Performance Analysis for Identification of (Sub)task-Level Parallelism in Java

R.Stahl, R.Pasko, L.Rijnders, D.Verkest, S.Vernalde,
R.Lauwereins and F.Catthoor

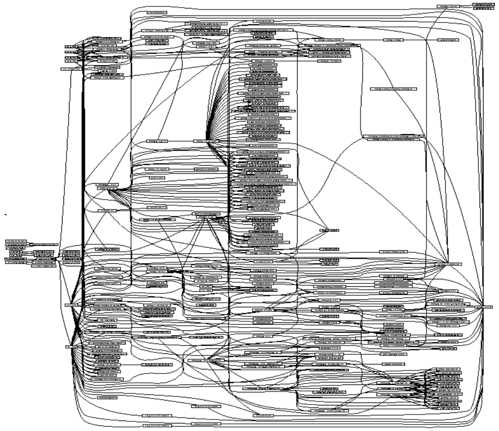
IMEC

Leuven, Belgium

Performance Analysis: need for embedded system program optimisation

original program

(single-threaded)



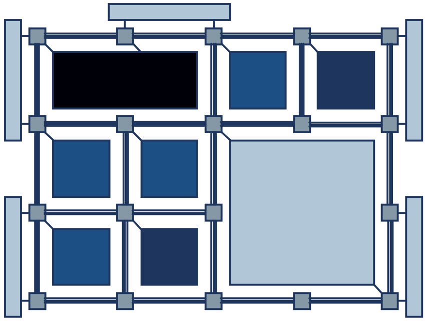
?

MAPPING

(optimal)

multiprocessor platform

(heterogeneous)



Outline

- ➢ **Introduction**
- ❑ Parallel Performance Analysis (PPA)
  - ❑ Pre-processing
  - ❑ Profiler
  - ❑ Post-processing
- ❑ Results & Conclusions

We do task-level parallelism extraction from object-oriented programs

= high-level platform-independent transformations

“high-level”

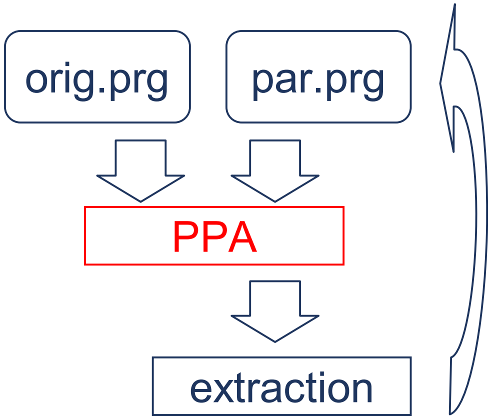
- looking at the high-level program structures e.g. classes, methods

“platform-independent”

- positive effect for multiprocessor systems in general

thus, we have to:

- **identify dominant parts of the program**
- extract task-level parallelism
- **evaluate the transformation effect**



Performance analysis requirements

program perspective:

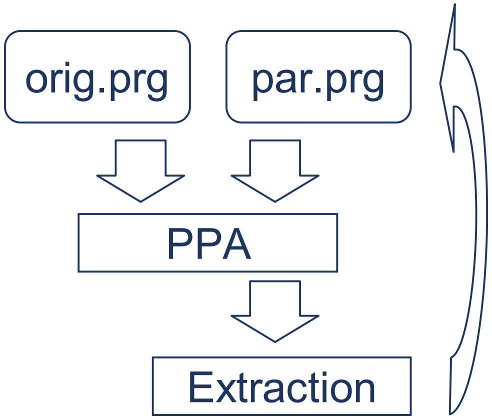
- ❑ **same environment** for original and transformed programs
  (to take equal measure for both)

platform perspective:

- ❑ **exposing the parallel behaviour**
  (to evaluate the optimisation effects)

designer's perspective:

- ❑ as fast as possible
  (minimal run-time overhead)
- ❑ running on any platform
  (most preferably on my computer)
- ❑ easy to use



Concept of virtual time

program:

