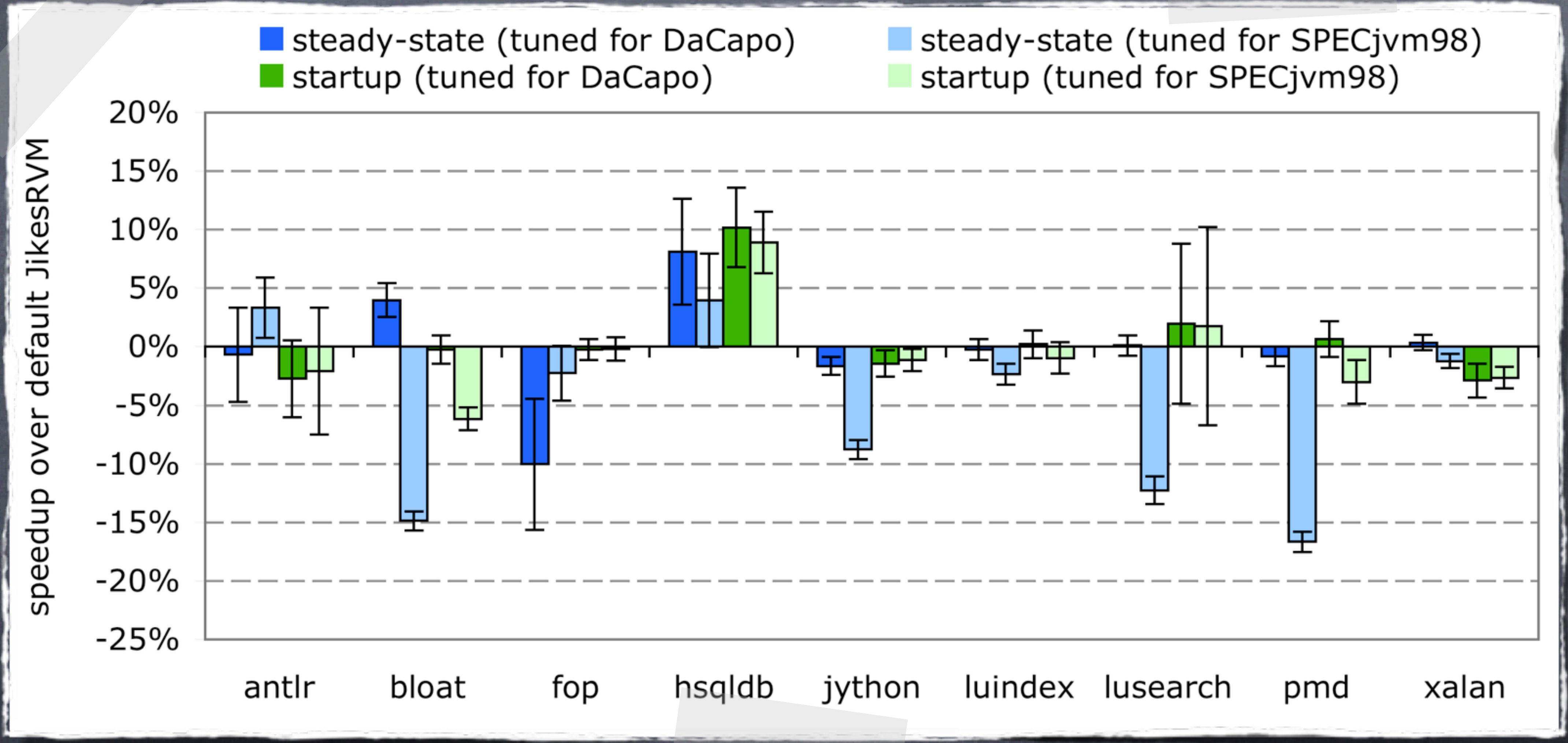
tune for DaCapo, evaluate with SPECjvm98



