Optimization Coaching for Fork/Join Applications on the Java Virtual Machine

Eduardo Rosales

Advisor: Prof. Walter Binder

Research area: Parallel applications, performance analysis

PhD stage: Planner

EuroDW 2018
April 23, 2018
Porto, Portugal

Università della Svizzera italiana

Optimization Coaching for Fork/Join Applications on the Java Virtual Machine

- **The problem:** despite the complexities associated with developing and tuning fork/join applications, *there is little work focused on assisting developers in optimizing such applications on the JVM*.

- **Relevance:** fork/join parallelism has an increasing popularity among developers targeting the JVM. It has been integrated to support parallel processing on the Java library, thread management in JVM languages and a variety of parallel applications based on Actors, MapReduce, etc.

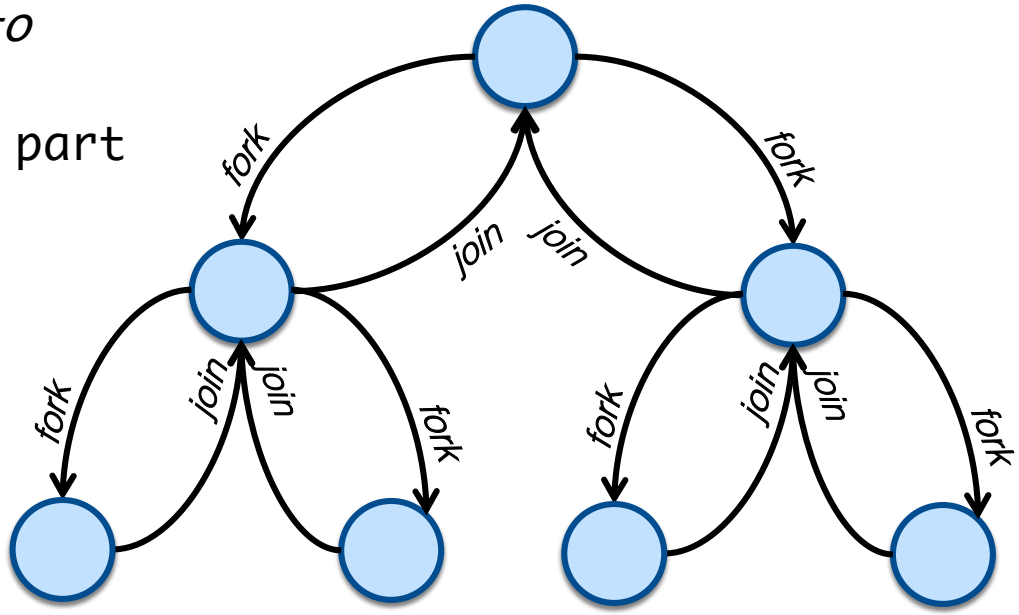
- **Our proposal:** coaching developers towards optimizing fork/join applications by diagnosing performance issues on such applications and further suggest concrete code refactoring to solve them.

- **Expected outcome:** in contrast to the manual experimentation often required to tune fork/join applications on the JVM, we devise a tool able to automatically assist developers in optimizing a fork/join application.

Fork/join Application

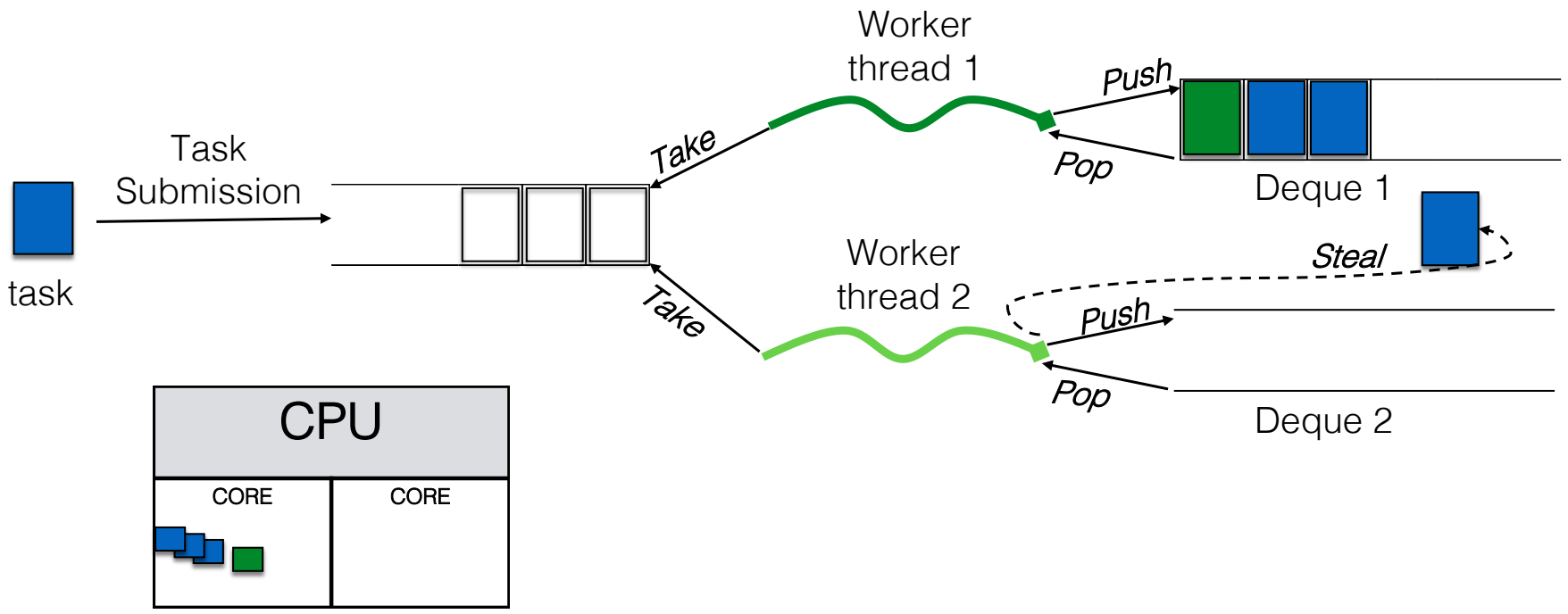
- What is a fork/join application?

```
solve(Problem problem) {
  if (problem is small)
    directly solve problem sequentially
  else {
    recursively split problem into
    independent parts:
    fork new tasks to solve each part
    join all forked tasks
  }
}
```