□ **executing in one environment**

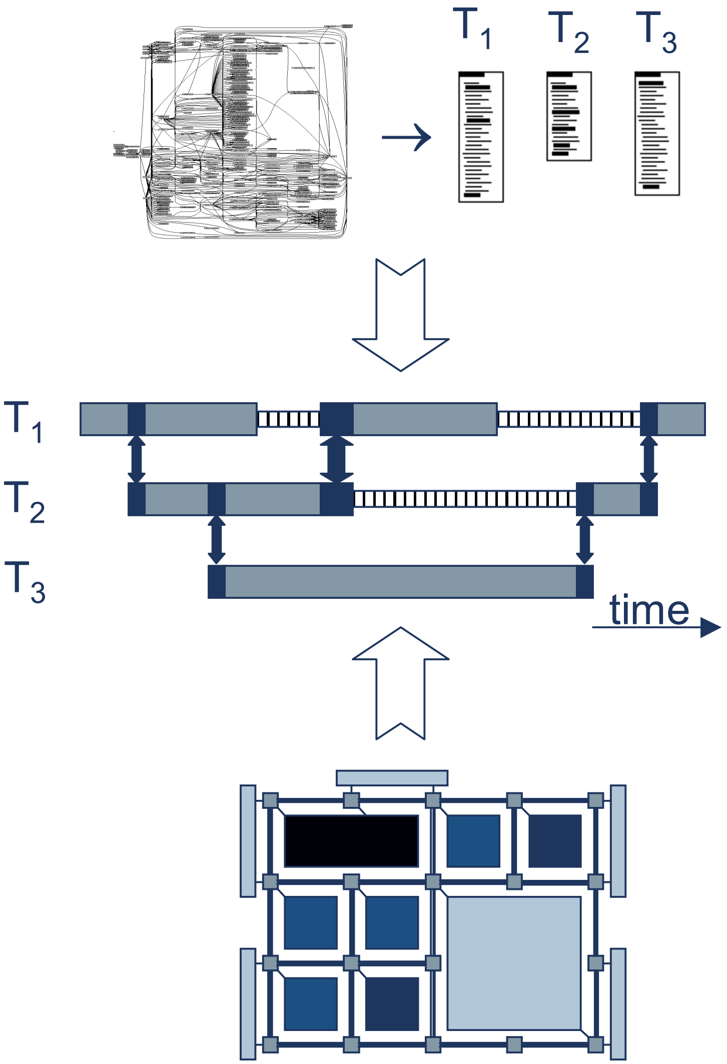
□ *with minimal run-time overhead*

virtual time
(virtually parallel execution)

platform:

□ **simulating parallel behaviour**

□ *on any platform*

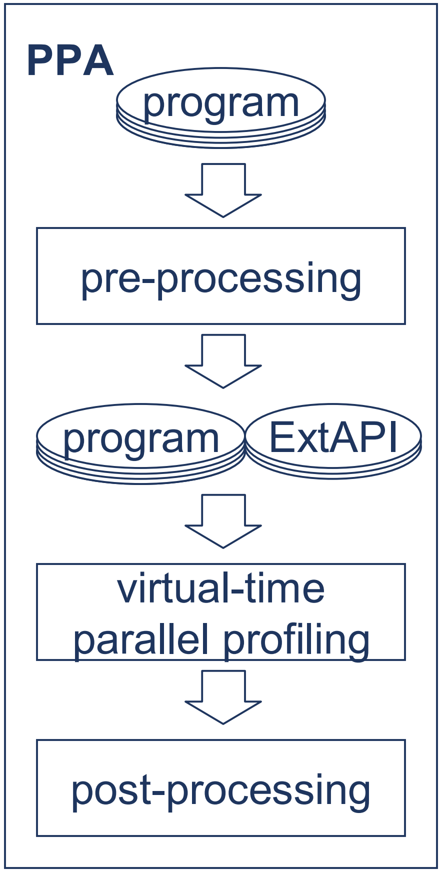


Outline

- ❑Introduction
- ➢**Parallel Performance Analysis (PPA)**
    - ❑ Pre-processing
    - ❑ Profiler
    - ❑ Post-processing
- ❑Results & Conclusions

