Accurate Fork-join Profiling on the Java Virtual Machine

Matteo Basso, Eduardo Rosales, Filippo Schiavio,
Andrea Rosà, Walter Binder

Università della Svizzera italiana, Switzerland

Euro-Par 2022
August 24, 2022

Introduction

- Fork-Join model in Java
  - Included in the Java Class Library since Java 7
  - At the core of many Java, Scala, Groovy, and Clojure frameworks
- Understanding and optimizing fork-join computations is crucial
- Dedicated profilers need to:
  - Collect specific fork-join metrics
    - E.g., task stealing, parent/child task relationships
  - Profile task granularity
    - A measure of the amount of computations performed by each task

Motivation

- There is no specific fork-join profiler for the Java Virtual Machine (JVM)
- Accurately profiling fork-join computations is challenging:
  - Task unforking
  - Task cancellation
  - Task reinitialization
- Existing tools for task-granularity profiling on the JVM:
  - High overhead
  - Significant measurement perturbations
  - Inaccurate profiles

Contributions

- New profiling model capturing any legitimate (non-erroneous) use of the Java fork-join framework
  - Including specific fork-join metrics and task granularity
  - Accurately detecting parent/child relationships between tasks
    - Multiple fork-join computations concurrently execute in the same fork-join pool
- Implementation of profiling model in the `wosp` profiler
- Evaluation of accuracy and overhead of `wosp`
  - Including comparison with the task-granularity profiler `FJProf` [1]

[1] E. Rosales et al., "FJProf: Profiling Fork/Join Applications on the Java Virtual Machine." VALUETOOLS 2020.

Background - ForkJoinPool API

- Fork-join framework implementation in Java based on work-stealing
- Main abstractions:
  - Task (`ForkJoinTask`)
    - Task execution: `ForkJoinTask.exec`
  - Fork-join pool (`ForkJoinPool`)
- Given two tasks *p* and *c* such that *p* forks *c*
  - *p* is the parent task
  - *c* is the child (or subtask) of *p*

Background - Task Reuse

- Reusage of the same task instance to perform multiple executions
  - Useful to:
    - Reduce object allocations
    - Execute pre-constructed trees of tasks in loops
- `ForkJoinTask.reinitialize`
  - Resets the internal state of the task
  - Allowed if task was:
    - never forked, or
    - forked, executed, and all joins completed

Background - Task Unforking

- Unscheduling of a task which was previously forked
  - Useful to reduce the task-management overhead of the framework
  - Typically used to locally process tasks that could have been—but actually were not—stolen
- `ForkJoinTask.tryUnfork`
  - Allowed if task execution not already started in another thread

Background - Task Cancellation

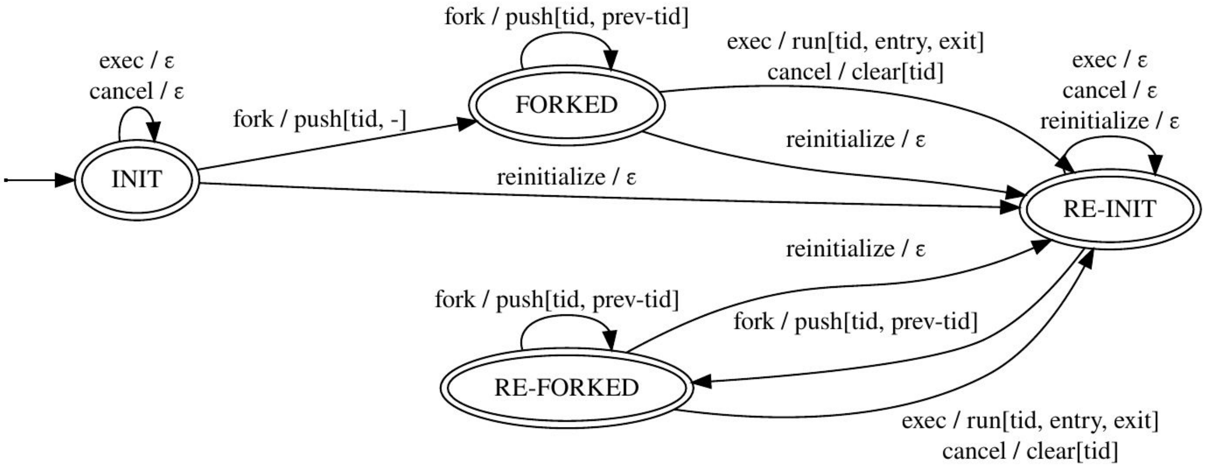
- Cancellation of task execution by the user
  - Useful for specific optimizations (e.g., short-circuiting)
- `ForkJoinTask.cancel`
  - May fail depending on the internal state of the task
    - e.g., if the task has already completed
  - Task is unscheduled and execution suppressed
  - Before subsequent usages, user must call `ForkJoinTask.reinitialize`

