Method-Level Phase Behavior in Java Workloads

Andy Georges, Dries Buytaert, Lieven Eeckhout and Koen De Bosschere

Ghent University

Presented by Bruno Dufour

dufour@cs.rutgers.edu

Rutgers University DCS

Outline

- Introduction & motivation
- Experimental setup
- Method-level phases
  - Profiling techniques
  - Data analysis
- Statistical techniques
- Results
- Conclusions

Introduction

- Java workload: Java application + Java Virtual Machine (JVM)
- Application and JVM interact at runtime
  - Application complexity is increasing
  - VM complexity is increasing
    - VM Implementation: (smart) interpreters, JITs & optimizations, ...
    - Runtime support: GC, thread scheduling, class loaders, finalizer mechanism, ...
- Problem: Need automated ways to analyze and understand Java workload behaviour
  - Focus on **low-level** behaviour characteristics (i.e. hardware performance metrics)

Method-level Phase Behaviour

- Relies on a strong correspondance between phases and code organisation
  - Behaviour of a method over time expected to have low variation
- Java is strongly object-oriented, methods are (on average):
  - short
  - frequently executed
- Methods should provide a good level of abstraction for phases.

Method-level Phase Behaviour (2)

- Goal: Cluster executed methods into *phases* based on runtime information (offline).
  - Collect timing information
  - Find method-level phases
  - Profile each phase to measure behaviour characteristics

Experimental Setup

- Hardware & Performance counters
- Virtual Machine
- Benchmarks

Experimental Setup – Hardware

- AMD Athlon XP 2.1 Ghz
  - 64 KB L1 I-Cache + 64 KB L1 D-Cache
  - 256 Kb (unified) L2 cache
  - ...
- 4 performance counter registers
  - Programmable
  - Can measure 60+ event types (cycles, retired instructions, cache misses, ...)
  - Used to compute hardware-level performance metrics
    - Normalize measurements # of retired instructions
- Performance API (PAPI) provides abstraction layer for increased portability

Experimental Setup – Virtual Machine

- Jikes Research Virtual Machine (RVM)
  - No interpretation (Pure JIT)
  - Implemented in Java
  - 3 compilation strategies:
    - **Baseline**: fast, unoptimized compilation.
    - **Optimizing**: slow, optimized compilation.
    - **Adaptive**: baseline first, then recompilation of hot methods as needed.
  - Generational GC
  - Variable number of **virtual processors**, i.e. kernel threads
  - Built-in support for harware counters
    - Counters monitored on per-thread basis

Experimental Setup – Benchmarks

	Benchmark	Description
SPECjvm98	Compress	Modified Lempel-Ziv compression/decompression
	Jess	Expert shell system
	Raytrace	Raytracer
	DB	Performs operations on memory-resident database
	Javac	JDK compiler (1.0.6)
	Mpegaudio	mp3 decoder
	Mtrt	Multithreaded version of Raytrace
	Jack	Java parser generator (now JavaCC)
	PseudoJBB	Modified warehouse simulation program

Method-Level Phases

- **Phase**: set of parts of program execution that exhibit similar characteristics.
  - Not necessarily temporally adjacent.
- Requirements:
  - Distinguish app/JVM
  - Distinguish between various parts of JVM
  - Recognize application phases
- Approach: Consider method + callees (subtrees rooted at m in call graph)
  - Coarse granularity limits runtime profiling
  - Granularity sufficiently fined-grained to identify phases

Method-Level Phases (2)

- Offline analysis
- Additional Goals
  - Complete temporal coverage
  - Unintrusive profiling
  - Compact traces
  - Rich traces

Data Gathering

- Strategy (overview):
  - Step 1 (online): Measure total number of clock cycles spent in each method
  - Step 2 (offline): Aggregate data from step 1. Build dynamic call graph annoted with result from step 1, and use it to identify phases.
  - Step 3 (online): Measure performance metrics for each phase.

Instrumentation

- Methods compiled as
  - **Prologue/Epilogue**: Used to implement setup method execution (e.g. calling conventions).
  - **Method body**: original body of method.
- Instrumentation supported by all Jikes RVM compilers
  - Instrumentation introduces new GC points
  - Must ensure that all stack maps are updated before running instrumented code
  - On-stack replacement (OSR) is supported.

Instrumentation (2)

- Counter values reset in prologue, read in epilogue
  - Includes all callees
  - Prologue/epilogue effect on counters attributed to caller
    - Claimed to be negligible in practice
  - Uses trace per-thread cyclic trace buffers for efficiency
    - Writing buffers to disk handled concurrently
- Handling exceptions:
  - Exceptions bypass epilogue
  - Need to instrument exception handling mechanism

Generating trace data

- Maximum of 35 bytes per record (37 with thread info)
  - 4 bits for event type
  - 4 bits for # of counters
  - 4 bytes for method ID
  - 8 bytes per counter
  - (Optional: 2 bytes for thread ID)
- Using a single file per thread requires serializing traces
- Can skip instrumenting methods that:
  - are shorter than 50 bytecodes, and
  - don't have a back-edge (i.e. no possibility of looping)

Instrumenting VM services

- Finalizer, GC and optimizer run in dedicated threads
  - Easily profiled using built-in technology
- Profiling compiler needs special VM modification

Phase Identification

- θ_{weight}: Method total time threshold.
- θ_{grain}: Method average time threshold.
- c_T: Total execution time (in clock cycles)
- c_m: Total execution time for method m.
- p_{total}: Portion of total execution time attributed to m

$$c_m = (p_{\text{total}})(c_T)$$

- p_{average}: $\frac{1}{\text{number of calls to } m}$

$$c_m = (p)(c_T)$$

- Goal: $p_{\text{total}} > \theta_{\text{weight}}, p_{\text{average}} > \theta_{\text{grain}}$

Statistical Techniques

- Need to quantify amount of intra-phase variation
  - Use Coefficient of Variation (CoV)

$$V = \frac{\sigma}{\mu}$$

  - CoV measures deviation of a variable from its mean
- Need to quantify inter-phase variations
  - Use ANOVA (ANalysis Of VAriance) technique
  - Compute p-value based on level of significance
  - Most p-values less than 10^-16 (i.e. more variation between phases than within phases)

Results

Selecting θ_{weight} and θ_{grain}

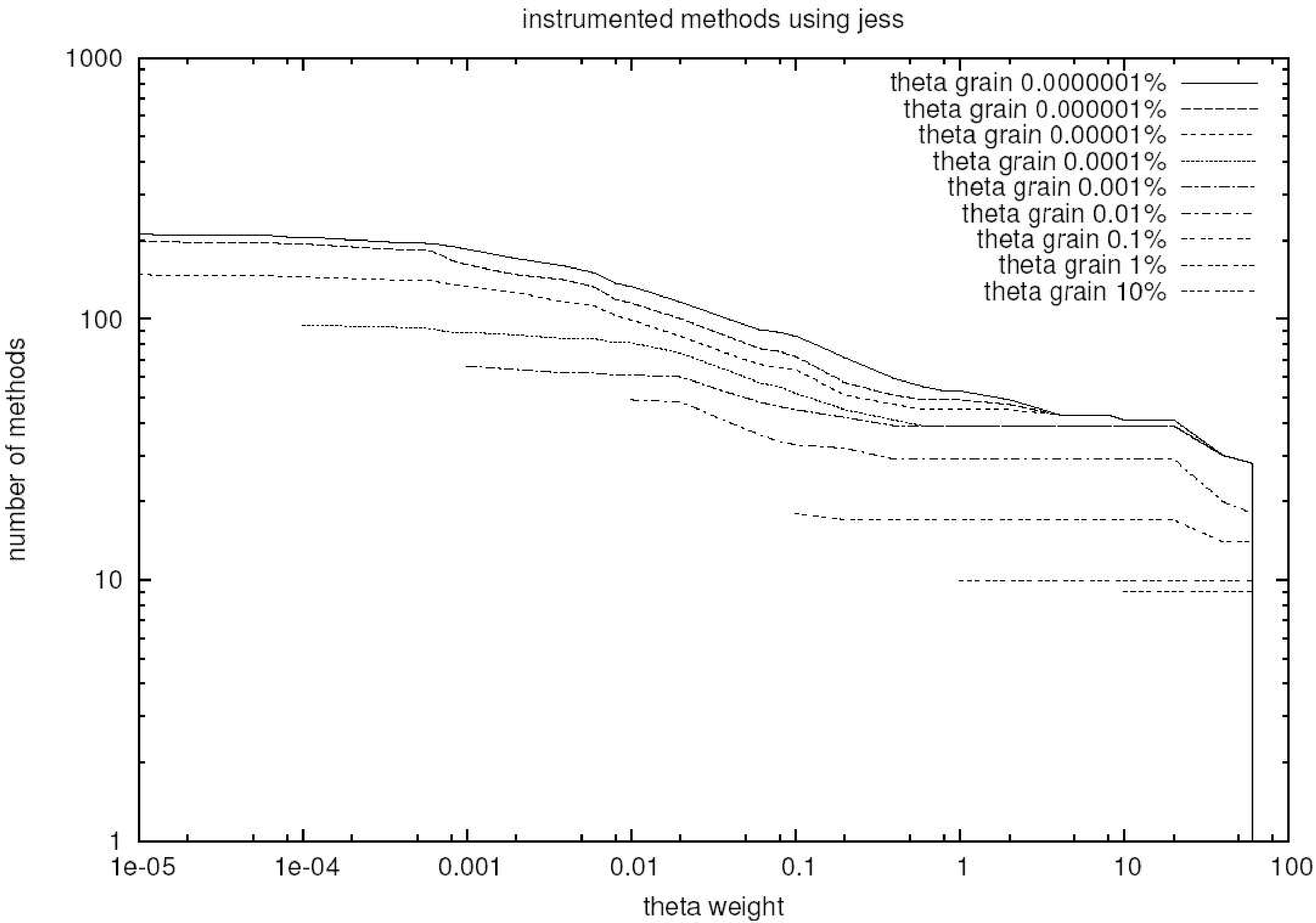
- θ_{weight} and θ_{grain} affect
  - Profiling cost
  - Precision
- Must find a tradeoff values based on
  - Maximum acceptable overhead
  - Required level of information
  - Application
- Estimate overhead as $\frac{\text{profiled method invocations}}{\text{total method invocations}}$
  - Choose overhead close to 1% (paper says $< 1\%$)