Parallel Performance Analysis

- ❑pre-processing
  - ❑user-controlled instrumentation
  - ❑program transformation for profiler
- ❑parallel profiler
  - ❑implements the run-time support for the concept of virtual time
    - ➢executing program
    - ➢simulating parallel behaviour
- ❑post-processing
  - ❑critical-path analysis
  - ❑feedback for the parallelism extraction



Pre-processing enables selective profiling based on user interest

user indicates the important parts of the program

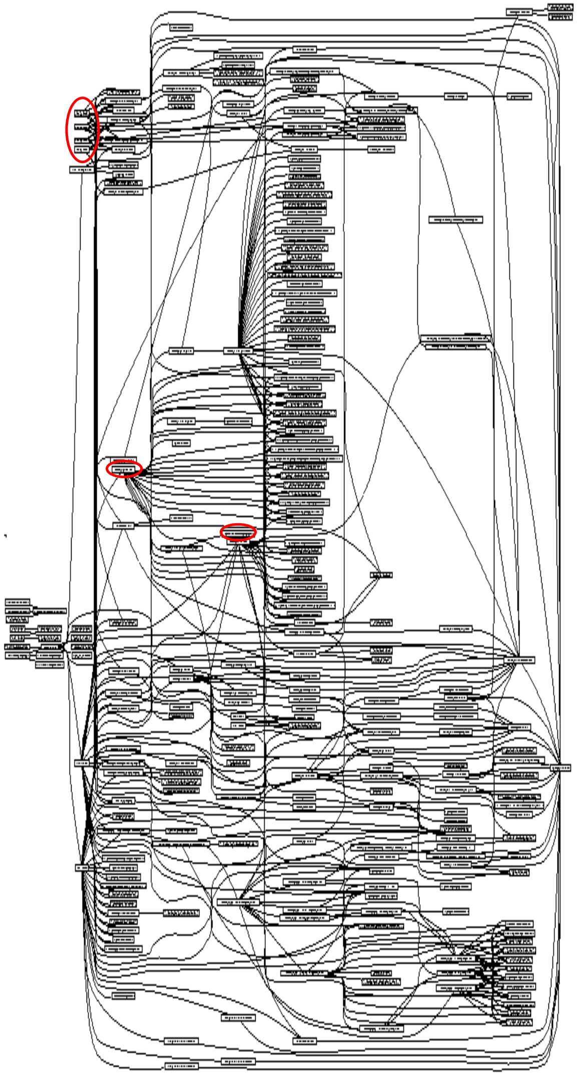
1. top-level methods and loops accumulating most of the computation
2. looking inside in more detail

instrumentation:

inserts profiler-specific code to reflect user's interest

profiling modes:

- ❑ full profile for n top levels in the call graph
- ❑ selective profile
- ❑ sub-graph (branch) profile
- ❑ cumulative vs. non-cumulative method profile



Pre-processing adapts program for the parallel profiler

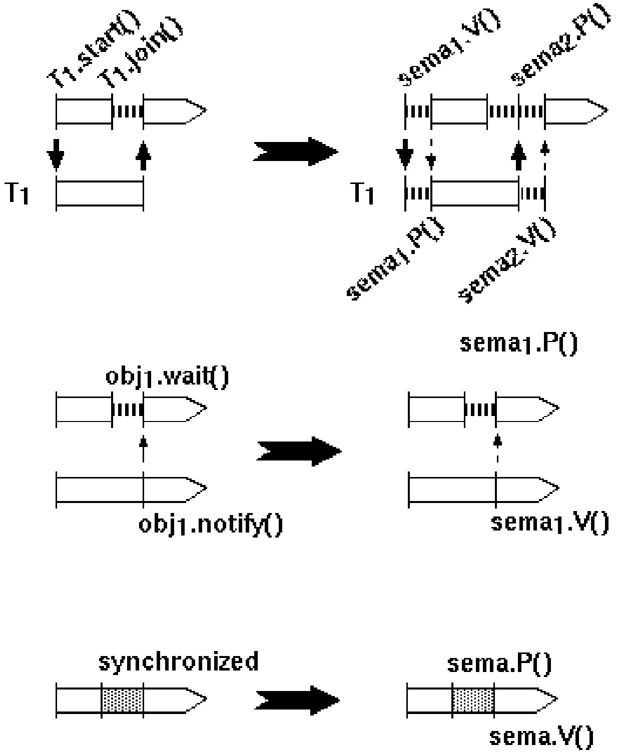
virtual time is based on passing time stamps between tasks of the program

two possible situations:

- ❑task-creation
  - ❑ sending time stamp from parent to child
- ❑task-synchronisation
  - ❑ updating the stamp between synced peers

solution = transformation of the Java synchronisation primitives into profiler-specific one (binary semaphore)

- ❑reducing extentions to run-time system
- ❑reducing run-time overhead



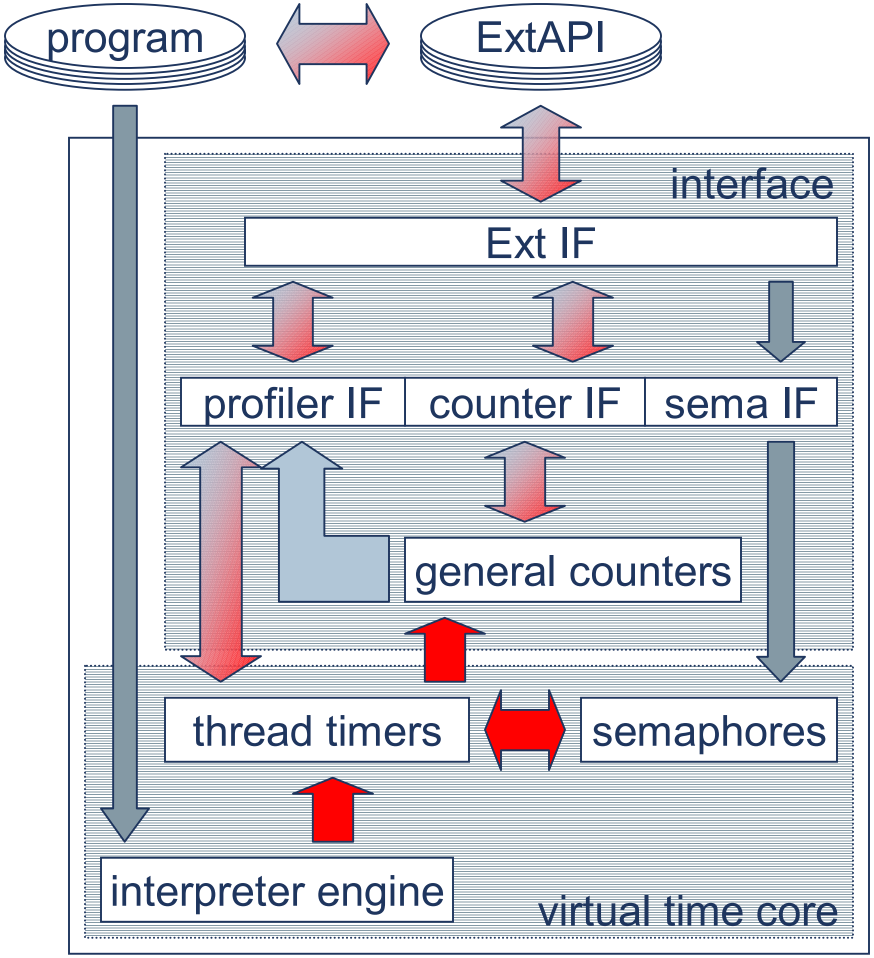
Parallel profiler implements run-time support for the virtual time concept

virtual time core

- ❑ passing appropriate time stamps between cooperating threads

interface

- ❑ providing control over the parallel profiler
- ❑ passing information between the program and the profiler