Profiling Model - Focus and Goals

- ➢ We focus on the execution of tasks that have been forked
  - Tasks that have been arranged for parallel execution
- ➢ We disregard the sequential execution of children tasks
  - We incorporate the granularity of any direct synchronous method invocations into the granularity of their parent tasks

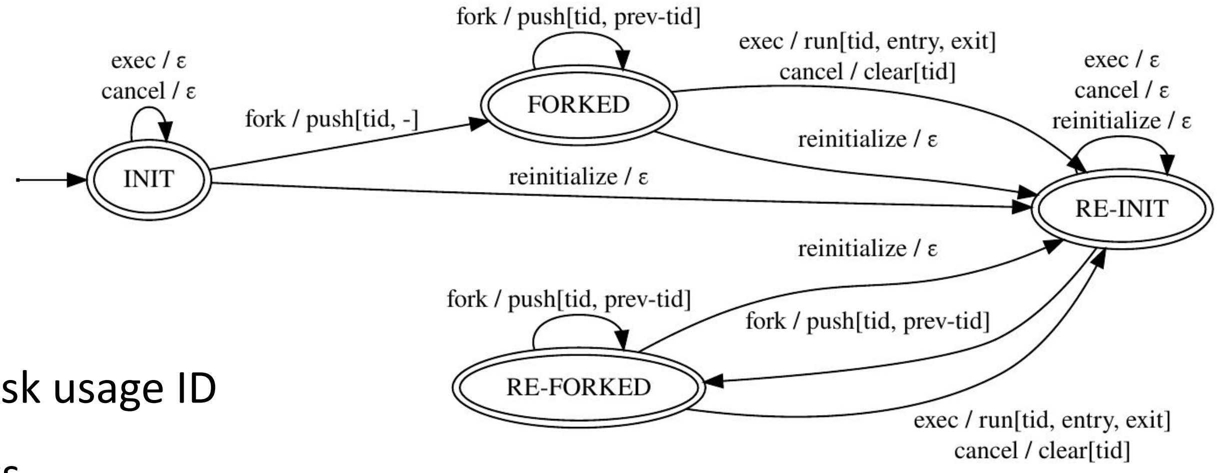
Profiling Model - Task State Machine

- ➢ We model each fork-join task as a finite state machine



- ➢ Four states: INIT, FORKED, RE-INIT, and RE-FORKED
- ➢ Transitions: events
  - Four events: *fork*, *exec*, *cancel*, and *reinitialize*
  - Trace record produced as output

