Profiling and Optimizing Java Streams

Eduardo Rosales, Matteo Basso, Andrea Rosà, Walter Binder

Università della Svizzera italiana

<Programming> 2023
March 16, 2023

INTRODUCTION

Stream API

- ***Stream API*** (package *java.util.stream*)
  - Data processing
    - MapReduce-style transformations

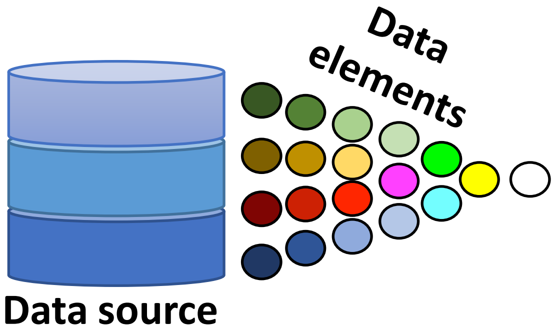
  - Two key abstractions:
    - ***Stream***
    - ***Stream pipeline***

Stream

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

Stream

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

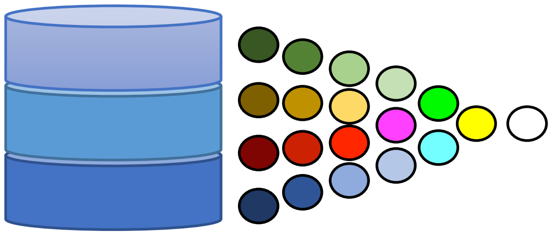


- A ***stream*** represents a sequence of data elements coming from a ***data source***

Pipeline

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

Data source

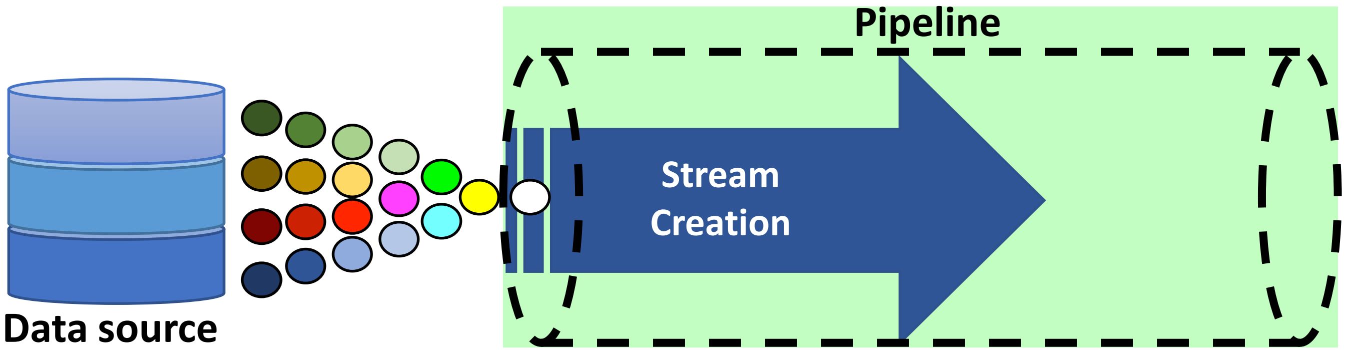


Data source

Pipeline

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

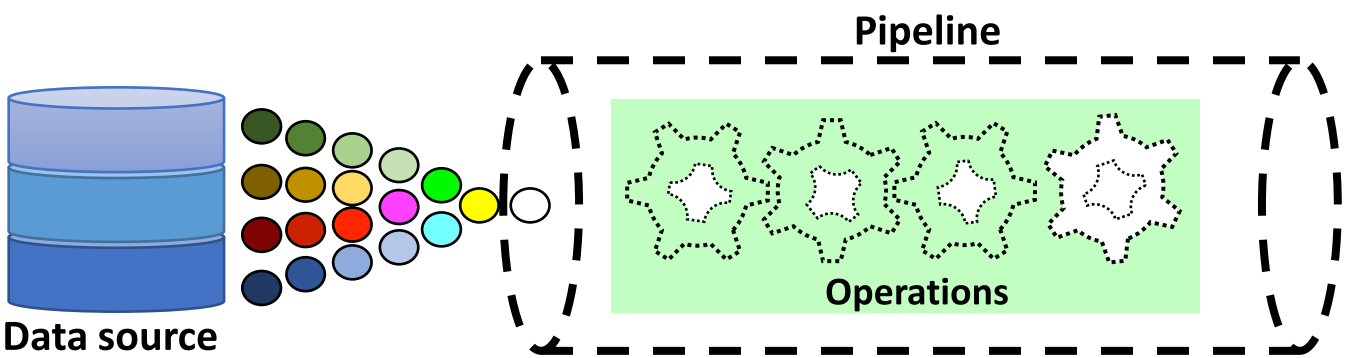
Stream creation



- *A* ***pipeline*** is associated with the stream

Operations

```
transactionList.stream()
	.parallel()
	.filter(t -> t.getStatus() == Transaction.VALID)
	.map(Transaction::getID)
	.collect(Collectors.toSet());
```