JIT compiler tuned for DaCapo performs well for SPECjvm98

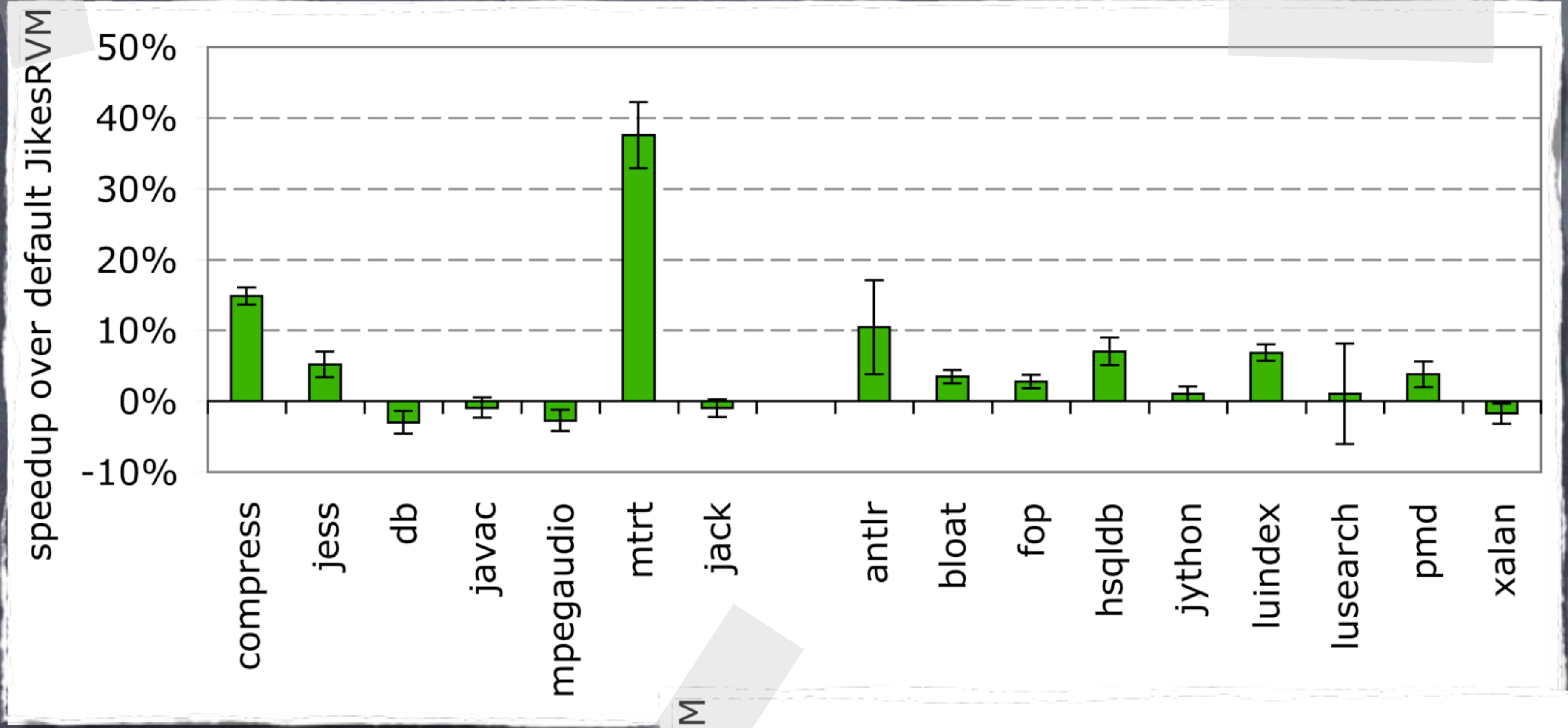
Cross-validation

tune for SPECjvm98, evaluate with DaCapo