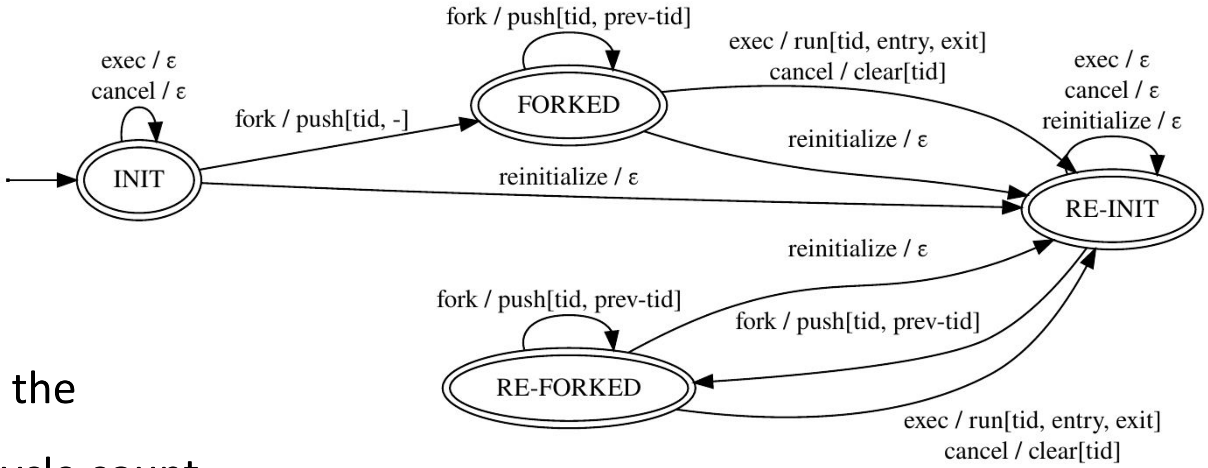
Profiling Model - Task State Machine



- Three traces records:
  push[tid, prev-tid],
  clear[tid],
  and *run[tid, entry, exit]*
- *tid* refers to a unique task usage ID
  - Sequence of events
  - Generated upon the occurrence of each *fork*
  - The same task instance may be associated to multiple IDs due to task reuse
  - Reconstruction of task lifecycle done by chaining *push* trace records

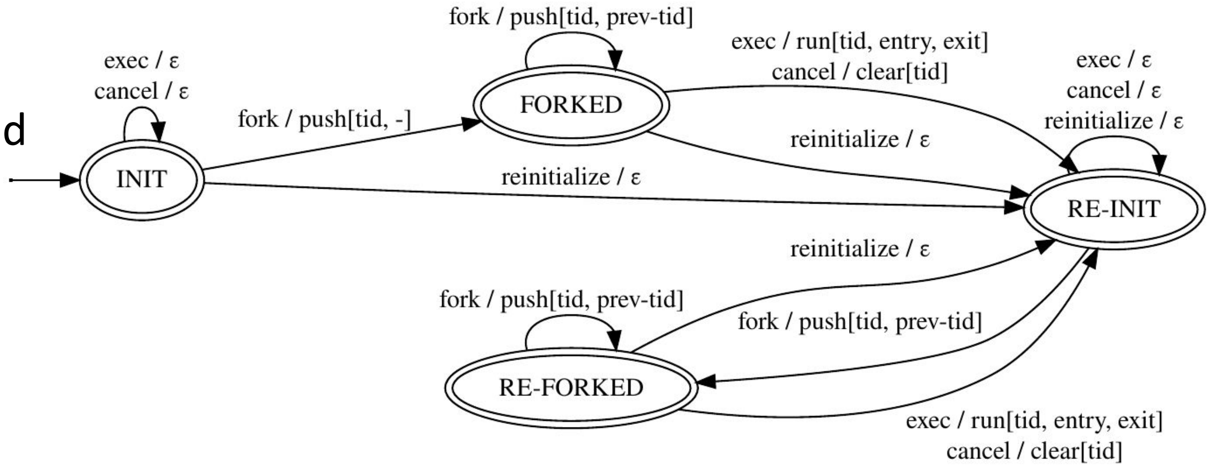
Profiling Model - Task State Machine

- Three traces records:
  push[tid, prev-tid],
  clear[tid],
  and *run[tid, entry, exit]*
- *entry* and *exit* represent the thread-local reference-cycle count
  - The clock cycles elapsed during thread execution until when the measurement was performed
  - Used as a measure of task granularity



