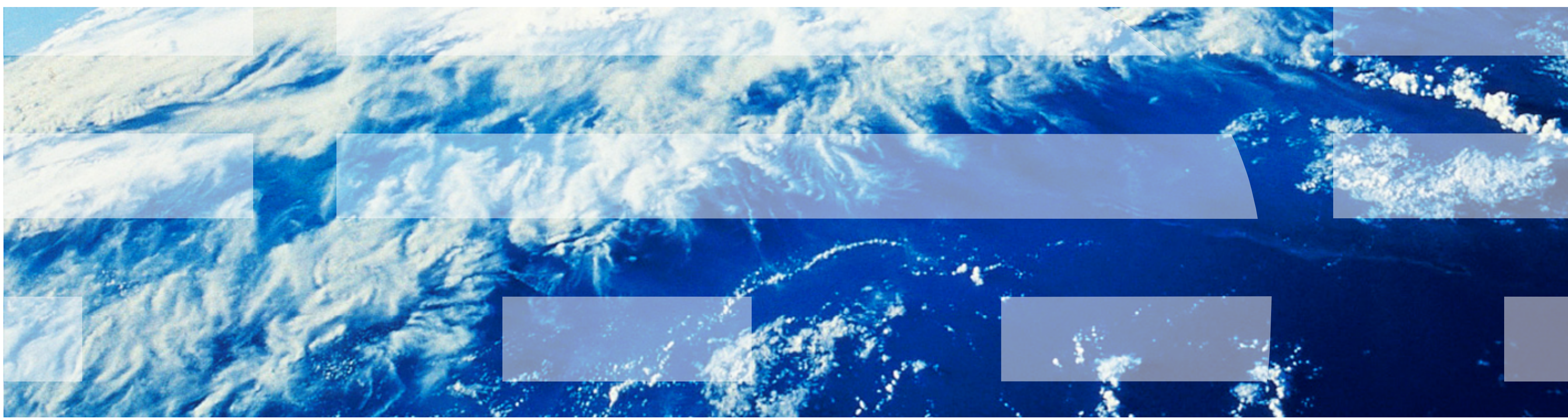
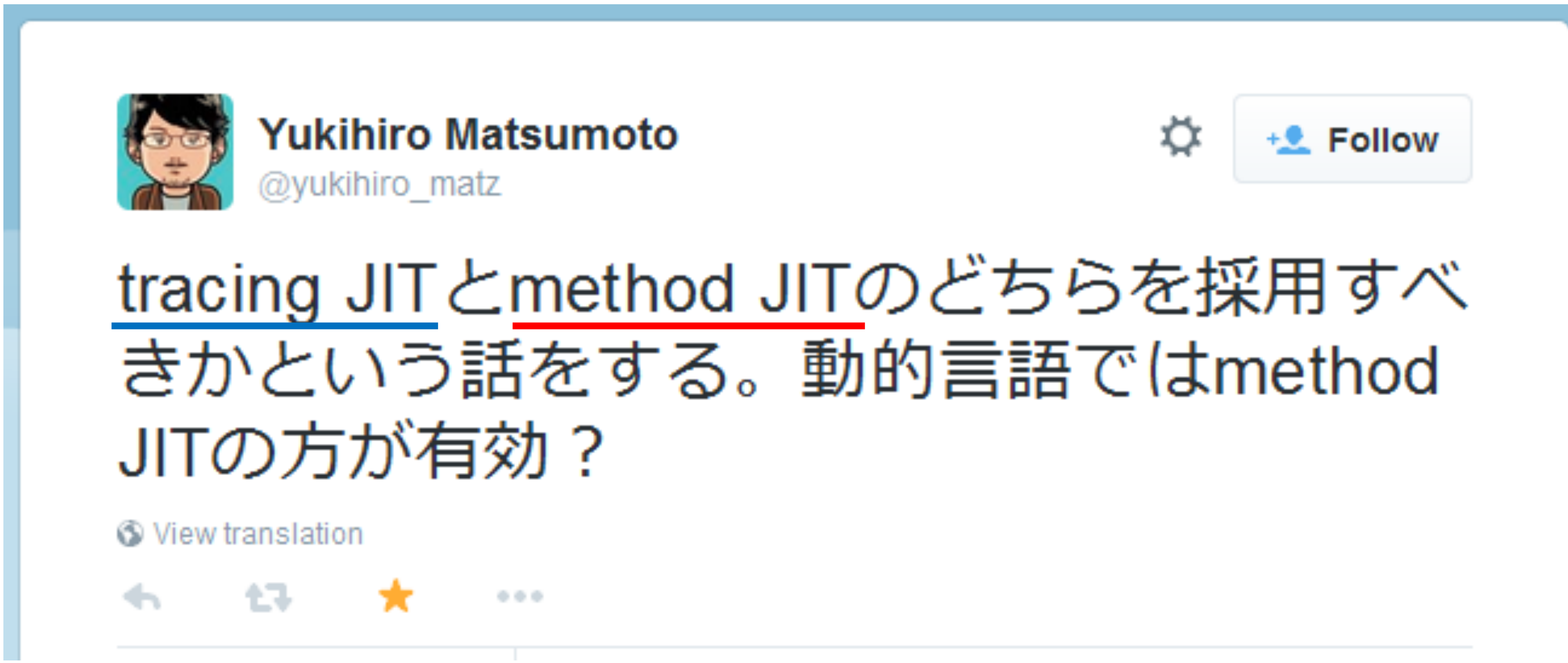
数理・計算科学特論C