■ program control

🟥 changing time information

🟦 passing time information

Profiler core – minimal functionality enabling parallel profiling

interpreter

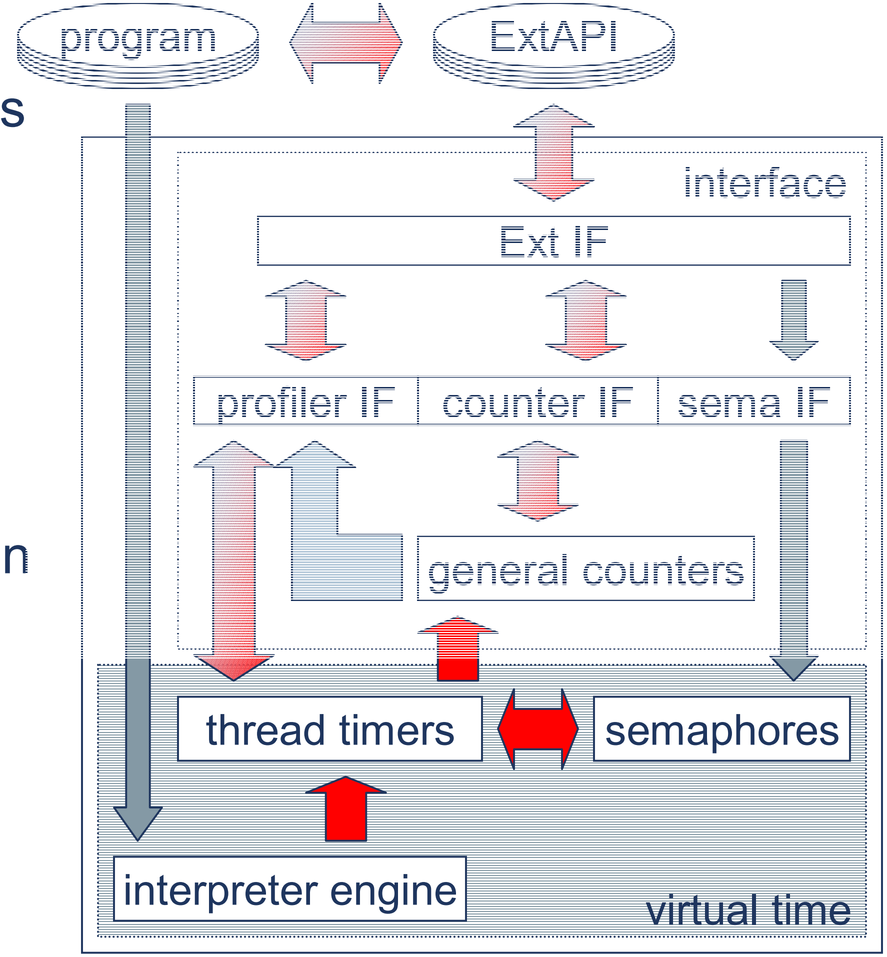
- ❑Java interpreter
- ❑extended to enable operations on thread timers
- ❑having configurable time unit for different processors

thread timers

- ❑single timer per thread
- ❑storing proper time information in the proper thread timer