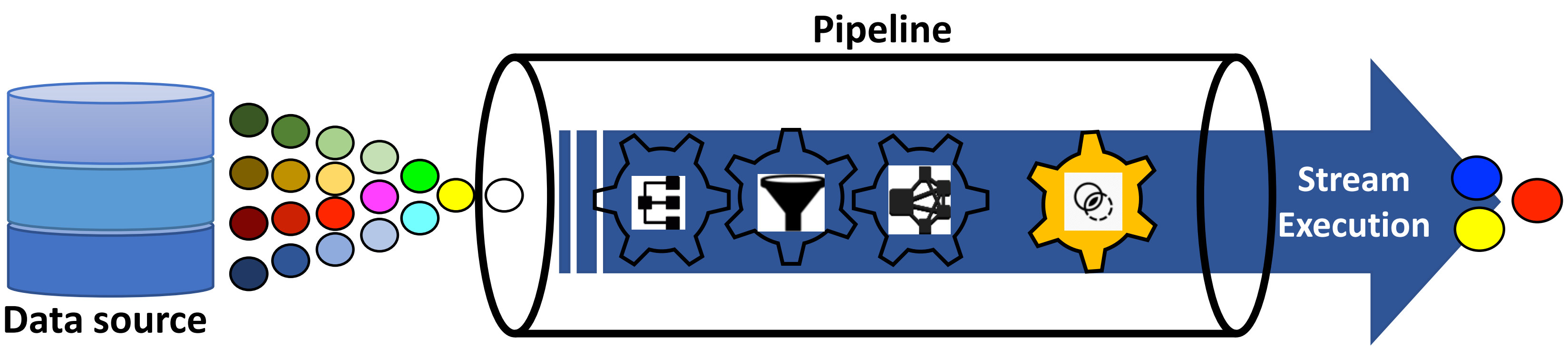


- The pipeline can contain **operations**

Execution Mode

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

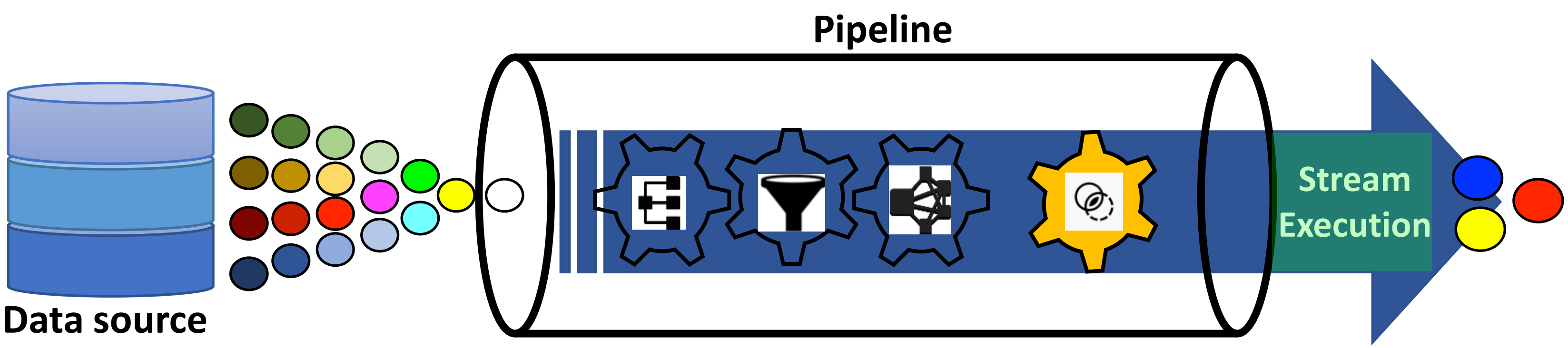
Execution Mode	
sequential	parallel



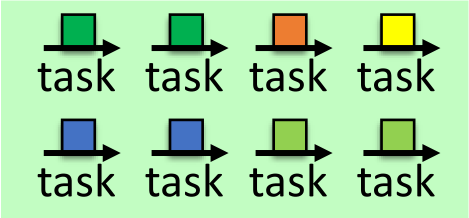
Tasks

```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

Execution Mode	
sequential	parallel



- Parallel streams may spawn ***tasks***
- Tasks are executed in a ***fork/join pool*** by threads called ***workers***



Optimization of Streams

- Stream code suffers from important performance penalties when compared to imperative code [1, 2, 3]

- These penalties are mainly caused by:
  - **Abstraction overheads**
    - Due to extra object allocations and reclamations
  - **Abundant virtual method calls**
    - Can prevent *Just-In-Time* (JIT) compiler optimizations

[1] Biboudis et al. *Clash of the Lambdas*. ICOOOLPS'14.
[2] Kiselyov et al. *Stream Fusion, to Completeness*. ACM SIGPLAN Notices, 2017.
[3] Møller et al. *Eliminating Abstraction Overhead of Java Stream Pipelines Using Ahead-of-Time Program Optimization*. OOPSLA'20.

Optimization of Streams

To mitigate stream-related overheads and optimize streams ***developers need means*** to study the **runtime behavior** of streams

Optimization of Streams

Current approaches to stream optimization:

- Mainly rely on static analysis techniques
  - Overlook **runtime information** key to spot stream-related performance issues
- Suffer from important limitations to detect all streams used by a Java application

Optimization of Streams – Research Gap

- **Research gap:**

There is a lack of dedicated tools able to dynamically analyze stream processing on the JVM to help developers locate streams that impair good performance

Contributions

- Propose a technique enabling cycle-accurate stream profiling
  - Accurately measure the computations performed by a stream in terms of **reference cycles** (*cycles* for short)
- Analyze stream processing in *Renaissance* [4]
- Optimize stream-related performance issues in *Renaissance*
- Conduct an evaluation on accuracy and overhead

[4] Prokopec et al. *Renaissance: Benchmarking Suite for Parallel Applications on the JVM*. PLDI'19.

PROFILING AND OPTIMIZING JAVA STREAMS

Cycle-accurate Stream Profiling

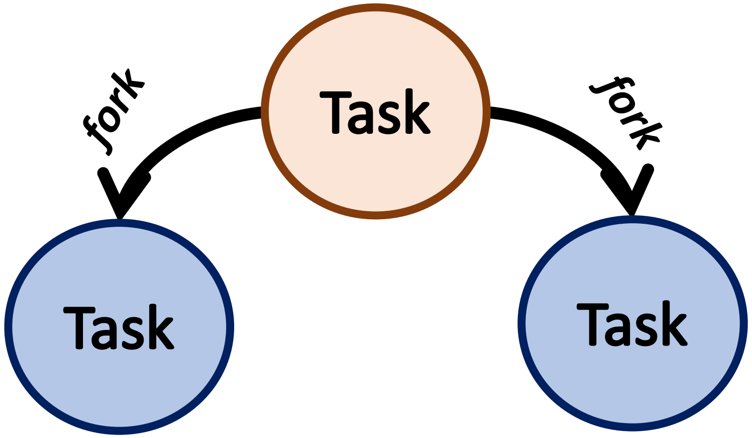
- Our technique to profile stream executions is implemented in ***StreamProf***, the first dedicated stream profiler for the JVM

- Features:
  - Detects every form of stream execution
  - Shows the impact of stream processing on application performance
  - Generates accurate profiles that help developers detect problematic streams

Profiling Model

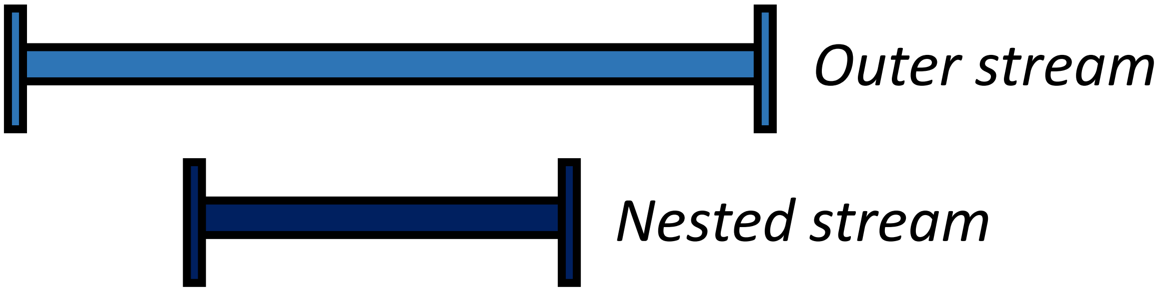
StreamProf targets **stream executions**:

- **Sequential stream executions:**
  - Carried out by the current thread
- **Parallel stream executions:**
  - Typically involve the execution of **tasks**
    - Tasks can be executed by multiple **workers**



Profiling Model

- ***Nested stream***: A stream whose execution is triggered by (and occurs during the execution of) another ***outer stream***



  - Multiple nesting levels are possible