プログラミング言語処理系の最先端実装技術

Trace Compilation



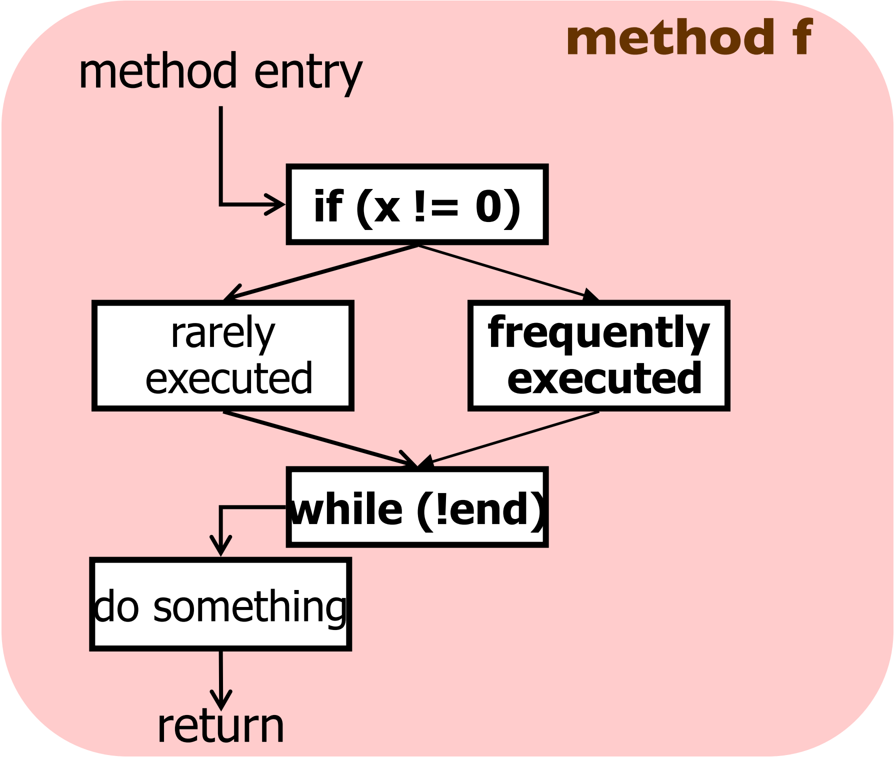
Trace JIT vs. Method JIT



https://twitter.com/yukihiro_matz/status/533775624486133762

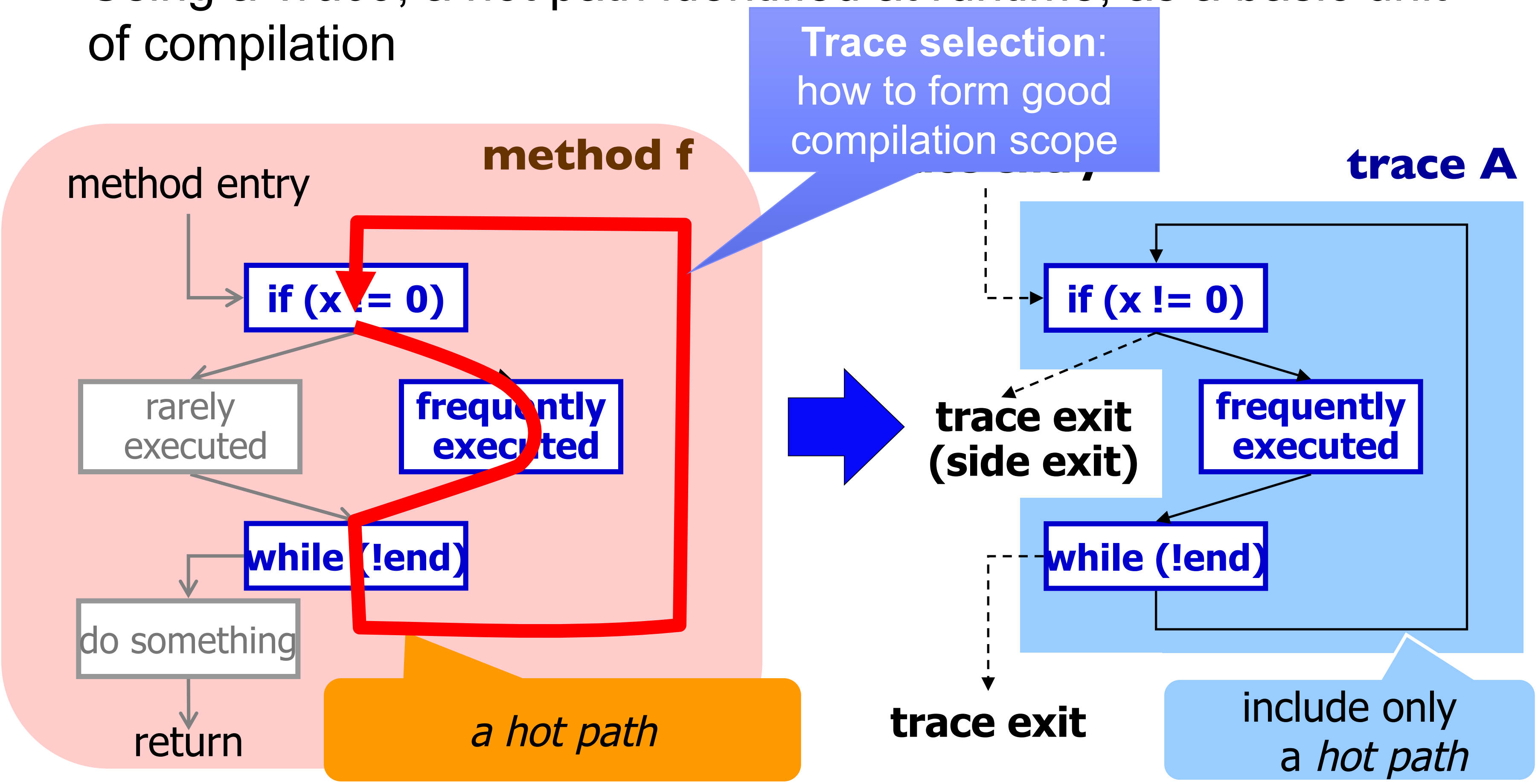
Background: Trace-based Compilation

- Using a *Trace*, a hot path identified at runtime, as a basic unit of compilation



Background: Trace-based Compilation

- Using a *Trace*, a hot path identified at runtime, as a basic unit of compilation



A Brief History of Trace-based Compilation (1)

- Trace-based compilation was first introduced by binary translators and optimizers
  - because method structures are not available in binaries
  - e.g. Dynamo (PLDI’00), BOA/DAISY (MICRO’99 etc.)
- Dynamo demonstrated optimization potentials
  - average 10% speedup over binaries compiled at –O level
  - improvements came mostly from better code layout and simple folding
- Trace-based compilation is still commonly used in binary instrumentation tools, translators today
  - e.g. DynamoRIO, Strata, Intel Pin

A Brief History of Trace-based Compilation (2)

- Also, trace-based compilation has gained popularity in dynamic scripting languages
  - because it can potentially provide more opportunities for type specialization
  - e.g. TraceMonkey (PLDI'09), PyPy (ICOOOLPS'09), LuaJIT, SPUR (OOPSLA'10)
- TraceMonkey (used in Firefox 3.5 - 10) is the first trace-JIT for JavaScript
  - demonstrated 2x to 20x speedups on loop-intensive scripts
- PyPy is the first Python trace-JIT
  - use trace compilation for aggressive specialization of generic operations/data

A Brief History of Trace-based Compilation (3)

- HotpathVM (VEE'06), YETI (VEE'07), Maxpath (PPPJ'10), HotSpot-based trace-JIT (PPPJ'11), Dalvik JIT of Android 2.2+ are trace-JITs targeting Java and variants
- HotpathVM emphasizes its efficiency in resource constrained systems where full-blown JIT compilation is not practical
- YETI showed that the trace-based compilation eased the development of a JIT compiler by allowing incremental implementation of JIT compiler
- HotSpot-based trace-JIT showed performance improvement over (method-based) HotSpot client compiler (closer to region-based compilation approach)

Trace? Region?

- The definition of "trace" often differs by authors!
- Typically,
  - traces are generated based on runtime path profile information
  - control flow merge is not allowed within a trace
    - but, control flow divergence is sometimes allowed (a.k.a. trace tree)
- Region-based compilation [1], is a similar approach. A region is often a subset of a method excluding cold code. (But again no definitive definition of "region")

[1] R. E. Hank et al., "Region-Based Compilation: An Introduction and Motivation", Micro95.

Outline

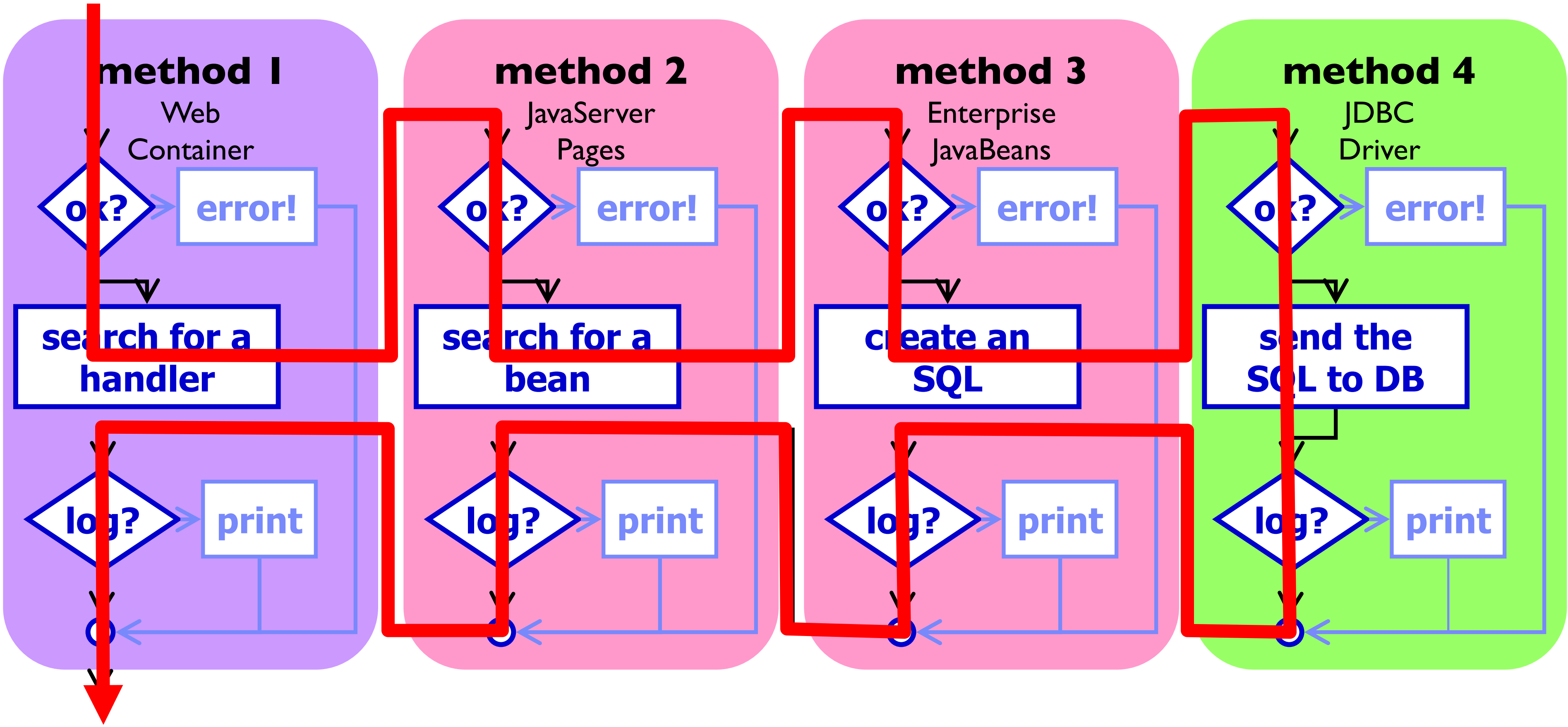
- Back ground
- Overview of our trace-JIT
- Trace Selection and Performance
- Summary