The Java Fork/Join Framework

- **The Java fork/join framework** [1] is the implementation enabling fork/join applications on the JVM
  - It implements the *work-stealing* [2] scheduling strategy:

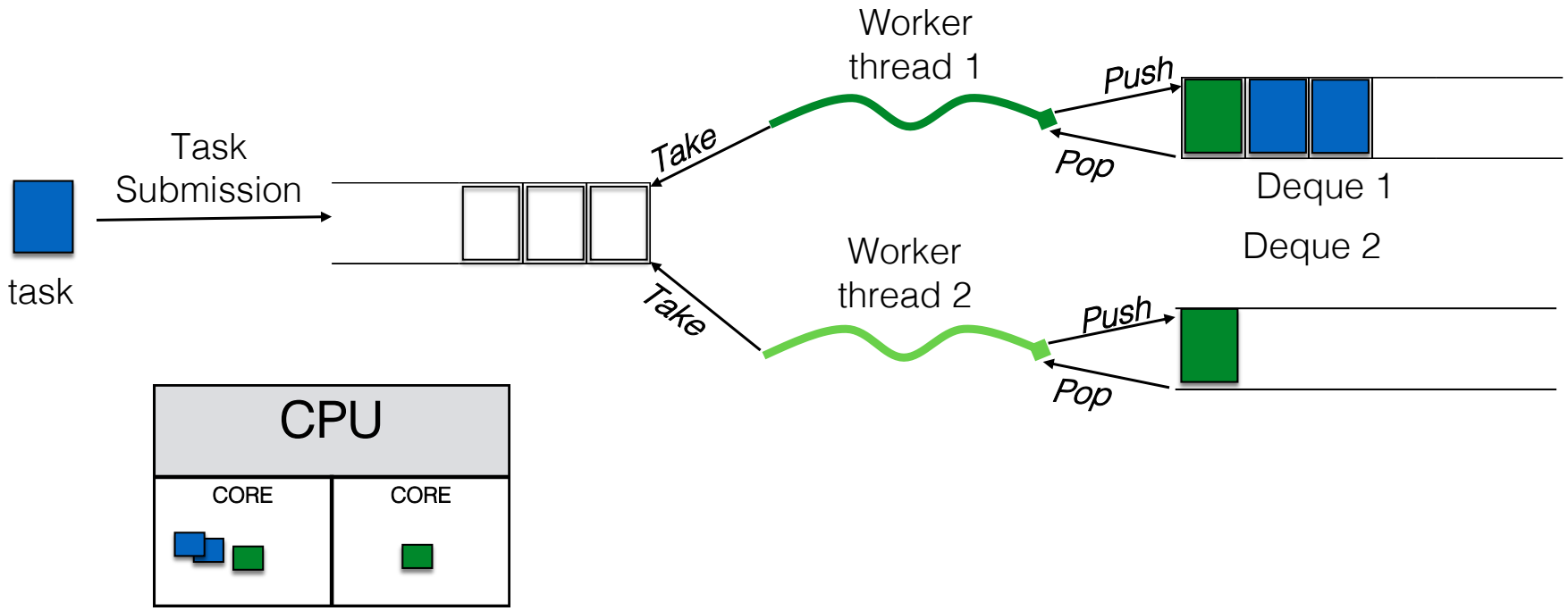


[1] D. Lea. *A Java Fork/Join Framework*. JAVA 2000.

[2] Burton et al. *Executing Functional Programs on a Virtual Tree of Processors*. FPCA 1981.

The Java Fork/Join Framework

- **The Java fork/join framework** [1] is the implementation enabling fork/join applications on the JVM
  - It implements the *work-stealing* [2] scheduling strategy:



[1] D. Lea. *A Java Fork/Join Framework*. JAVA 2000.

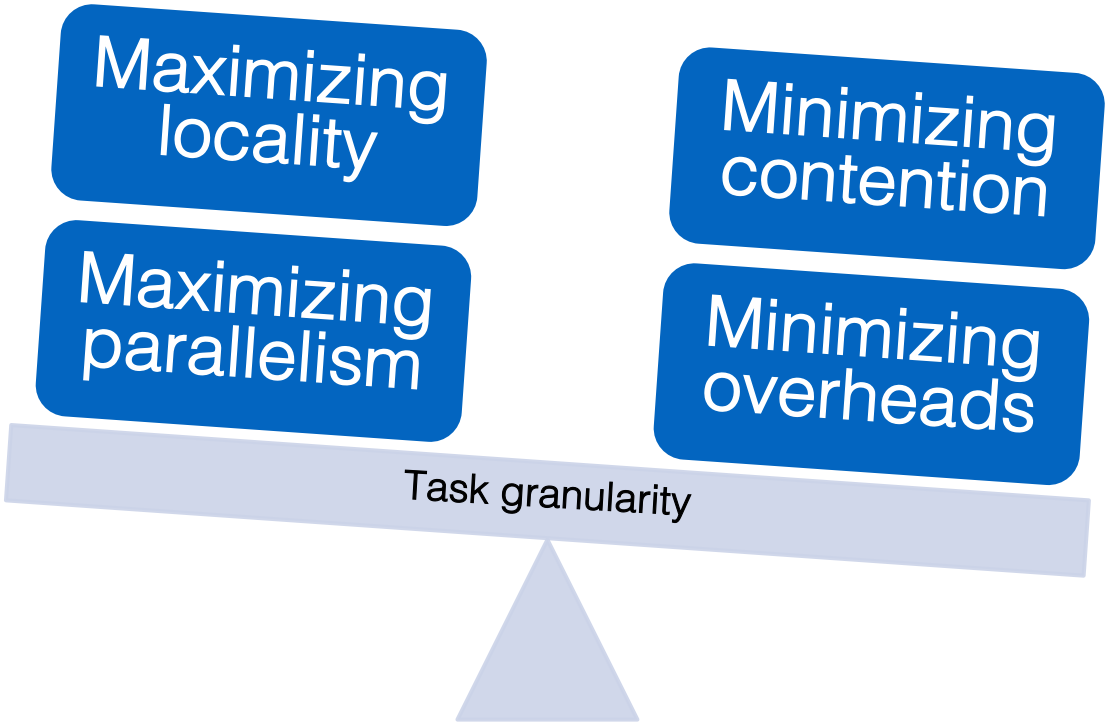
[2] Burton et al. *Executing Functional Programs on a Virtual Tree of Processors*. FPCA 1981.