Profiling Model

- ***Location***: Fully qualified name of the caller of the method executing the stream

ExampleClass.exampleMethod

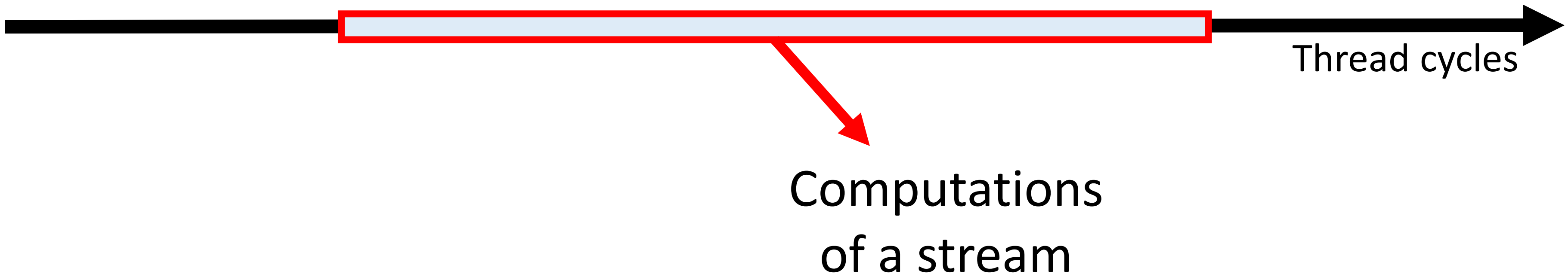
```
transactionList.stream()
        .parallel()
        .filter(t -> t.getStatus() == Transaction.VALID)
        .map(Transaction::getID)
        .collect(Collectors.toSet());
```

Profiling Model

We model stream execution around the concept of *span*

***Span*: The interval in which a stream is executed by a thread**

Profiling Model



Profiling Model

Profiling Model

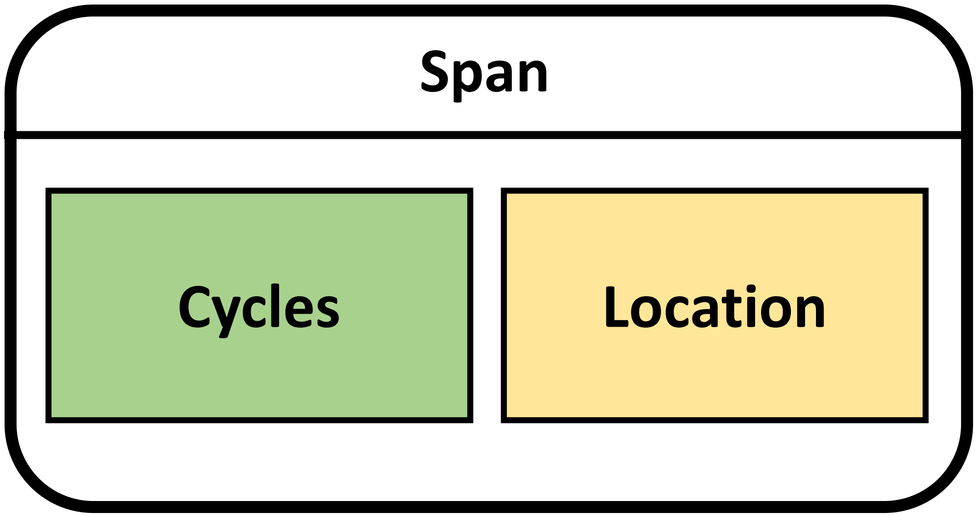
Profiling Model

Profiling Model

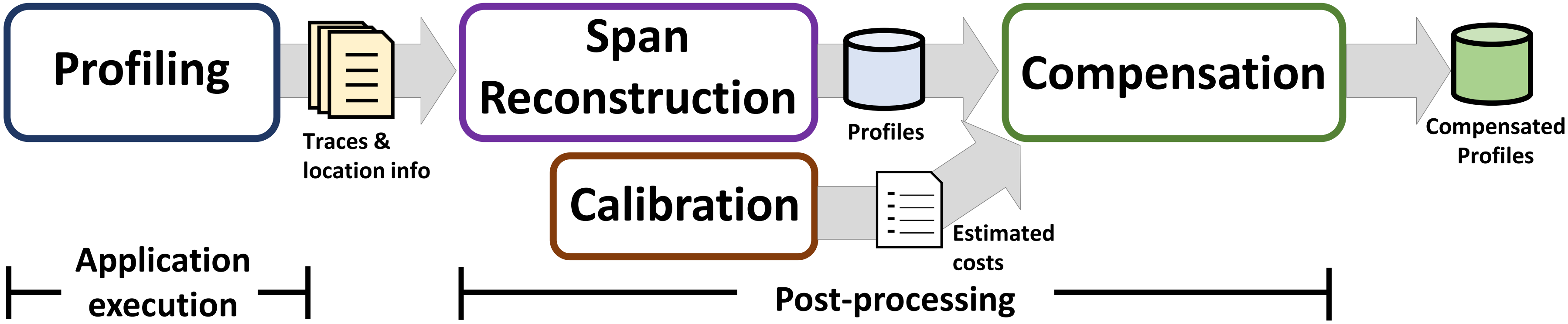
measured cycles **=** $\mathbf{cycles_{end} - cycles_{begin}}$

Profiling Model

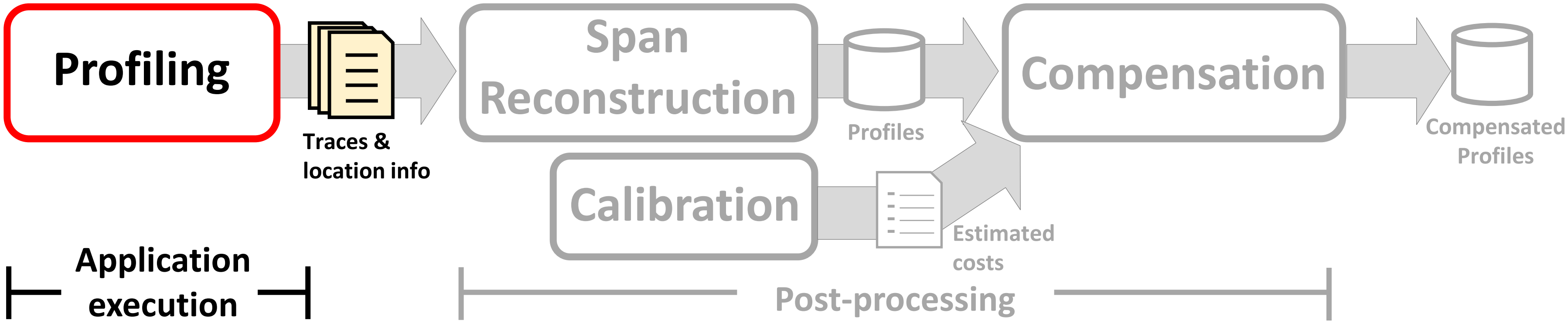
- For each span we compute the corresponding cycles and location



Profiling Methodology



Profiling Methodology

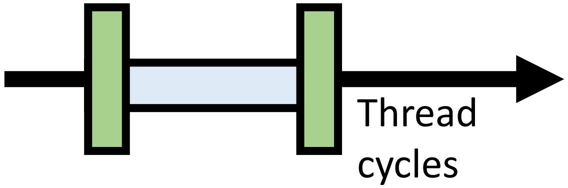


- **Profiling:** Profile every span and associate it with the respective measured cycles and location

Profiling Methodology

Difference when profiling sequential and parallel streams:

- **Sequential stream execution:**
  - Carried out by a single thread
  - Execution represented by a single span

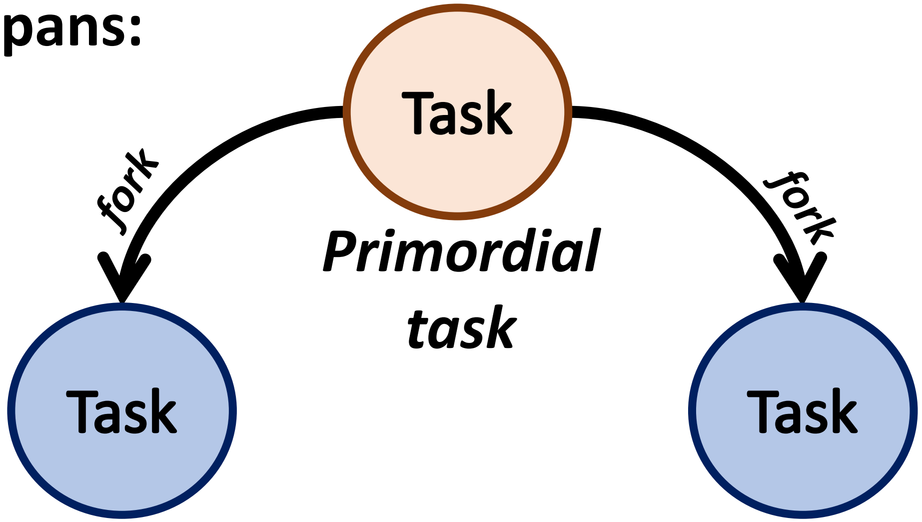


  - The measured cycles in the span can be directly attributed to the sequential stream execution
    - **No stream ID required** to identify a sequential stream
  - ***Anonymous span***:
    - Represents a sequential stream execution

- **Parallel stream execution:**
  - Typically carried out by multiple tasks → multiple spans
    - To aggregate the measured cycles of all spans
      - A **stream ID required** to identify the parallel stream execution
  - ***Named spans***: All spans associated with the same parallel stream execution

Profiling Methodology

- **Two named spans:**



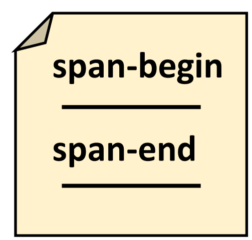
- ***Primordial span:*** Represents the primordial task
  - Associated with a new unique stream ID
- ***Support span:*** Any named span other than the primordial span
  - Retrieves the already generated stream ID

Profiling Methodology

The instrumentation relies on a ***tracer***:

- Reads cycle counters and stores the values in the form of 2 events:

- ***span-begin*** → **cycles**$_{\mathbf{begin}}$
- ***span-end*** → **cycles**$_{\mathbf{end}}$



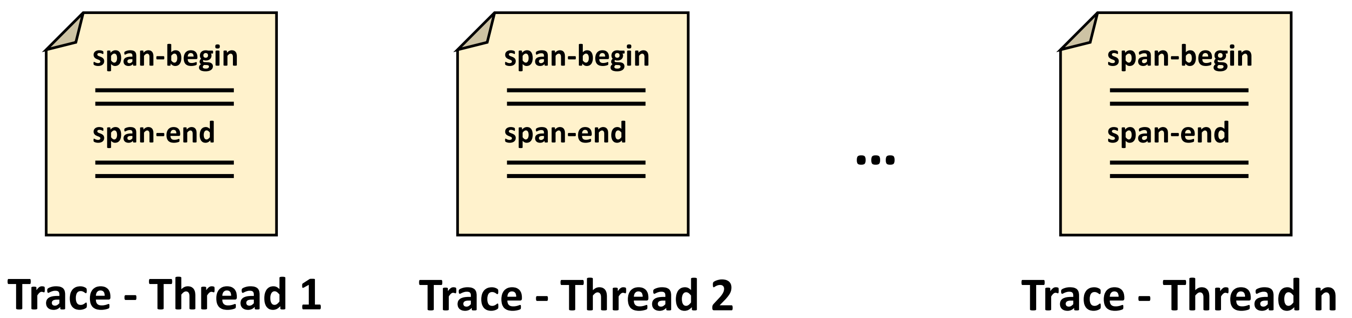
Trace

Profiling Methodology

- The tracer maintains thread-local buffers
  - Stores span information associated to a thread in memory
    - No synchronization (only when allocating new buffers)

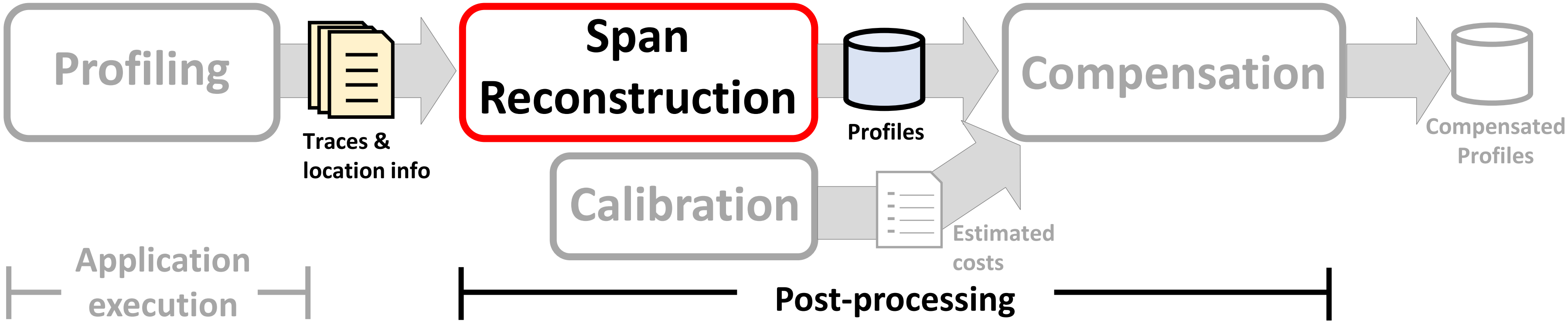
Profiling Methodology

- The tracer dumps data upon application completion:
  - Dumps one trace per thread participating in stream processing

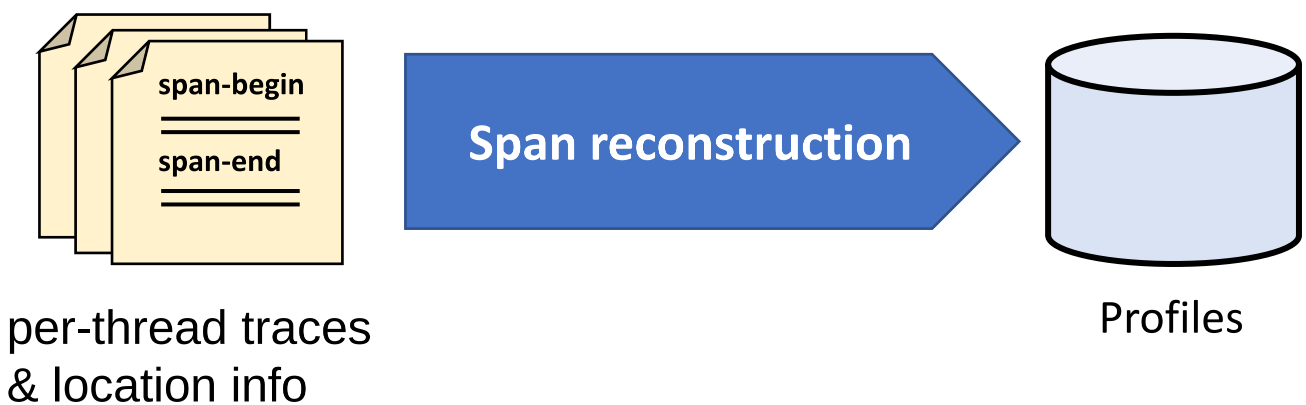


  - Traces are **post-processed offline**

Profiling Methodology



- **Span reconstruction:** each stream profile is reconstructed from the *span-begin* and *span-end* events stored in the dumped traces



Measured cycles computation:

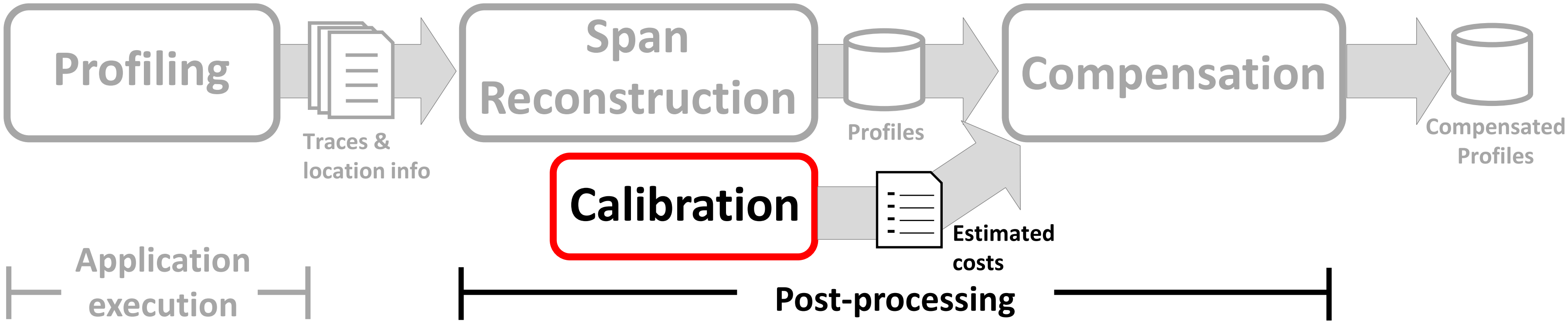
- **Sequential streams:**