Our Trace-based Java JIT

- Problem statement
  - How to optimize large-scale Java applications with deep (>100) call chains and a flat execution profile?
- Why trace compilation?
  - Tracing may create larger compilation scopes than conventional inlining, especially across method boundaries
  - Tracing may provide context-sensitive profiling information
- Our approach
  - Develop a trace-JIT based on existing method-based Java JIT
- 2-year effort by 3 members
  - Hiroshi Inoue, Hiroshige Hayashizaki (IBM Research – Tokyo)
  - Peng Wu (IBM Research – Watson)

Motivating Example

- A trace can span multiple methods
  - Free from method boundaries
  - In large server workloads, there are deep (>100) layers of methods

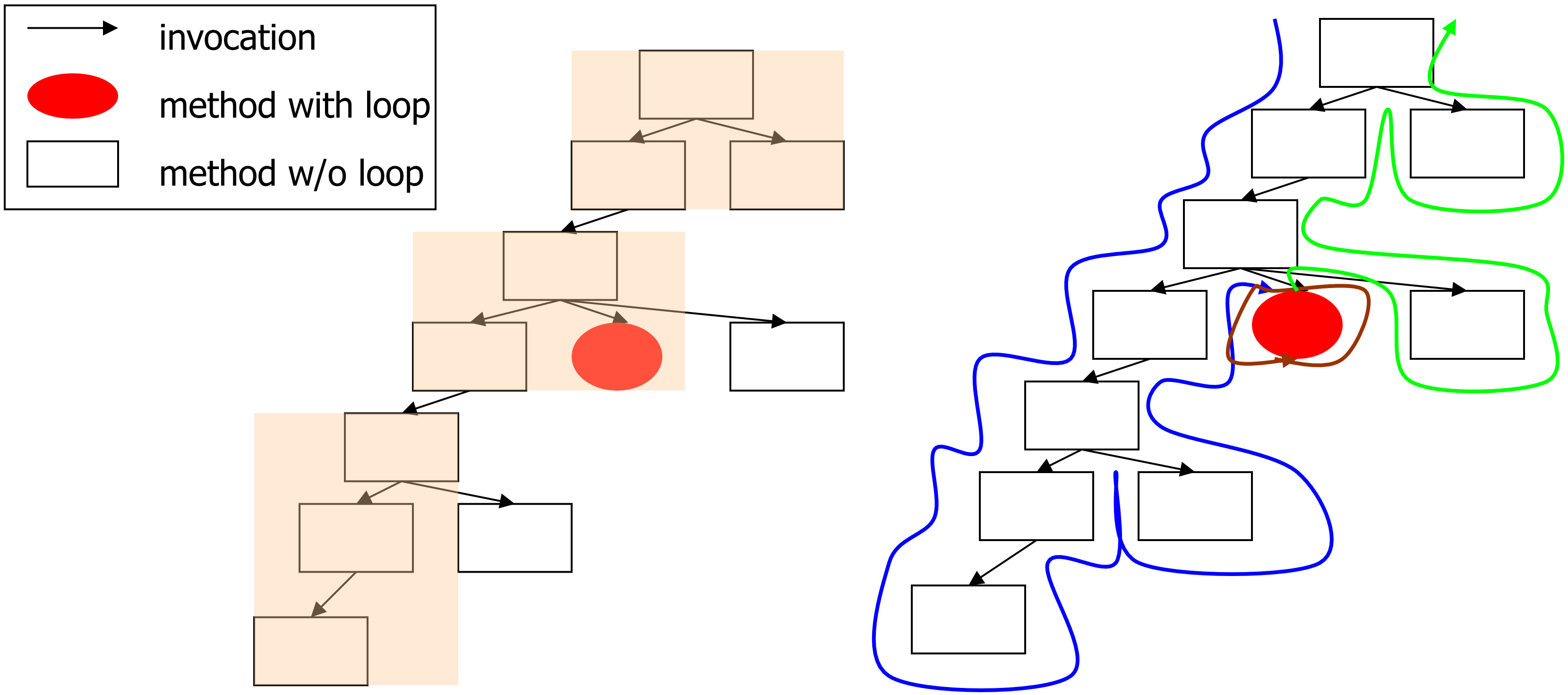


Our Questions

1. Can trace-JIT break method boundaries more effectively?
2. Can trace-JIT produce better codes?
3. Can trace-JIT compile more efficiently (i.e., compile time & code size)?
4. Can a Java trace-JIT beat a Java method-JIT?

