

saarland-informatics-campus.de

Facile: Fast, Accurate, and Interpretable Basic-Block Throughput Prediction

Andreas Abel, Saarland University*

Shrey Sharma, Saarland University

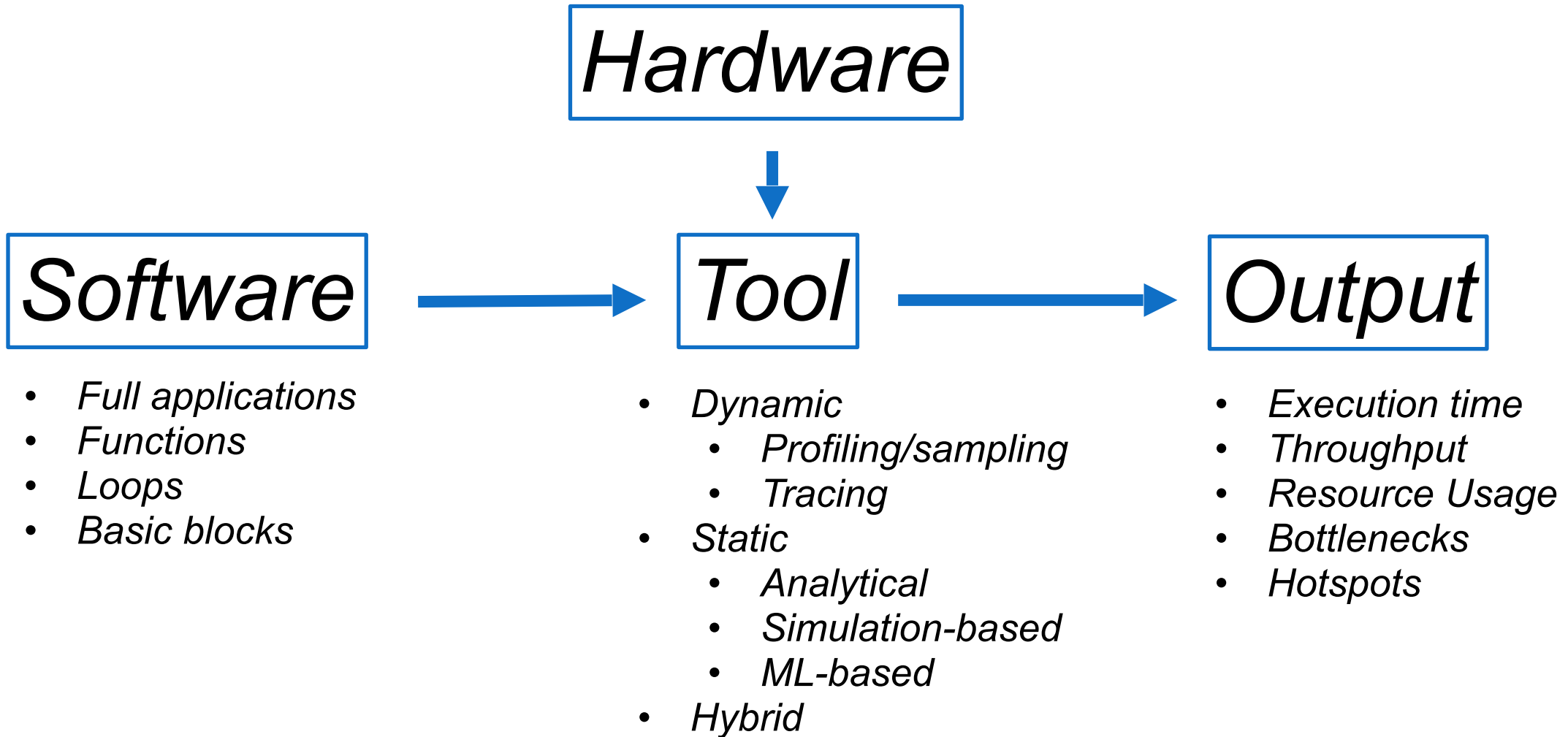
Jan Reineke, Saarland University

* now at Google

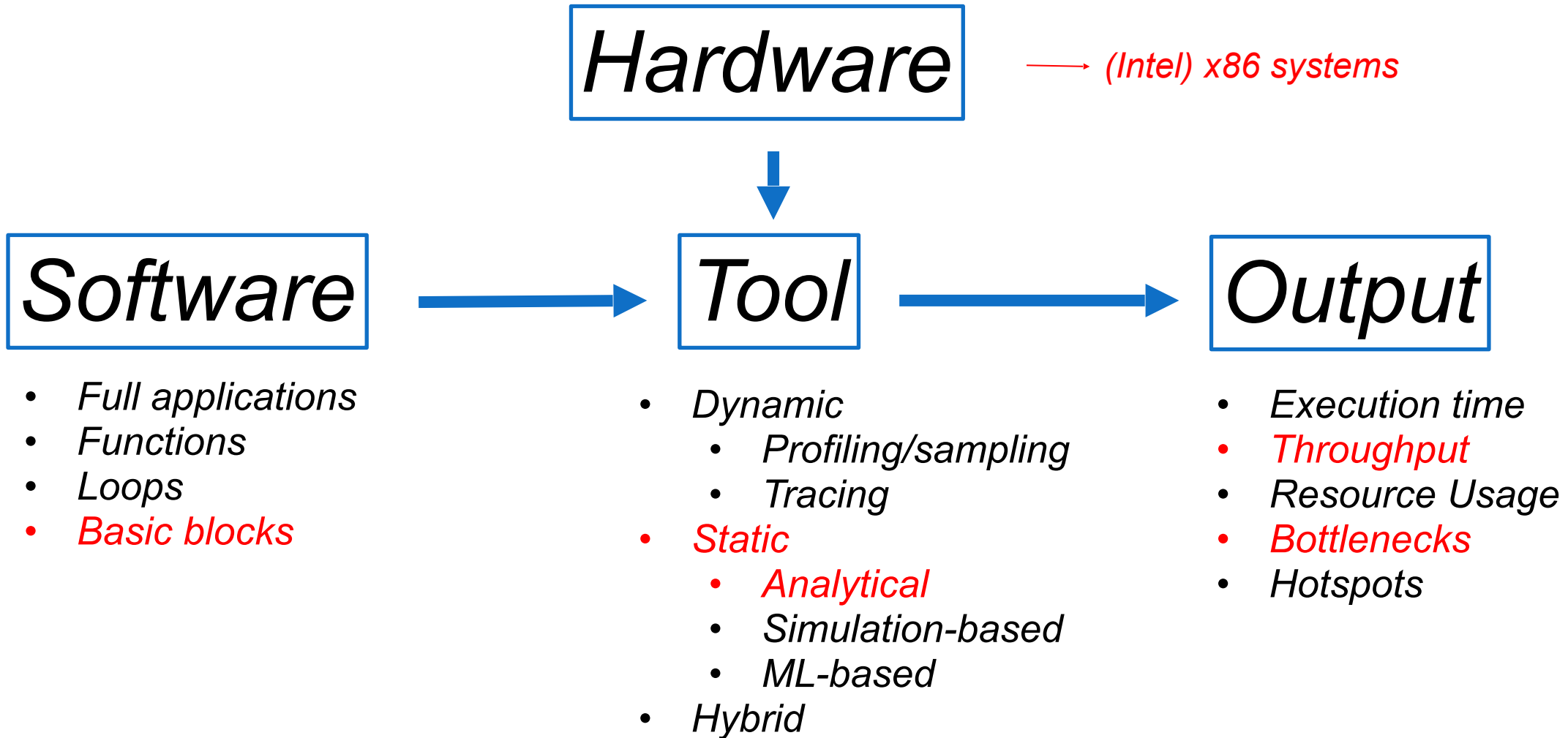


This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 101020415)"

Performance Prediction/Analysis



Performance Prediction/Analysis



x86 Basic-Block Throughput Prediction



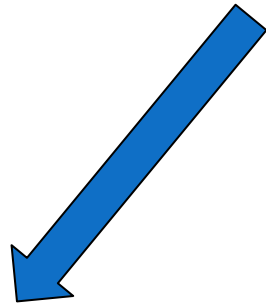
Intel® Architecture Code Analyzer

(IACA)

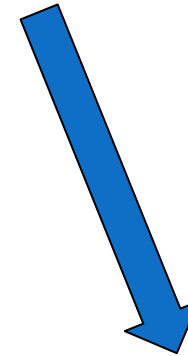
Instruction THroughput Estimator using
MAchine Learning (ITHEMAL)