  - Computed from the single anonymous span

Anonymous span

- **Parallel streams:**
  - Computed from the respective multiple named spans

Primordial span + **Support span** + **Support span**

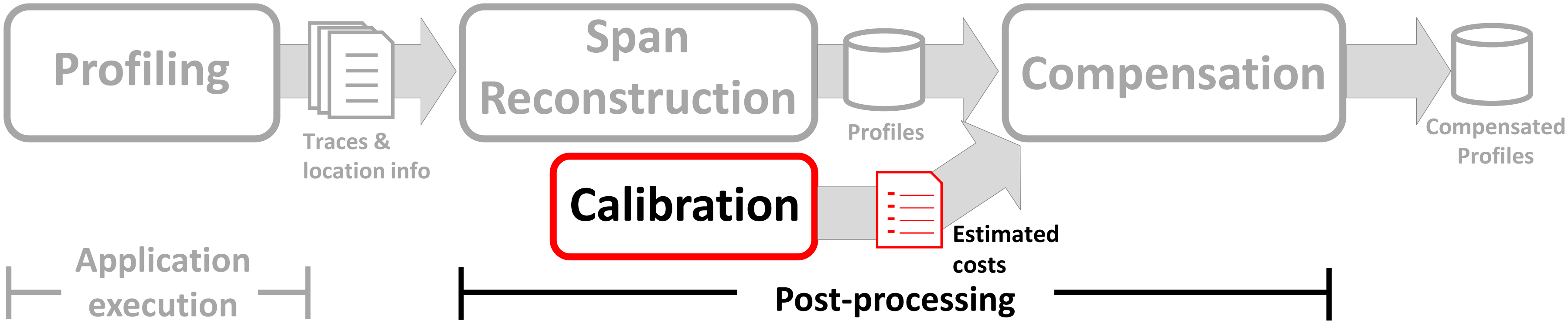
Profiling Methodology



- The inserted instrumentation code introduces extra cycles that are included in the measured cycles of each span
  - Decreasing profile accuracy

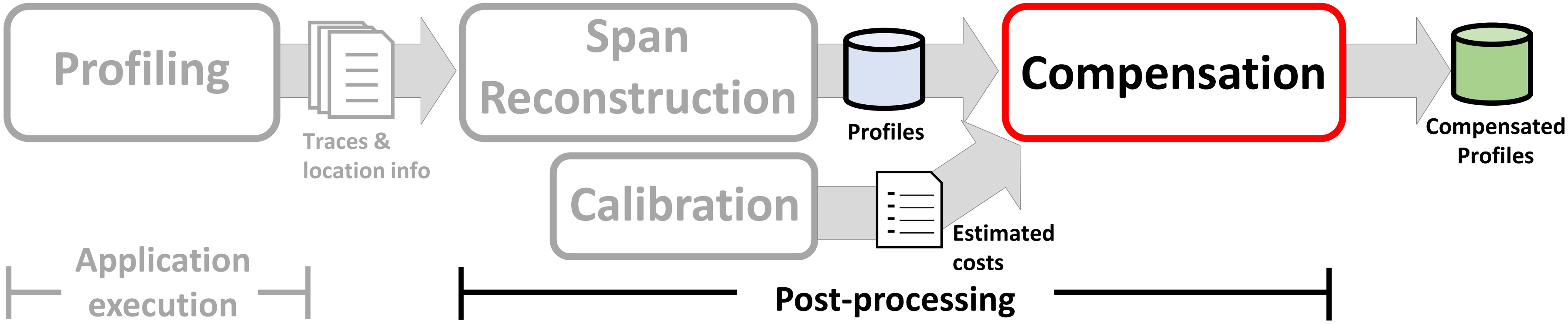
- To remove these extra cycles they need to be estimated, which is the aim of the **calibration** phase

Profiling Methodology



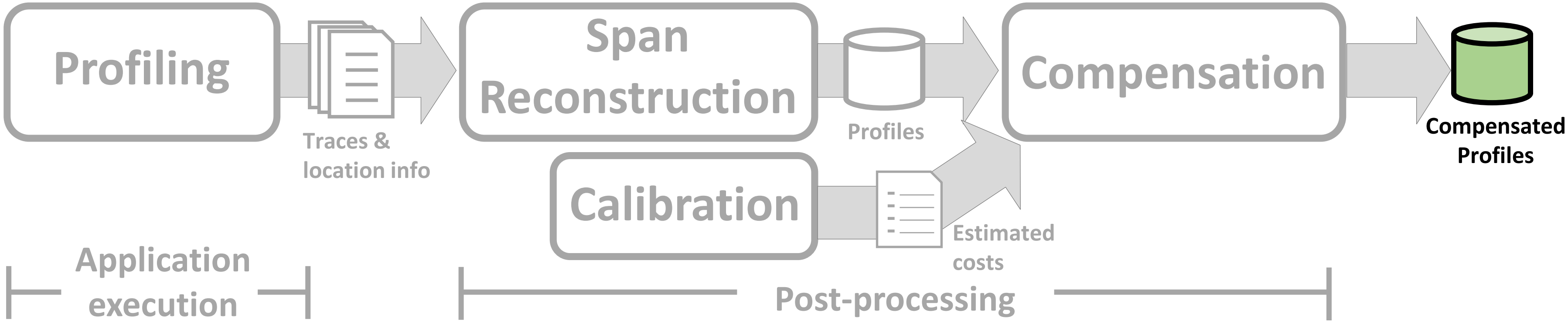
- **Calibration:** Produces ***estimated costs***, i.e., estimations of the extra cycles required to profile spans
  - The estimated costs are approximated by constants

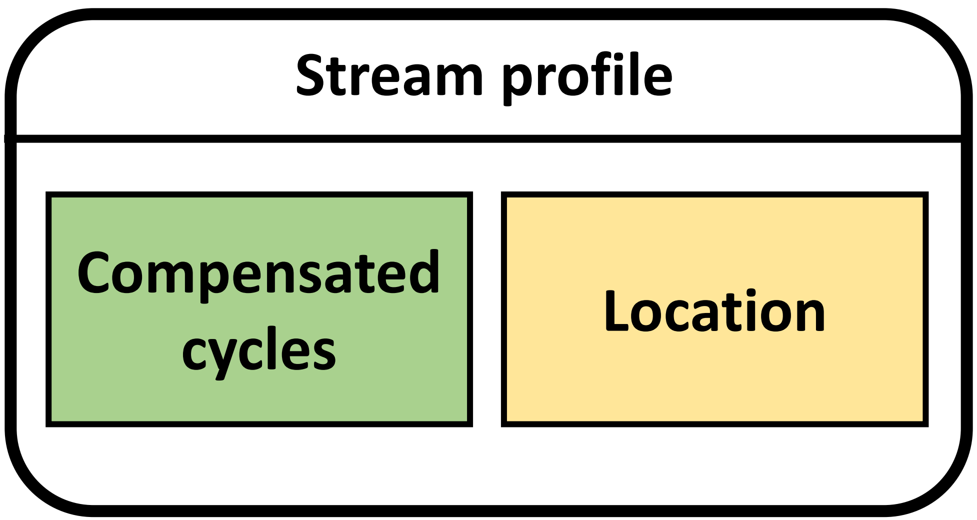
Profiling Methodology



- **Compensation:** Produce ***compensated cycles,*** i.e., the measured cycles for a span after the removal of both:
  - Estimated costs (the output of the calibration)
  - ***Nested cycles***, i.e., cycles elapsed due to nested spans

Profiling Methodology





Optimizing Streams - Renaissance

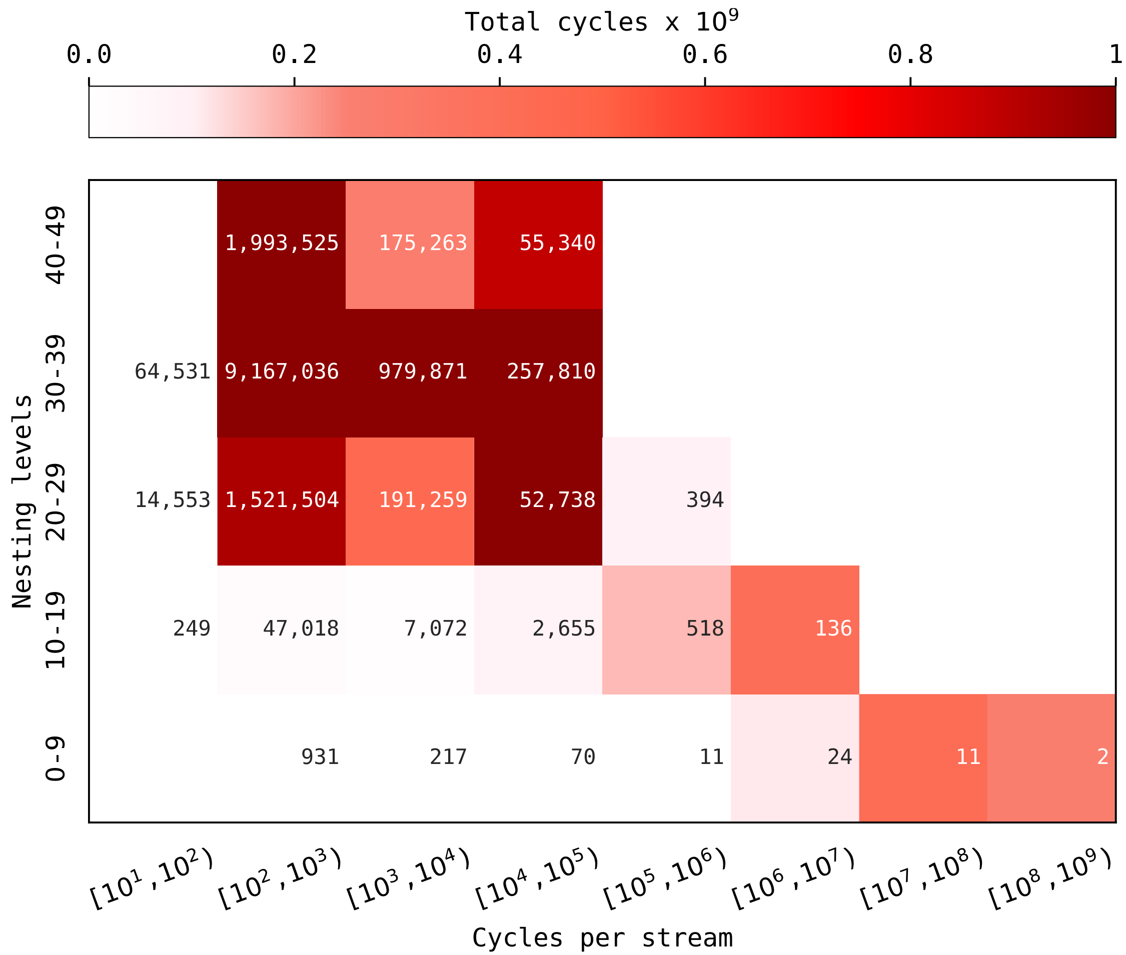
Using **StreamProf** to optimize stream processing:

- Target applications:
  - Stream-based workloads from *Renaissance* [8]
    - *mnemonics*
    - *par-mnemonics*
- Analysis targets **steady state**
- Testbed:
  - M_1: **8 core**, 128 GB RAM
  - M_2: **18 cores**, 256 GB RAM

[8] Prokopec et al. *Renaissance: Benchmarking Suite for Parallel Applications on the JVM*. PLDI'19.

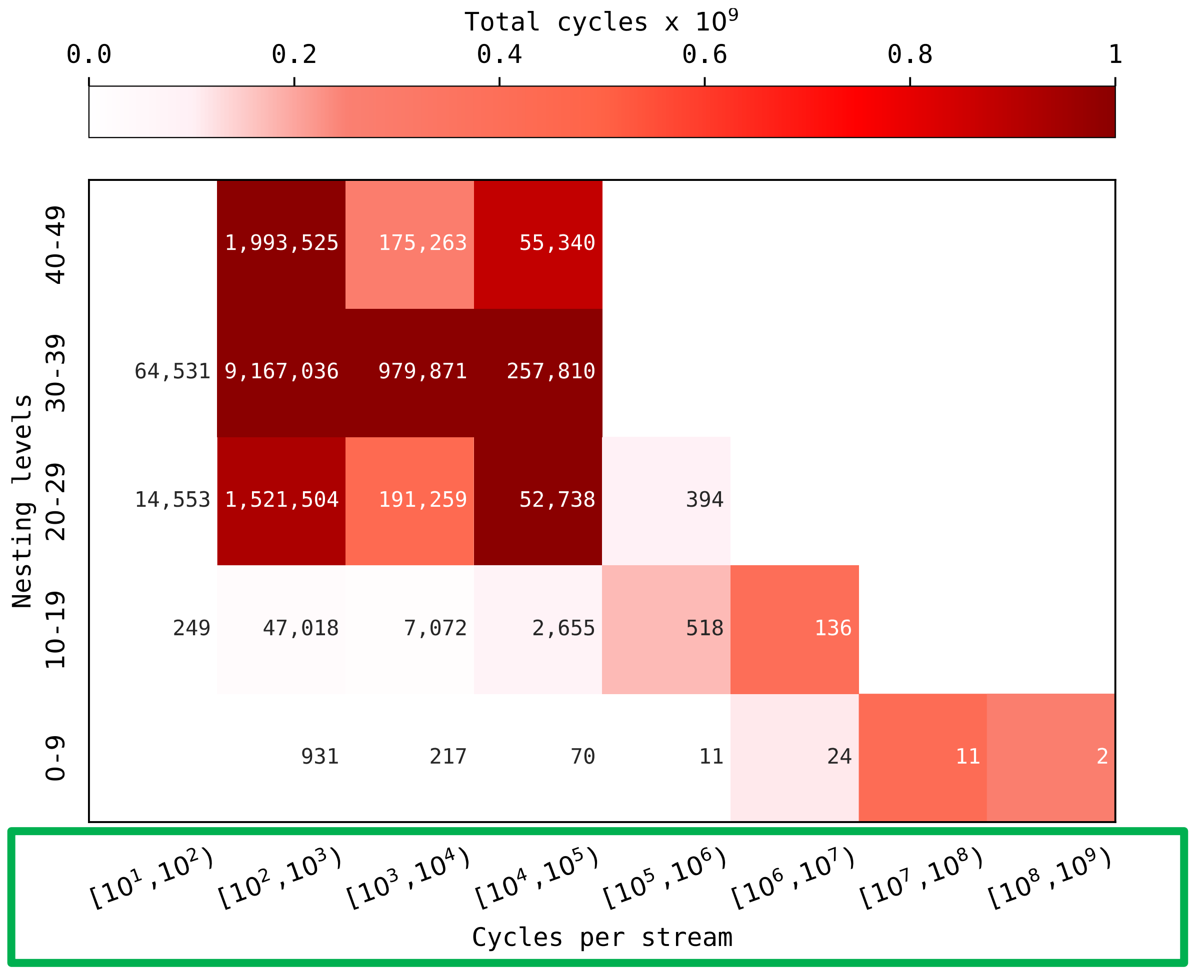
OPTIMIZATION 1:
REMOVING UNNEEDED STREAMS

Profiling and Optimizing Streams – Optimization I



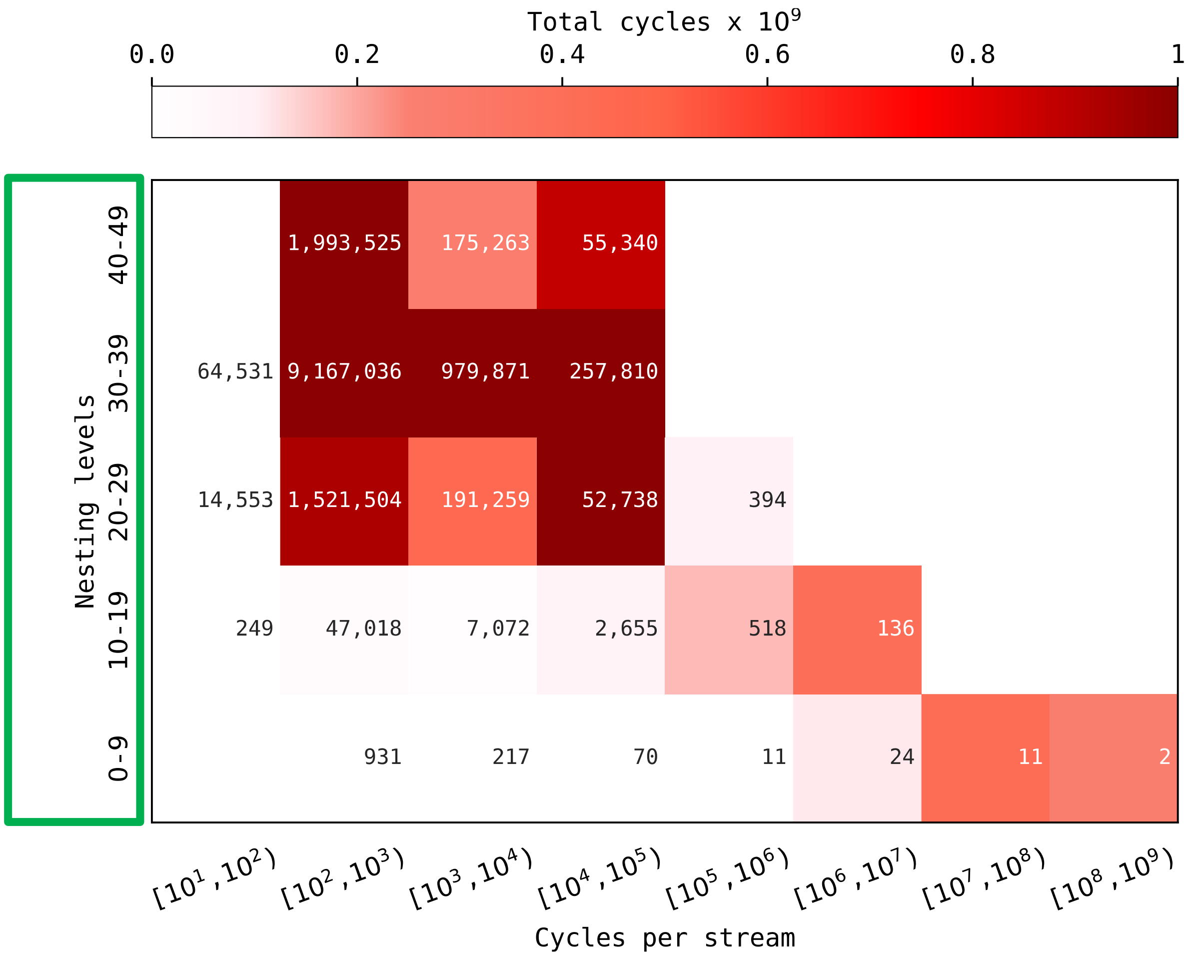
- Customized heatmap of stream executions in *mnemonics*

Profiling and Optimizing Streams – Optimization I



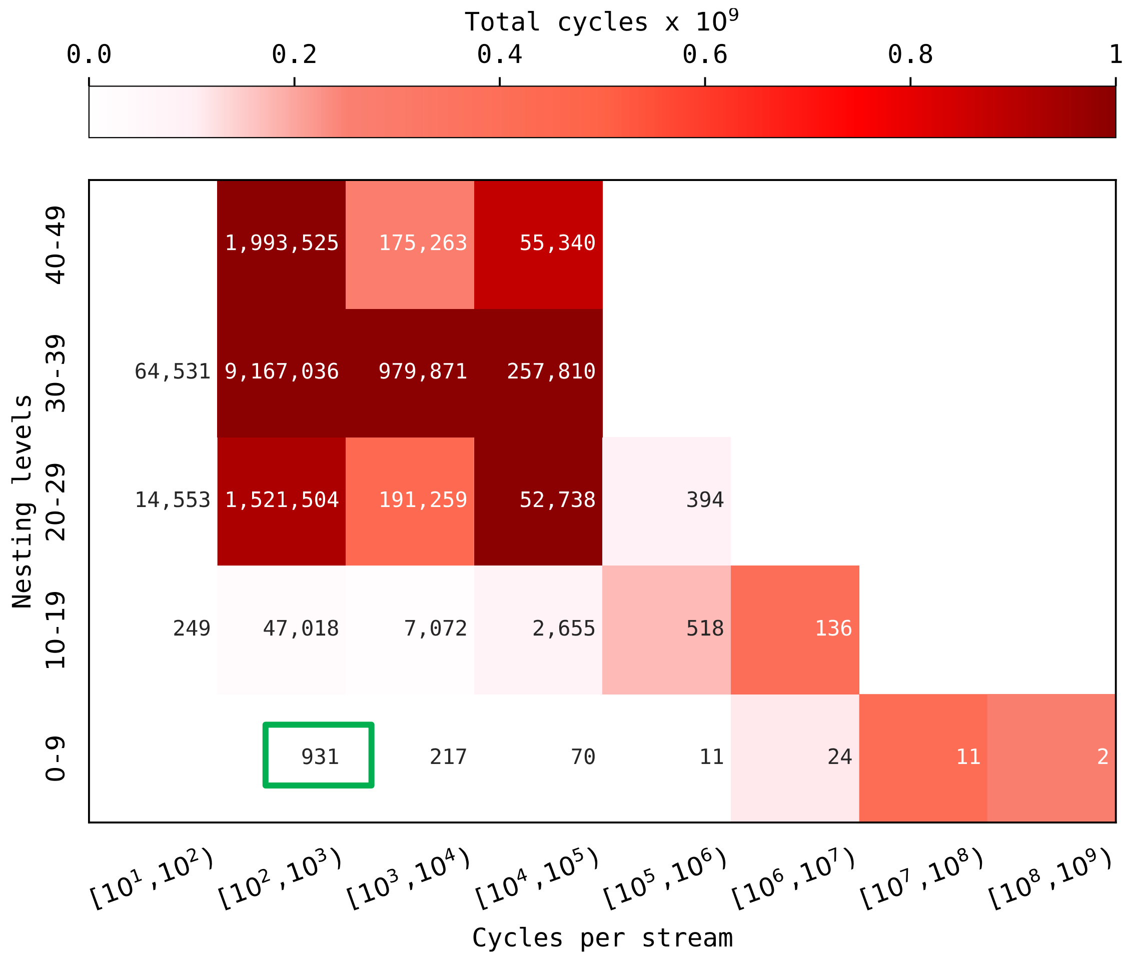
x-axis: streams are grouped by their *compensated cycles*

Profiling and Optimizing Streams – Optimization I



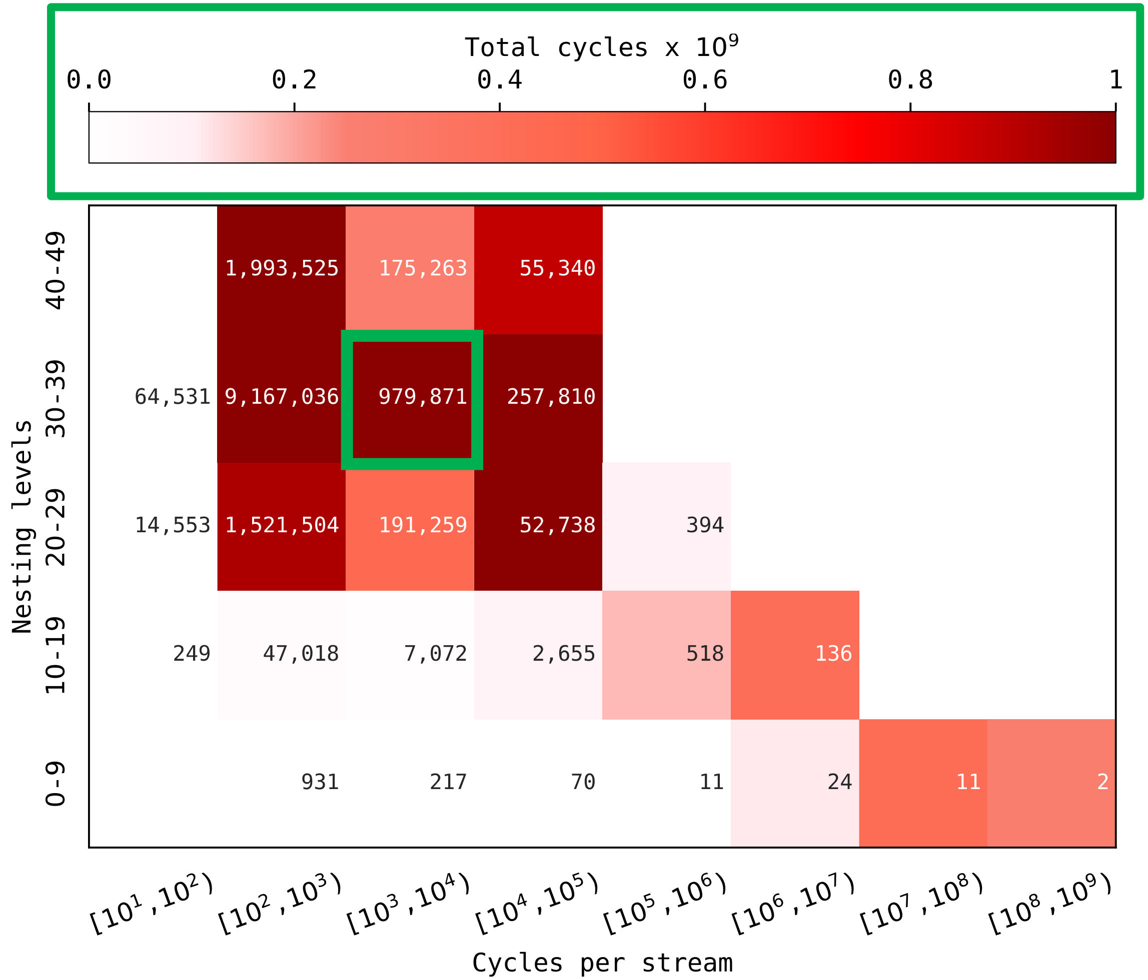
y-axis: streams are grouped by their *nesting level*

Profiling and Optimizing Streams – Optimization I



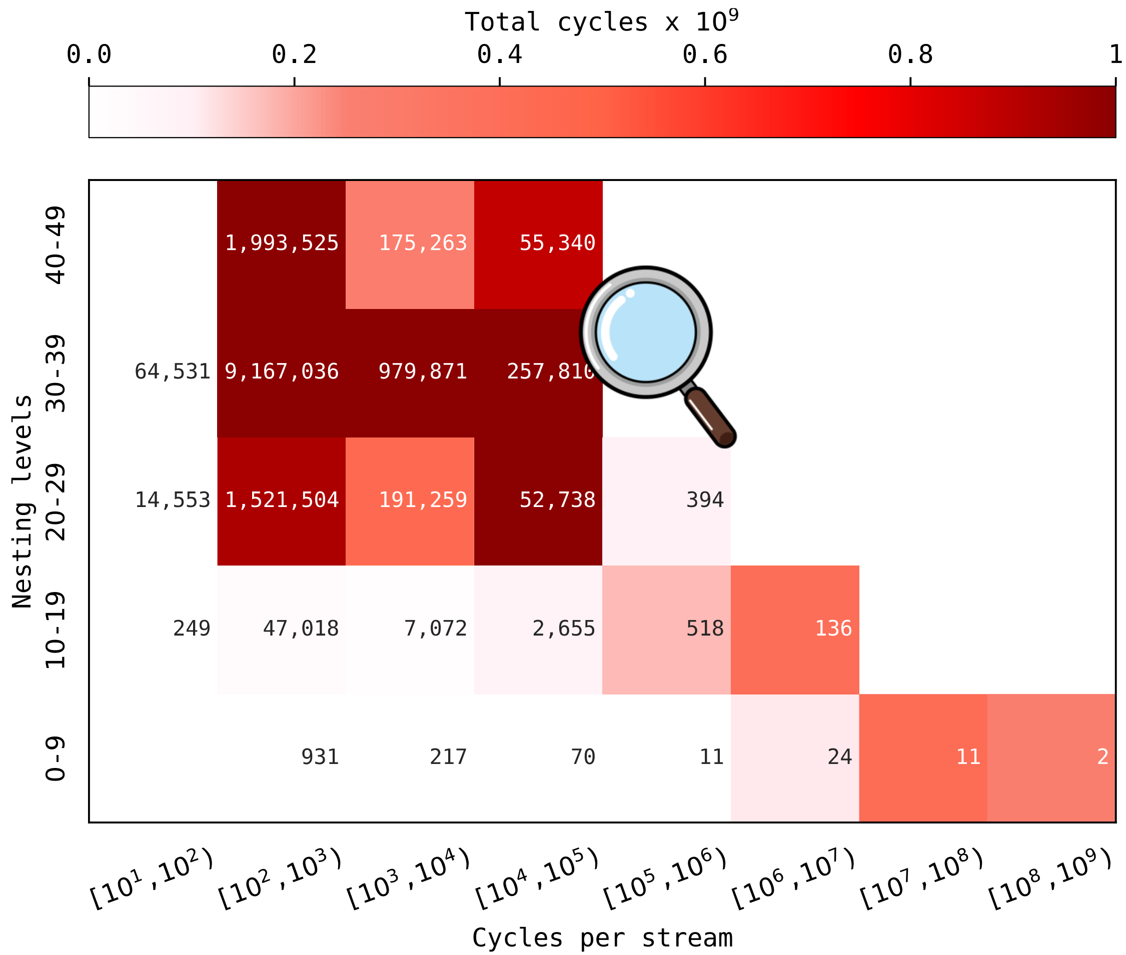
Cell: reports the number of stream executions for a given range of *nesting levels* and *cycles*

Profiling and Optimizing Streams – Optimization I



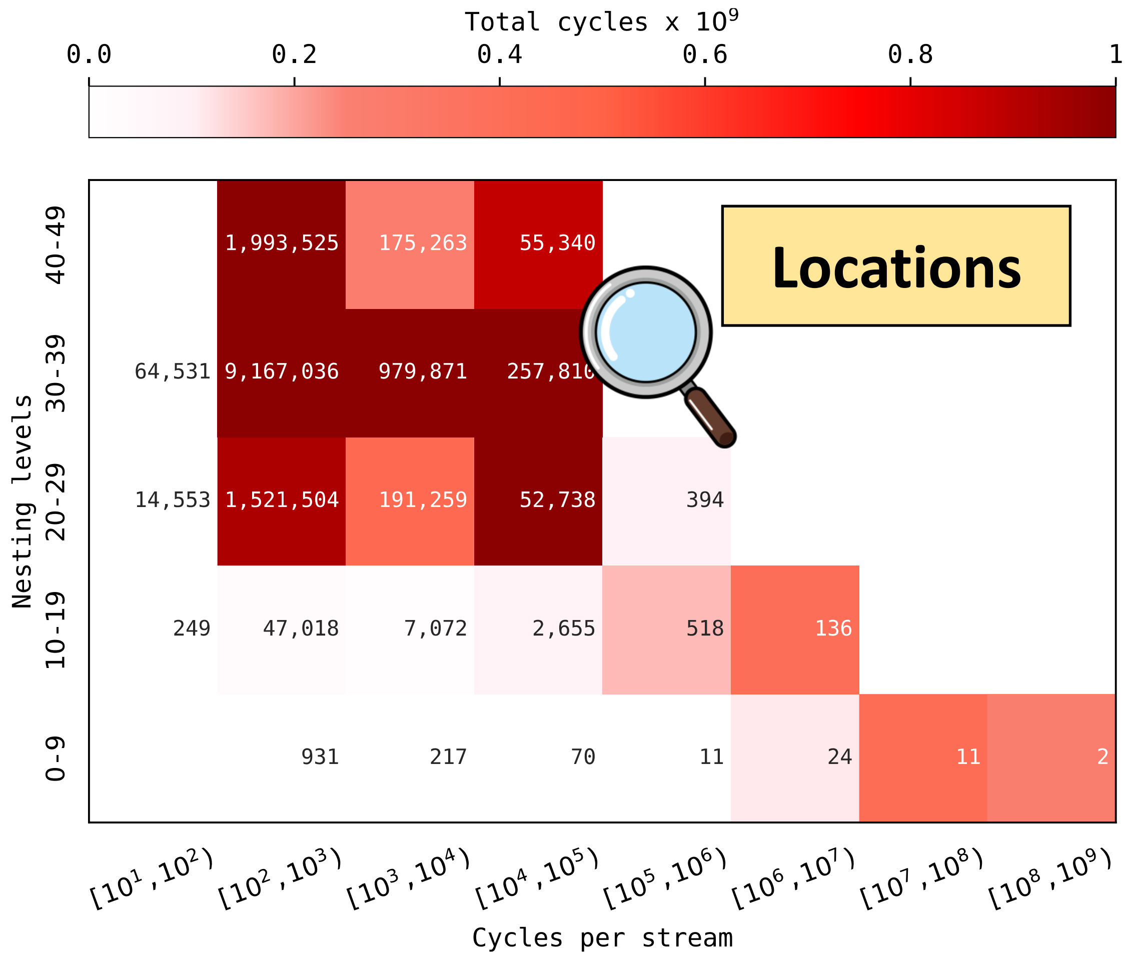
Heat: the color of a cell indicates the *total cycles of all the stream executions in the group*

Profiling and Optimizing Streams – Optimization I



Darker regions: good targets for stream code optimizations

Profiling and Optimizing Streams – Optimization I



Hot locations: locations in application code responsible for most of the stream processing

Profiling and Optimizing Streams – Optimization I

- **Optimization I**:
  - **Target:** *mnemonics* and *par-mnemonics*

- *MnemonicsCoderWithStream.**wordForNum**:*
  - Recursive method
  - Many stream executions

    ⚠ **Finding:** These streams always produce the same result
    - The result does not depend on any input of the current recursion level

- **Optimization:** Move the streams out the recursive code