The Java Fork/Join Framework

- Supports parallel processing in the Java library:
  - `java.util.Array`
  - `java.util.streams` (package)
  - `java.util.concurrent.CompletableFuture<T>`
- Supports thread management for other JVM languages:
  - *Scala*
  - *Apache Groovy*
  - *Clojure*
- Supports diverse fork/join parallelism, including applications based on *Actors* and *MapReduce*

The Java Fork/Join Framework

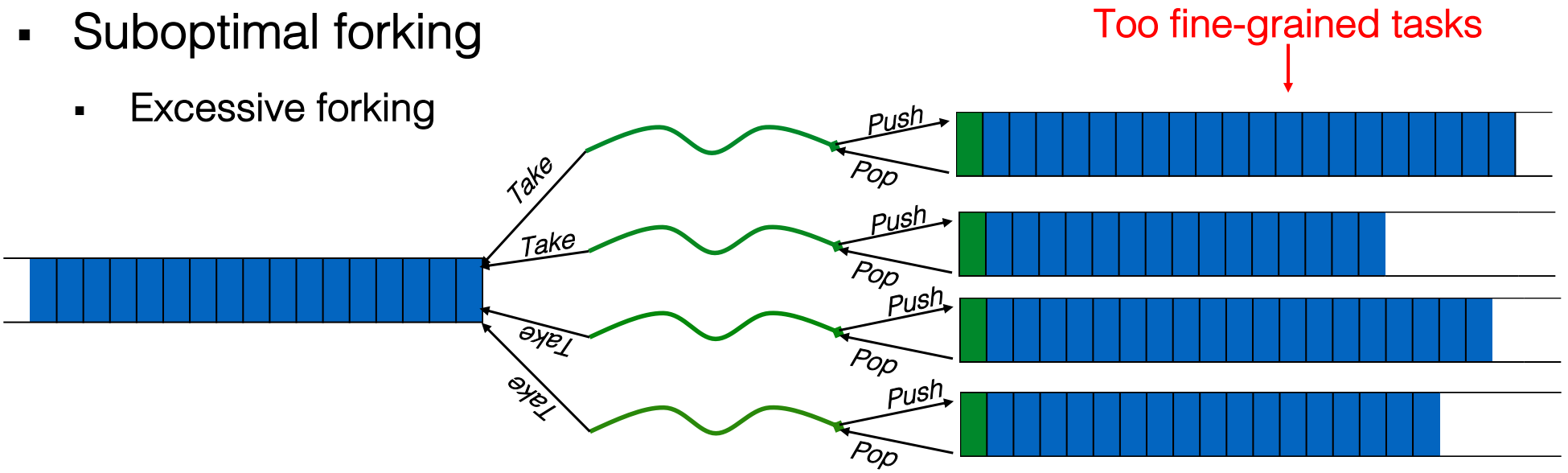
- *Many of the design forces encountered when implementing fork/join designs surround task granularity at four levels* [3]*:*

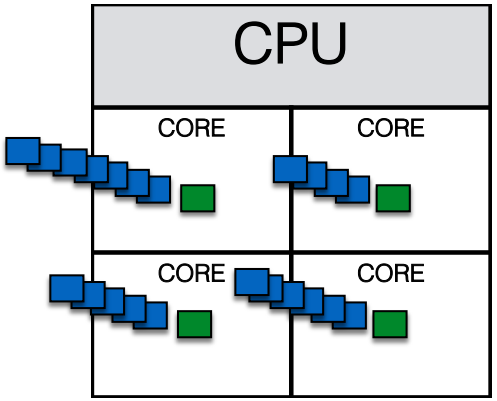


[3] D. Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Professional, 2nd edition, 1999.