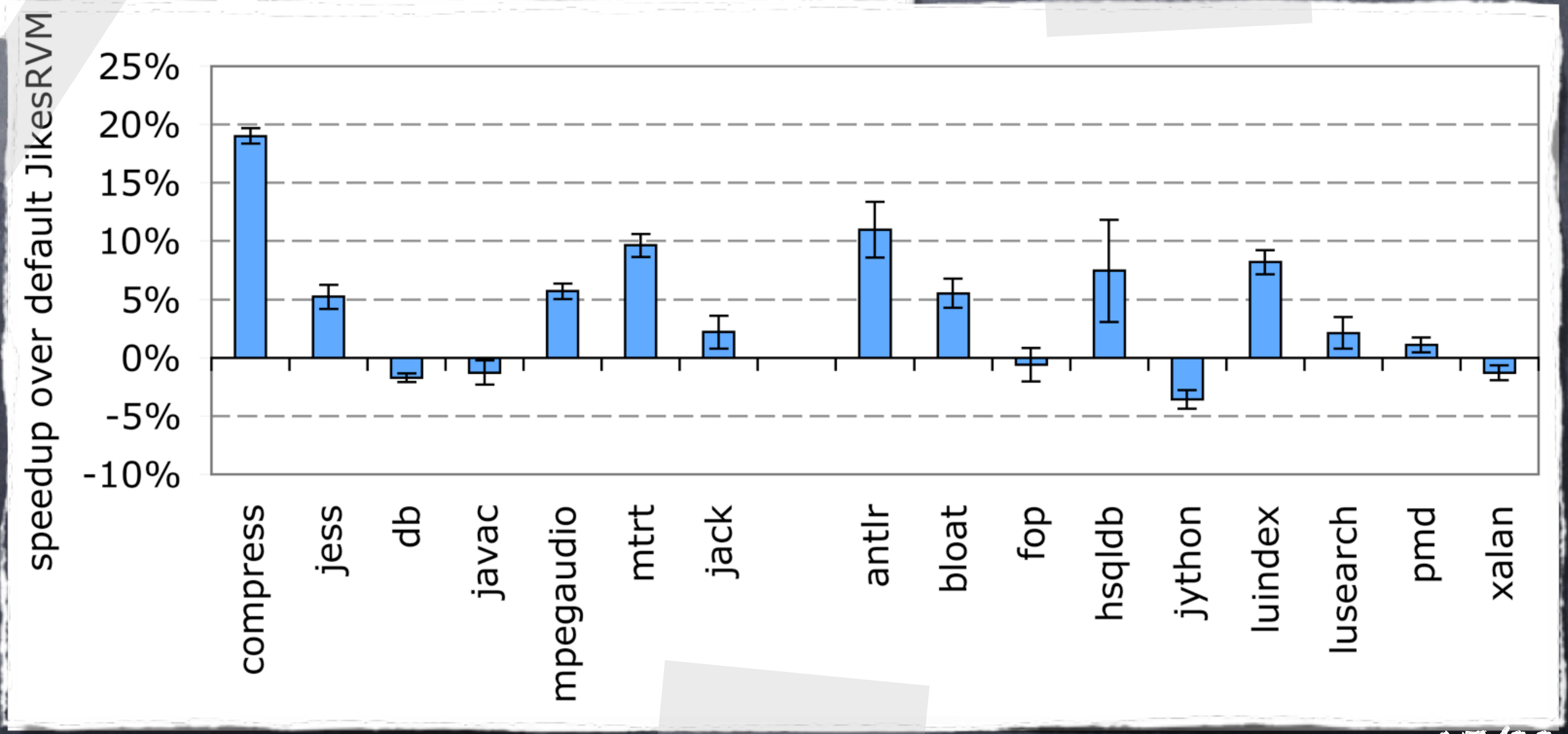
JIT compiler tuned for SPECjvm98 ~~performs well~~ for DaCapo $\longrightarrow$ DaCapo is a lot more complex !!!