  ☑ **Benefit:** Reduce overheads due to unneeded stream executions

Profiling and Optimizing Streams – Optimization I

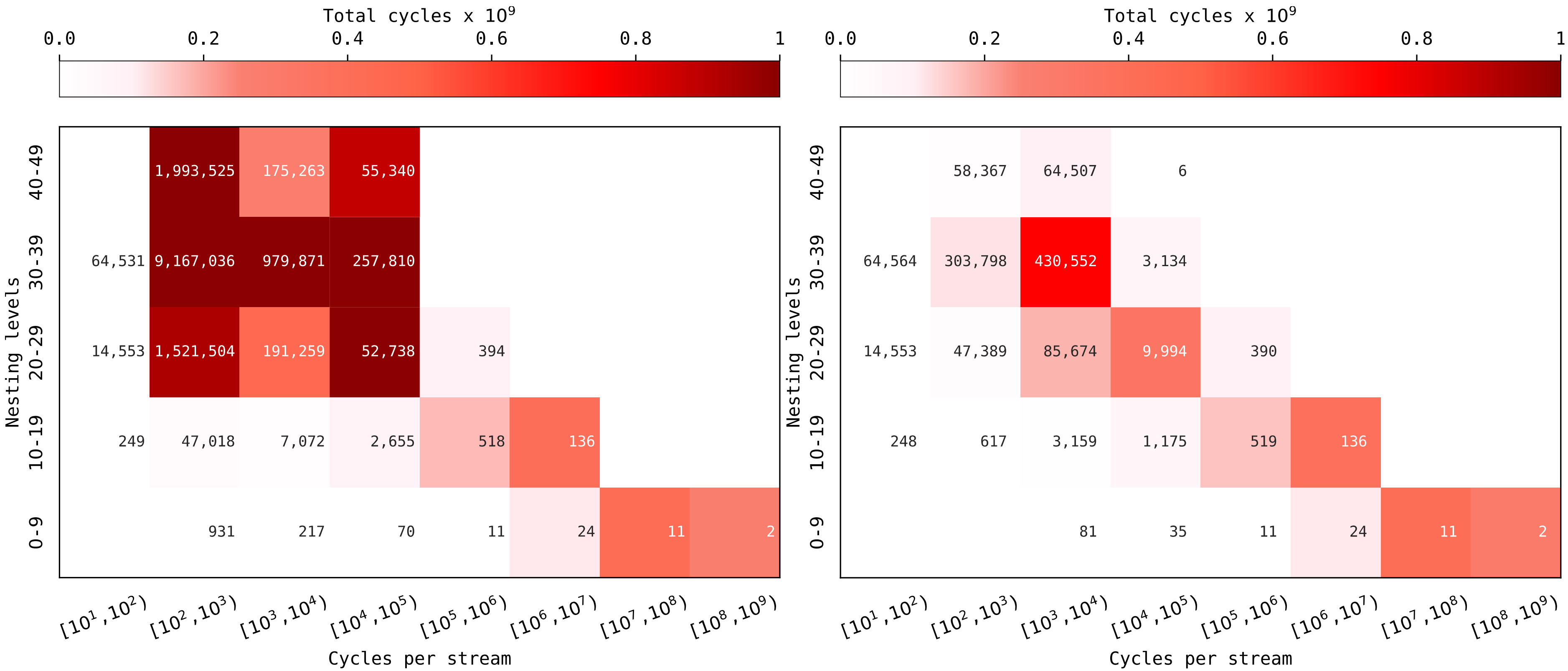
- *MnemonicsCoderWithStream.**lambda$encode$9**:*
  - Many stream executions

    ⚠ **Finding:** These streams are always created from an empty set
    - The empty set never changes
    - The streams do not contribute in any form to the workload output

- **Optimization:** Remove the unneeded streams

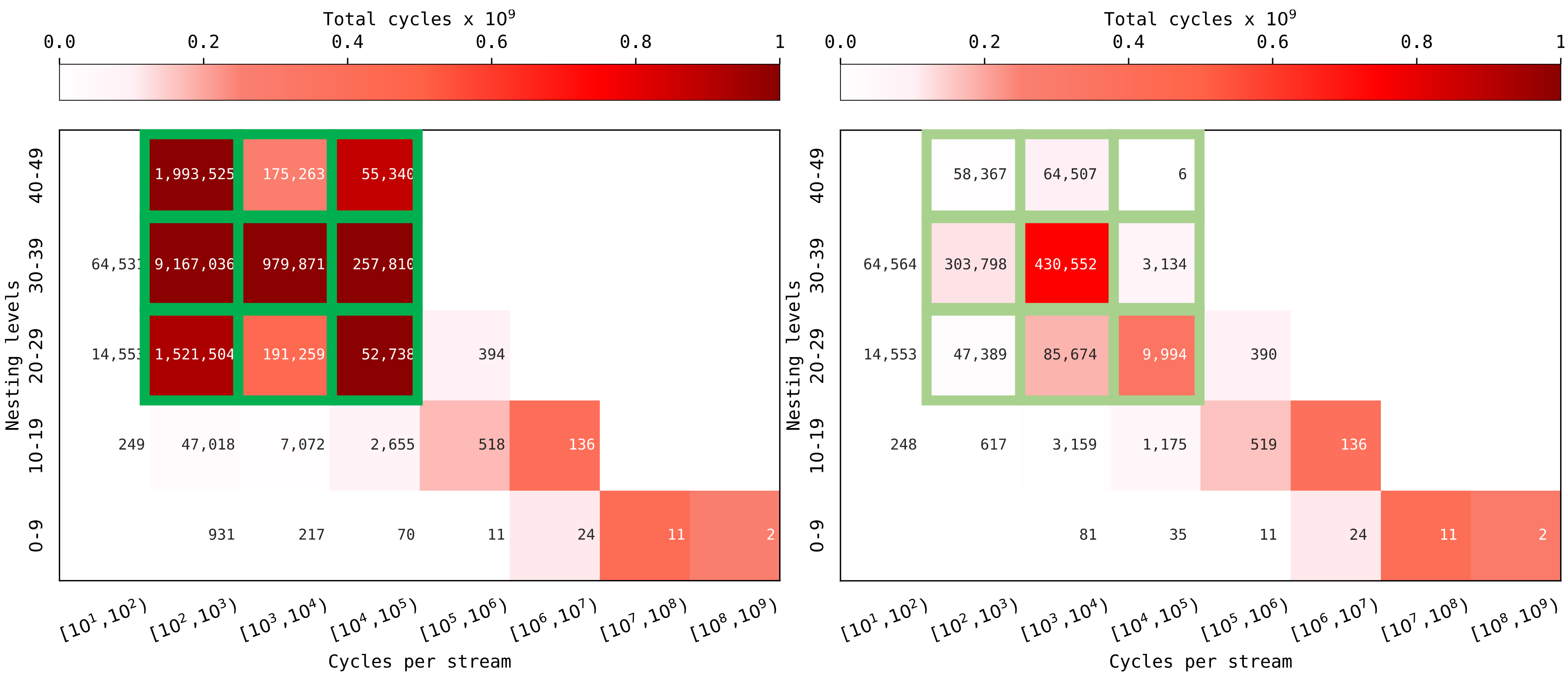
  ☑ **Benefit:** Reduce overheads due to unneeded stream executions

Profiling and Optimizing Streams – Optimization I



- Heatmaps of the original (***left***) and optimized (***right***) *mnemonics*

Profiling and Optimizing Streams – Optimization I



- Heatmaps of the original (***left***) and optimized (***right***) *mnemonics*

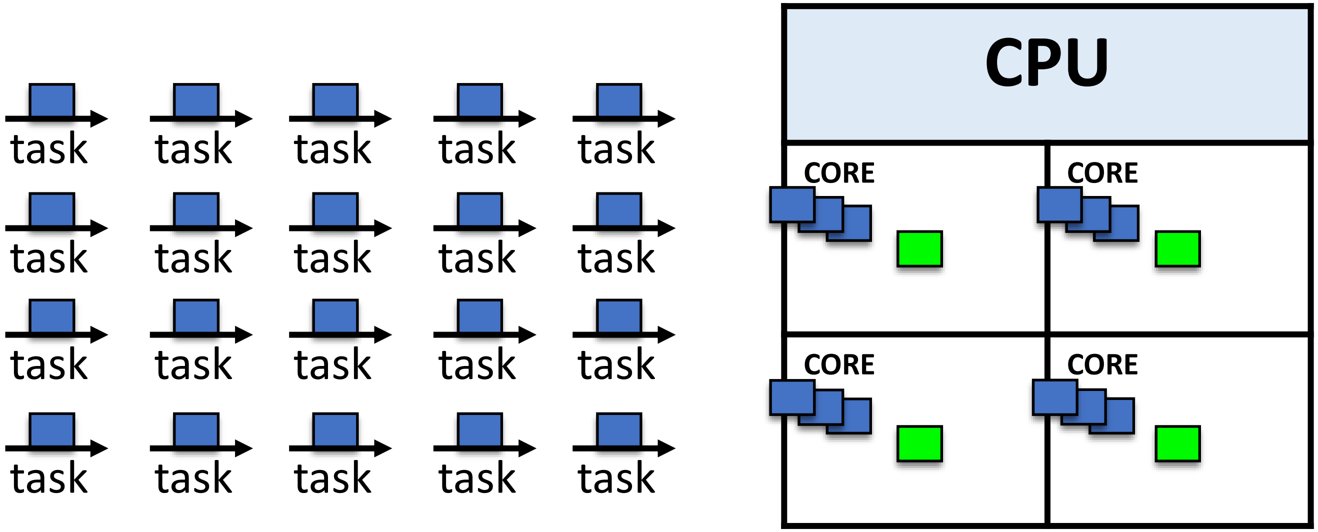
OPTIMIZATION 2:
IMPROVING LOAD IMBALANCE

- **Optimization II**:
  - **Target:** *par-mnemonics*
  - *StreamProf* reports the distribution of cycles per worker

⚠ **Finding:** Only 2 workers execute more than **99.99%** of the total cycles (on both machines)

- **Optimization:** Tune *task granularity* to improve load balance

☑ **Benefit:** Improve parallel stream execution performance



Speedups

Benchmark	Version	Machine	Time [ms]	Speedup Factor	Speedup 95% CI
mnemonics	orig	M_1	4,944.72		
		M_2	3,303.64		
	opt 1	M_1	1,200.70	**4.09**	(3.99, 4.19)
		M_2	981.97	**3.36**	(3.32, 3.40)
par-mnemonics	orig	M_1	4,419.53		
		M_2	2,977.83		
	opt 1	M_1	1,106.22	**3.98**	(3.89, 4.04)
		M_2	905.19	**3.27**	(3.20, 3.33)
	opt 2	M_1	880.09	**5.00**	(4.91, 5.10)
		M_2	764.63	**3.88**	(3.80, 3.96)

Speedups

Benchmark	Version	Machine	Time [ms]	Speedup Factor	Speedup 95% CI
mnemonics	orig	M_1	4,944.72		
		M_2	3,303.64		
	opt 1	M_1	1,200.70	**4.09**	(3.99, 4.19)
		M_2	981.97	**3.36**	(3.32, 3.40)
par-mnemonics	orig	M_1	4,419.53		
		M_2	2,977.83		
	opt 1	M_1	1,106.22	**3.98**	(3.89, 4.04)
		M_2	905.19	**3.27**	(3.20, 3.33)
	opt 2	M_1	880.09	**5.00**	(4.91, 5.10)
		M_2	764.63	**3.88**	(3.80, 3.96)

Speedups

Benchmark	Version	Machine	Time [ms]	Speedup Factor	Speedup 95% CI
mnemonics	orig	M_1	4,944.72		
		M_2	3,303.64		
	opt 1	M_1	1,200.70	**4.09**	(3.99, 4.19)
		M_2	981.97	**3.36**	(3.32, 3.40)
par-mnemonics	orig	M_1	4,419.53		
		M_2	2,977.83		
	opt 1	M_1	1,106.22	**3.98**	(3.89, 4.04)
		M_2	905.19	**3.27**	(3.20, 3.33)
	opt 2	M_1	880.09	**5.00**	(4.91, 5.10)
		M_2	764.63	**3.88**	(3.80, 3.96)

Speedups

Benchmark	Version	Machine	Time [ms]	Speedup Factor	Speedup 95% CI
mnemonics	orig	M_1	4,944.72		
		M_2	3,303.64		
	opt 1	M_1	1,200.70	**4.09**	(3.99, 4.19)
		M_2	981.97	**3.36**	(3.32, 3.40)
par-mnemonics	orig	M_1	4,419.53		
		M_2	2,977.83		
	opt 1	M_1	1,106.22	**3.98**	(3.89, 4.04)
		M_2	905.19	**3.27**	(3.20, 3.33)
	opt 2	M_1	880.09	**5.00**	(4.91, 5.10)
		M_2	764.63	**3.88**	(3.80, 3.96)

Speedups

Benchmark	Version	Machine	Time [ms]	Speedup Factor	Speedup 95% CI
mnemonics	orig	M_1	4,944.72		
		M_2	3,303.64		
	opt 1	M_1	1,200.70	**4.09**	(3.99, 4.19)
		M_2	981.97	**3.36**	(3.32, 3.40)
par-mnemonics	orig	M_1	4,419.53		
		M_2	2,977.83		
	opt 1	M_1	1,106.22	**3.98**	(3.89, 4.04)