Trace Selection vs. Method Inlining

ASSUMPTION: when a call graph is too big to be fully inlined into the root node



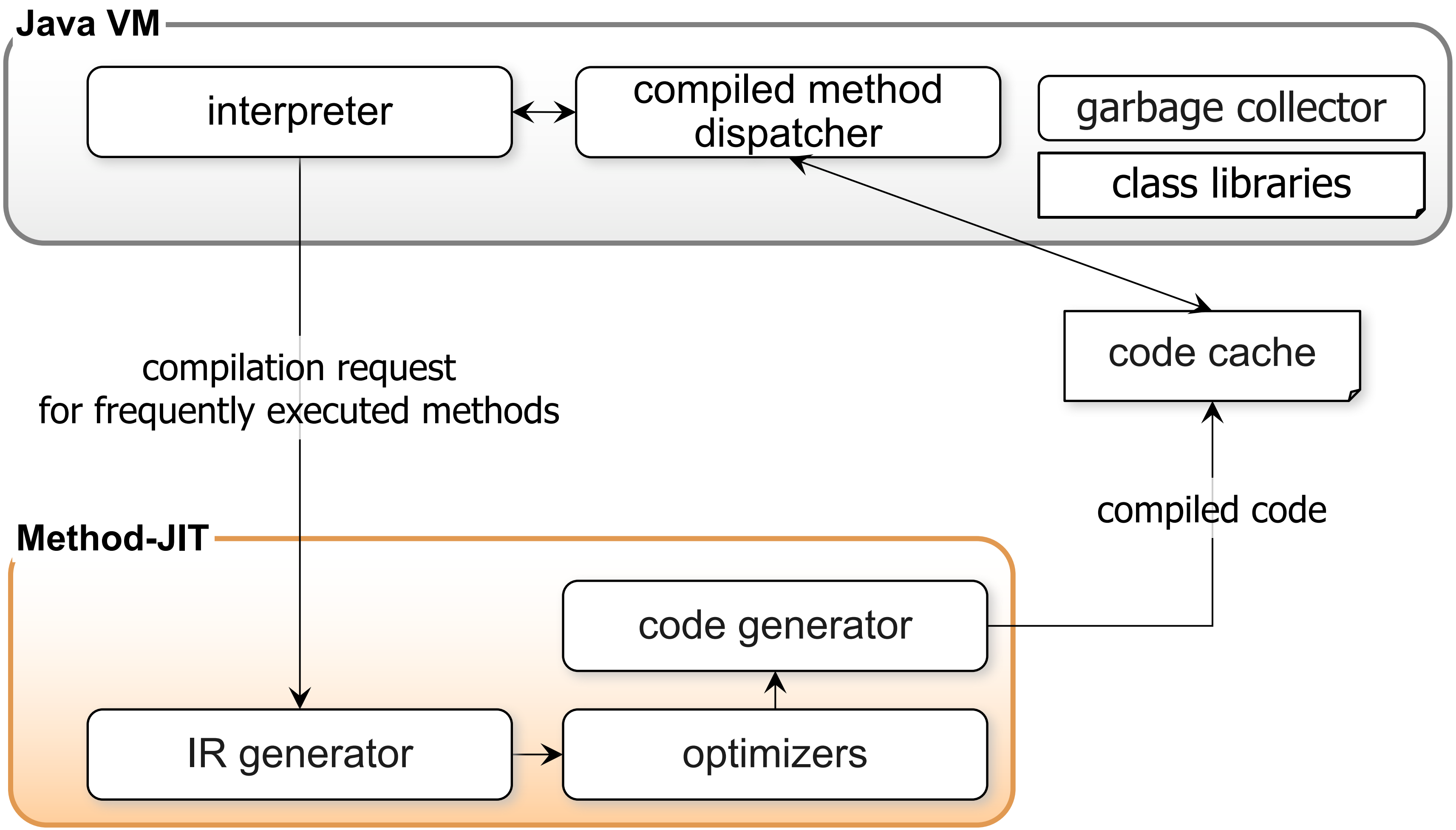
Method inlining forms

hierarchical regions

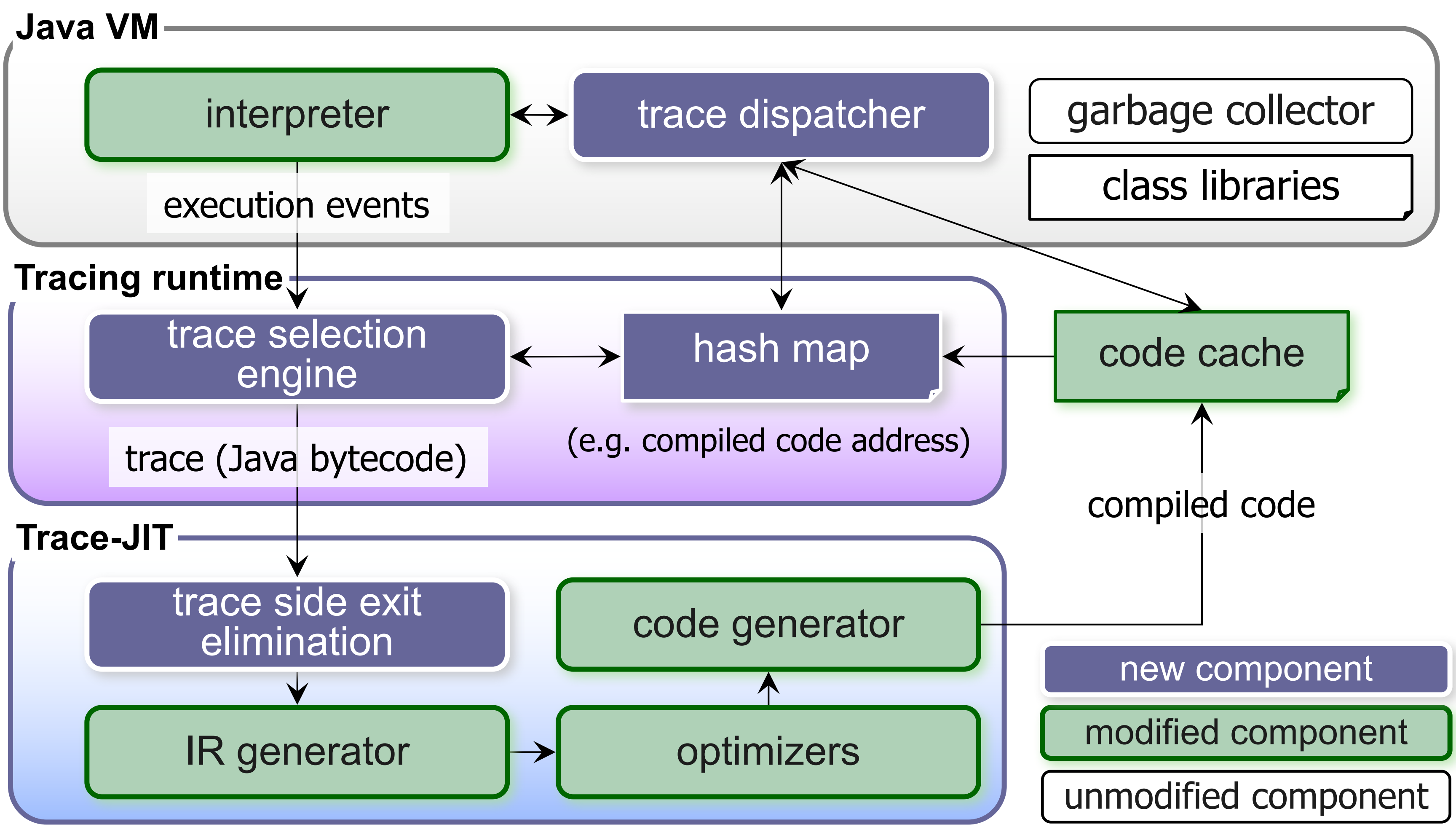
Trace selection forms contiguous regions