Overhead Estimation

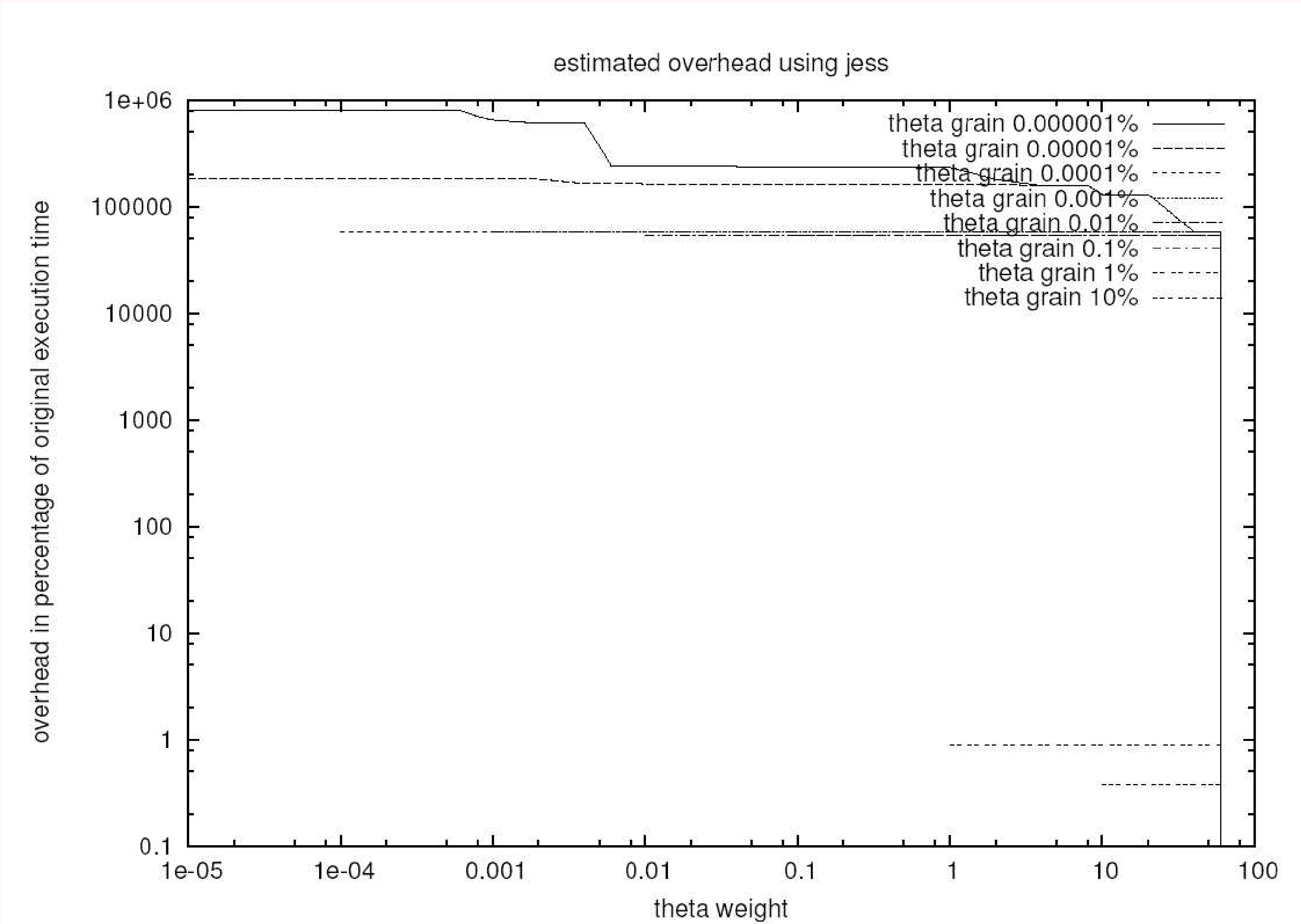
- How good is the overhead estimate?

Benchmark	Est.	Measured
Compress	1.84%	1.82%
Jess	1.22%	1.27%
DB	7.17%	5.61%
Javac	2.61%	2.11%
Mpegaudio	10.75%	3.52%
Mtrt	24.68%	7.83%
Jack	3.98%	4.28%
PseudoJBB	3.69%	6.65%

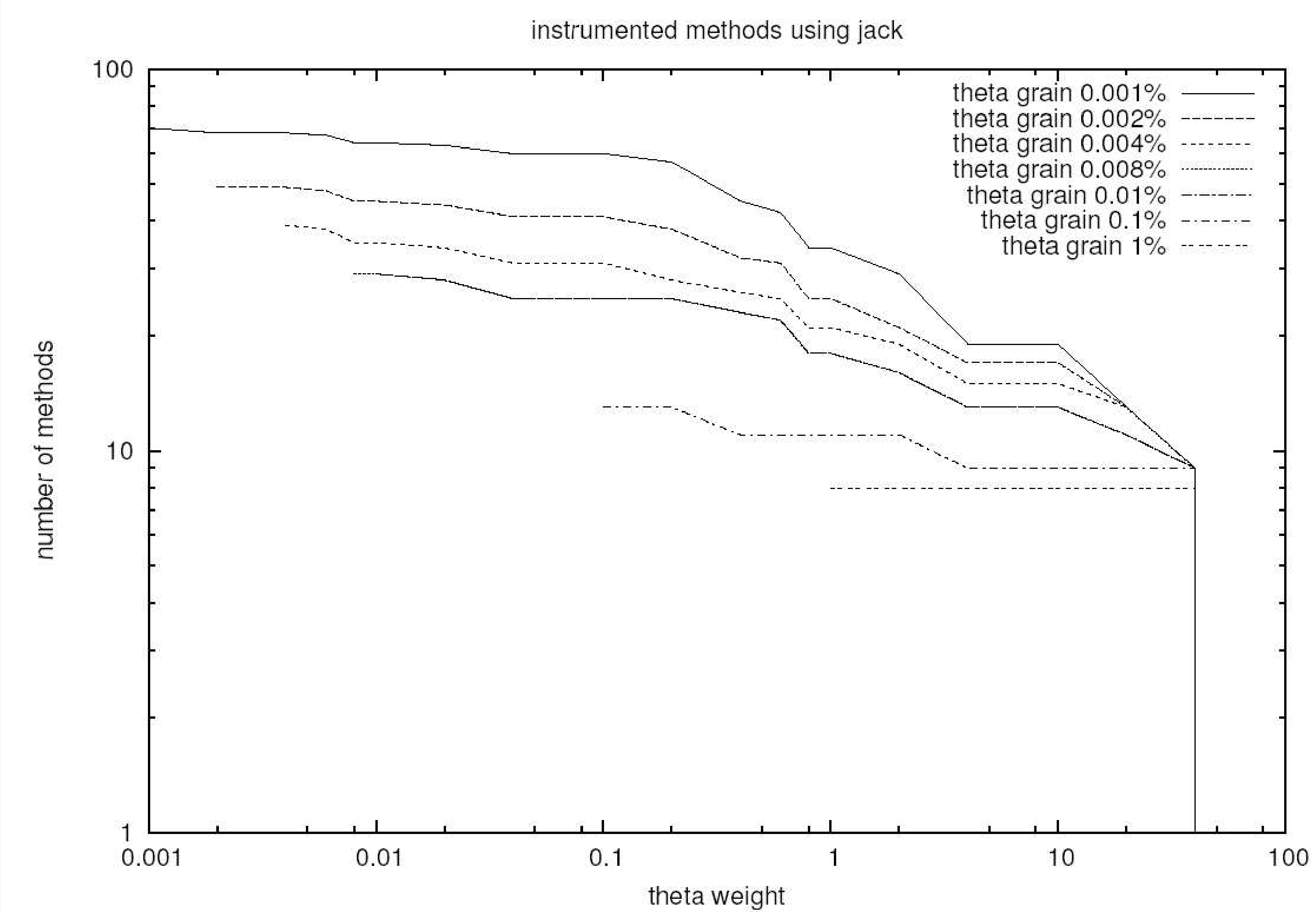
Instrumented Method (Jess)