Example of a common performance issues 1/4

- Suboptimal forking
  - Excessive forking



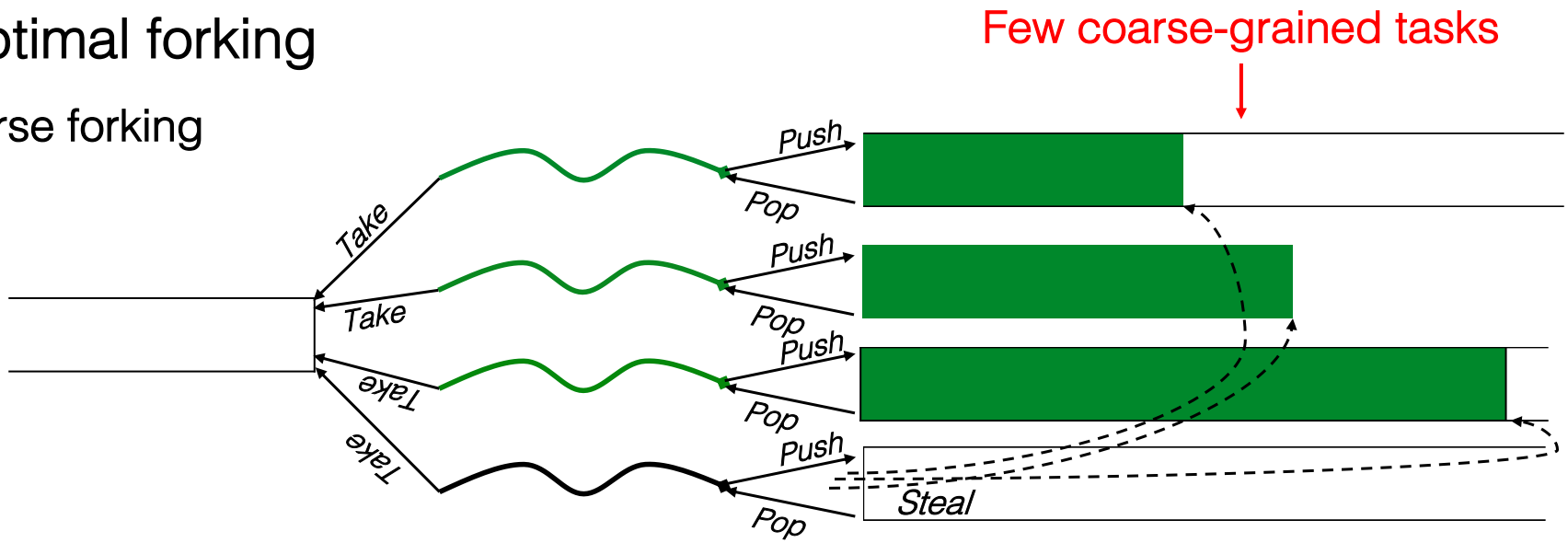


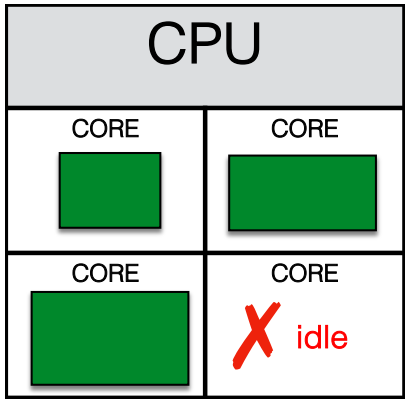
✗ Parallelization overheads due to excessive:

- Deque accesses
- Object creation/reclaiming

Example of a common performance issues 2/4

- Suboptimal forking
  - Sparse forking





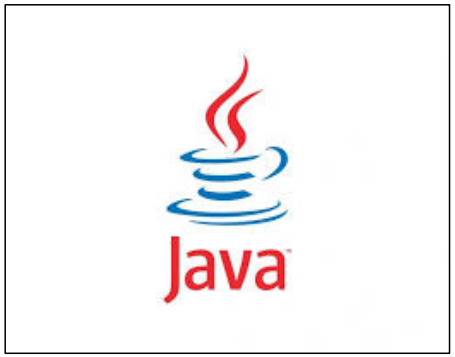
✗ Missed parallelization opportunities:

- Low CPU utilization
- Load imbalance

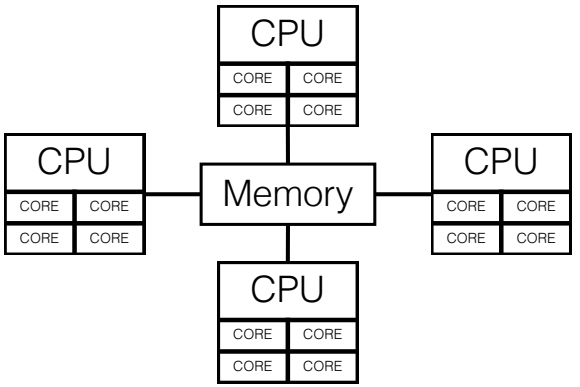
The problem

Despite the complexities associated with developing and tuning fork/join applications, *there is little work focused on assisting developers towards optimizing such applications on the JVM*.

The scope:



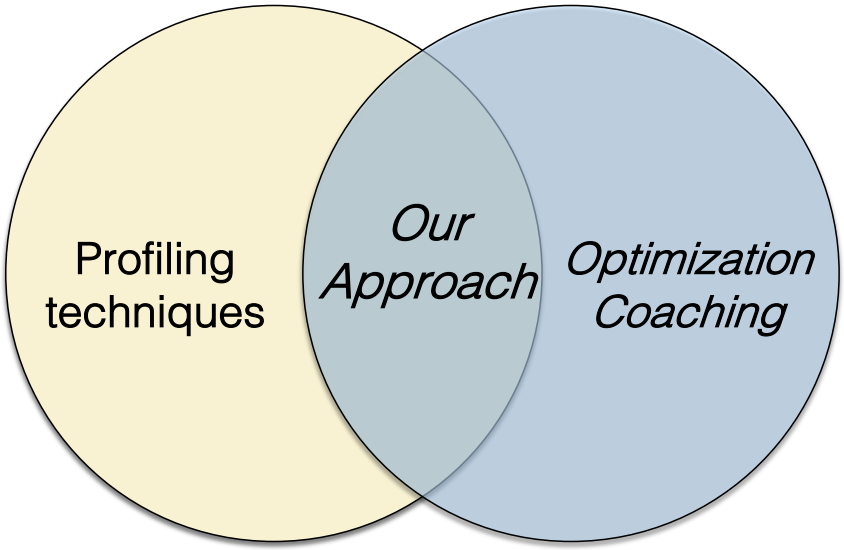
Fork/join applications running in a single JVM