Application-specific tuning



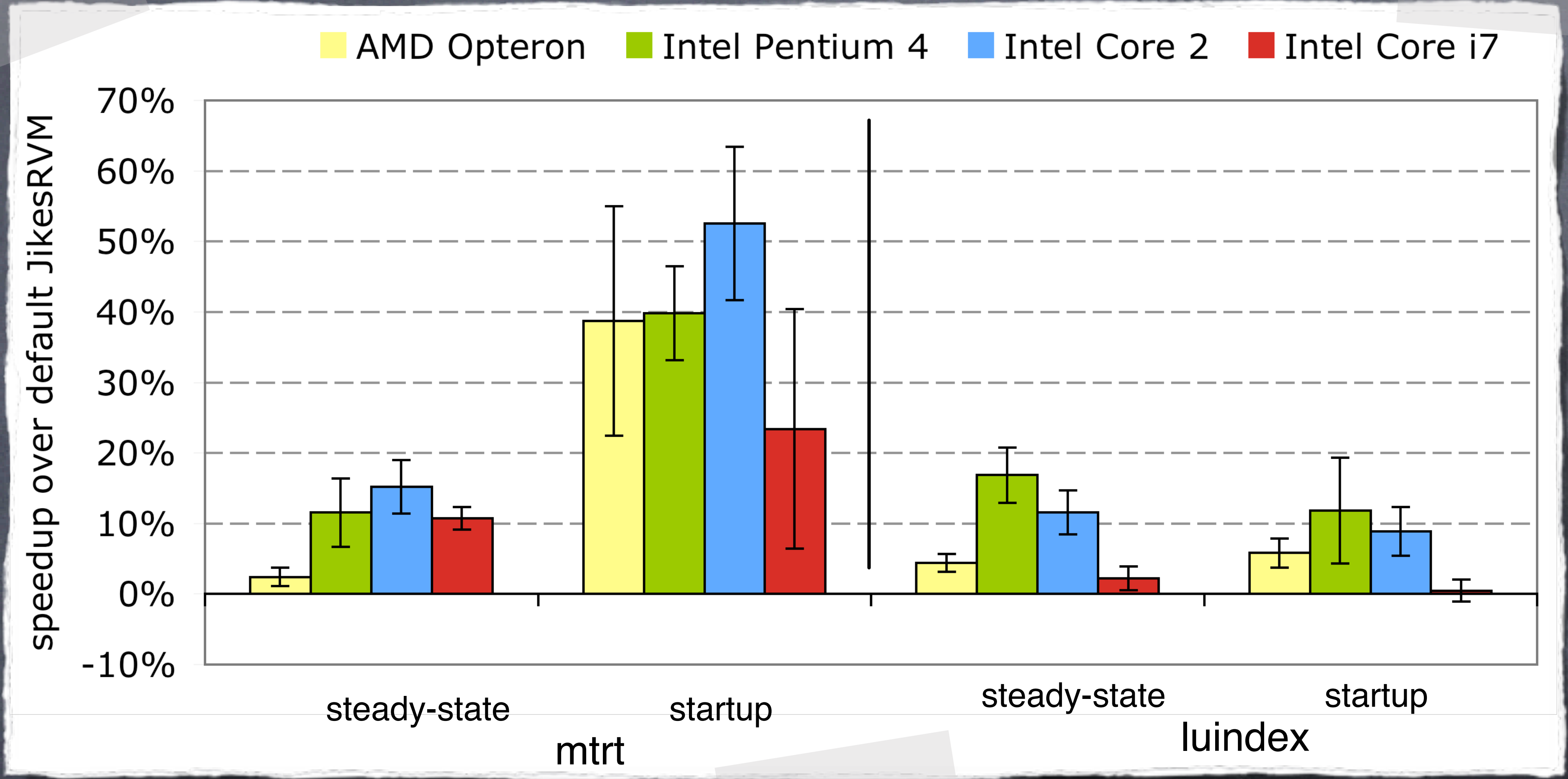
startup

steady-state



significant speedups for several benchmarks by specializing the JIT compiler for one single application