uiCA - The uops.info Code Analyzer

Use cases for basic block predictors

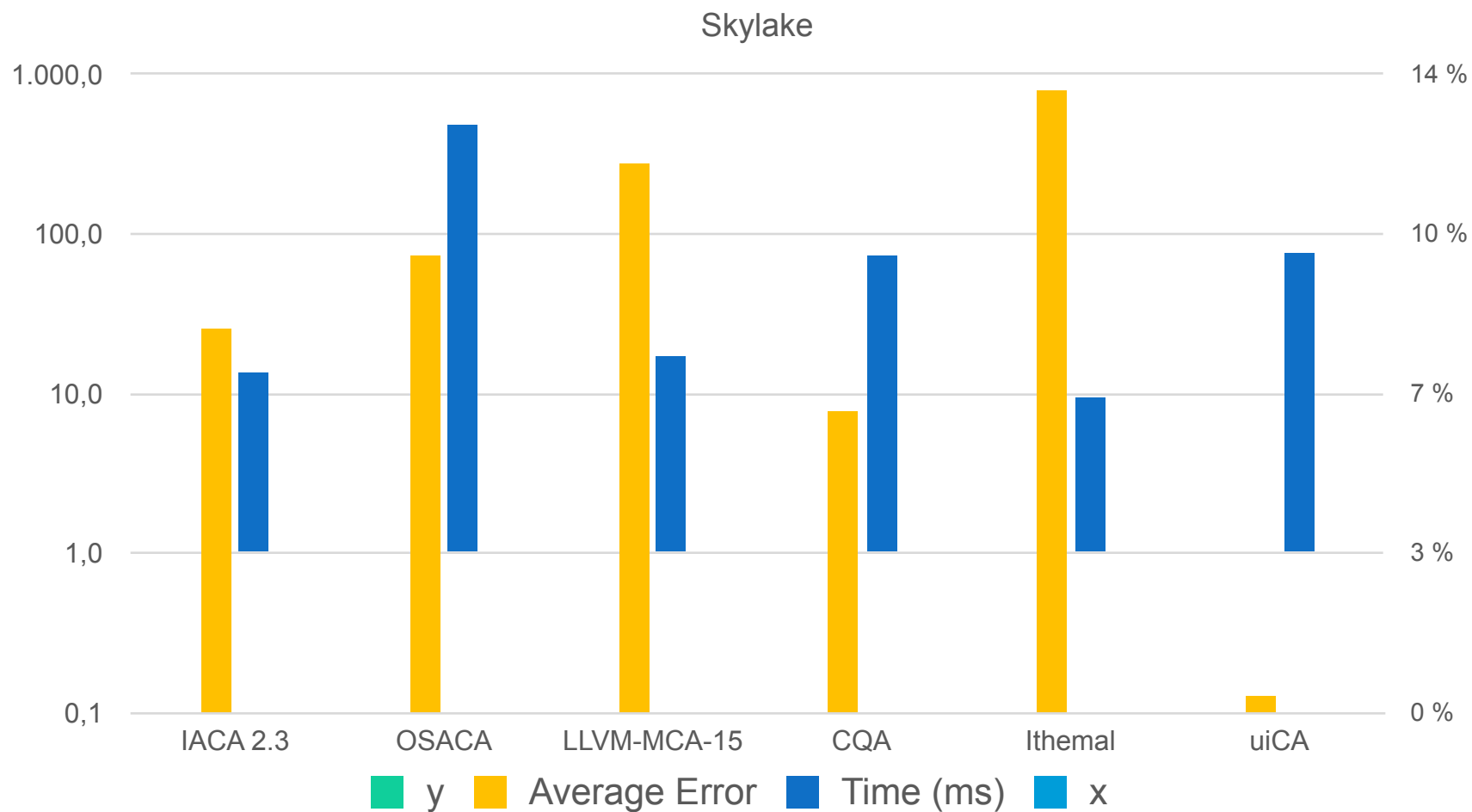


*Manual performance
analysis/optimization*

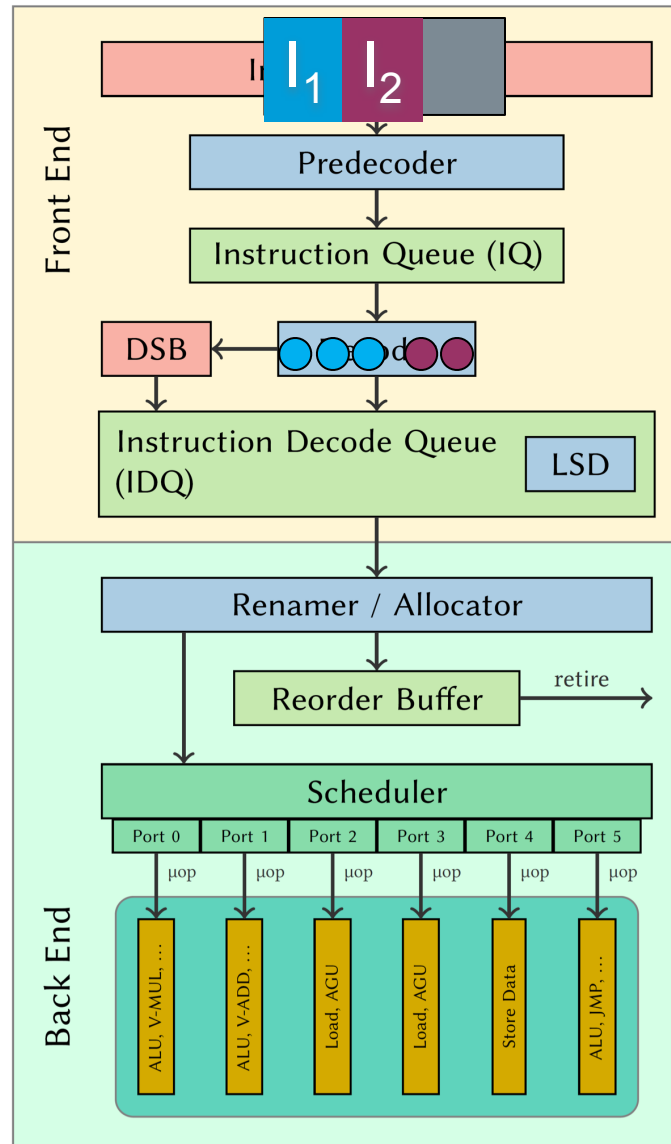


*As a cost model for
compilers/superoptimizers*

State of the art



Background: Pipeline of Intel Core CPUs



Analytical throughput predictor

*in cycles
per iteration* ↙ $TP = \max\{\text{FrontEnd}, \text{Issue}, \text{Ports}, \text{Precedence}\}$