		M_2	905.19	**3.27**	(3.20, 3.33)
	opt 2	M_1	880.09	**5.00**	(4.91, 5.10)
		M_2	764.63	**3.88**	(3.80, 3.96)

Speedups

Benchmark	Version	Machine	Time [ms]	Speedup Factor	Speedup 95% CI
mnemonics	orig	M_1	4,944.72		
		M_2	3,303.64		
	opt 1	M_1	1,200.70	**4.09**	(3.99, 4.19)
		M_2	981.97	**3.36**	(3.32, 3.40)
par-mnemonics	orig	M_1	4,419.53		
		M_2	2,977.83		
	opt 1	M_1	1,106.22	**3.98**	(3.89, 4.04)
		M_2	905.19	**3.27**	(3.20, 3.33)
	opt 2	M_1	880.09	**5.00**	(4.91, 5.10)
		M_2	764.63	**3.88**	(3.80, 3.96)

- ***speedup*** $= \dfrac{\textit{exec time original workload}}{\textit{exec time optimized workload}}$

Speedups

Benchmark	Version	Machine	Time [ms]	Speedup Factor	Speedup 95% CI
mnemonics	orig	M_1	4,944.72		
		M_2	3,303.64		
	opt 1	M_1	1,200.70	**4.09**	(3.99, 4.19)
		M_2	981.97	**3.36**	(3.32, 3.40)
par-mnemonics	orig	M_1	4,419.53		
		M_2	2,977.83		
	opt 1	M_1	1,106.22	**3.98**	(3.89, 4.04)
		M_2	905.19	**3.27**	(3.20, 3.33)
	opt 2	M_1	880.09	**5.00**	(4.91, 5.10)
		M_2	764.63	**3.88**	(3.80, 3.96)

- 95% confidence intervals (*CI*)

Speedups

Benchmark	Version	Machine	Time [ms]	Speedup Factor	Speedup 95% CI
mnemonics	orig	M_1	4,944.72		
		M_2	3,303.64		
	opt 1	M_1	1,200.70	**4.09**	(3.99, 4.19)
		M_2	981.97	**3.36**	(3.32, 3.40)
par-mnemonics	orig	M_1	4,419.53		
		M_2	2,977.83		
	opt 1	M_1	1,106.22	**3.98**	(3.89, 4.04)
		M_2	905.19	**3.27**	(3.20, 3.33)
	opt 2	M_1	880.09	**5.00**	(4.91, 5.10)
		M_2	764.63	**3.88**	(3.80, 3.96)

- ☑ Removing unneeded stream processing (***opt1***) improves the performance of both *mnemonics* and *par-mnemonics*
  - ☑ *Only 6* lines of code changed

Speedups

Benchmark	Version	Machine	Time [ms]	Speedup Factor	Speedup 95% CI
mnemonics	orig	M_1	4,944.72		
		M_2	3,303.64		
	opt 1	M_1	1,200.70	**4.09**	(3.99, 4.19)
		M_2	981.97	**3.36**	(3.32, 3.40)
par-mnemonics	orig	M_1	4,419.53		
		M_2	2,977.83		
	opt 1	M_1	1,106.22	**3.98**	(3.89, 4.04)
		M_2	905.19	**3.27**	(3.20, 3.33)
	opt 2	M_1	880.09	**5.00**	(4.91, 5.10)
		M_2	764.63	**3.88**	(3.80, 3.96)

- ☑ Improving load balance (***opt2***) results in performance gains in *par-mnemonics*
  - ☑ 2 lines of code changed

Speedups

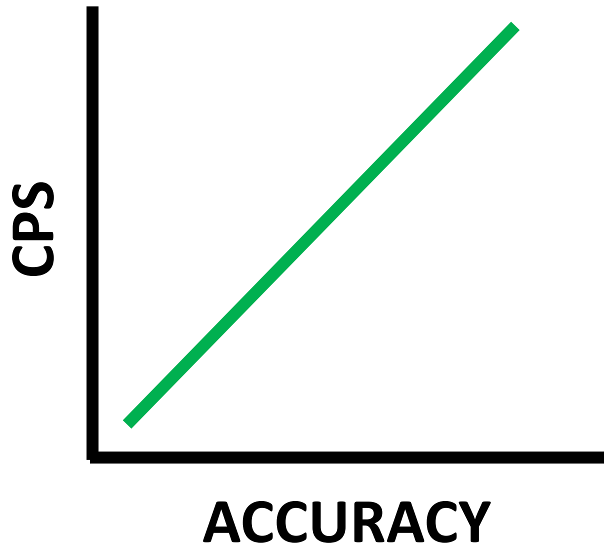
Benchmark	Version	Machine	Time [ms]	Speedup Factor	Speedup 95% CI
mnemonics	orig	M_1	4,944.72		
		M_2	3,303.64		
	opt 1	M_1	1,200.70	**4.09**	(3.99, 4.19)
		M_2	981.97	**3.36**	(3.32, 3.40)
par-mnemonics	orig	M_1	4,419.53		
		M_2	2,977.83		
	opt 1	M_1	1,106.22	**3.98**	(3.89, 4.04)
		M_2	905.19	**3.27**	(3.20, 3.33)
	opt 2	M_1	880.09	**5.00**	(4.91, 5.10)
		M_2	764.63	**3.88**	(3.80, 3.96)

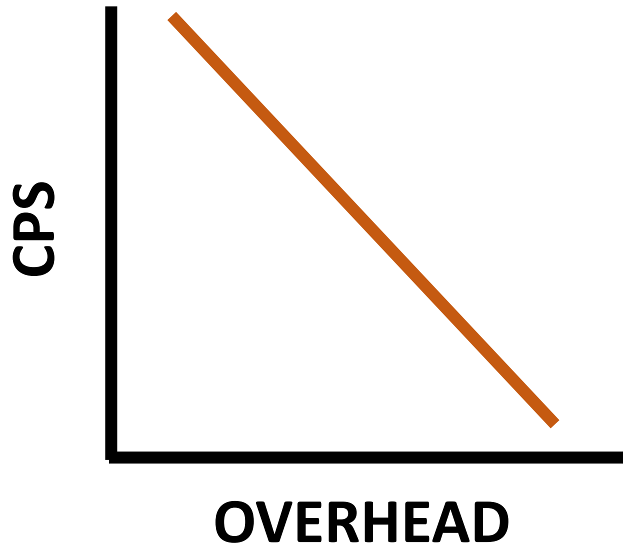
☑ Average speedup is **3.94x** considering all workloads on both machines

ACCURACY AND OVERHEAD EVALUATION

Accuracy and Overhead Evaluation

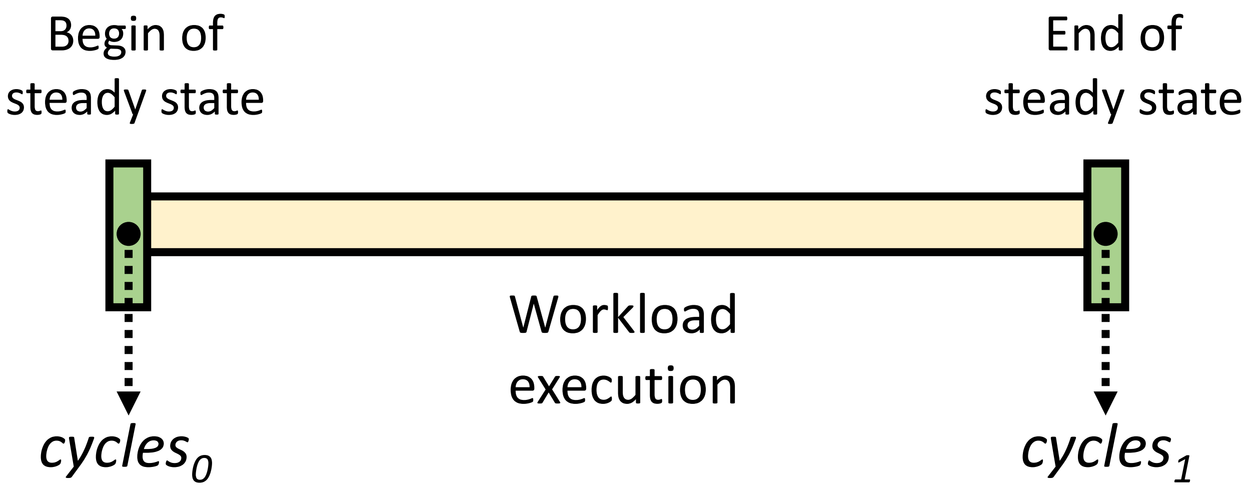
- **Average *Cycles Per Span* (CPS)**
  - **Expectation:**
    - For workloads with a low CPS
      - the relative cost of the inserted instrumentation code
        - is higher than for workloads with a higher CPS





Accuracy and Overhead Evaluation

- **Baseline computation:**
  - To compute CPS and accuracy, we need a *baseline*
  - All evaluated workloads take place only within stream executions