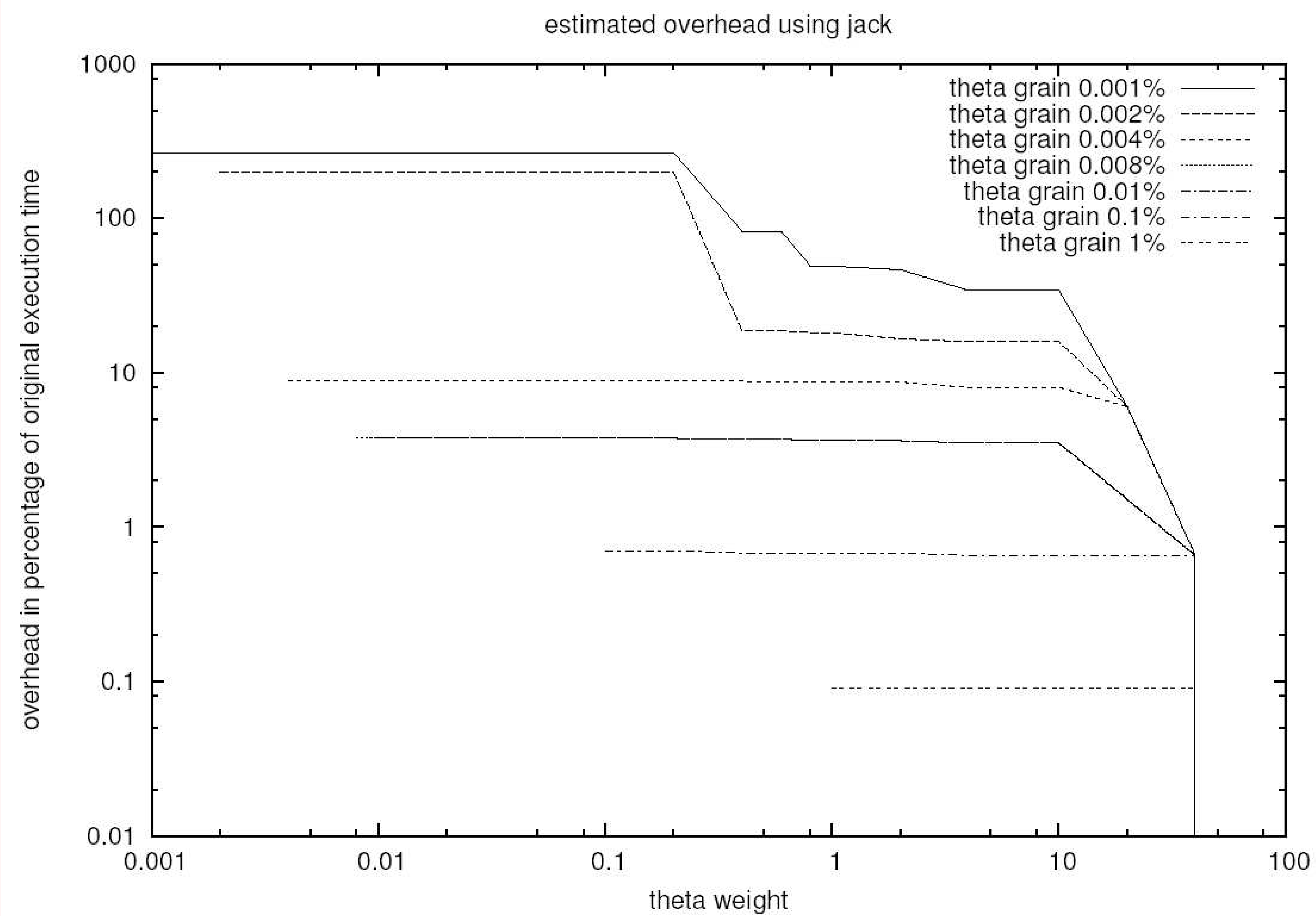
Estimated Overhead (Jess)



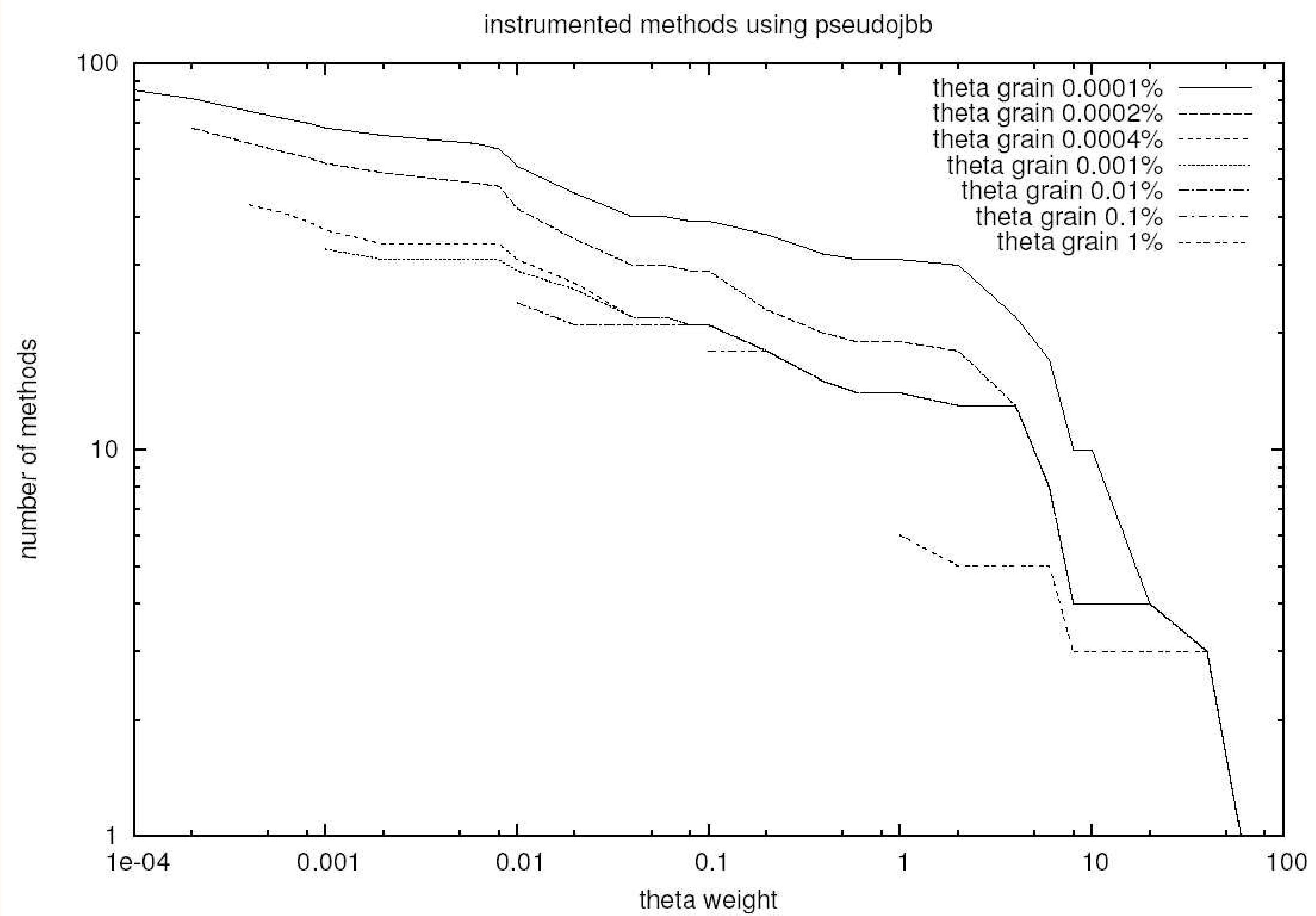
Instrumented Methods (Jack)