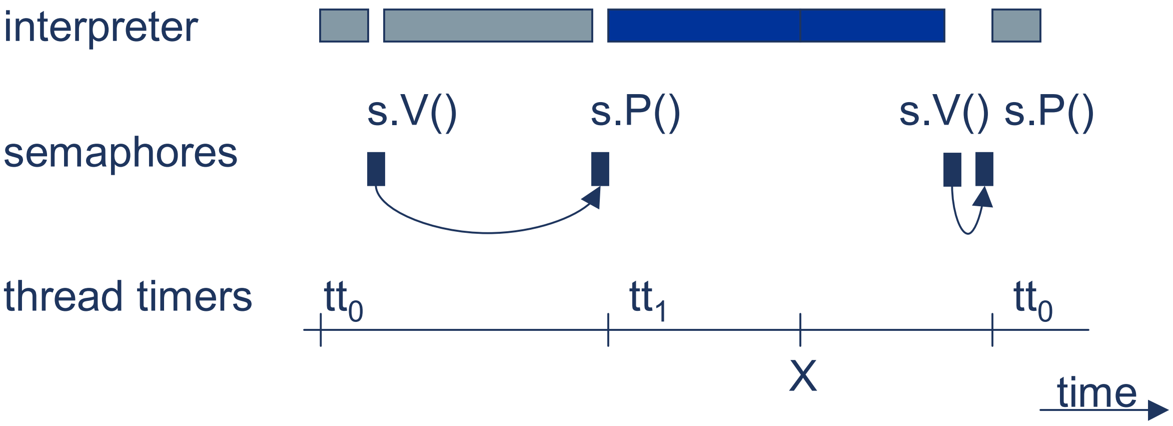
semaphores

- ❑the only way to pass time information between different thread timers

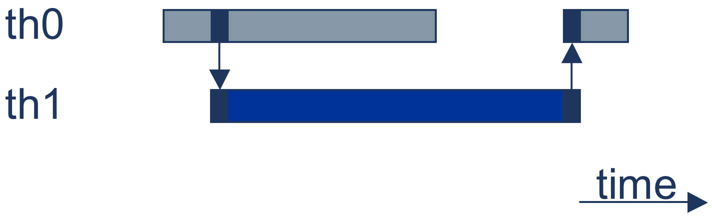


How it works ...

real execution sequence



virtual execution sequence



Extension interface enables full run-time control over the profiler

general purpose counters

init, inc/dec, set/rst, get

- ❑typical usage:
  - ❑per method timer
  - ❑method call counter

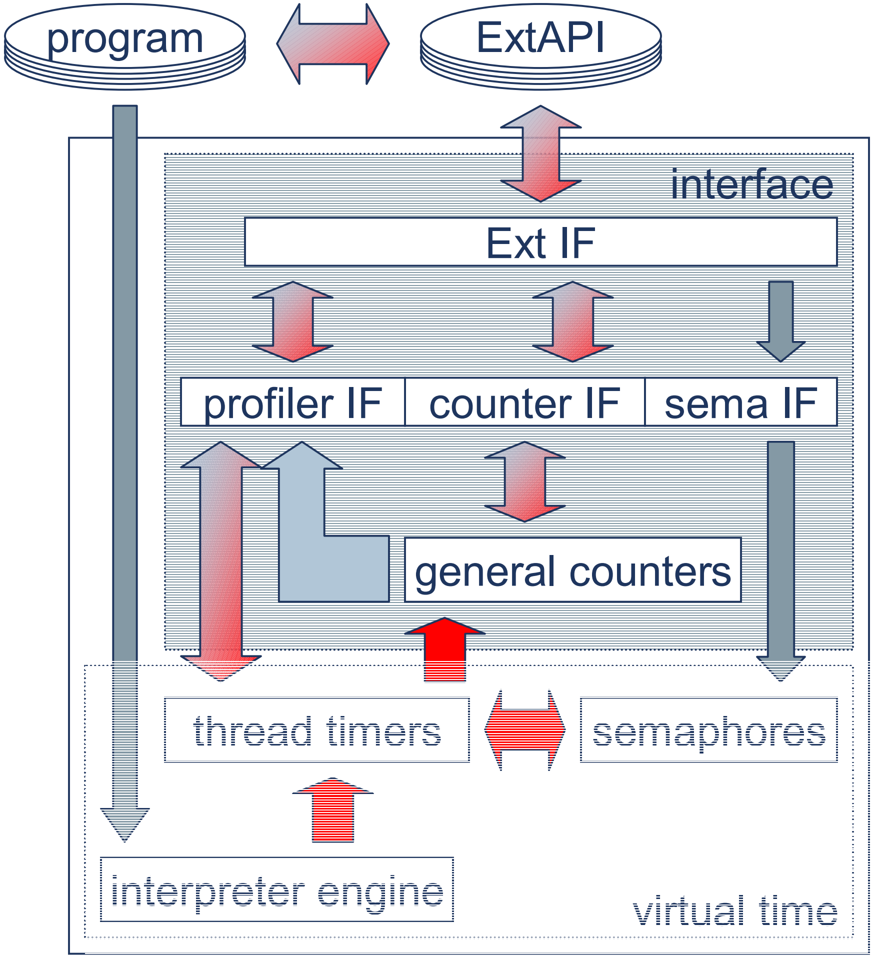
semaphore interface

init(state), P(), V()