  - Baseline: total cycles elapsed by all threads involved in stream processing



$$\boldsymbol{baseline = cycles_1 - cycles_0}$$

Accuracy and Overhead Evaluation

- **CPS computation:**

$$\boldsymbol{CPS = baseline \,/\, total\_spans}$$

- **Accuracy computation:**

$$\mathbf{RE} = \left| \frac{\boldsymbol{baseline - compensated\_cycles}}{\boldsymbol{baseline}} \right|$$

$$\mathbf{Accuracy = 1 - RE}$$

Accuracy and Overhead Evaluation

- Testbed:
  - All stream-based workloads from *Renaissance*
    - *mnemonics*
    - *par-mnemonics*
    - *scrabble*
  - OpenJDK [5]
    - Collection of stream-based workloads released by the developers of OpenJDK
  - JEDI [6]
    - Benchmark suite consisting of TPCH-queries recast to Java Streams

[5] https://github.com/usi-dag/jdk20u/tree/master/test/micro/org/openjdk/bench/java/util/stream

[6] https://github.com/usi-dag/JEDI

ACCURACY EVALUATION

Accuracy Evaluation

Benchmark	Version	Machine	CPS	Accuracy [%]
mnemonics	orig	M_1	915	**87.99**
		M_2	679	**95.18**
	opt 1	M_1	2,972	**98.99**
		M_2	2,699	**99.54**
par-mnemonics	orig	M_1	1,027	**88.27**
		M_2	761	**95.83**
	opt 1	M_1	3,290	**99.68**
		M_2	2,989	**99.66**
	opt 2	M_1	3,241	**99.42**
		M_2	3,436	**94.98**
scrabble	orig	M_1	1,481	**89.97**
		M_2	2,151	**97.13**

Accuracy Evaluation

Benchmark	Version	Machine	CPS	Accuracy [%]
mnemonics	orig	M_1	915	**87.99**
		M_2	679	**95.18**
	opt 1	M_1	2,972	**98.99**
		M_2	2,699	**99.54**
par-mnemonics	orig	M_1	1,027	**88.27**
		M_2	761	**95.83**
	opt 1	M_1	3,290	**99.68**
		M_2	2,989	**99.66**
	opt 2	M_1	3,241	**99.42**
		M_2	3,436	**94.98**
scrabble	orig	M_1	1,481	**89.97**
		M_2	2,151	**97.13**

Accuracy Evaluation

Benchmark	Version	Machine	CPS	Accuracy [%]
mnemonics	orig	M_1	915	**87.99**
		M_2	679	**95.18**
	opt 1	M_1	2,972	**98.99**
		M_2	2,699	**99.54**
par-mnemonics	orig	M_1	1,027	**88.27**
		M_2	761	**95.83**
	opt 1	M_1	3,290	**99.68**
		M_2	2,989	**99.66**
	opt 2	M_1	3,241	**99.42**
		M_2	3,436	**94.98**
scrabble	orig	M_1	1,481	**89.97**
		M_2	2,151	**97.13**

- Accuracy shown as percentage

Accuracy Evaluation

Benchmark	Version	Machine	CPS	Accuracy [%]
mnemonics	orig	M_1	**915**	**87.99**
		M_2	**679**	**95.18**
	opt 1	M_1	**2,972**	**98.99**
		M_2	**2,699**	**99.54**
par-mnemonics	orig	M_1	**1,027**	**88.27**
		M_2	**761**	**95.83**
	opt 1	M_1	**3,290**	**99.68**
		M_2	**2,989**	**99.66**
	opt 2	M_1	**3,241**	**99.42**
		M_2	**3,436**	**94.98**
scrabble	orig	M_1	**1,481**	**89.97**
		M_2	**2,151**	**97.13**

- Positive correlation between CPS and accuracy
  - PCC: **0.71**, considering all workloads on M_1 and M_2

Accuracy Evaluation

Benchmark	Version	Machine	CPS	Accuracy [%]
mnemonics	orig	M_1	915	**87.99**
		M_2	679	**95.18**
	opt 1	M_1	2,972	**98.99**
		M_2	2,699	**99.54**
par-mnemonics	orig	M_1	1,027	**88.27**
		M_2	761	**95.83**
	opt 1	M_1	3,290	**99.68**
		M_2	2,989	**99.66**
	opt 2	M_1	3,241	**99.42**
		M_2	3,436	**94.98**
scrabble	orig	M_1	1,481	**89.97**
		M_2	2,151	**97.13**
JEDI (mean)		M_1		**91.55**
		M_2		**90.34**
OpenJDK (mean)		M_1		**94.82**
		M_2		**96.73**

OVERHEAD EVALUATION

Overhead Evaluation