– blue, brown, green

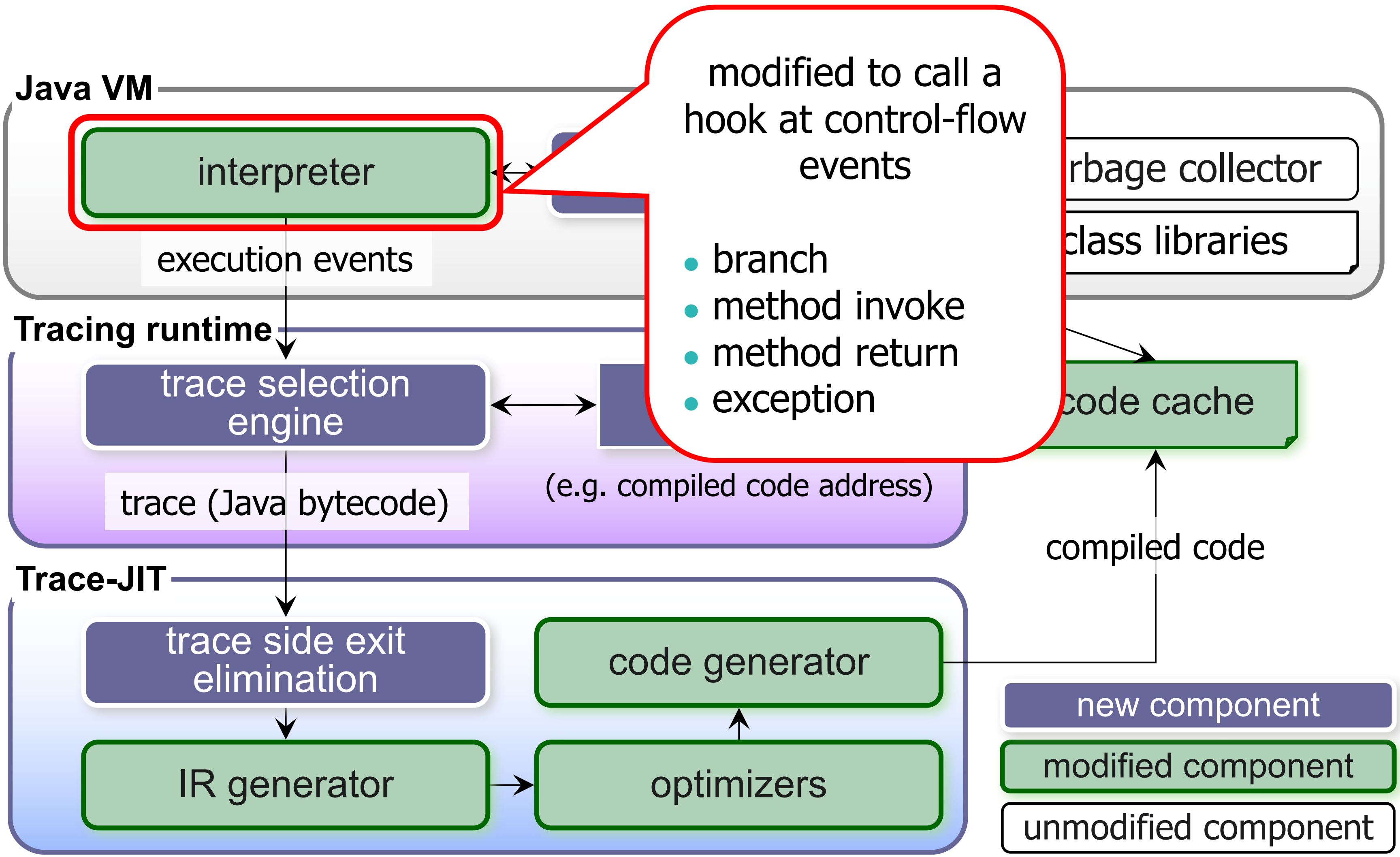
Baseline Method-JIT Components



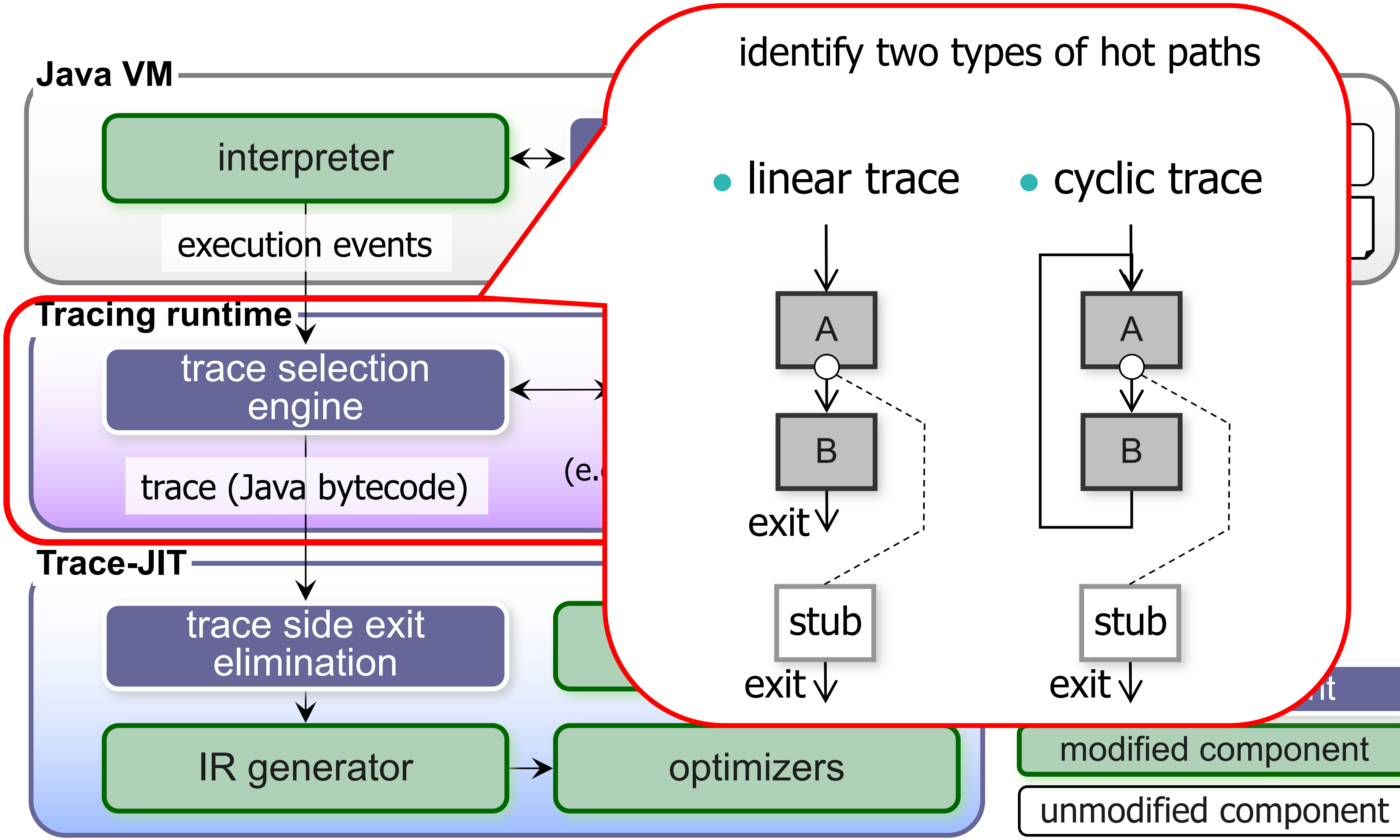
Our Trace-JIT Architecture



Our Trace-JIT Architecture



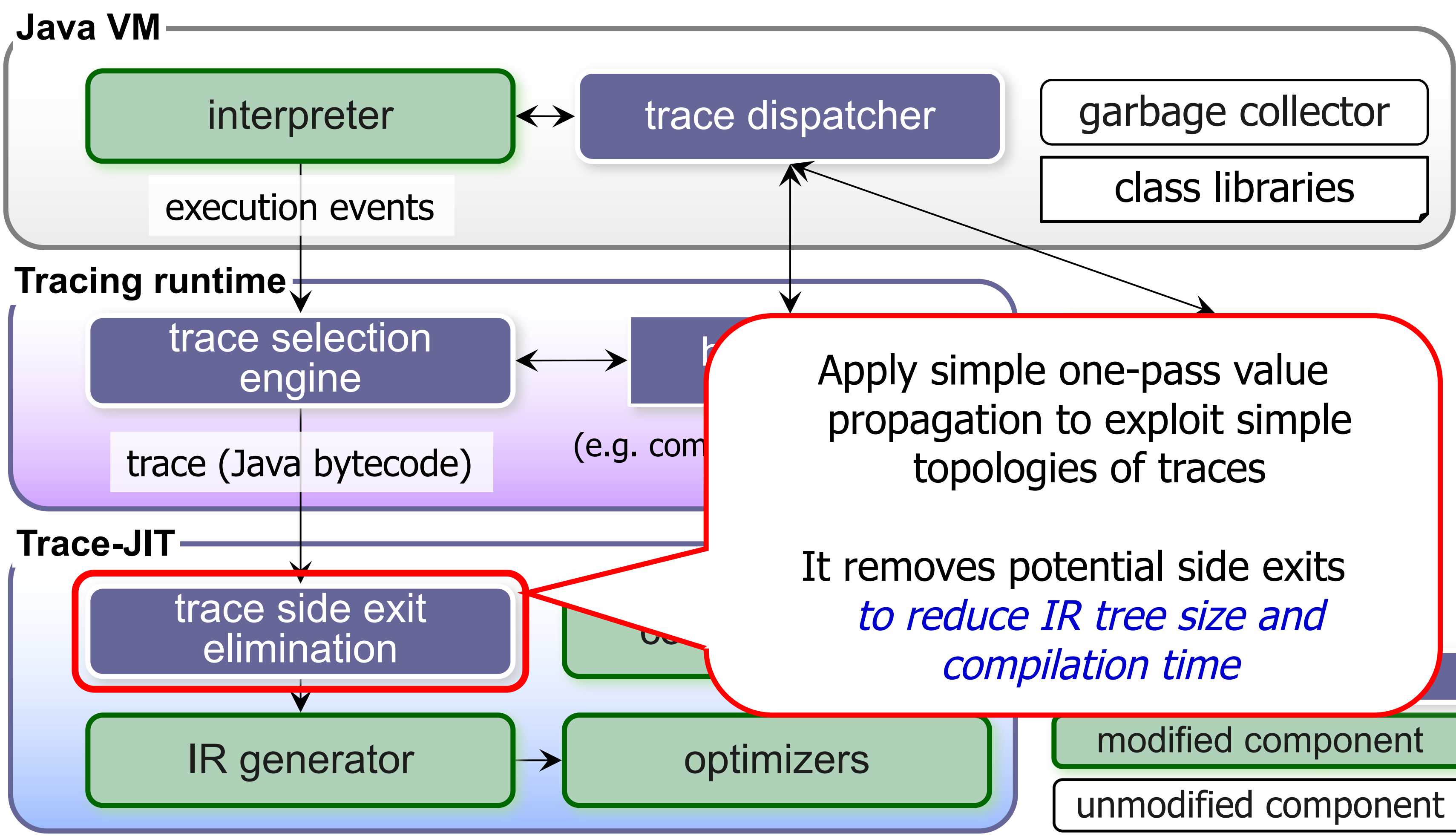
Our Trace-JIT Architecture



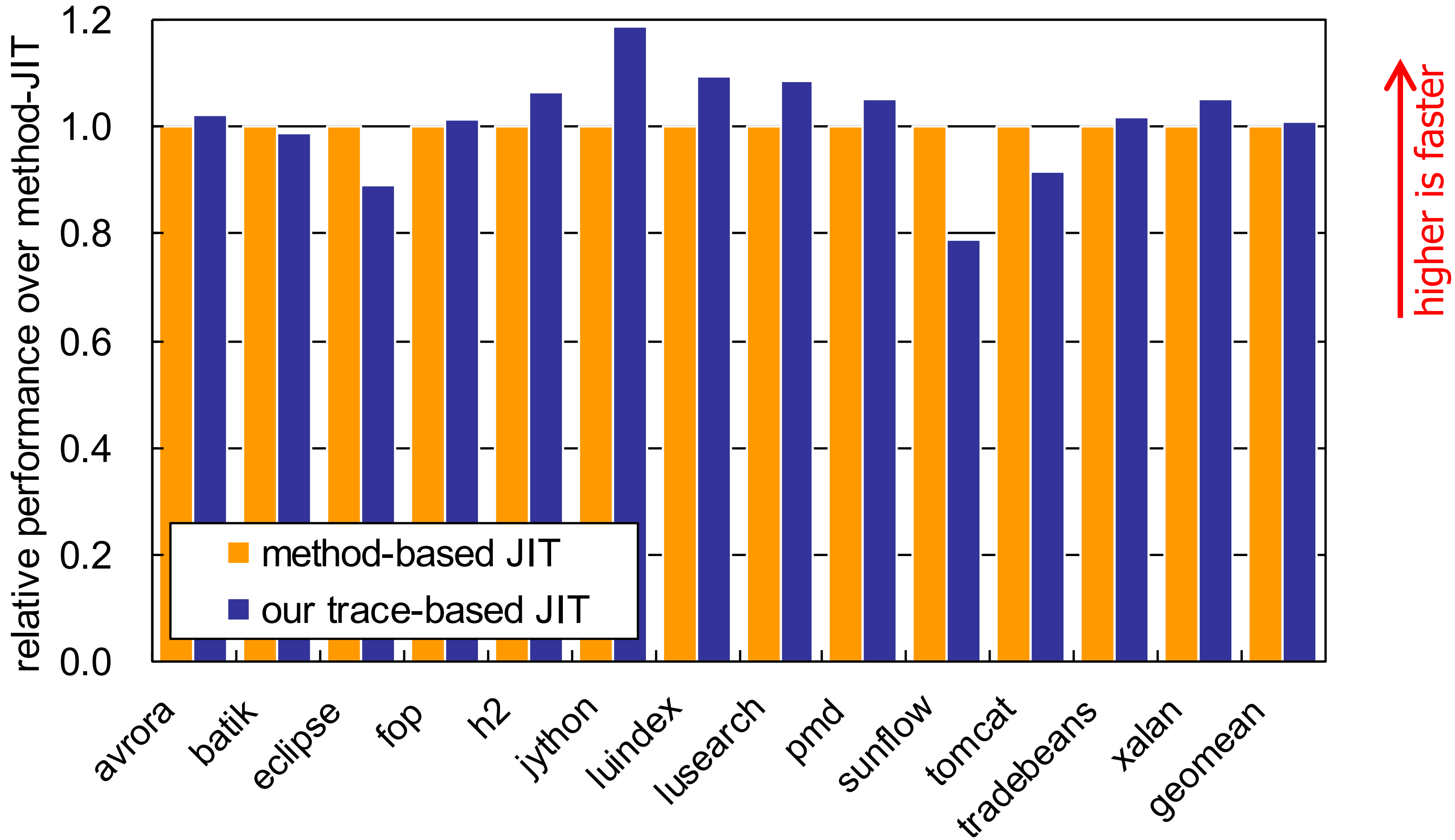
Trace Selection

1. Identify a hot trace head
   - a taken target of a backward branch
   - a bytecode that follows a exit point of an existing trace
2. Record next execution path starting from the trace head
3. Stop recording when the trace being recorded:
   - forms a cycle (loop)
   - reaches pre-defined maximum length
   - calls or returns to a JNI (native) method
   - throws an exception

Our Trace-JIT Architecture



Peak Performance (DaCapo-9.12)



- Trace-JIT was almost comparable to the method-JIT on average
- 21% slower to 19% faster

Outline