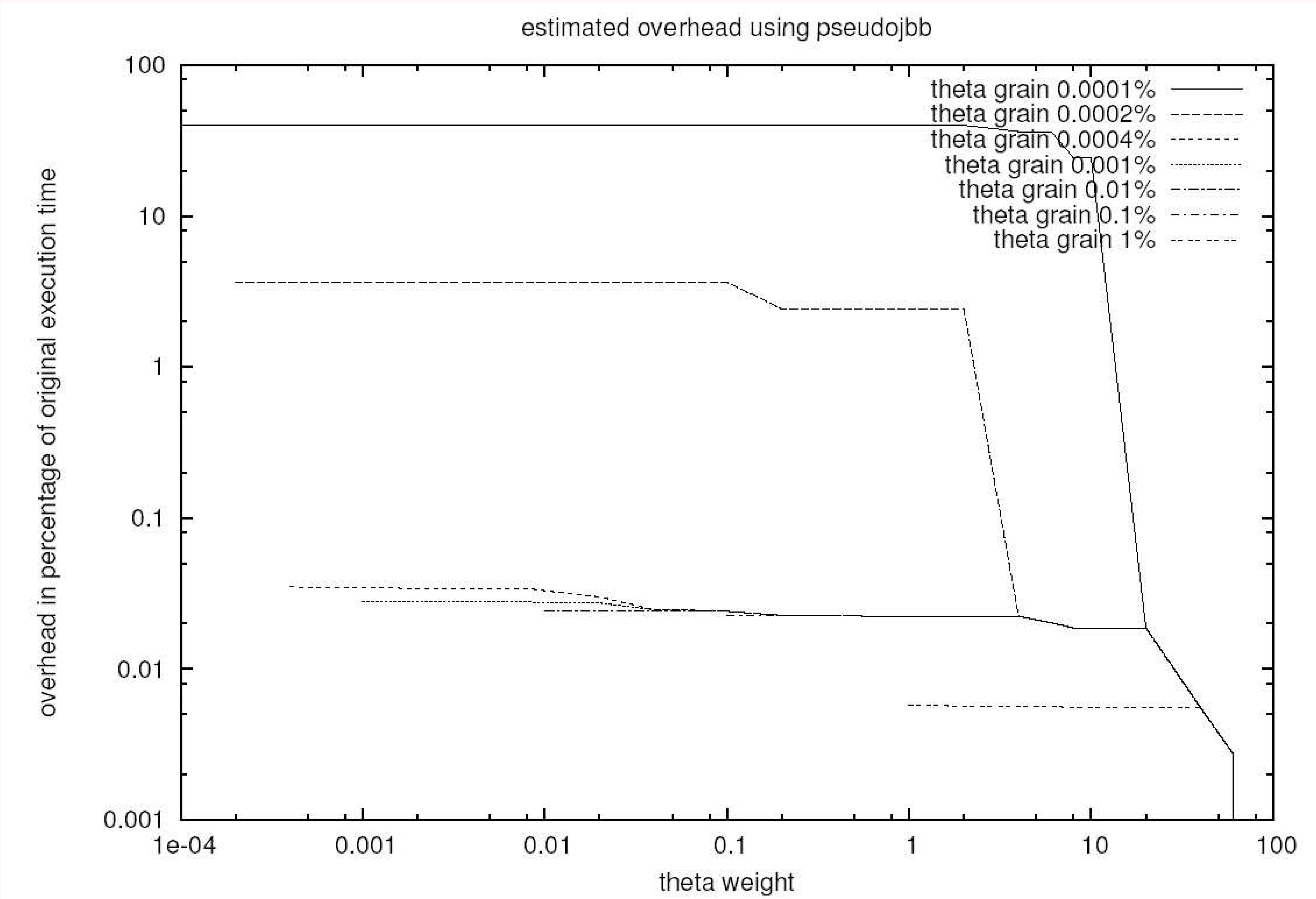
Estimated Overhead (Jack)



Instrumented Methods (PseudoJBB)



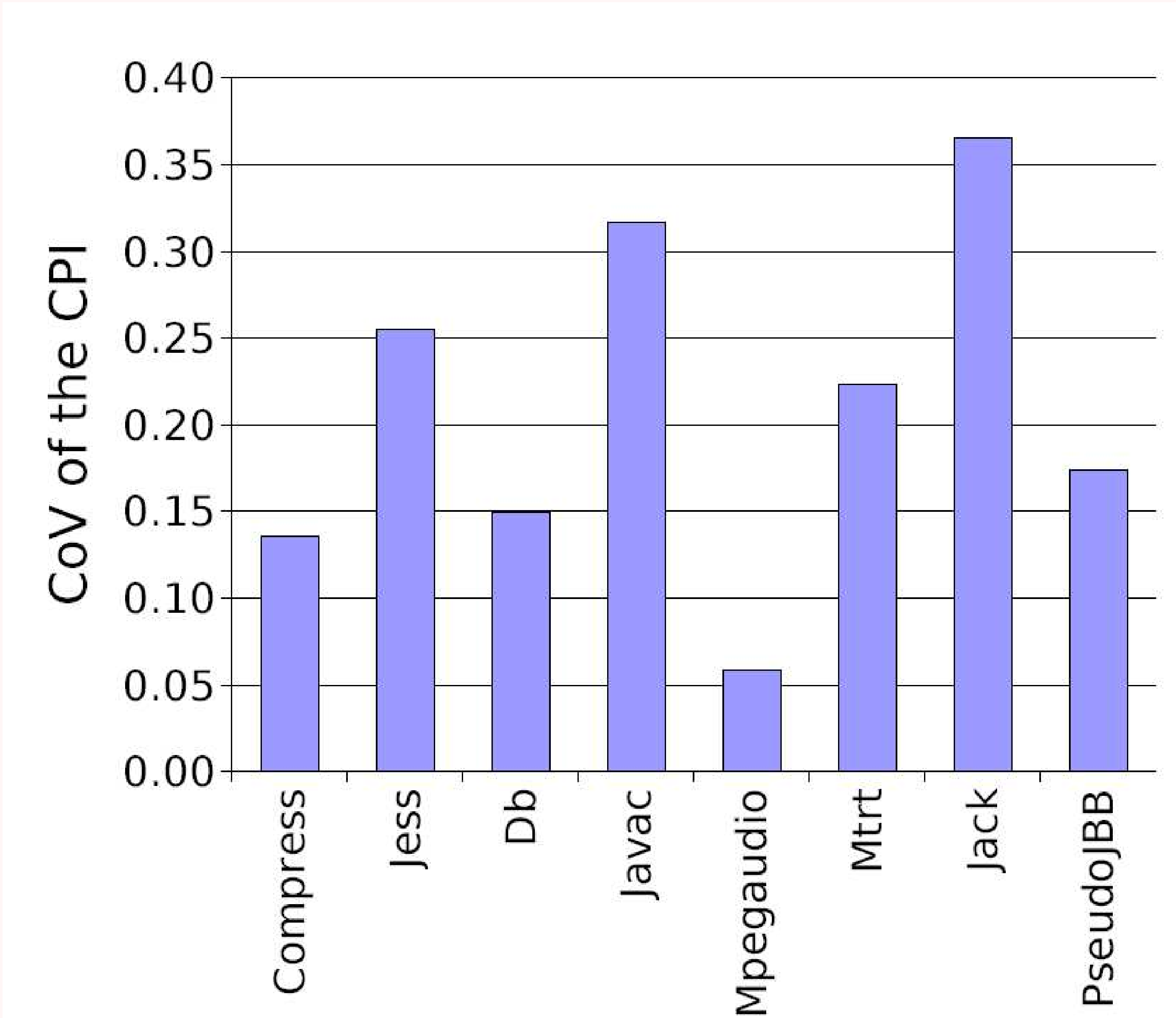
Estimated Overhead (PseudoJBB)