Benchmark	Version	Machine	Time [ms]	CPS	Overhead Factor	Overhead 95% CI
mnemonics	orig	M_1	4,944.72	915	**1.55**	(1.51, 1.60)
		M_2	3,303.64	679	**1.46**	(1.44, 1.49)
	opt 1	M_1	1,200.70	2,972	**1.16**	(1.15, 1.16)
		M_2	981.97	2,699	**1.12**	(1.12, 1.13)
par-mnemonics	orig	M_1	4,419.53	1,027	**1.51**	(1.47, 1.54)
		M_2	2,977.83	761	**1.41**	(1.38, 1.45)
	opt 1	M_1	1,106.22	3,290	**1.13**	(1.13, 1.14)
		M_2	905.19	2,989	**1.11**	(1.11, 1.12)
	opt 2	M_1	880.09	3,241	**1.10**	(1.10, 1.11)
		M_2	764.63	3,436	**1.09**	(1.08, 1.09)
scrabble	orig	M_1	310.84	1,481	**1.43**	(1.42, 1.44)
		M_2	187.09	2,151	**1.23**	(1.22, 1.23)

Overhead:

- ***slowdown factor =*** $\dfrac{\textbf{\textit{exec time workload with profiling}}}{\textbf{\textit{exec time workload without profiling}}}$

Università della Svizzera italiana

Overhead Evaluation

Benchmark	Version	Machine	Time [ms]	CPS	Overhead Factor	Overhead 95% CI
mnemonics	orig	M_1	4,944.72	**915**	**1.55**	(1.51, 1.60)
		M_2	3,303.64	**679**	**1.46**	(1.44, 1.49)
	opt 1	M_1	1,200.70	**2,972**	**1.16**	(1.15, 1.16)
		M_2	981.97	**2,699**	**1.12**	(1.12, 1.13)
par-mnemonics	orig	M_1	4,419.53	**1,027**	**1.51**	(1.47, 1.54)
		M_2	2,977.83	**761**	**1.41**	(1.38, 1.45)
	opt 1	M_1	1,106.22	**3,290**	**1.13**	(1.13, 1.14)
		M_2	905.19	**2,989**	**1.11**	(1.11, 1.12)
	opt 2	M_1	880.09	**3,241**	**1.10**	(1.10, 1.11)
		M_2	764.63	**3,436**	**1.09**	(1.08, 1.09)
scrabble	orig	M_1	310.84	**1,481**	**1.43**	(1.42, 1.44)
		M_2	187.09	**2,151**	**1.23**	(1.22, 1.23)

- Negative correlation between CPS and overhead
  - PCC: **-0.96**, considering all workloads on M_1 and M_2

Overhead Evaluation

Benchmark	Version	Machine	Time [ms]	CPS	Overhead Factor	Overhead 95% CI
mnemonics	orig	M_1	4,944.72	915	**1.55**	(1.51, 1.60)
		M_2	3,303.64	679	**1.46**	(1.44, 1.49)
	opt 1	M_1	1,200.70	2,972	**1.16**	(1.15, 1.16)
		M_2	981.97	2,699	**1.12**	(1.12, 1.13)
par-mnemonics	orig	M_1	4,419.53	1,027	**1.51**	(1.47, 1.54)
		M_2	2,977.83	761	**1.41**	(1.38, 1.45)
	opt 1	M_1	1,106.22	3,290	**1.13**	(1.13, 1.14)
		M_2	905.19	2,989	**1.11**	(1.11, 1.12)
	opt 2	M_1	880.09	3,241	**1.10**	(1.10, 1.11)
		M_2	764.63	3,436	**1.09**	(1.08, 1.09)
scrabble	orig	M_1	310.84	1,481	**1.43**	(1.42, 1.44)
		M_2	187.09	2,151	**1.23**	(1.22, 1.23)
JEDI (mean)		M_1			**1.13**	
		M_2			**1.15**	
OpenJDK (mean)		M_1			**1.07**	
		M_2			**1.06**	

Limitations

- **Limitations of the technique**:
  - Additional sources of perturbation (e.g., prevention of JIT compiler optimizations) are not compensated
  - The profiles produced using our technique are platform dependent and require the availability of per-thread virtualized reference-cycle counters

Summary of Contributions

- New technique enabling cycle-accurate profiling of stream executions
  - Implemented in *StreamProf*, a novel stream profiler for the JVM
    - Uses a perturbation-compensation technique
- Analysis of *Renaissance*, revealing previously unknown stream-related performance issues
- Optimization of two stream-based workloads from *Renaissance*
  - Average speedup: 3.94x
- Evaluation results show that our profiling technique is efficient and yields accurate profiles

Thanks a lot for your attention!

☺