Analytical throughput predictor: Issue

in cycles per iteration \swarrow $TP = \max\{\text{FrontEnd}, \text{Issue}, \text{Ports}, \text{Precedence}\}$

\searrow

$\text{Issue} = \frac{n}{i}$

\swarrow *number of uops of the benchmark*

\searrow *issue width*

The diagram illustrates the components of the analytical throughput predictor. At the top, the formula $TP = \max\{\text{FrontEnd}, \text{Issue}, \text{Ports}, \text{Precedence}\}$ is shown. A blue arrow points from the text 'in cycles per iteration' to the 'TP' term. A green oval highlights the 'Issue' term in the set. A green arrow points from this 'Issue' term down to the 'Issue' term in the formula $\text{Issue} = \frac{n}{i}$. From this formula, a blue arrow points from 'n' to the text 'number of uops of the benchmark', and a red arrow points from 'i' to the text 'issue width'.

Analytical throughput predictor: Front end

in cycles per iteration \swarrow

$$TP = \max\{\text{FrontEnd}, \text{Issue}, \text{Ports}, \text{Precedence}\}$$

\swarrow

$$\text{FrontEnd} = \begin{cases} \max\{\text{Predec}, \text{Dec}\} & \text{if benchmark is affected by the JCC erratum} \\ \text{LSD} & \text{else if LSD is enabled and } \#\mu\text{ops} \leq \text{IDQWidth} \\ \text{DSB} & \text{else} \end{cases}$$

\rightarrow

$$\text{LSD} = \frac{\left\lceil \frac{n \cdot u}{i} \right\rceil}{u}$$

number of uops \swarrow *unrolling factor* \swarrow *issue width* \swarrow

Analytical throughput predictor: Front end

in cycles per iteration \swarrow

$$TP = \max\{\text{FrontEnd}, \text{Issue}, \text{Ports}, \text{Precedence}\}$$

\swarrow

$$\text{FrontEnd} = \begin{cases} \max\{\text{Predec}, \text{Dec}\} & \text{if benchmark is affected by the JCC erratum} \\ \text{LSD} & \text{else if LSD is enabled and } \#uops \leq IDQWidth \\ \text{DSB} & \text{else} \end{cases}$$

\swarrow

number of uops \swarrow *length (in bytes)*

$$\text{DSB} = \begin{cases} \lceil \frac{n}{w} \rceil & l < 32, \\ \frac{n}{w} & l \geq 32. \end{cases}$$

\nwarrow *DSB width*

Analytical throughput predictor: Front end

in cycles per iteration \swarrow

$$TP = \max\{\text{FrontEnd}, \text{Issue}, \text{Ports}, \text{Precedence}\}$$

\swarrow

$$\text{FrontEnd} = \begin{cases} \max\{\text{Predec}, \text{Dec}\} & \text{if benchmark is affected by the JCC erratum} \\ \text{LSD} & \text{else if LSD is enabled and } \#\mu\text{ops} \leq \text{IDQWidth} \\ \text{DSB} & \text{else} \end{cases}$$

Analytical throughput predictor: Ports


*in cycles
per iteration* \swarrow $TP = \max\{\text{FrontEnd}, \text{Issue}, \text{Ports}, \text{Precedence}\}$

Examples: 0|1|5 $\rightarrow 1/3$



0|1 1|5 $\rightarrow 2/3$

- In general: can be solved using a linear program
- We developed a simpler, more efficient heuristic approach