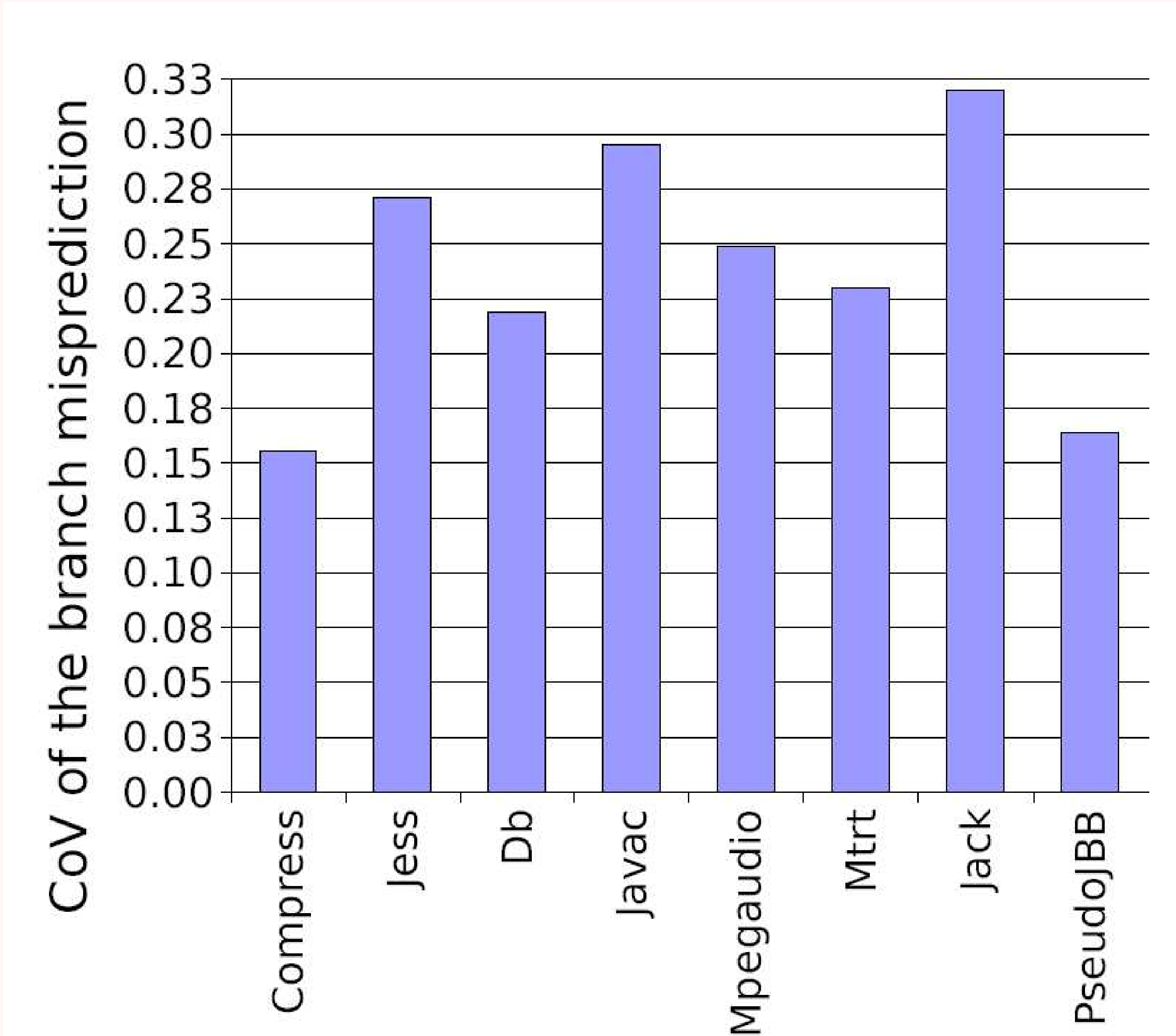
Variability between and within Phases

- CoV
- Boxplots

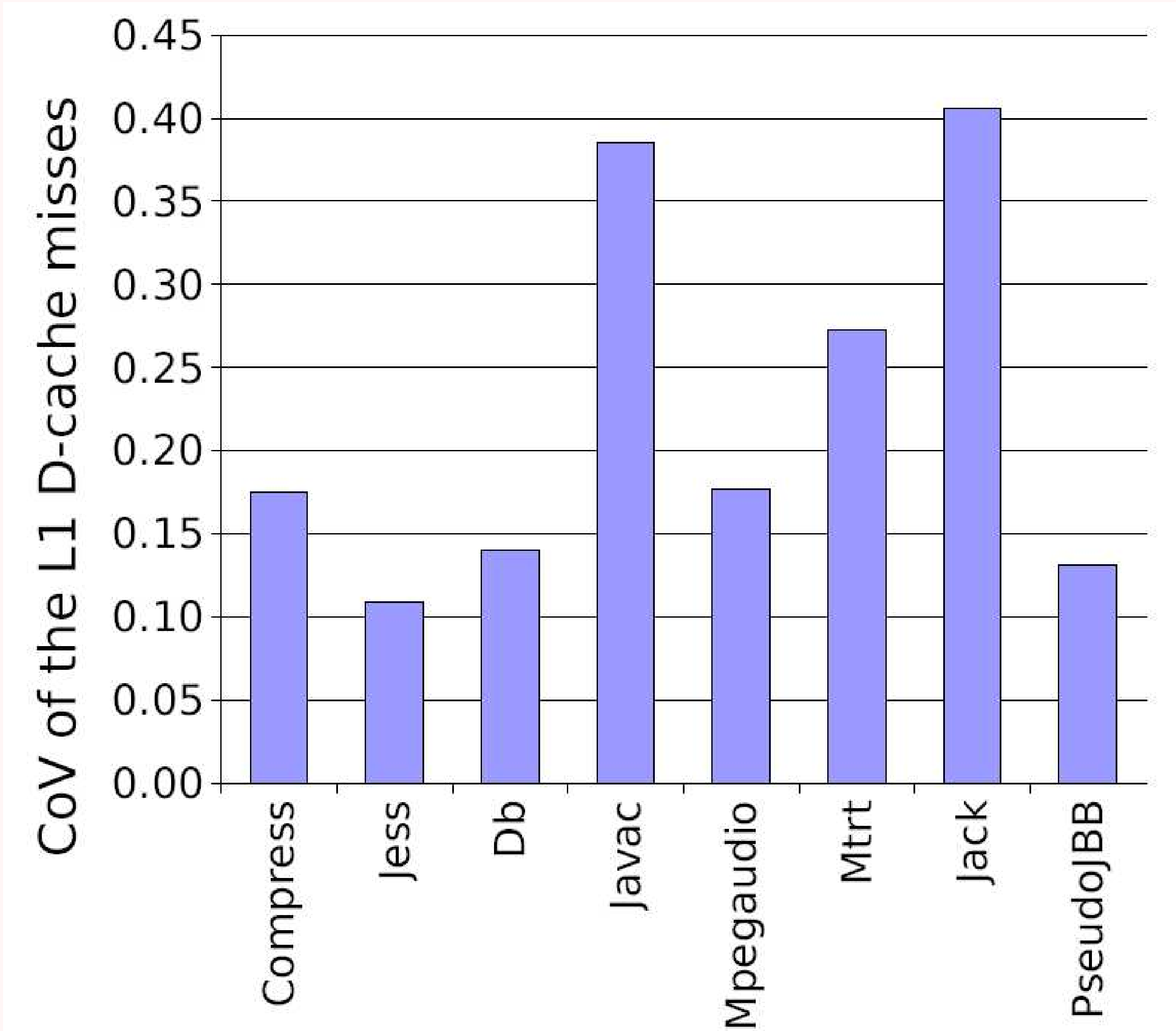
CoV of CPI


0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
CoV of the CPI
Compress
Jess
Db
Javac
Mpegaudio
Mtrt
Jack
PseudoJBB