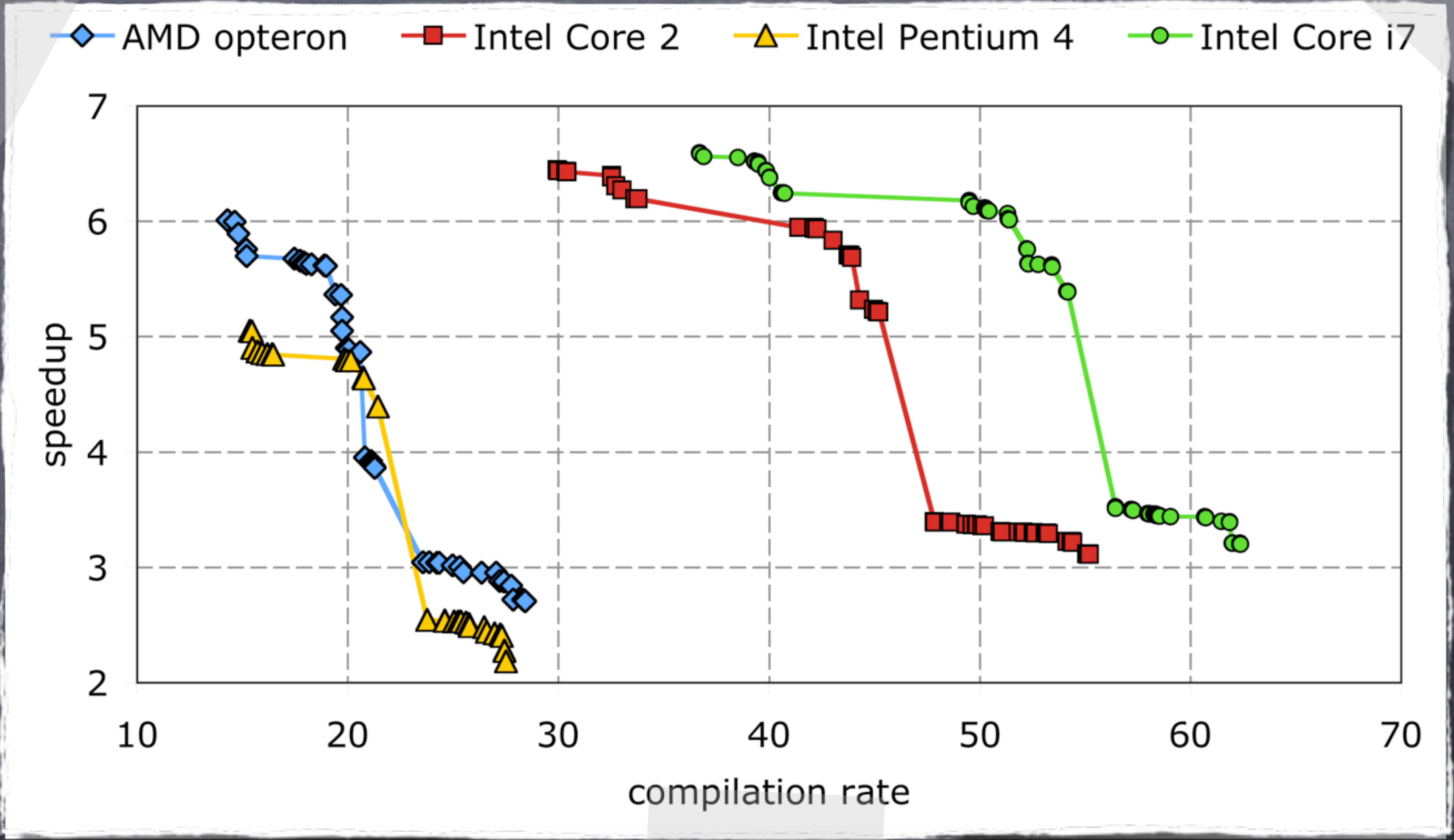
Cross-platform evaluation



significant speedups for different hardware platforms

Retuning for a different platform

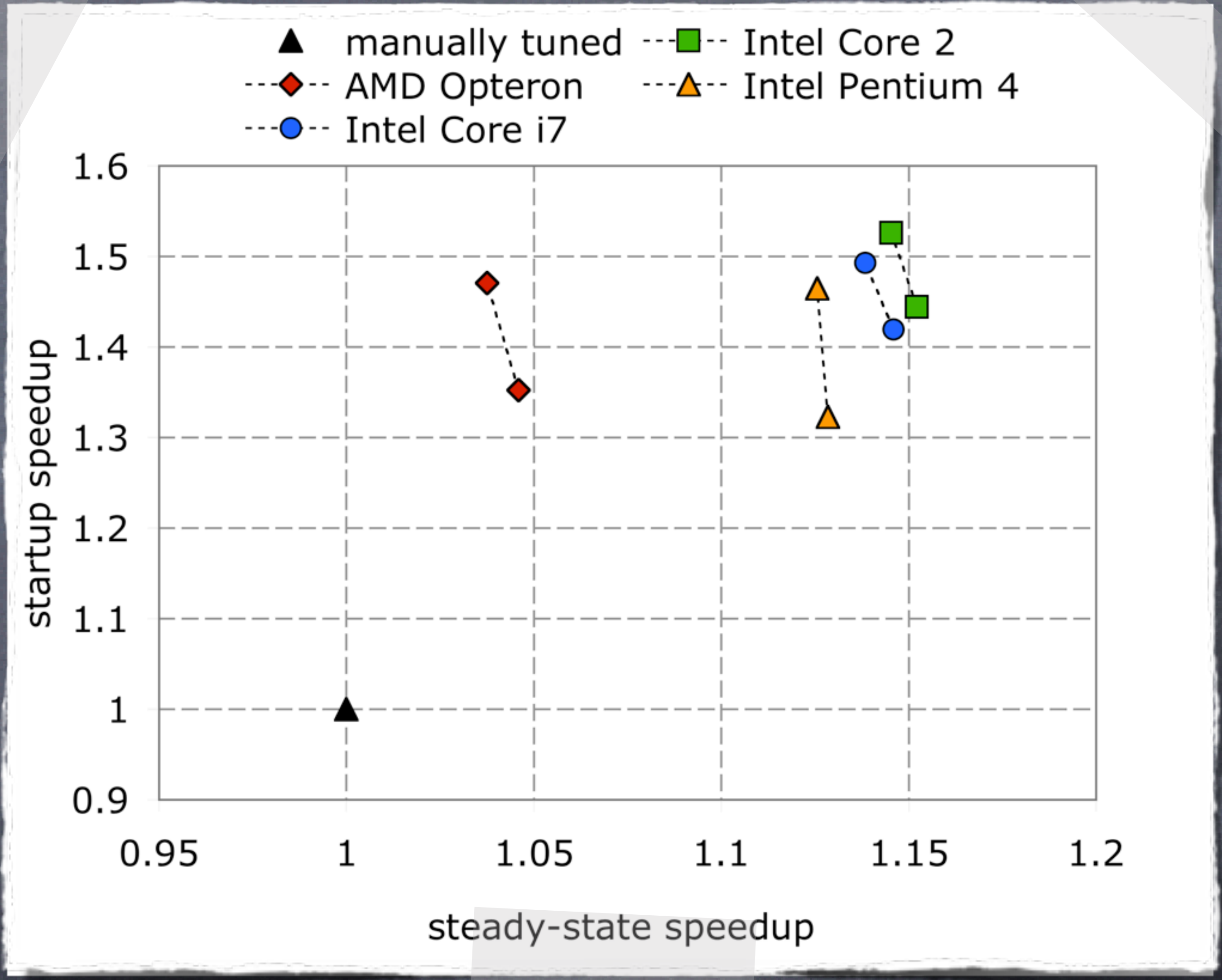
optimization plans



different platforms result in different tradeoffs

Retuning for a different platform

cross-validation of JIT compiler tuned for mtrt @ Intel Core 2



<u>retuning</u> for a new platform is important

to obtain to best possible performance

Exploration time

- evaluating an optimization plan or JIT compiler setting takes time
    - execute (all) application(s) multiple times
    - embarrassingly parallel (per generation)
- global tuning for SPECjvm98 and DaCapo
    - step 1: +/- 550 hours, step 2: 1320 hours
    - with sufficient resources: about 3 days
- application-specific tuning: matter of hours
- feasible, but room for improvement
    - limit number of evaluations
    - partial evaluation (e.g., only some benchmarks)

Conclusions

automatically tuning a JIT compiler is feasible

- average performance is competitive with a manually tuned JIT compiler
- tuning the JIT compiler for one application yields significant speedups
- retuning for a different set of applications, or a different platform, is important to obtain really good performance

Automated Just-In-Time Compiler Tuning

Kenneth Hoste, Andy Georges and Lieven Eeckhout
Computer Systems Lab
Ghent University, Belgium
kenneth.hoste@elis.ugent.be

CGO 2010
April 26th 2010