A single shared-memory multicore

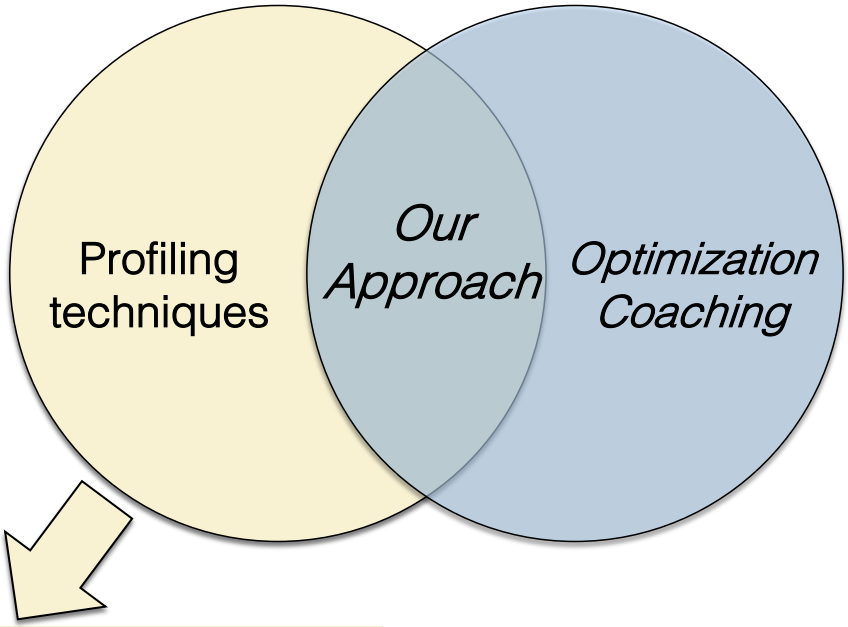
Our Approach

In contrast to manual experimentation used to tune a fork/join application, we propose an approach based on:



Our Approach

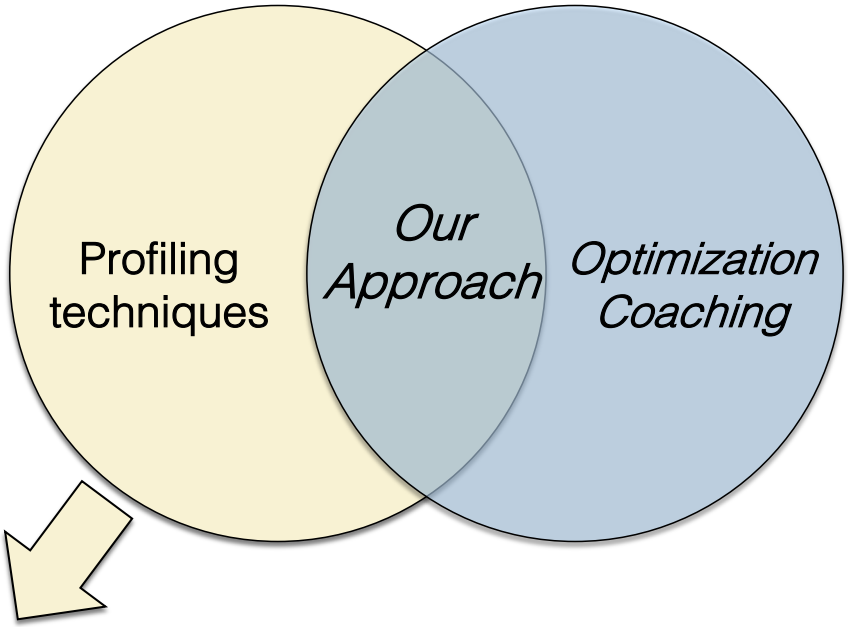
In contrast to manual experimentation often used to tune a fork/join application, we propose an approach based on:



Static and dynamic analysis to **automatically diagnose performance issues**

Our Approach

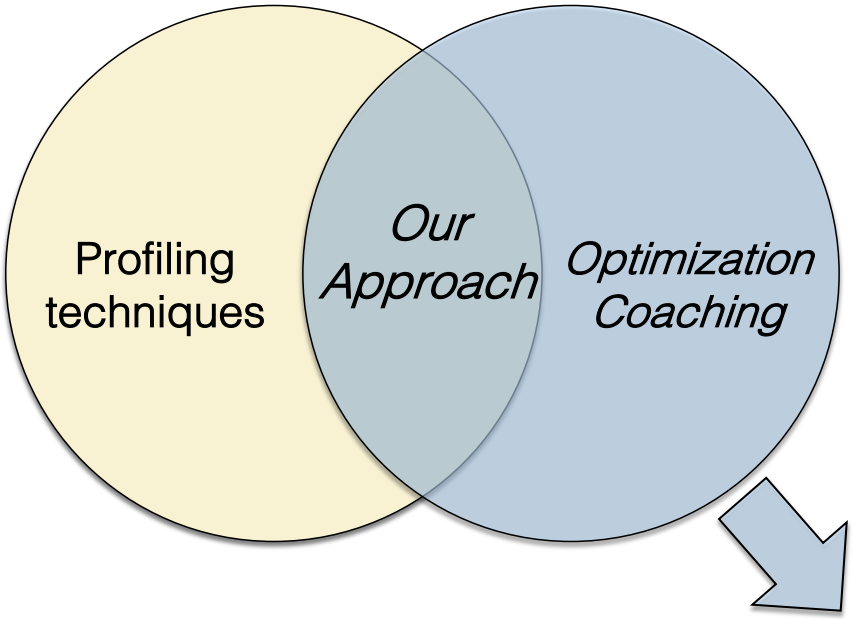
In contrast to manual experimentation often used to tune a fork/join application, we propose an approach based on:



- **Static analysis:** to automatically inspect the source code to detect fork/join anti patterns.
- **Dynamic analysis:** to automatically diagnose performance issues noticeable at runtime (e.g., suboptimal forking, excessive garbage collection, low CPU usage, contention).