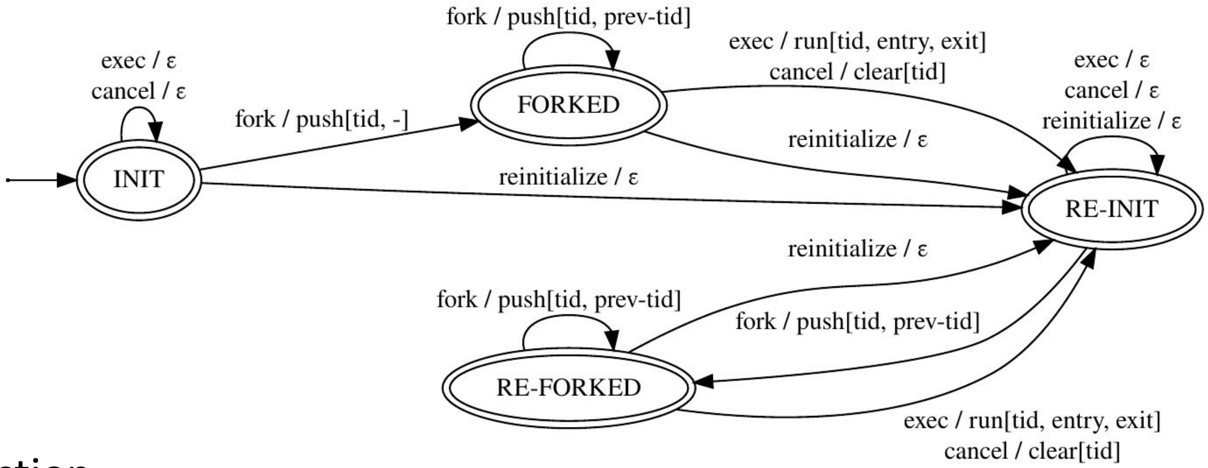
Profiling Model - Task State Machine

- The *run[tid, entry, exit]* trace record is composed of two sub-records *run_begin[tid, entry]* and *run_end[exit]*
  - Support nesting runs
  - *run_begin* and *run_end* are always balanced



Profiling Model - Task State Machine

- ➢ No *unfork* event
- ➢ Unforked tasks will be either
    - ● Executed
    - ● Discarded
- ➢ Leads to overhead reduction



Profiling Model - Work Stealing

- Each trace record contains a reference to the thread that produced it
- If a *push* and a *run* associated to the same ID π are produced by different threads *t0* and *t1*, we can conclude that *t1* has stolen the task associated to π from *t0*

Profiling Model - Nested Executions

- Trace records of a task *i* may appear between the *run_begin* and the *run_end* records of another task *o*
  - Nested task execution
    - *o* is the outer task, *i* is the inner task
  - Takes place because of
    - Parent/child executions (*fork*, *unfork*, and then *exec*)
    - Work stealing
- Nested executions are crucial to correctly compute the task granularity

Profiling Model - Parent/Child Rel.

- ➢ Outer tasks may not be parent tasks of their corresponding inner tasks
  - A push of a task c occurring within the run of another task *p* indicates that *p* is the parent task of *c*

Implementation

- We implement our model in a profiler called `wosp`
- `wosp` is composed of three main components
  - The instrumentation
  - The tracing agent
  - The postprocessor

Implementation - Metrics

- Task granularity
- Parent/child relationships (task dependencies)
- Number of tasks stolen from/by a given thread (task-stealing rate)
- Load balance
- Task execution nesting
- Task-reuse rate

Implementation - Instrumentation

- `wosp` is based on DiSL [1]
  - A load-time out-of-process Java bytecode instrumentation framework
- High accuracy and low overhead is of paramount importance
  - Minimal instrumentation
  - Instrumentation code that minimizes online processing
  - Thread-local data structures

---

[1] L. Marek et al, "DiSL: A Domain-Specific Language for Bytecode Instrumentation". AOSD 2012.

Implementation - Tracing Agent

- To produce trace records, the instrumentation code calls a tracing agent attached to the executing JVM via the Java Native Interface (JNI)
  - Thread-local traces
  - Thread-local buffers
    - Allocated at VM startup
    - Acquired when needed
  - Buffered data is dumped to binary files only at JVM shutdown
- Reference cycles are collected per thread using the PAPI [1] library

[1] http://icl.utk.edu/papi

Implementation - Postprocessor

- ➢ After the application execution, a Java application reads and decodes the binary traces
- ➢ Decoding exploits a stack of *`run_begin`* records
  - *`run_begin`*: pushed on the stack
  - *`run_end`*: the corresponding *`run_begin`* is popped from the stack
- ➢ Task granularity of each task
- ➢ Parent/child relationships
  - Decoding a *`push[child-id]`* while *`run_begin[parent-id]`* is at the top of the stack

Evaluation

- Evaluated metrics:
  - Accuracy (in terms of total task granularity)
  - Profiling overhead
- We compare `wosp` with the task-granularity profiler `FJProf` [1]
- We target the Renaissance [2] and Aeminium [3] benchmark suites
  - Workloads that make use of the peculiar features of the Java fork-join framework