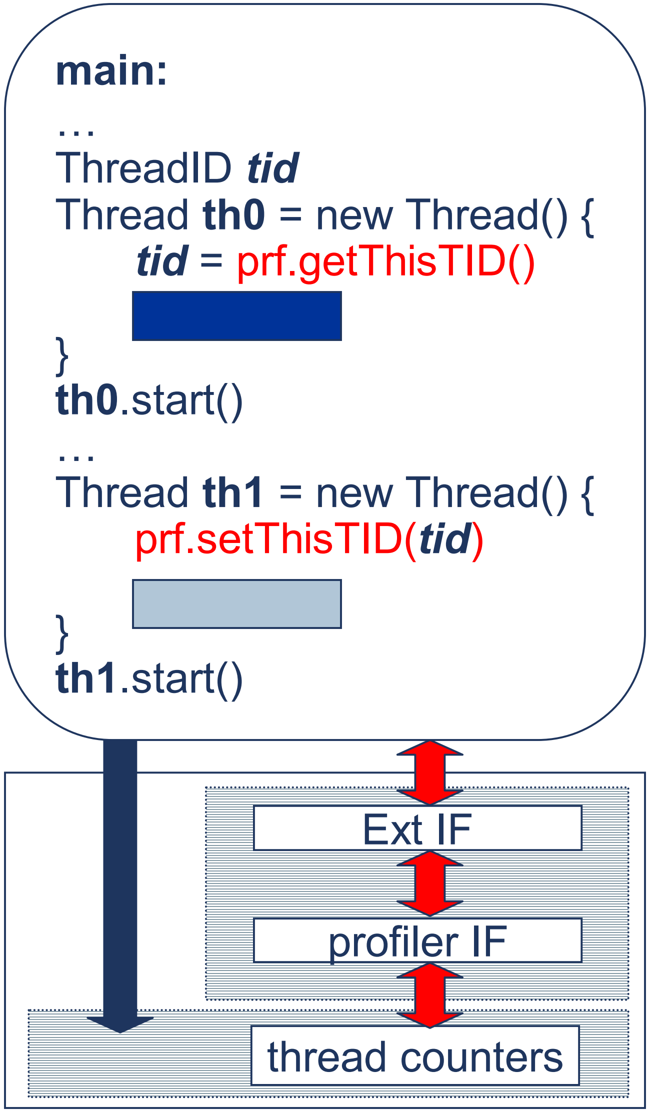
profiler interface

reification

- ❑statistical information
- ❑configuring profiling mode

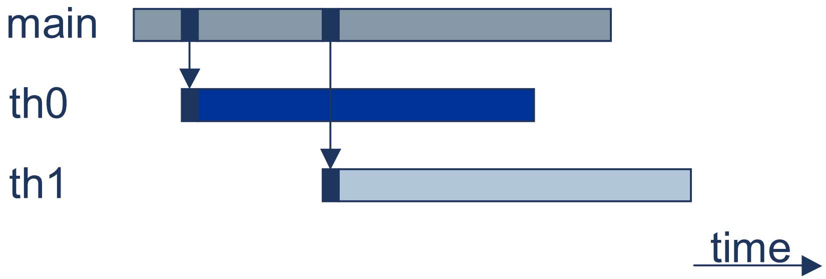


Example: simulating fixed number of processors via profiler interface

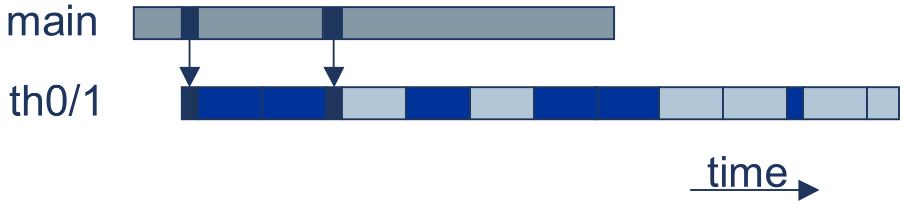


```
main:
…
ThreadID tid
Thread th0 = new Thread() {
      tid = prf.getThisTID()

}
th0.start()
…
Thread th1 = new Thread() {
      prf.setThisTID(tid)

}
th1.start()
```

non-shared tid = unlimited no. of processors



shared tid = fixed number of processors



notes:

- ❑th0 and th1 share the same thread counter
- ❑threads are schedules by the JVM scheduler

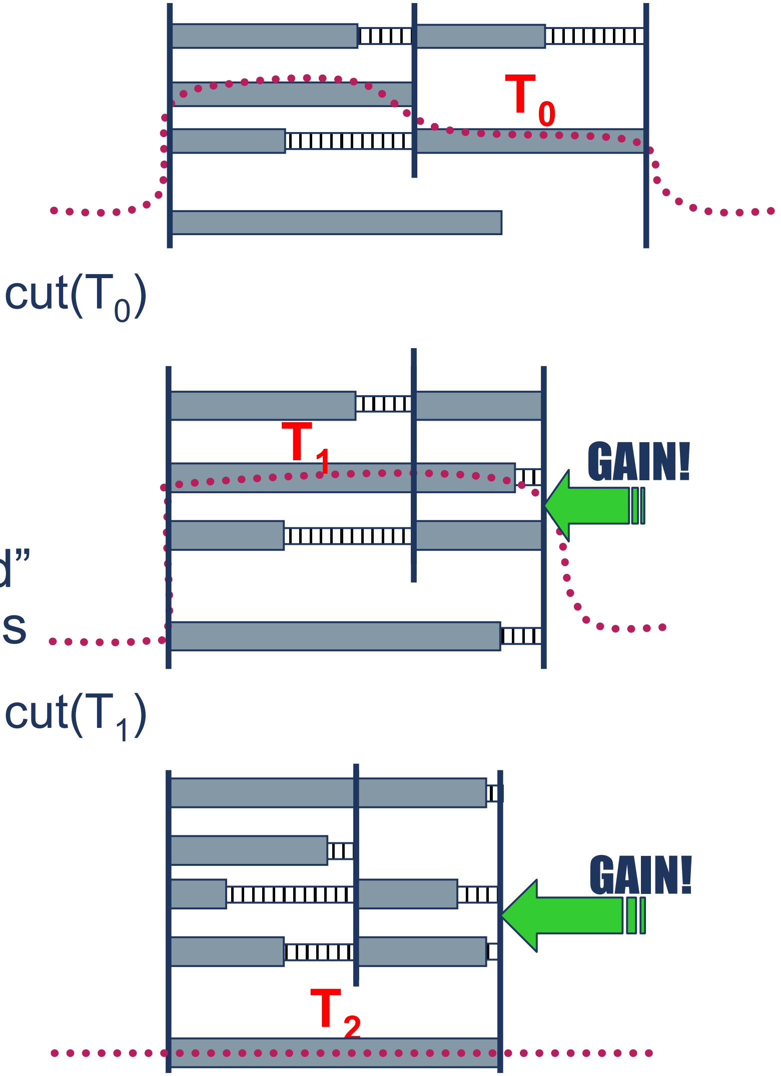
Post-processing analysis indicates the potential to improve

critical-path analysis

the most critical part is the one where reduction in its execution time has the highest impact on overall execution time of the program

task balance

the ideal partitioning creates parallel sub-tasks with “balanced” execution time, i.e., their idleness is minimised



Experimental results

	speedup	# th	idle [%]	T_{ins} [s]	notes
MPEG player	~ 2.3	5	20	30	imperative, data-dominant, static
3D engine v1	~ 4.1	8	23	31	OOD, modular, interactive
3D engine v2	~ 4.6	18	36	31	
javac v1	1.1 – 1.2	7 - 12	0	210	OOD, recursion, complex
javac v2	1.4 – 1.9	21 - 32	25 - 34	210	
javac v3	1.8 – 2.3	21 - 32	21 - 32	210	

Conclusions

- ❑parallel performance analysis framework for task-level parallelism extraction
- ❑concept of virtual time simulating parallel behaviour of multithreaded programs
- ❑common execution environment for original and transformed programs
- ❑run-time overhead < 3%

Future:

- ❑data-access analysis

Thank you