Our Approach

In contrast to manual experimentation often used to tune a fork/join application, we propose an approach based on:

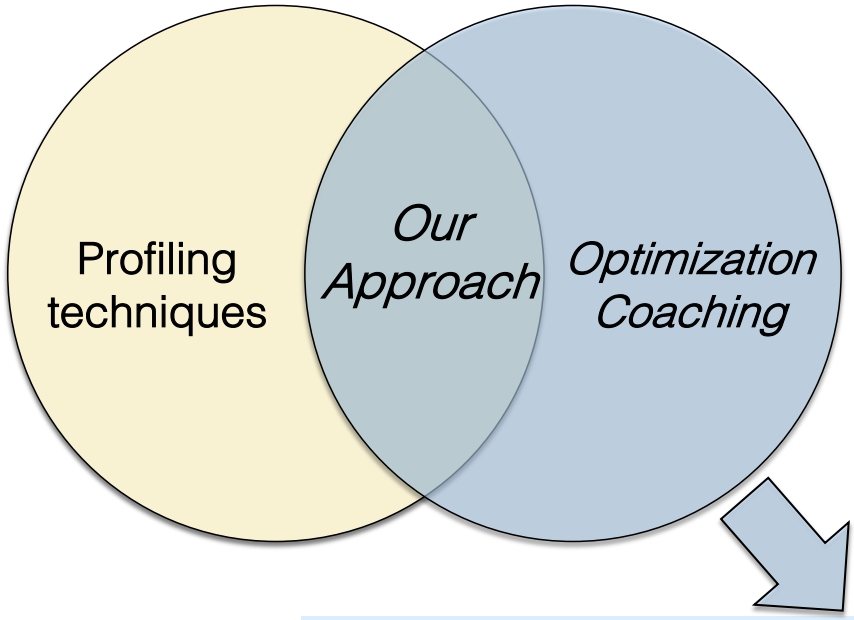


Optimization coaching [4]: processing the output generated by the compiler's optimizer to suggest concrete code modifications that may enable the compiler to achieve missed optimizations.

[4] St-Amour et al. *Optimization Coaching: Optimizers Learn to Communicate with Programmers*. OOPSLA 2012.

Our Approach

In contrast to manual experimentation often used to tune a fork/join application, we propose an approach based on:



Inspired by Optimization Coaching the goal is **automatically suggesting concrete code modifications to solve the detected issues**

Future Work

- Methodology for the automatic diagnosing of performance issues:
  - Define a model to characterize fork/join tasks
  - Characterize all tasks spawned by a fork/join application
  - Determine the metrics and entities worth to consider to automatically diagnose performance issues
- Methodology for the automatic suggestion of optimizations:
  - Automatic recognition of fork/join anti patterns and matching to concrete suggestions to avoid them
- Validation of the results:
  - Discover fork/join workloads, suitable for validating both aforementioned methodologies

BACKUP SLIDES.

Related Work

- Analysis of parallel applications on the JVM
  - A number of parallelism profilers focus on the JVM [9][10]

JProfiler

YourKit Java Profiler

Intel vTune

Java Mission Control

The goal	Characterizing processes or threads over time.
Limitations	○ None of the existing tools targets fork/join applications.

[9] Adhianto et al. *HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs*. Concurr. Comput.: Pract. Exper., 22(6): pp. 685–701, 2010.

[10] Teng et al. *THOR: a Performance Analysis Tool for Java Applications Running on Multicore Systems*. IBM Journal of Research and Development, 54(5):4:1–4:17, 2010.

Related Work

- Assisted optimization of applications
  - "*Optimization Coaching*" was first coined to describe techniques to optimize Racket [4] and JavaScript [5] [6] applications

The goal	Report to the developer precise changes in the code that may enable the compiler's optimizer to achieve missed optimizations.
Limitations	○ The techniques were not designed for optimizing parallel applications. ○ The prototyped techniques target only specific compilers.

[4] St-Amour et al. *Optimization Coaching: Optimizers Learn to Communicate with Programmers*. OOPSLA 2012.

[5] St-Amour et al. *Optimization Coaching for Javascript*. ECOOP 2015.

[6] Gong et al. *JITProf: Pinpointing JIT-unfriendly JavaScript Code*. ESEC/FSE 2015.

Related Work

- Analyses on the use of concurrency on the JVM
  - Documenting fork/join anti patterns on the JVM [7][8]

The goal	Identification of common bad practices and bottlenecks on real fork/join applications.
Limitations	○ Focus on detecting performance issues by using code inspection (manual code inspection and static analysis). ○ Do not consider the granularity of the tasks spawned by the fork/join application. ○ Do not mentor the developer towards optimizing a fork/join application.

[7] De Wael et al. *Fork/Join Parallelism in the Wild: Documenting Patterns and Antipatterns in Java Programs Using the Fork/Join Framework*. PPPJ 2014.

[8] Pinto et al. *Understanding Parallelism Bottlenecks in ForkJoin Applications*. ASE 2017.

Challenges

- Automatic detection of performance issues and suggestion of fixes
  - Combination of program-analysis and machine-learning techniques to automatically identify performance problems, to pinpoint them in the source code, and to recommend concrete optimizations.

- Accurately measure granularity of each task
  - Recursion, fine-grained parallelism, task scheduling, exception handling, auxiliary task, etc.

- Low perturbation in metric collection
  - Use of efficient and reduced instrumentation code.
  - Avoid any heap allocation in the target application.

Ongoing Research

- **tgp**: a **T**ask-**G**ranularity **P**rofiler for multi-threaded, task-parallel applications executing on the JVM [11]
  - Features as a vertical profiler [12] collecting metrics from the full system stack at runtime to characterize task granularity
  - Shows the impact of task granularity on application and system performance
  - Generates actionable profiles [13]
    - Developers can optimize code portions suggested by **tgp**

[11] Rosales et al. *tgp: a Task-Granularity Profiler for the Java Virtual Machine*. APSEC 2017.

[12] M. Hauswirth et al. *Vertical Profiling: Understanding the Behavior of Object-oriented Applications*. OOPSLA 2004.