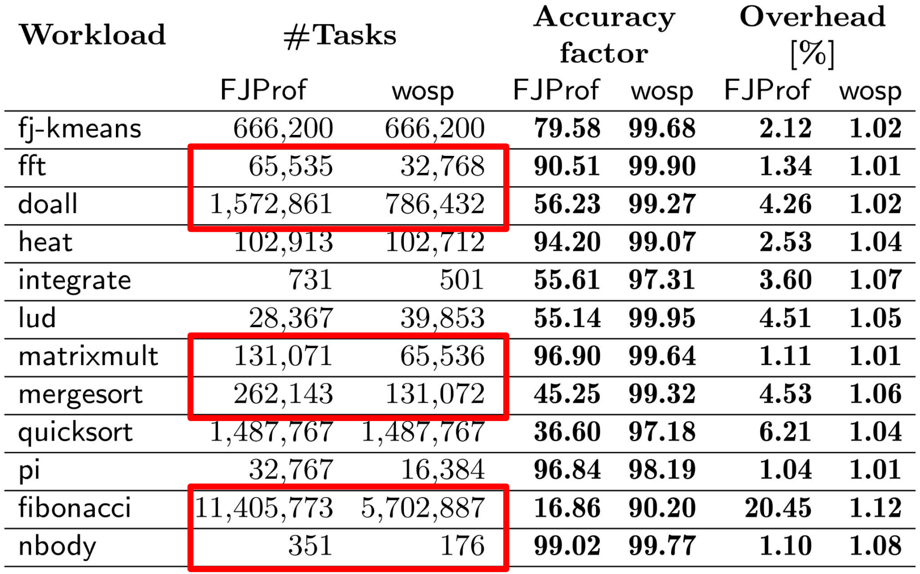
[1] E. Rosales et al., "FJProf: Profiling Fork/Join Applications on the Java Virtual Machine." VALUETOOLS, 2020.

[2] A. Prokopec et al, "Renaissance: Benchmarking Suite for Parallel Applications on the JVM". PLDI, 2019.

[3] A. Fonseca et al, "Evaluation of Runtime Cut-off Approaches for Parallel Programs". VECPAR, 2016.

Evaluation - Number of Tasks

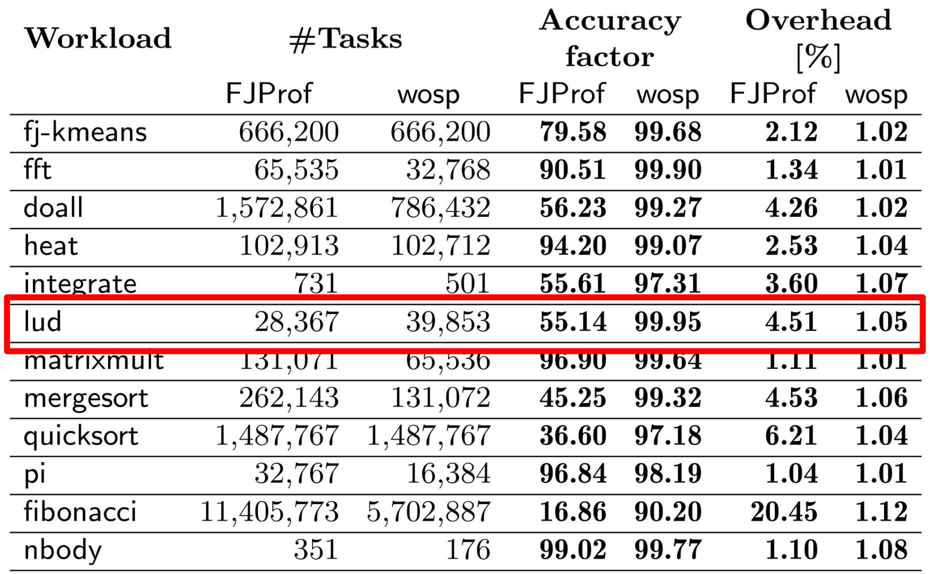
- In many workloads, the number of tasks reported by `FJProf` is twice the one reported by `wosp`
  - Differences in the profiling models
  - In these workloads, tasks split the work into two parts
    - One child task is executed sequentially while the other is forked



