

# A Trace-based Java JIT Compiler Retrofitted from a Method-based Compiler



*Hiroshi Inoue*<sup>†</sup>, Hiroshige Hayashizaki<sup>†</sup>,  
Peng Wu<sup>‡</sup> and Toshio Nakatani<sup>†</sup>

<sup>†</sup> IBM Research – Tokyo

<sup>‡</sup> IBM Research – T.J. Watson Research Center

# Goal and Motivation

- Goals

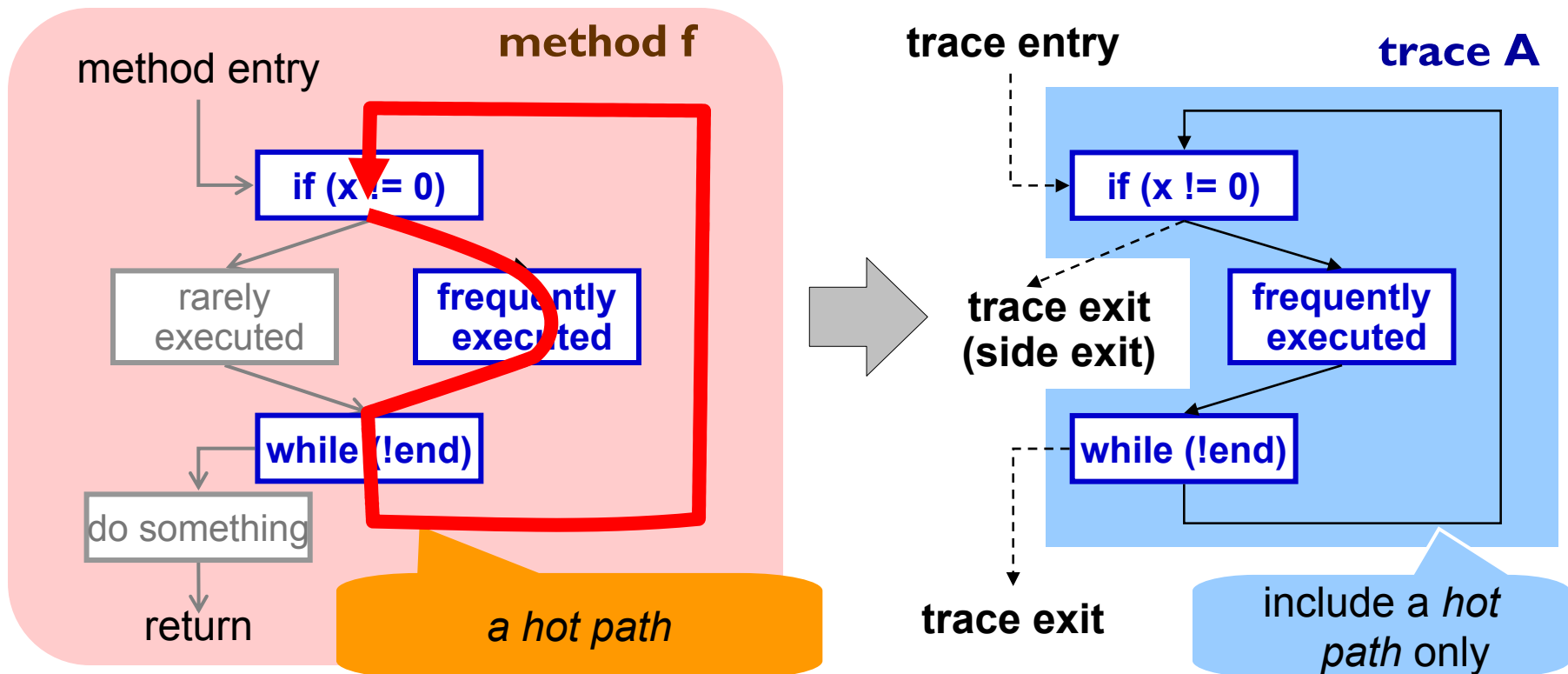
1. develop an efficient *trace-based Java JIT compiler (trace-JIT)* based on existing mature *method-based JIT compiler (method-JIT)*
2. understand the benefits and drawbacks of the trace-JIT against the method-JIT

- Why not method-JIT?