Analytical throughput predictor: Precedence

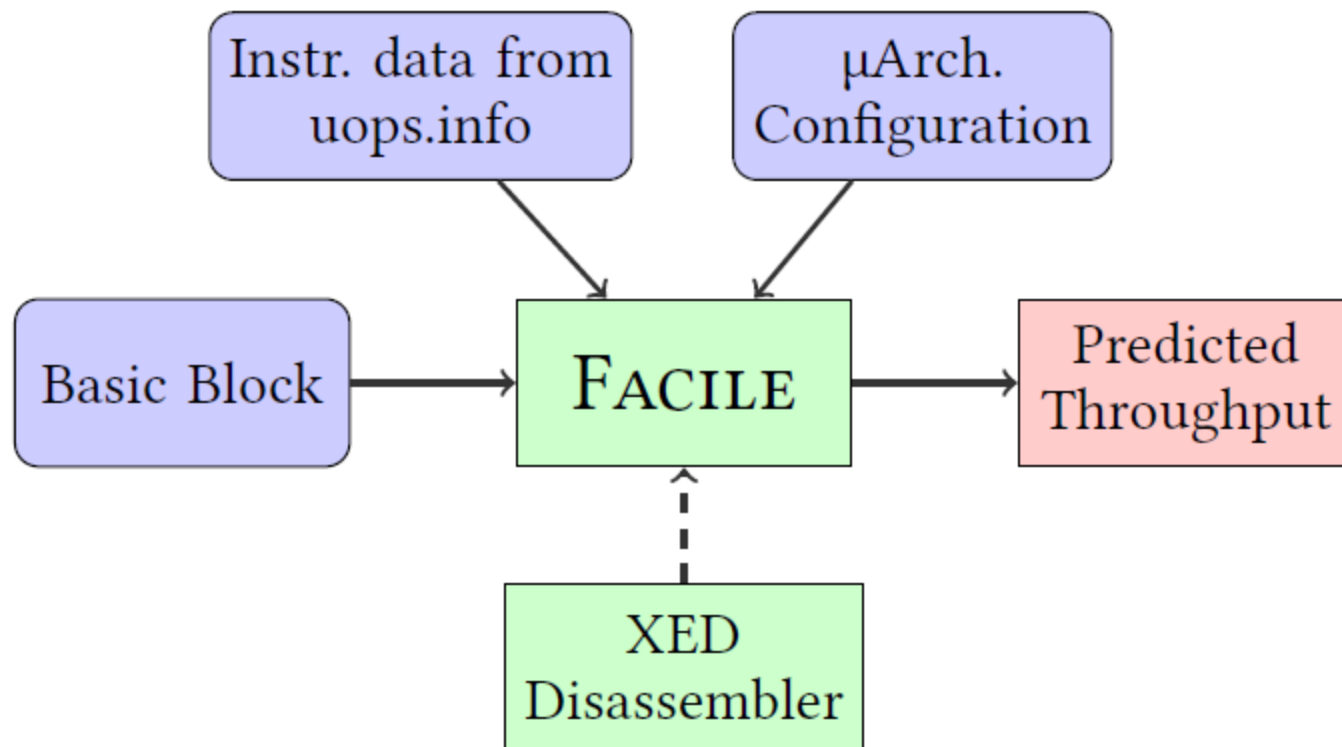
*in cycles
per iteration*  $TP = \max\{\text{FrontEnd}, \text{Issue}, \text{Ports}, \text{Precedence}\}$

loop:

dec rax

dec rax 

jnz loop

Implementation



uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures

Andreas Abel and Jan Reineke
[abel,reineke]@cs.uni-saarland.de
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany

Abstract

Modern microarchitectures are some of the world's most complex man-made systems. As a consequence, it is increasingly difficult to predict, explain, let alone optimize the performance of software running on such microarchitectures. As a basis for performance predictions and optimizations, we would need faithful models of their behavior, which are, unfortunately, seldom available.

In this paper, we present the design and implementation of a tool to construct faithful models of the latency, throughput, and port usage of x86 instructions. To this end, we first discuss common notions of instruction throughput and port usage, and introduce a more precise definition of latency that, in contrast to previous definitions, considers dependencies between different pairs of input and output operands. We then develop novel algorithms to infer the latency, throughput, and port usage based on automatically-generated microbenchmarks that are more accurate and precise than existing work.

To facilitate the rapid construction of optimizing compilers and tools for performance prediction, the output of our tool is provided in a machine-readable format. We provide experimental results for processors of all generations of Intel's Core architecture, i.e., from Nehalem to Coffee Lake, and discuss various cases where the output of our tool differs considerably from prior work.

ACM Reference Format:

Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, April 13–17, 2019, Providence, RI, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. ASPLOS '19, April 13–17, 2019, Providence, RI, USA. © 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-6240-5/19/04...\$15.00 <https://doi.org/10.1145/3297858.3304062>

Providence, RI, USA, ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3297858.3304062>

1 Introduction

Developing tools that predict, explain, or even optimize the performance of software is challenging due to the complexity of today's microarchitectures. Unfortunately, this challenge is exacerbated by the lack of a precise documentation of their behavior. While the high-level structure of modern microarchitectures is well-known and stable across multiple generations, lower-level aspects may differ considerably between microarchitecture generations and are generally not as well documented. An important aspect with a relatively strong influence on performance is how ISA instructions decompose into micro-operations (uops); which ports these uops may be executed on; and what their latencies are.

Knowledge of this aspect is required, for instance, to build precise performance-analysis tools like CQA [8], Kerncraft [18], or lvm-mca [6]. It is also useful when configuring cycle-accurate simulators like Zesto [27], gem5 [7], McSim+ [3] or ZSim [31]. Optimizing compilers, such as LLVM [26] and GCC [15], can profit from detailed instruction characterizations to generate efficient code for a specific microarchitecture. Similarly, such knowledge is helpful when manually fine-tuning a piece of code for a specific processor.

Unfortunately, information about the port usage, latency, and throughput of individual instructions at the required level of detail is hard to come by. Intel's processor manuals [23] only contain latency and throughput data for a number of "commonly-used instructions." They do not contain information on the decomposition of individual instructions into uops, nor do they state the execution ports that these uops can use.