Workload	**#Tasks**		**Accuracy factor**		**Overhead [%]**	
	FJProf	wosp	FJProf	wosp	FJProf	wosp
fj-kmeans	666,200	666,200	**79.58**	**99.68**	**2.12**	**1.02**
fft	65,535	32,768	**90.51**	**99.90**	**1.34**	**1.01**
doall	1,572,861	786,432	**56.23**	**99.27**	**4.26**	**1.02**
heat	102,913	102,712	**94.20**	**99.07**	**2.53**	**1.04**
integrate	731	501	**55.61**	**97.31**	**3.60**	**1.07**
lud	28,367	39,853	**55.14**	**99.95**	**4.51**	**1.05**
matrixmult	131,071	65,536	**96.90**	**99.64**	**1.11**	**1.01**
mergesort	262,143	131,072	**45.25**	**99.32**	**4.53**	**1.06**
quicksort	1,487,767	1,487,767	**36.60**	**97.18**	**6.21**	**1.04**
pi	32,767	16,384	**96.84**	**98.19**	**1.04**	**1.01**
fibonacci	11,405,773	5,702,887	**16.86**	**90.20**	**20.45**	**1.12**
nbody	351	176	**99.02**	**99.77**	**1.10**	**1.08**

Evaluation - Number of Tasks

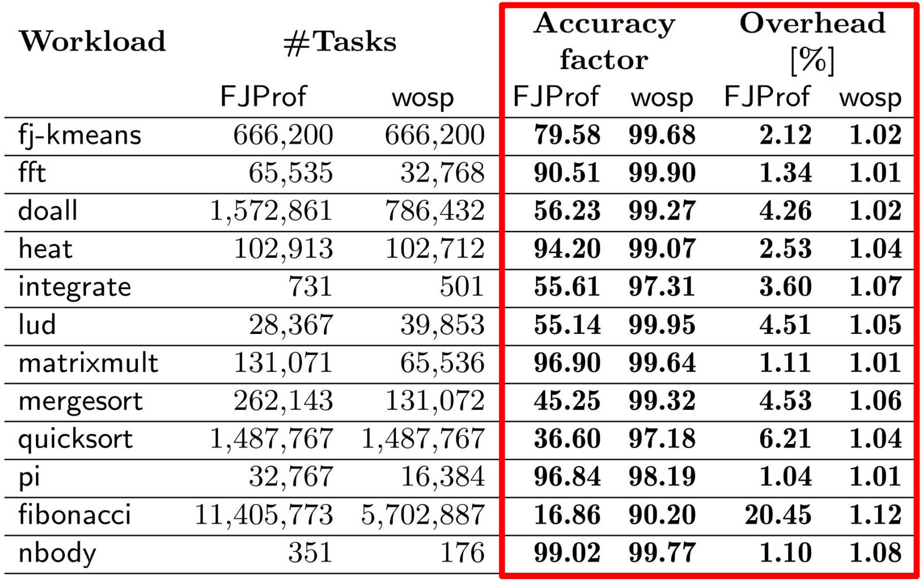
- `lud` is the only workload where `wosp` detects more tasks than `FJProf`
  - The overhead of `FJProf` significantly affects task unforking
  - `ForkJoinTask.tryUnfork` succeeds more frequently as threads are busy executing instrumentation code, instead of actively stealing



Workload	**#Tasks**		**Accuracy factor**		**Overhead [%]**	
	FJProf	wosp	FJProf	wosp	FJProf	wosp
fj-kmeans	666,200	666,200	**79.58**	**99.68**	**2.12**	**1.02**
fft	65,535	32,768	**90.51**	**99.90**	**1.34**	**1.01**
doall	1,572,861	786,432	**56.23**	**99.27**	**4.26**	**1.02**
heat	102,913	102,712	**94.20**	**99.07**	**2.53**	**1.04**
integrate	731	501	**55.61**	**97.31**	**3.60**	**1.07**
lud	28,367	39,853	**55.14**	**99.95**	**4.51**	**1.05**
matrixmult	131,071	65,536	**96.90**	**99.64**	**1.11**	**1.01**
mergesort	262,143	131,072	**45.25**	**99.32**	**4.53**	**1.06**
quicksort	1,487,767	1,487,767	**36.60**	**97.18**	**6.21**	**1.04**
pi	32,767	16,384	**96.84**	**98.19**	**1.04**	**1.01**
fibonacci	11,405,773	5,702,887	**16.86**	**90.20**	**20.45**	**1.12**
nbody	351	176	**99.02**	**99.77**	**1.10**	**1.08**