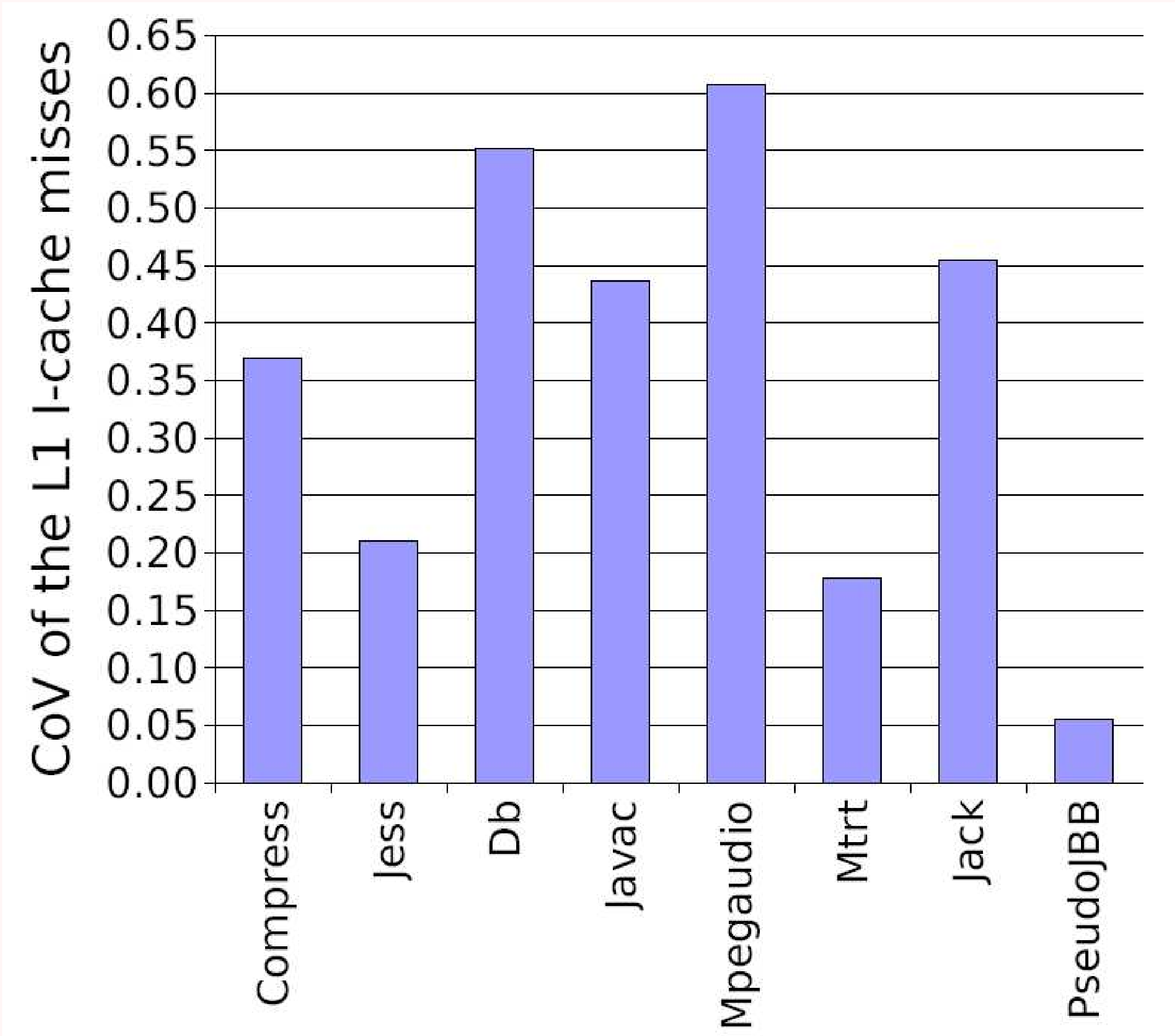
CoV of Branch Misprediction



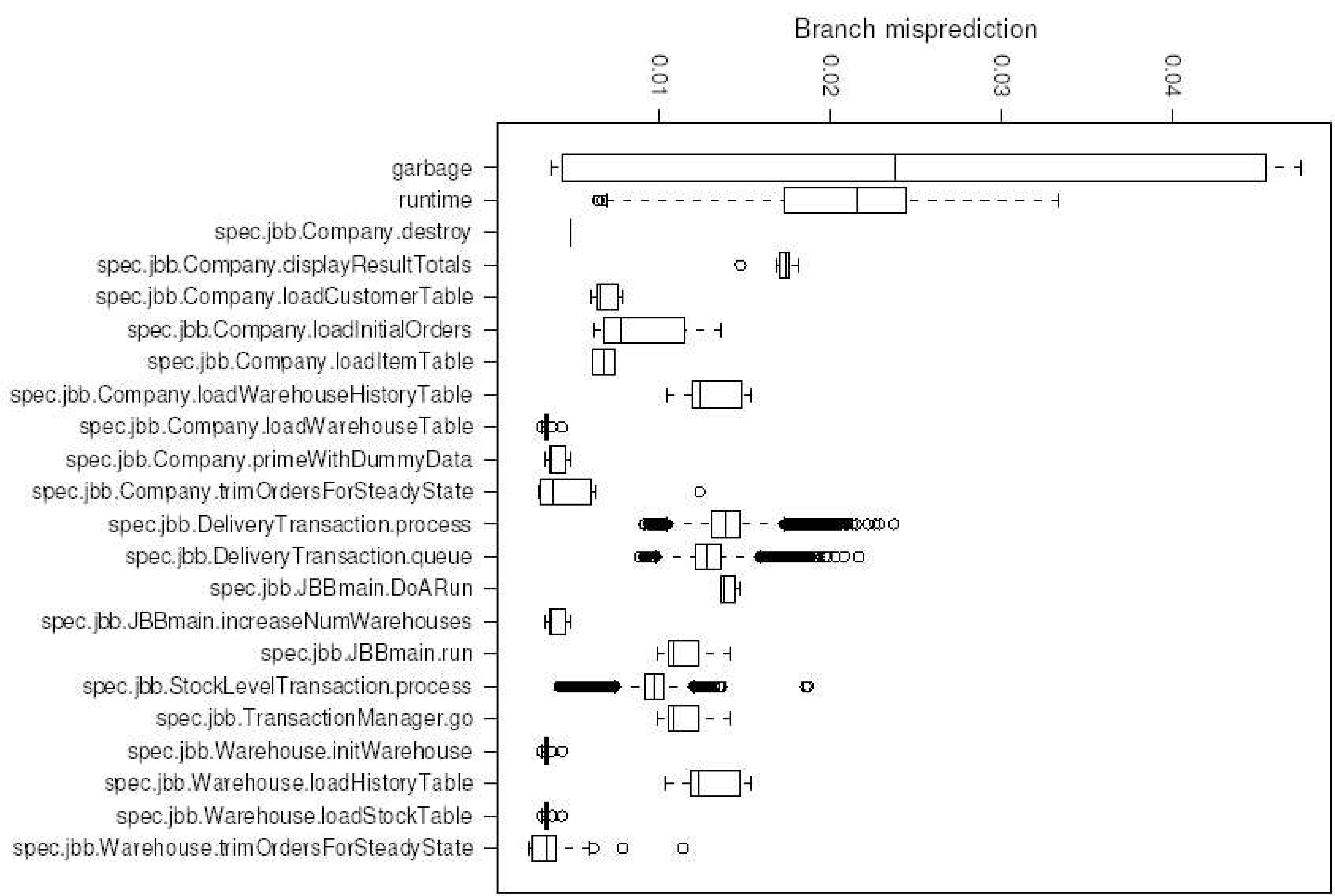
CoV of L1 D-Cache Miss



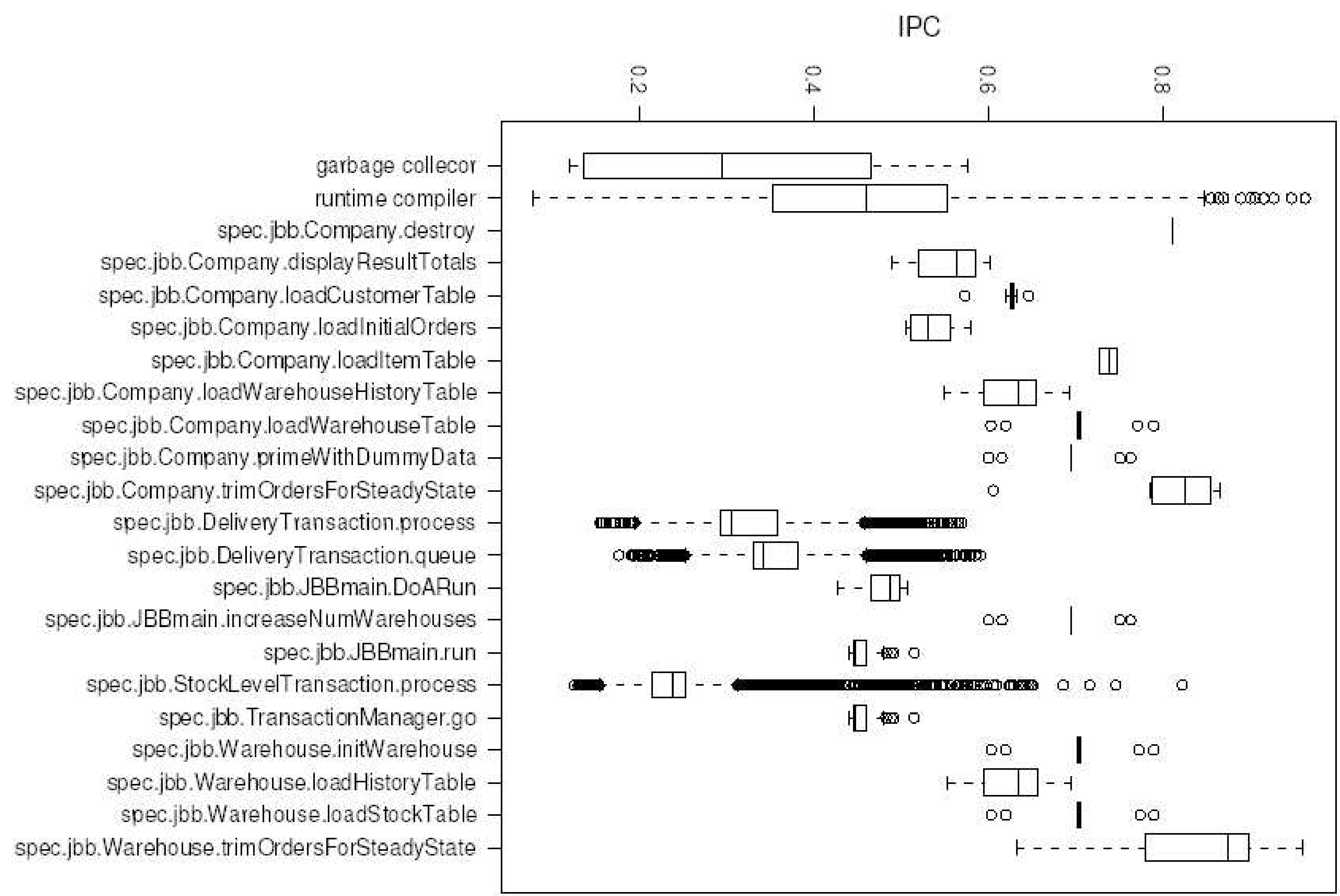
CoV of L1 I-Cache Miss



Branch Misprediction

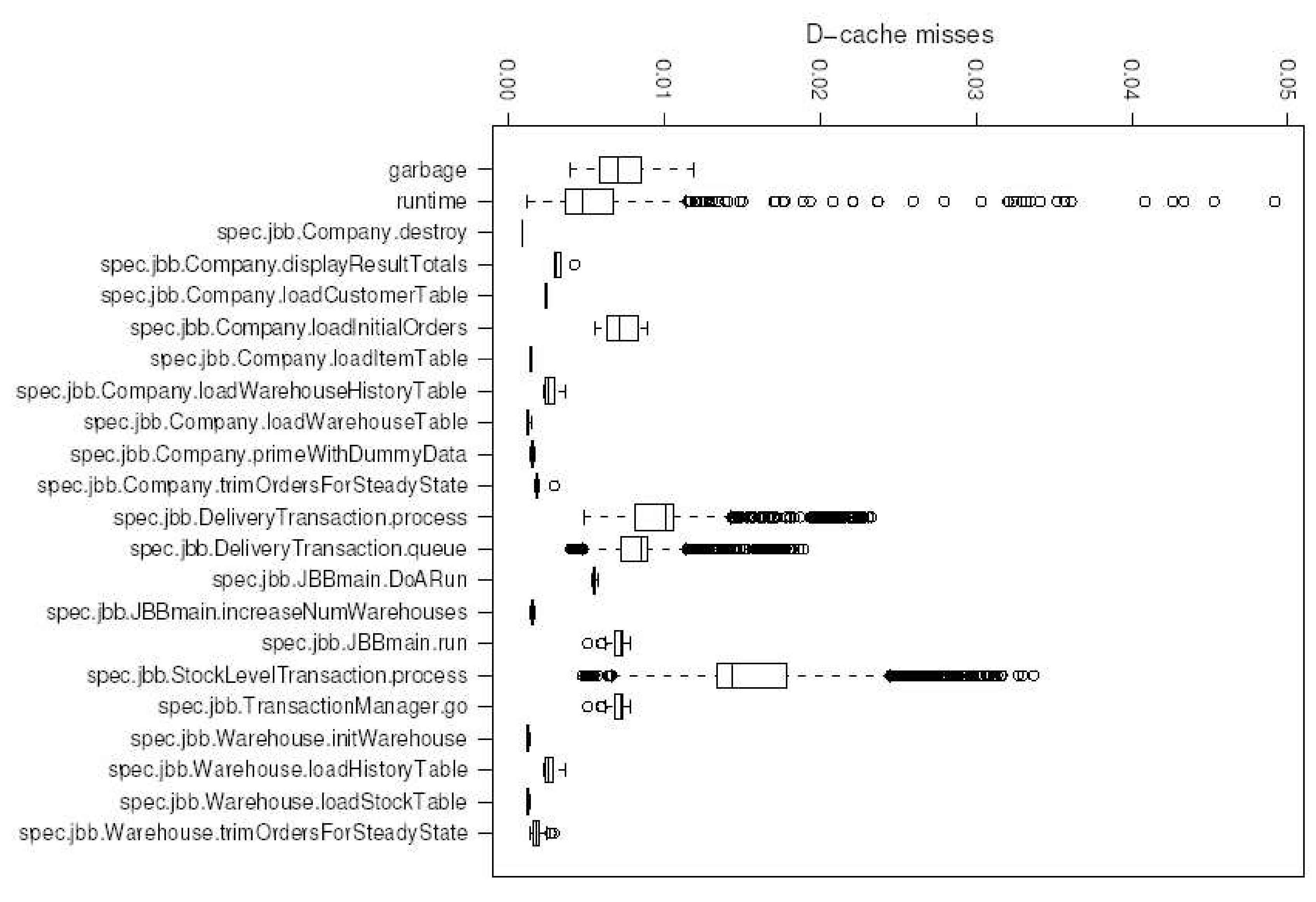


IPC

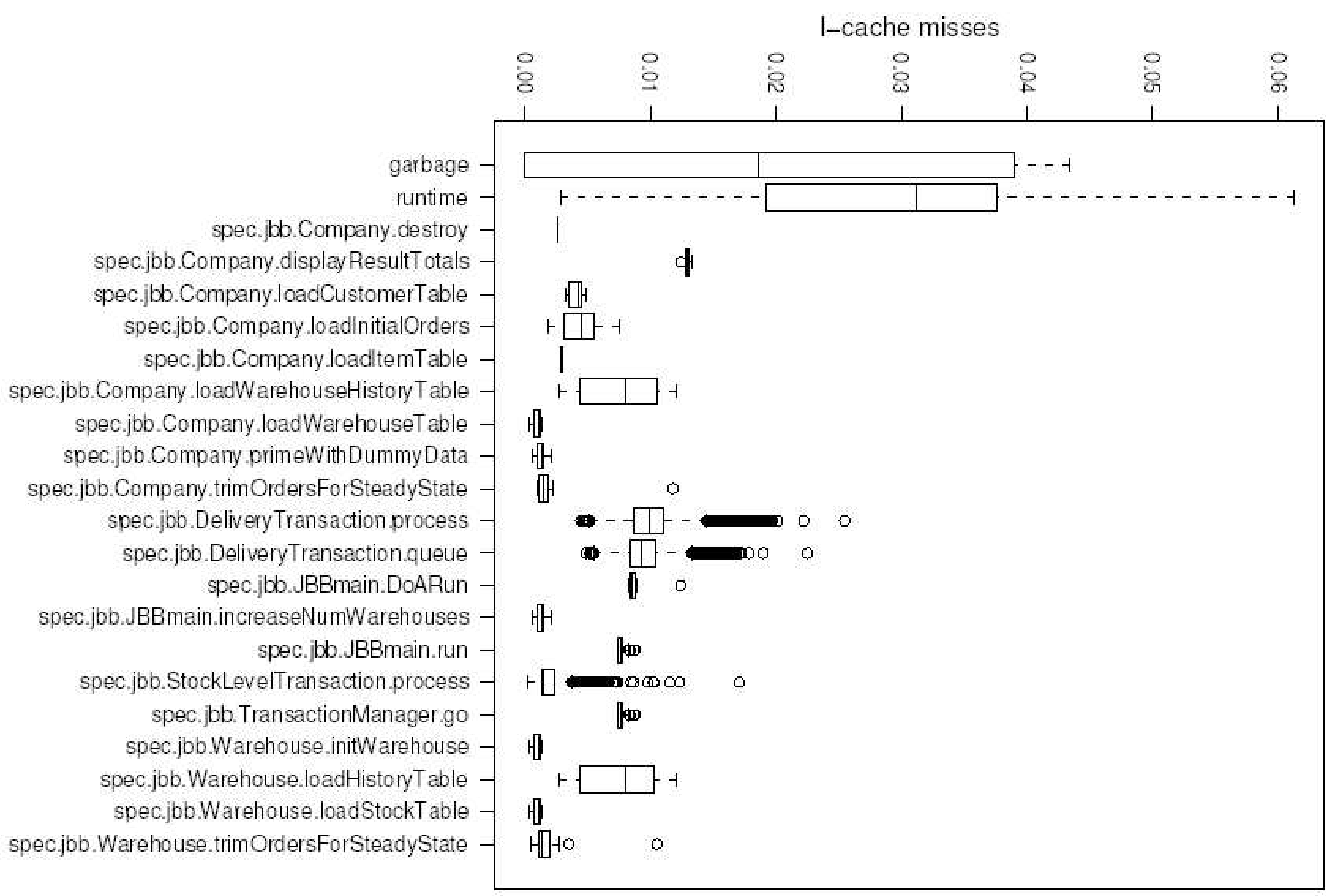


IPC
0.2
0.4
0.6
0.8
garbage collecor
runtime compiler
spec.jbb.Company.destroy
spec.jbb.Company.displayResultTotals
spec.jbb.Company.loadCustomerTable
spec.jbb.Company.loadInitialOrders
spec.jbb.Company.loadItemTable
spec.jbb.Company.loadWarehouseHistoryTable
spec.jbb.Company.loadWarehouseTable
spec.jbb.Company.primeWithDummyData
spec.jbb.Company.trimOrdersForSteadyState
spec.jbb.DeliveryTransaction.process
spec.jbb.DeliveryTransaction.queue
spec.jbb.JBBmain.DoARun
spec.jbb.JBBmain.increaseNumWarehouses
spec.jbb.JBBmain.run
spec.jbb.StockLevelTransaction.process
spec.jbb.TransactionManager.go
spec.jbb.Warehouse.initWarehouse
spec.jbb.Warehouse.loadHistoryTable
spec.jbb.Warehouse.loadStockTable
spec.jbb.Warehouse.trimOrdersForSteadyState