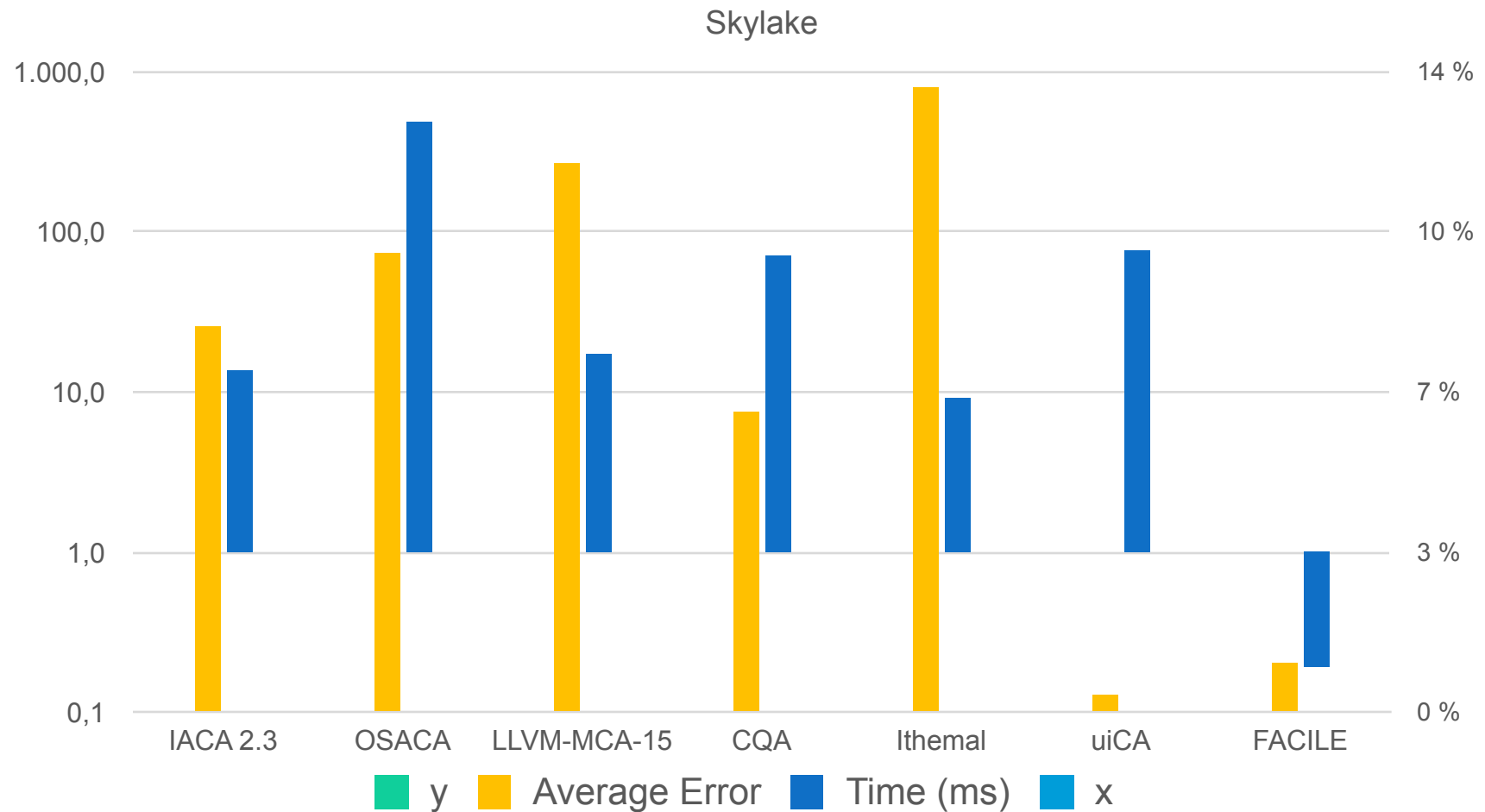
The only way to obtain accurate instruction characterizations for many recent microarchitectures is thus to perform measurements using microbenchmarks. Such measurements are aided by the availability of performance counters that provide precise information on the number of elapsed cycles and the cumulative port usage of instruction sequences. A relatively large body of work [1, 2, 4, 9, 10, 19, 28–30, 32, 33, 35, 36] uses microbenchmarks to infer properties of the memory hierarchy. Another line of work [5, 13, 14, 25] uses automatically generated microbenchmarks to characterize the energy

Automatic generation of microbenchmarks to measure the latency, throughput and execution port usage of individual instructions