[13] Mytkowicz et al. *Evaluating the Accuracy of Java Profilers*. PLDI 2010.

Ongoing Research

- We analyzed task granularity in DaCapo [14] and ScalaBench [15]
- We revealed fine- and coarse-grained tasks mainly in Java thread pools causing performance drawbacks [11]
- We optimized suboptimal task granularity in `pmd` and `lusearch` [16]
  - Speedups up to 1.53x (`pmd`) and 1.13x (`lusearch`)

[11] Rosales et al. *tgp: a Task-Granularity Profiler for the Java Virtual Machine*. APSEC 2017.

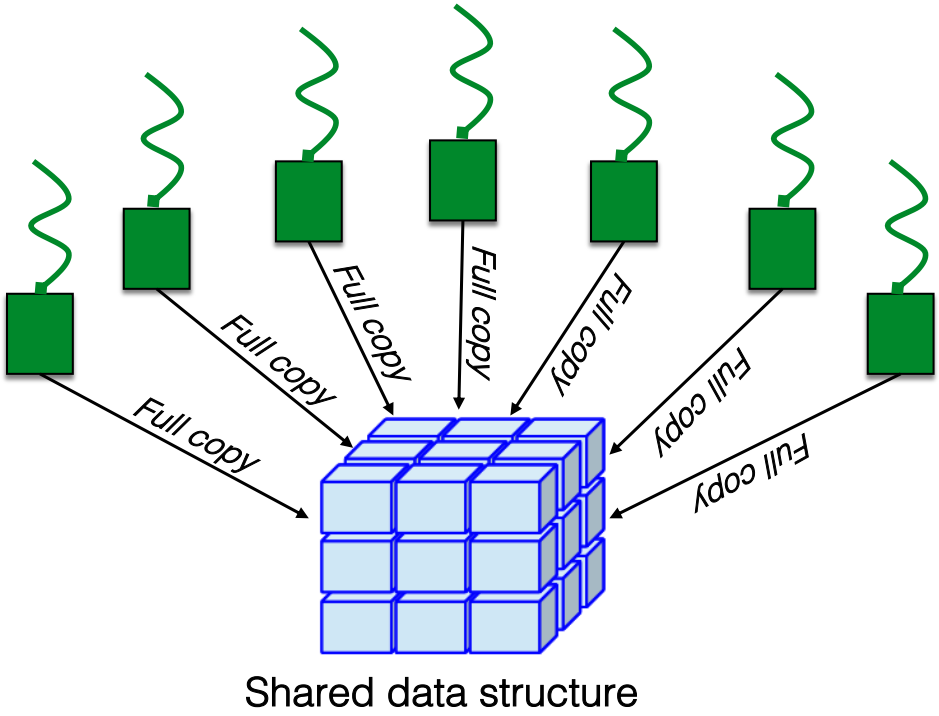
[14] Blackburn et al. *The DaCapo Benchmarks: Java Benchmarking Development and Analysis.* OOPSLA 2006.

[15] Sewe et al. *DaCapo con Scala: Design and Analysis of a Scala Benchmark Suite for the JVM .* OOPSLA 2011.

[16] Rosá, Rosales and Binder. *Analyzing and Optimizing Task Granularity on the JVM*. CGO 2018.

Example of a common performance issues

- Heavy copying on fork



✗ Parallelization overheads due to:

- Excessive object creation
- High memory load
- Frequent garbage collection

Automatic detection of performance issues

- **Static analysis:** analysis of source code for detecting **fork/join anti-patterns,** including:
  - **Heavy copy on fork** (e.g., detecting the use of methods such as `System.arraycopy, sublist`).
  - **Heavy merging on join** (e.g., detecting the use of methods such as `addAll, putAll`).
  - **Inappropriate synchronization** (e.g., detecting the use of improper synchronization during task execution, for example to wait for the result of another computation).
  - **The lack of a sequential threshold** (i.e., a threshold which determines whether a task will execute a sequential computation rather than forking parallel child tasks).

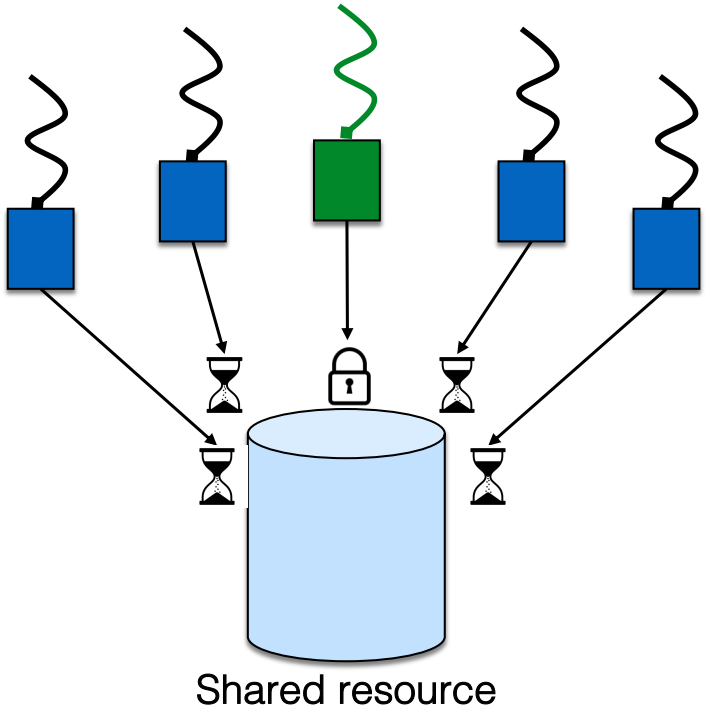
Automatic detection of performance issues