D-cache misses



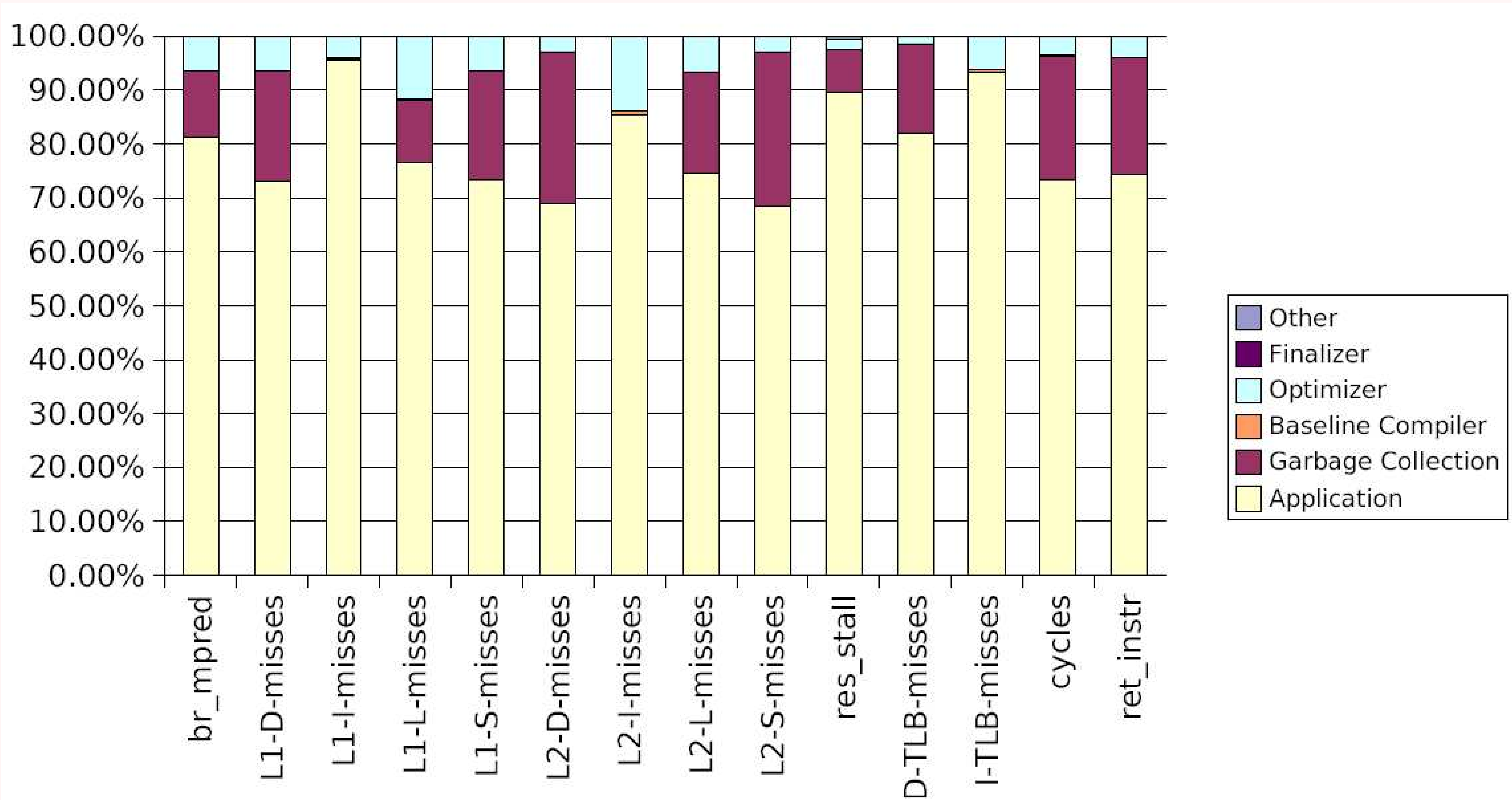
I-cache misses



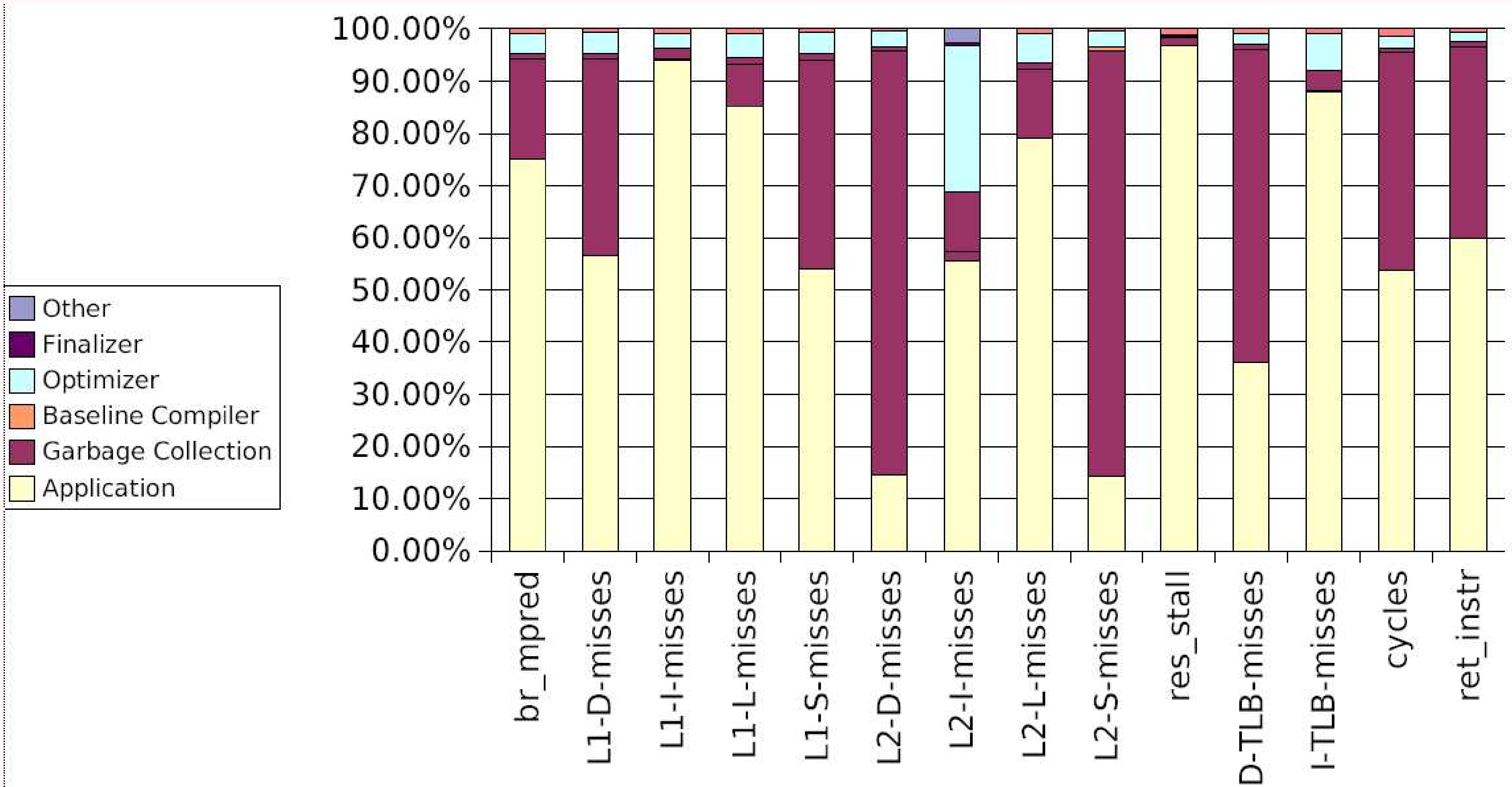
Analysis of method-level phase behaviour

- JVM vs app behaviour
- Application bottleneck analysis

JVM vs. app behaviour (PseudoJBB)



JVM vs. app behaviour (Jack)



Application bottleneck analysis

- 3 fundamental questions
  - What is the bottleneck?
    - List phases with highest CPI values
  - Why does it occur?
    - Investigate other counters for the same phase(s)
  - When does it occur?
    - Graph CPI over time
- Gives some insight, but still not always informative

Conclusions

- Method-level phase analysis works at an appropriate granularity level.
- Method-level phase behaviour analysis . . .
  - can reveal some low-level characteristics of Java workloads.
  - can be used to study the interaction between the JVM and the application.
  - can be used to bridge the gap between dynamic analysis results and source code.