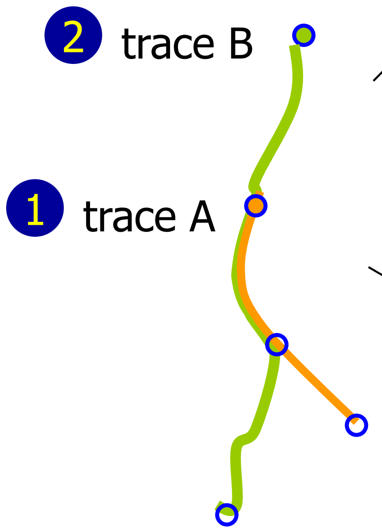
- Back ground
- Overview of our trace-JIT
- Trace Selection and Performance
- Summary

Trace Selection and Performance

- Block duplication is inherent to any trace selection algorithm
  - e.g., most blocks following any join-node are duplicated on traces
- Generating larger compilation scope by allowing more duplication is
  - ☺ key to achieve **higher peak performance**
    - more optimization opportunities for compilers
    - smaller trace transitioning overhead
  - ☹ but it may yield **longer compilation time**
    - costly source code analysis
    - more duplicated code among traces

➔We observed lots of duplication that does not help the performance

Understanding the Causes (I): Short-Lived Traces

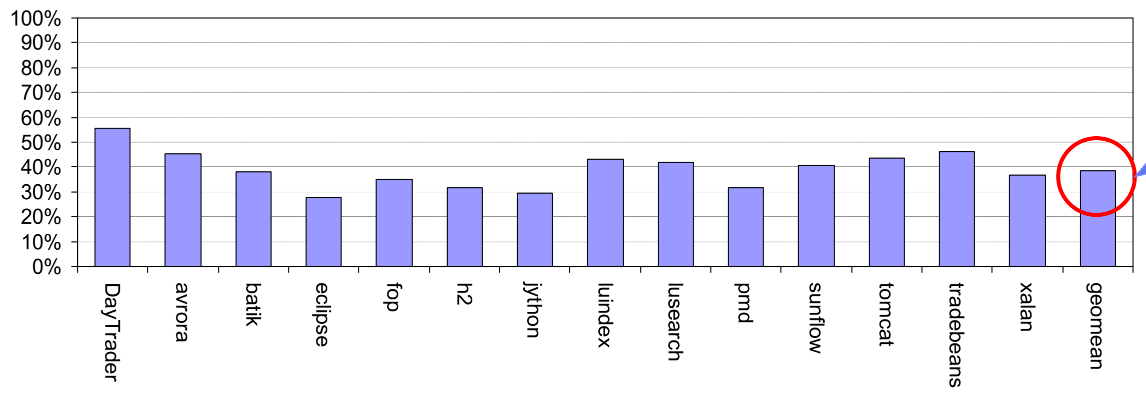


SYMPTON

- Trace A is formed first
- Trace B is formed later
- Afterwards, A is no longer entered