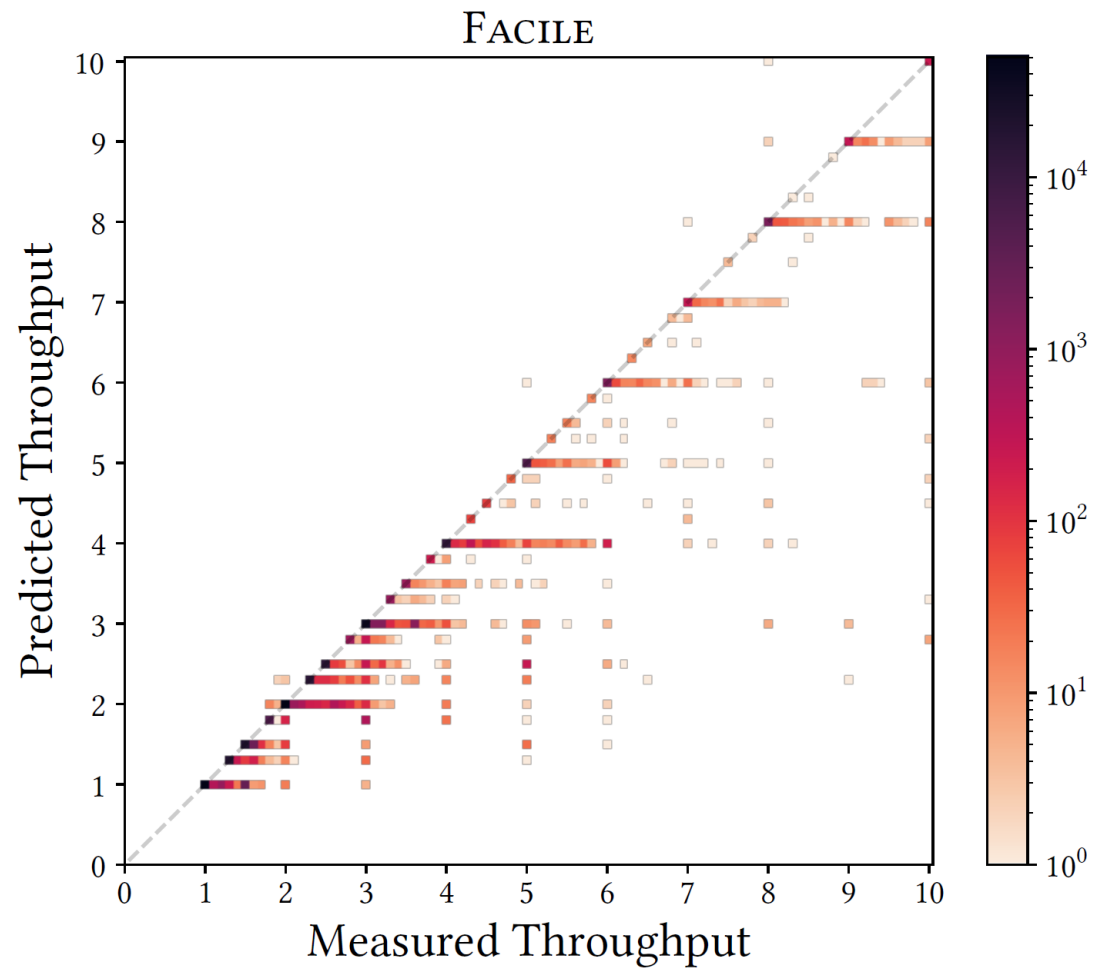
ASPLOS 2019

uops.info

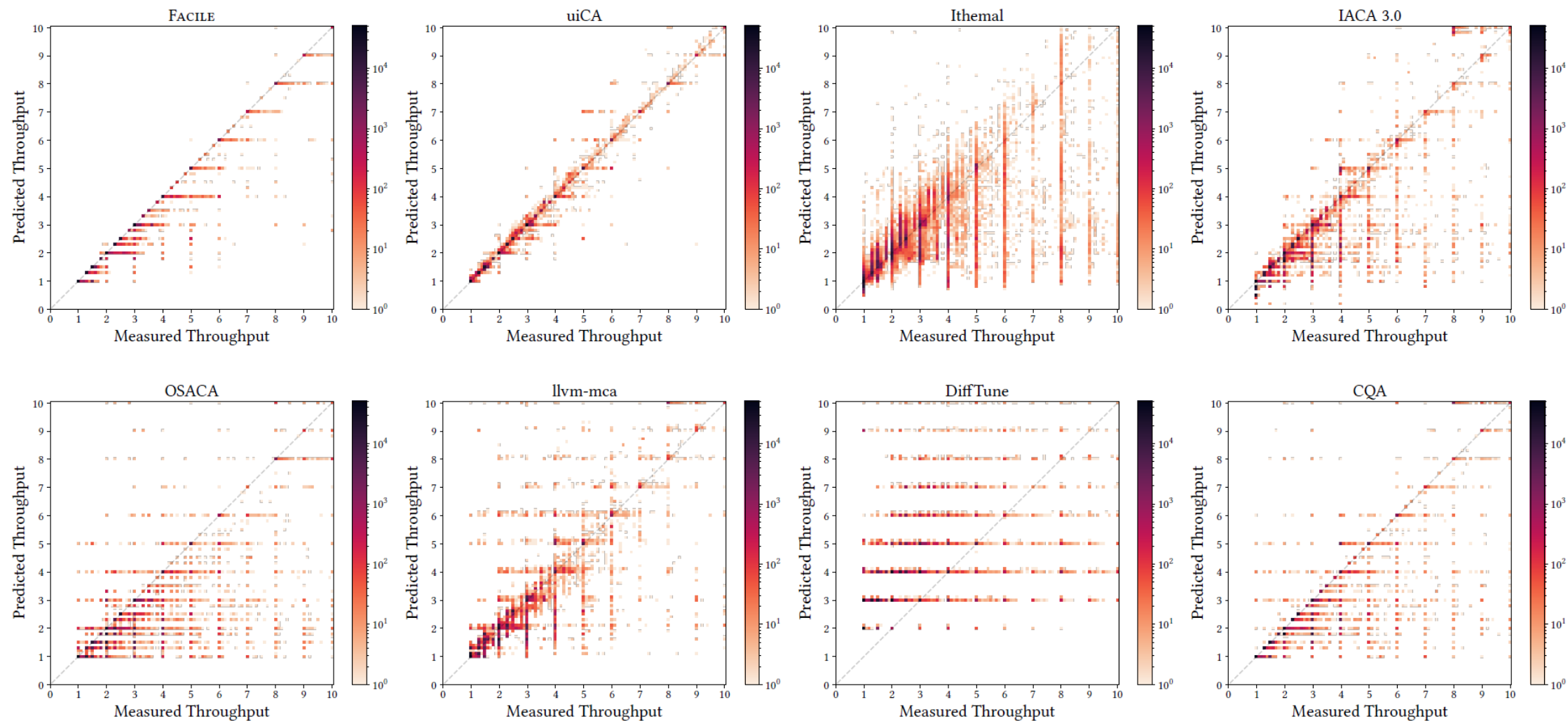
Evaluation



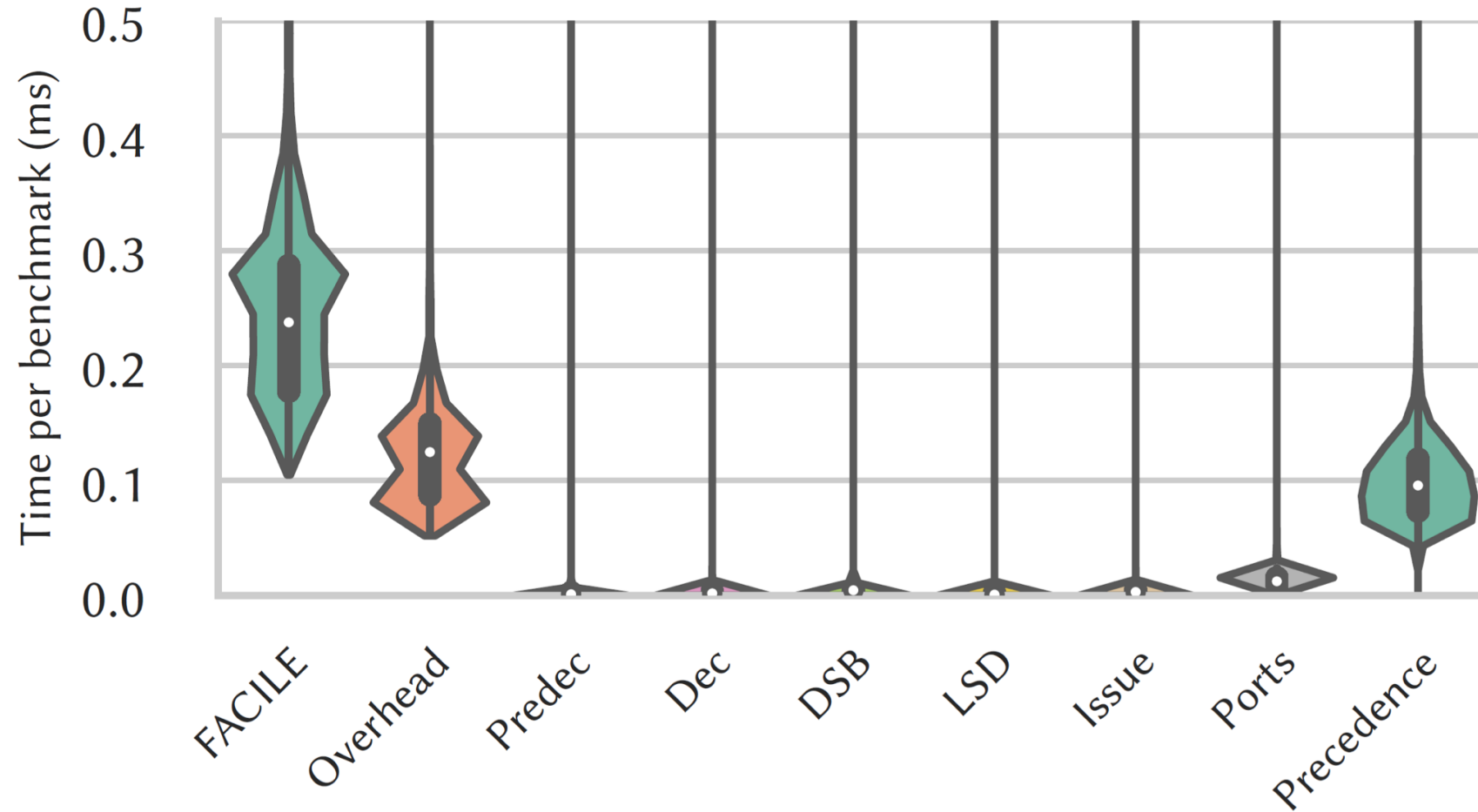
Accuracy



Accuracy



Execution time



Summary & Future Work

- Accurate and efficient open-source analytical basic-block TP predictor
- Provides insights into what the bottlenecks are
- Integrate into compilers/superoptimizers
- Combine with branch prediction or memory hierarchy models
- Combine static and dynamic analyses

github.com/andreas-abel/uiCA/blob/master/facile.py