- Limited optimization opportunities in larger application with a flat execution profile (no hot spots)
- Can trace-JIT provide more optimization opportunities than method-JIT for such applications?

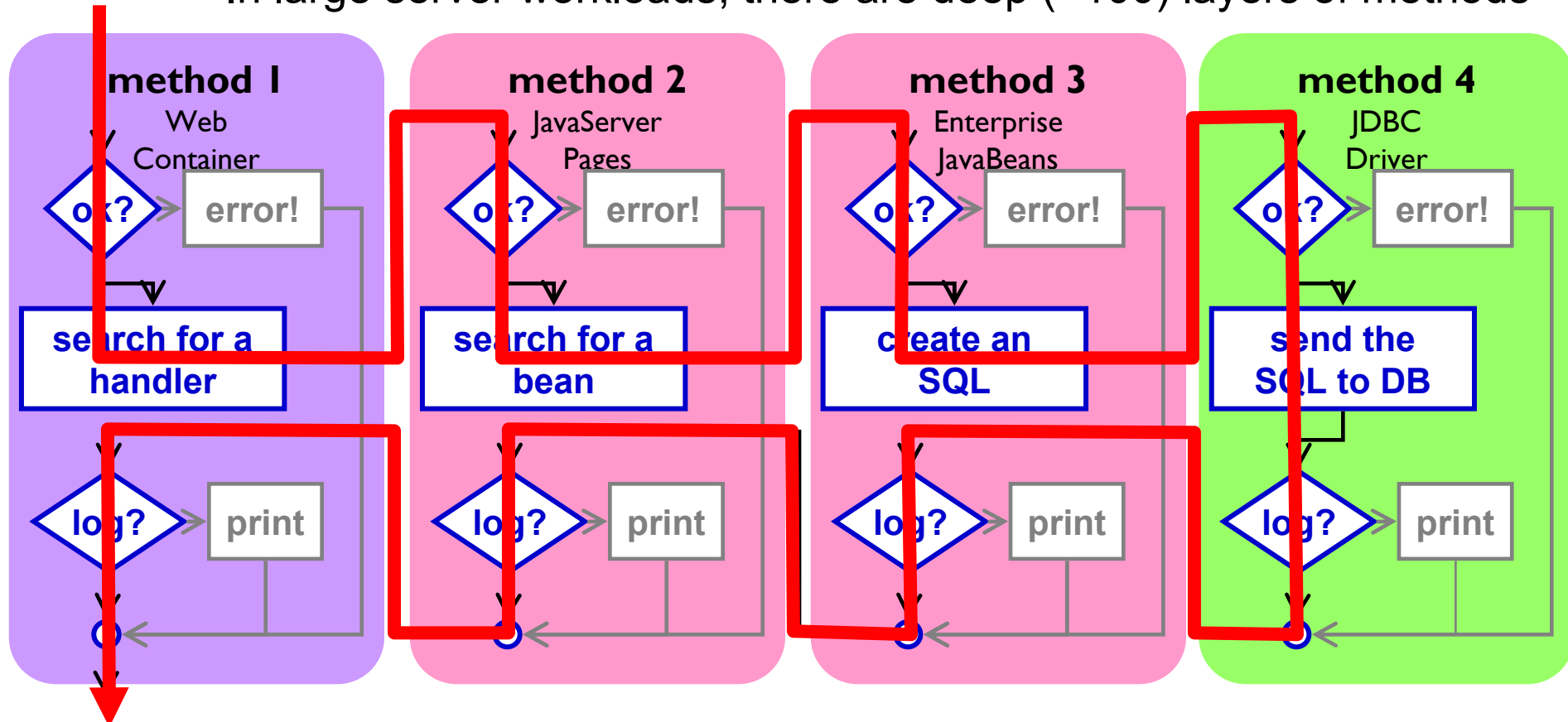