- **Dynamic analysis:** analysis of the fork/join application at runtime to deal with polymorphism and reflection along with detecting:
  - **Suboptimal forking** (i.e., the presence of too fine-grained tasks or few too coarse-grained tasks).
    - According to the Java fork/join framework documentation: "*a task should perform more than 100 and less than 10000 basic computational steps".* [17]
      - **Strategy:** collection and automatic of metrics from the full system stack at runtime:
        - **Framework-level metrics** (e.g., the number of already queued tasks via `getSurplusQueuedTaskCount` method).
        - **JVM-level metrics** (e.g., garbage collections)
        - **OS-level metrics** (e.g., CPU usage, memory load)
        - **Hardware Performance Counters** (e.g., reference cycles, machine instructions)

[17] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html

Example of a common performance issues

- Inappropriate sharing



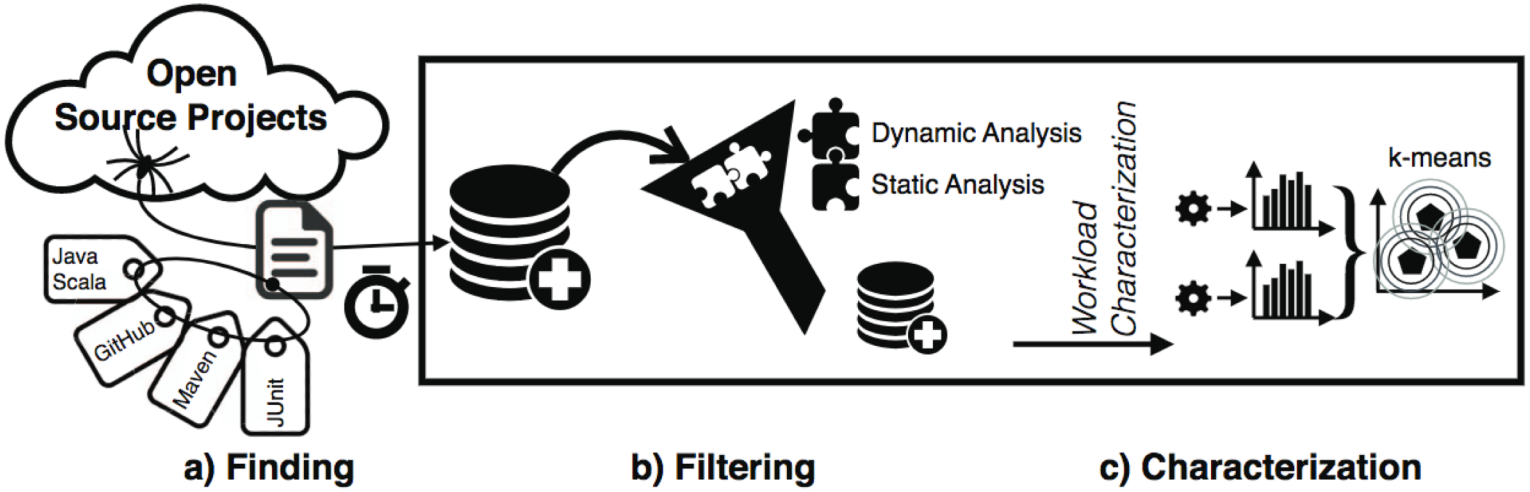
✗ Parallelization overheads due to significant:

- Thread synchronization
- Inter-thread communication

Automatic detection of performance issues

- **Dynamic analysis:** analysis of the fork/join application at runtime to detect:
  - **Inappropriate sharing of resources** (e.g., the use of shared objects, locks, files, data bases and other resources by several parallel tasks).
    - **Strategy:** collection and automatic analysis of performance metrics:
      - **VM-level metrics** (e.g., allocations in Java Heap)
      - **OS-level metrics** (e.g., context switches, cache misses, page faults)

Validation of the results



AutoBench [18], a toolchain combining:

- code repository crawling
- pluggable hybrid analyses, and
- workload characterization techniques

to discover candidate workloads satisfying the needs of domain-specific benchmarking.

[18] Zheng et al. *AutoBench: Finding Workloads that You Need Using Pluggable Hybrid Analyses*. SANER 2016.

Achievements

Publications

- **E. Rosales**, A. Rosà, and W. Binder. *tgp: a Task-Granularity Profiler for the Java Virtual Machine*. 24th Asia-Pacific Software Engineering Conference (APSEC'17), Nanjing, China, December 2017. IEEE Press, ISBN 978-1-5386-3681-7, pp. 570-575
- A. Rosà, **E. Rosales**, and W. Binder. *Accurate Reification of Complete Supertype Information for Dynamic Analysis on the JVM*. 16th International Conference on Generative Programming: Concepts & Experiences (GPCE'17), Vancouver, Canada, October 2017. ACM Press, ISBN 978-1-4503-5524-7, pp. 104-116.
- A. Rosà, **E. Rosales**, and W. Binder. *Analyzing and Optimizing Task Granularity on the JVM*. International Symposium on Code Generation and Optimization (CGO'18), Vienna, Austria, February 2018. ACM Press, ISBN 978-1-4503-5617-6, pp. 27-37.
- A. Rosà, **E. Rosales**, F. Schiavio, and W. Binder. *Understanding Task Granularity on the JVM: Profiling, Analysis, and Optimization*. Accepted to be presented on the Workshop on Modern Language Runtimes, Ecosystems, and VMs (MoreVMs'18), Nice, France, April 2018.
- **E. Rosales** and W. Binder. *Optimization Coaching for Fork/Join Applications on the Java Virtual Machine*. Accepted to be presented on the 12th EuroSys 2018 Doctoral Workshop (EuroDW'18), Porto, Portugal, April 2018.

Concepts

- **Reference cycle:** reference cycle elapses at the nominal frequency of the CPU, even if the actual CPU frequency is scaled up or down.

- **Context switches:** occurs when the kernel switches the processor from one thread to another—for example, when a thread with a higher priority than the running thread becomes ready.

- **Page fault:** is a type of exception raised by computer hardware when a running program accesses a memory page that is not currently mapped by the memory management unit (MMU) into the virtual address space of a process.