Evaluation - Accuracy and Overhead

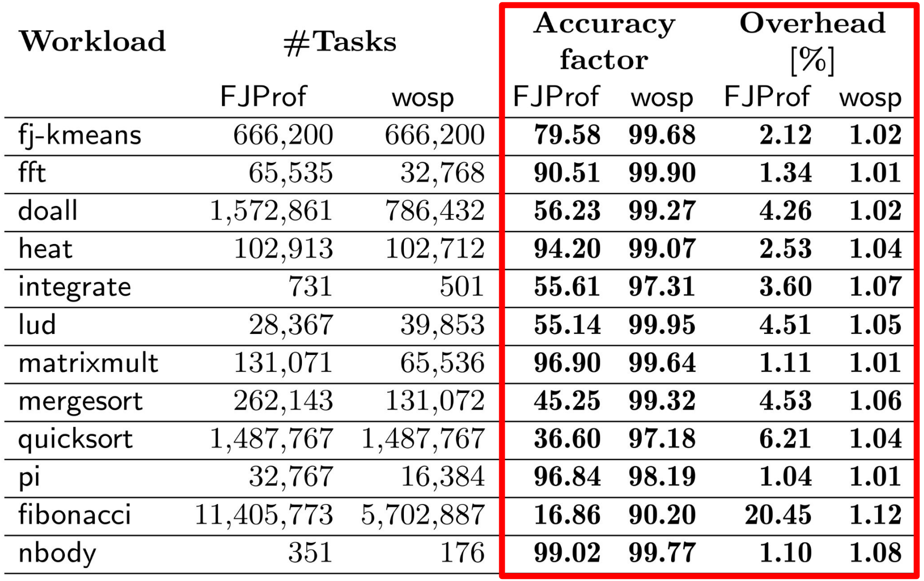
- `wosp` always achieves both a higher accuracy and lower overhead than `FJProf`
- The lowest accuracy and the highest overhead are experienced while profiling `fibonacci`



Workload	**#Tasks**		**Accuracy factor**		**Overhead [%]**	
	FJProf	wosp	FJProf	wosp	FJProf	wosp
fj-kmeans	666,200	666,200	**79.58**	**99.68**	**2.12**	**1.02**
fft	65,535	32,768	**90.51**	**99.90**	**1.34**	**1.01**
doall	1,572,861	786,432	**56.23**	**99.27**	**4.26**	**1.02**
heat	102,913	102,712	**94.20**	**99.07**	**2.53**	**1.04**
integrate	731	501	**55.61**	**97.31**	**3.60**	**1.07**
lud	28,367	39,853	**55.14**	**99.95**	**4.51**	**1.05**
matrixmult	131,071	65,536	**96.90**	**99.64**	**1.11**	**1.01**
mergesort	262,143	131,072	**45.25**	**99.32**	**4.53**	**1.06**
quicksort	1,487,767	1,487,767	**36.60**	**97.18**	**6.21**	**1.04**
pi	32,767	16,384	**96.84**	**98.19**	**1.04**	**1.01**
fibonacci	11,405,773	5,702,887	**16.86**	**90.20**	**20.45**	**1.12**
nbody	351	176	**99.02**	**99.77**	**1.10**	**1.08**

Evaluation - Accuracy and Overhead

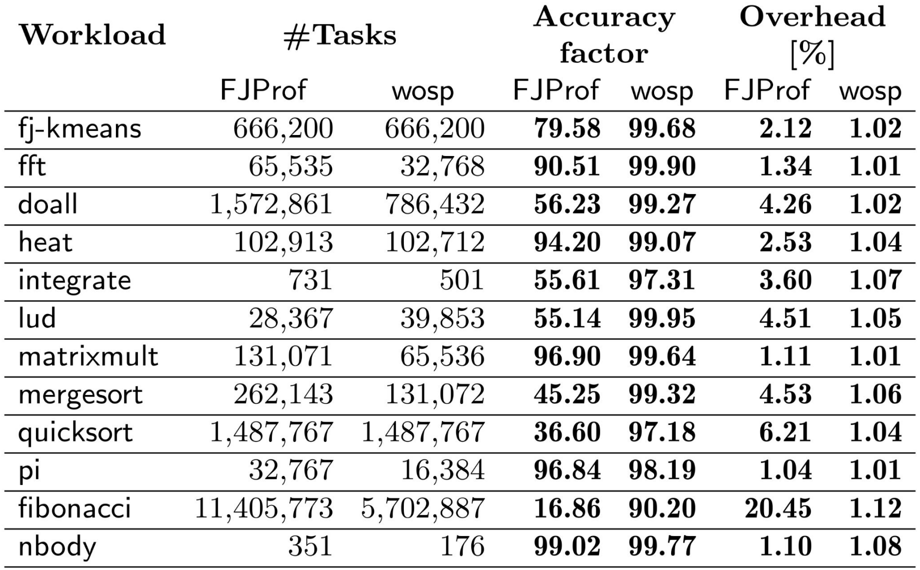
- General trend: the higher the number of tasks, the higher the overhead
- Exception: `integrate` and `lud` have relatively high overhead even if they use few tasks
  - Reason: task unforking succeeds frequently and tasks are not executed using the exec method