ROOT CAUSE

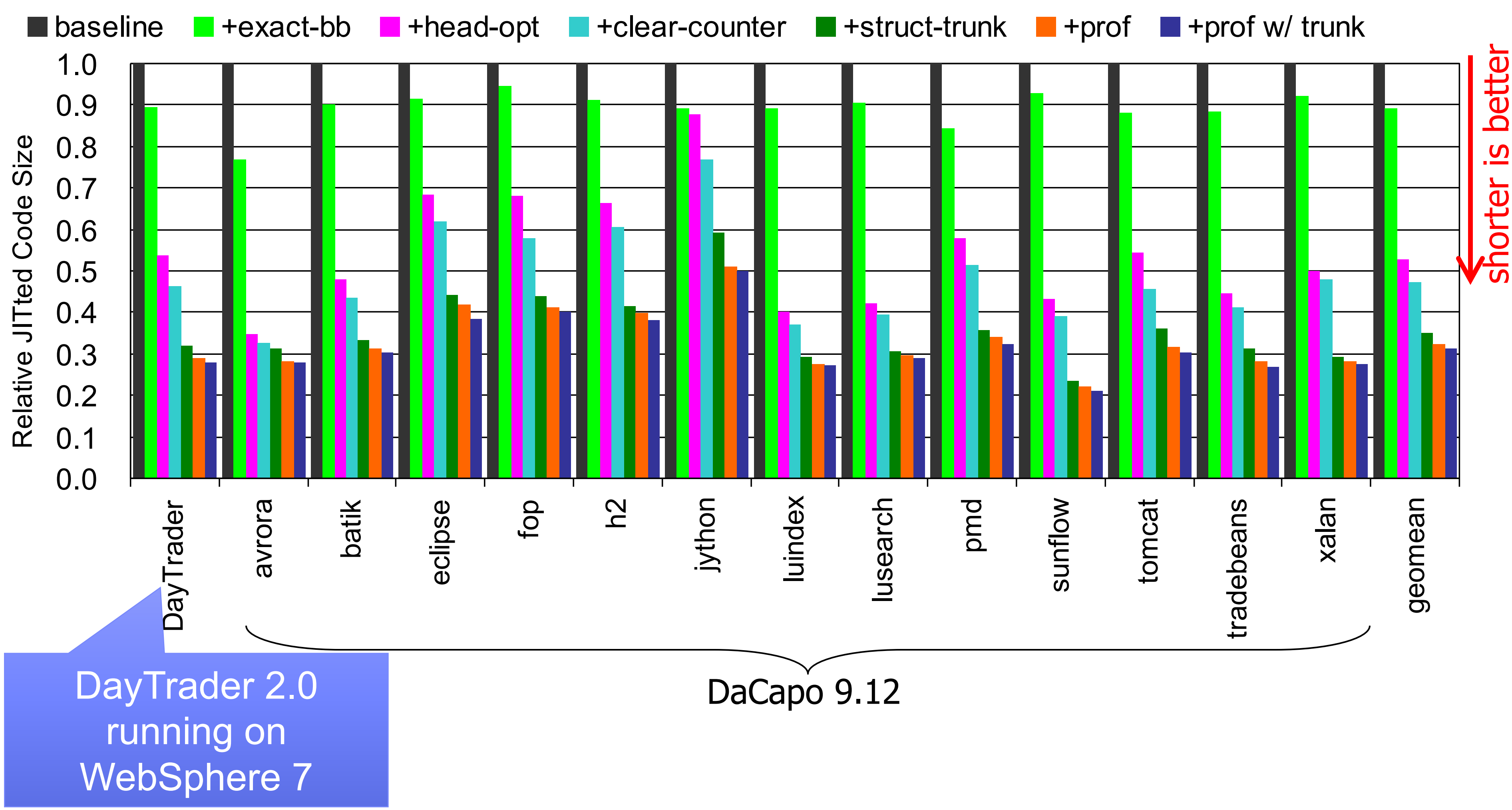
1. Trace A is formed before trace B, but node B dominates node A
2. Node A is part of trace B



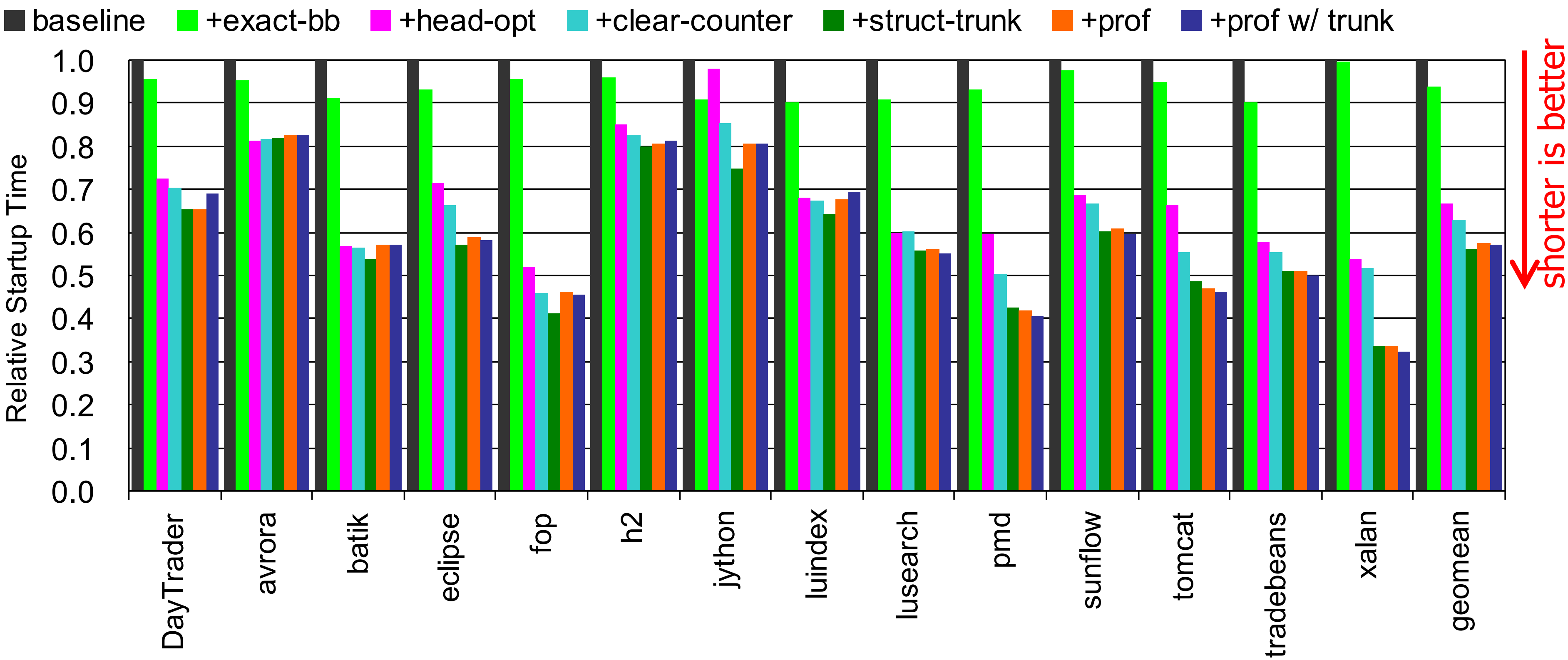
On average, 40% traces of DaCapo 9-12 are short lived

% traces selected by baseline algorithm with <500 execution frequency

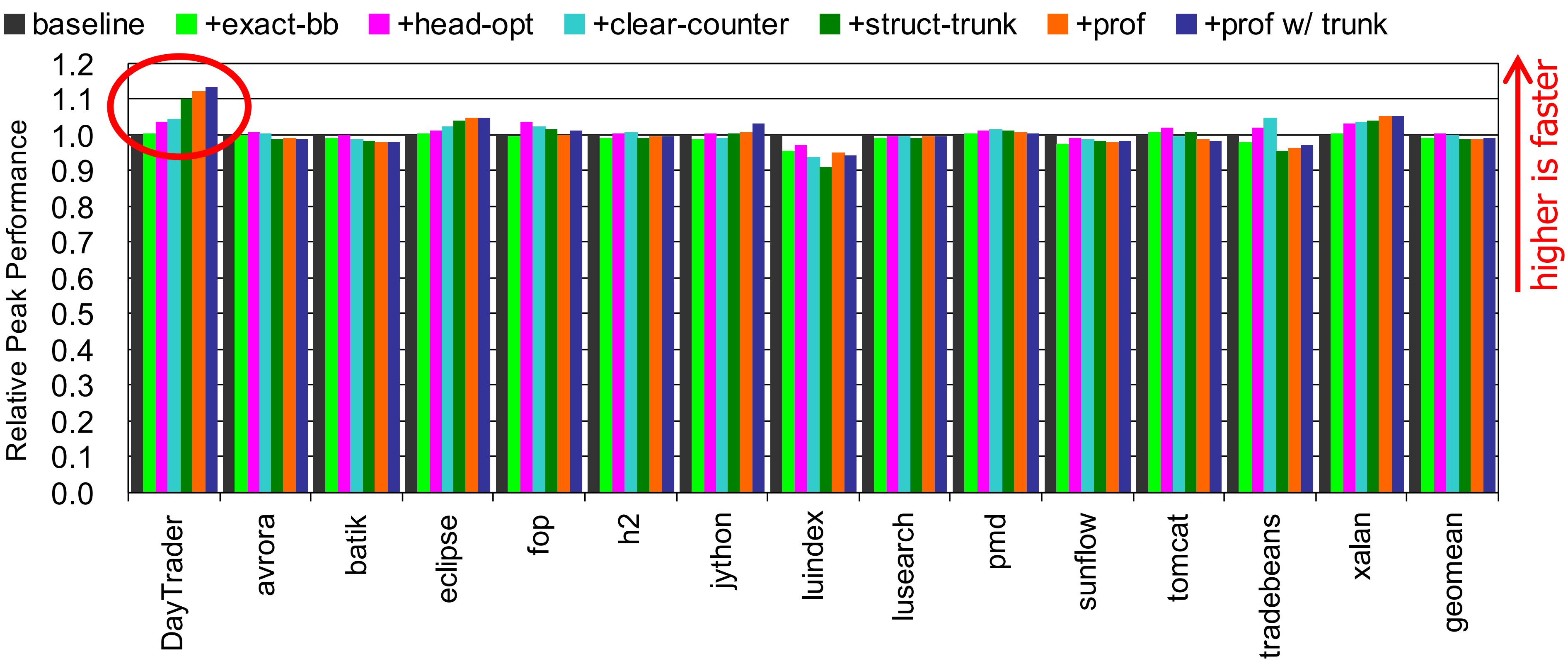
Compiled code size reduced by 70%



Startup time reduced by 45%



Peak performance was also improved in DayTrader!

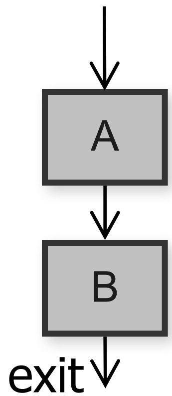


Trace Selection and Performance for Large-scale Applications

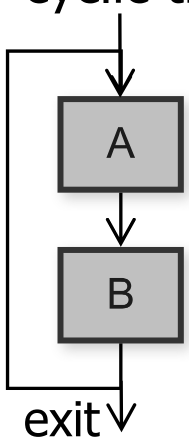
- Generating larger compilation scope by allowing more duplications
  - ☺ is key to achieve **higher peak performance**
    - more optimization opportunities for compilers
    - smaller trace transitioning overhead
  - ☹ but it may **hurt startup performance**
    - longer compilation time
    - more duplicated code among traces
  - ☹ also it may **hurt the peak performance for large applications**
    - Larger application tend to cause more instruction cache misses
    - ~20% of CPU cycles were wasted by I-cache misses in DayTrader

Generating longer trace also does not necessarily work: Supporting a loop in a trace

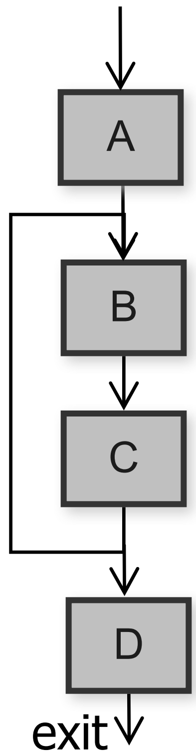
- linear trace



- cyclic trace



- *extended-form trace*



☺ fewer trace transition
➔L1 I$ miss: -10%
☺ potentially more optimization opportunity
➔but, no improvement observed

☹ more code duplication among traces
➔total code size: +20%
➔L3 Instruction miss: +6%

☹ **Performance: -1% in DayTrader**

+1% on average of DaCapo-9.12
(up to +2.5%)

Outline

- Back ground
- Overview of our trace-JIT
- Trace Selection and Performance
- A large-scale Java application with trace-JIT
- Summary

Questions and Our Answers

1. **Can trace-JIT break method boundaries more effectively?**
   – Workload dependent, e.g., trace JIT produces 2X larger scope than method-JIT for Jython, but 9% smaller scope for DayTrader
   – But simply enlarge the compilation scope does not help performance
2. **Can trace-JIT produce better codes?**
   – Retrofitting method-JIT optimizers for trace-JIT does not yield significant better codes beyond the benefit of larger scope
   – But opportunities may exist in new trace-specific optimizations
   – How to generate trace exit code is an interesting challenge unique to trace-JIT
3. **Can trace-JIT compile more efficiently (i.e., compile time & code size)?**
   – Yes. The simple topology of linear and cyclic traces can be compiled with much more efficiently
4. **Can a Java trace-JIT beat a Java method-JIT?**
   – It is not easy to beat a mature method-JIT. We feel that a specific type of workloads, such as Jython, respond better to trace-JIT than method-JIT

How about for other languages?

- TraceMonkey (JavaScript)
  - http://hacks.mozilla.org/2010/03/improving-javascript-performance-with-jagermonkey/
  - "That the approach that we've taken with tracing tends to interact poorly with certain styles of code."
  - "That when we're able to "stay on trace" (more on this later) TraceMonkey wins against every other engine."
- Pyston (Python)
  - https://tech.dropbox.com/2014/04/introducing-pyston-an-upcoming-jit-based-python-implementation/
  - "Whether or not the same performance advantage holds for Python is an open question, but since the two approaches are fundamentally incompatible, the only way to start answering the question is to build a new method-at-a-time JIT."

Lessons Learned

- Trace selection algorithm has a big impact on performance and code size
  - more flexible than the method inlining and hence is an interesting tool to evaluate the effect of code duplication
- What did not work for us
  - extending trace-scope by allowing non-linear structures (e.g., trace grouping, trace tree) does not yield any performance improvement for DayTrader
- Possible future steps
  - opportunities may exist in new trace-specific optimizations
    - e.g., allocation removal [Bolz ’10], aggressive redundancy elimination
  - improving profile accuracy
    - profile accuracy is more important in trace-JIT than method-JIT

Our Publication on trace-JIT

- Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani, "A Trace-based Java JIT Compiler Retrofitted from a Method-based Compiler", CGO 2011
  - Focus on trace-optimization aspect of the JIT, discussed the scope mismatch problem
- Hiroshige Hayashizaki, Peng Wu, Hiroshi Inoue, Mauricio Serrano and Toshio Nakatani, "Improving the Performance of Trace-based Systems by False Loop Filtering", ASPLOS 2011
  - Focus on trace selection algorithm, fragmentation of traces due to false loop problems
- Peng Wu, Hiroshige Hayashizaki, Hiroshi Inoue, and Toshio Nakatani, "Reducing Trace Selection Footprint for Large-scale Java Applications with no Performance Loss", OOPSLA 2011
  - Focus on trace selection algorithm, the code duplication problem
- Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani, "Adaptive Multi-Level Compilation in a Trace-based Java JIT Compiler", OOPSLA 2012
  - Focus on adaptive compilation of trace-JIT