Workload	**#Tasks**		**Accuracy factor**		**Overhead [%]**	
	FJProf	wosp	FJProf	wosp	FJProf	wosp
fj-kmeans	666,200	666,200	**79.58**	**99.68**	**2.12**	**1.02**
fft	65,535	32,768	**90.51**	**99.90**	**1.34**	**1.01**
doall	1,572,861	786,432	**56.23**	**99.27**	**4.26**	**1.02**
heat	102,913	102,712	**94.20**	**99.07**	**2.53**	**1.04**
integrate	731	501	**55.61**	**97.31**	**3.60**	**1.07**
lud	28,367	39,853	**55.14**	**99.95**	**4.51**	**1.05**
matrixmult	131,071	65,536	**96.90**	**99.64**	**1.11**	**1.01**
mergesort	262,143	131,072	**45.25**	**99.32**	**4.53**	**1.06**
quicksort	1,487,767	1,487,767	**36.60**	**97.18**	**6.21**	**1.04**
pi	32,767	16,384	**96.84**	**98.19**	**1.04**	**1.01**
fibonacci	11,405,773	5,702,887	**16.86**	**90.20**	**20.45**	**1.12**
nbody	351	176	**99.02**	**99.77**	**1.10**	**1.08**

Evaluation - Accuracy and Overhead

- Average accuracy
  - `wosp`: 98.25%
  - `FJProf`: 61.69%
- Average overhead factor
  - `wosp`: 1.04×
  - `FJProf`: 2.91×



Workload	**#Tasks**		**Accuracy factor**		**Overhead [%]**	
	FJProf	wosp	FJProf	wosp	FJProf	wosp
fj-kmeans	666,200	666,200	**79.58**	**99.68**	**2.12**	**1.02**
fft	65,535	32,768	**90.51**	**99.90**	**1.34**	**1.01**
doall	1,572,861	786,432	**56.23**	**99.27**	**4.26**	**1.02**
heat	102,913	102,712	**94.20**	**99.07**	**2.53**	**1.04**
integrate	731	501	**55.61**	**97.31**	**3.60**	**1.07**
lud	28,367	39,853	**55.14**	**99.95**	**4.51**	**1.05**
matrixmult	131,071	65,536	**96.90**	**99.64**	**1.11**	**1.01**
mergesort	262,143	131,072	**45.25**	**99.32**	**4.53**	**1.06**
quicksort	1,487,767	1,487,767	**36.60**	**97.18**	**6.21**	**1.04**
pi	32,767	16,384	**96.84**	**98.19**	**1.04**	**1.01**
fibonacci	11,405,773	5,702,887	**16.86**	**90.20**	**20.45**	**1.12**
nbody	351	176	**99.02**	**99.77**	**1.10**	**1.08**

Conclusions

- We presented a novel profiling model for fork-join computations on the JVM
  - Our model allows accurately profiling several specific fork-join metrics, while supporting the advanced features of the Java fork-join framework
- We presented `wosp`, a profiler implementing our model
- We showed that `wosp` achieves a notably higher accuracy than `FJProf`, while incurring much less overhead
- Our model helps in understanding performance and behaviour of fork-join applications

Future Work

- Conduct a large-scale characterization of Java fork-join applications
- Develop a visualization tool

Thanks for your attention

- Contacts:

  Matteo Basso

  matteo.basso@usi.ch