# Background: Trace-based Compilation

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



# Motivating Example

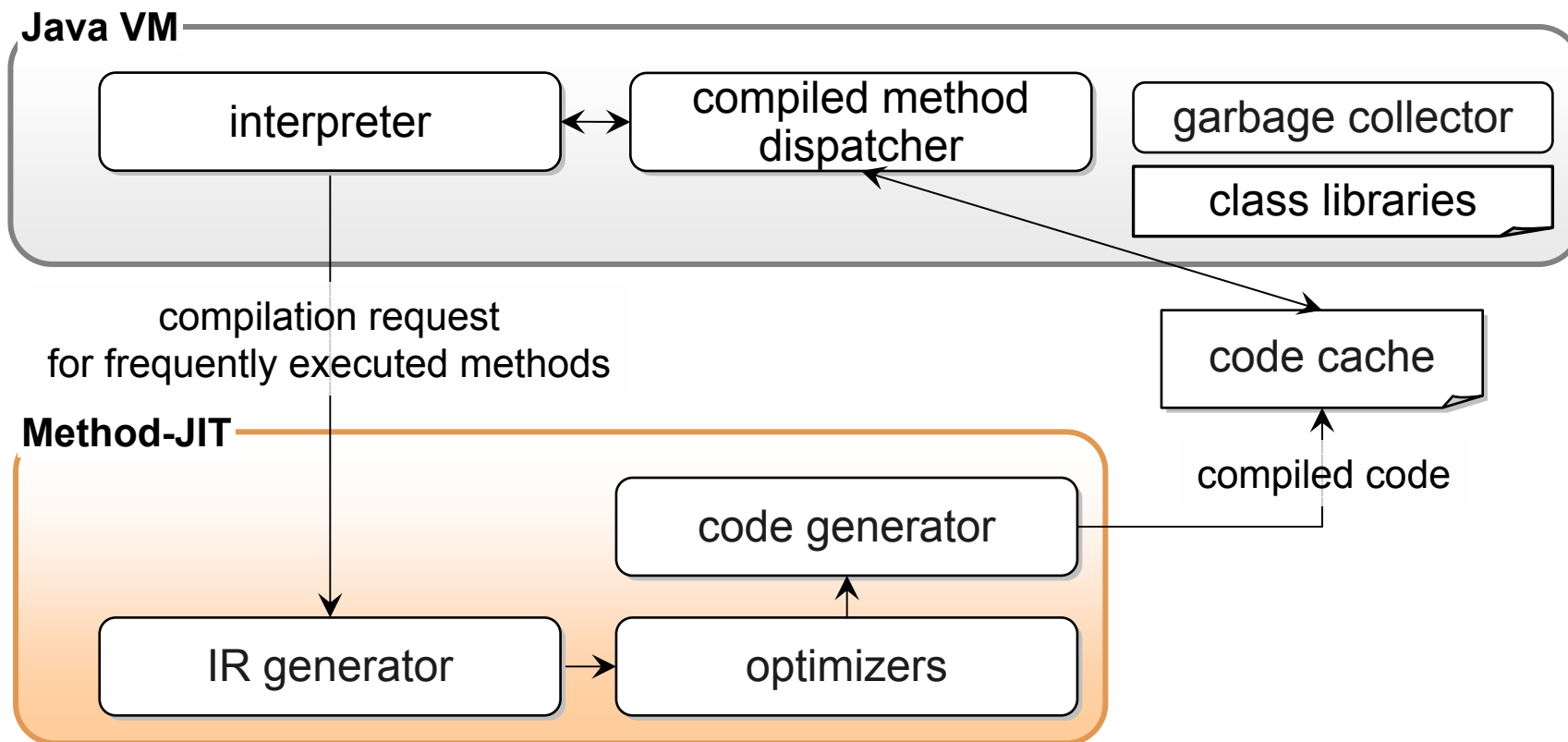
- A trace can span multiple methods
  - Free from method boundaries
  - In large server workloads, there are deep (>100) layers of methods



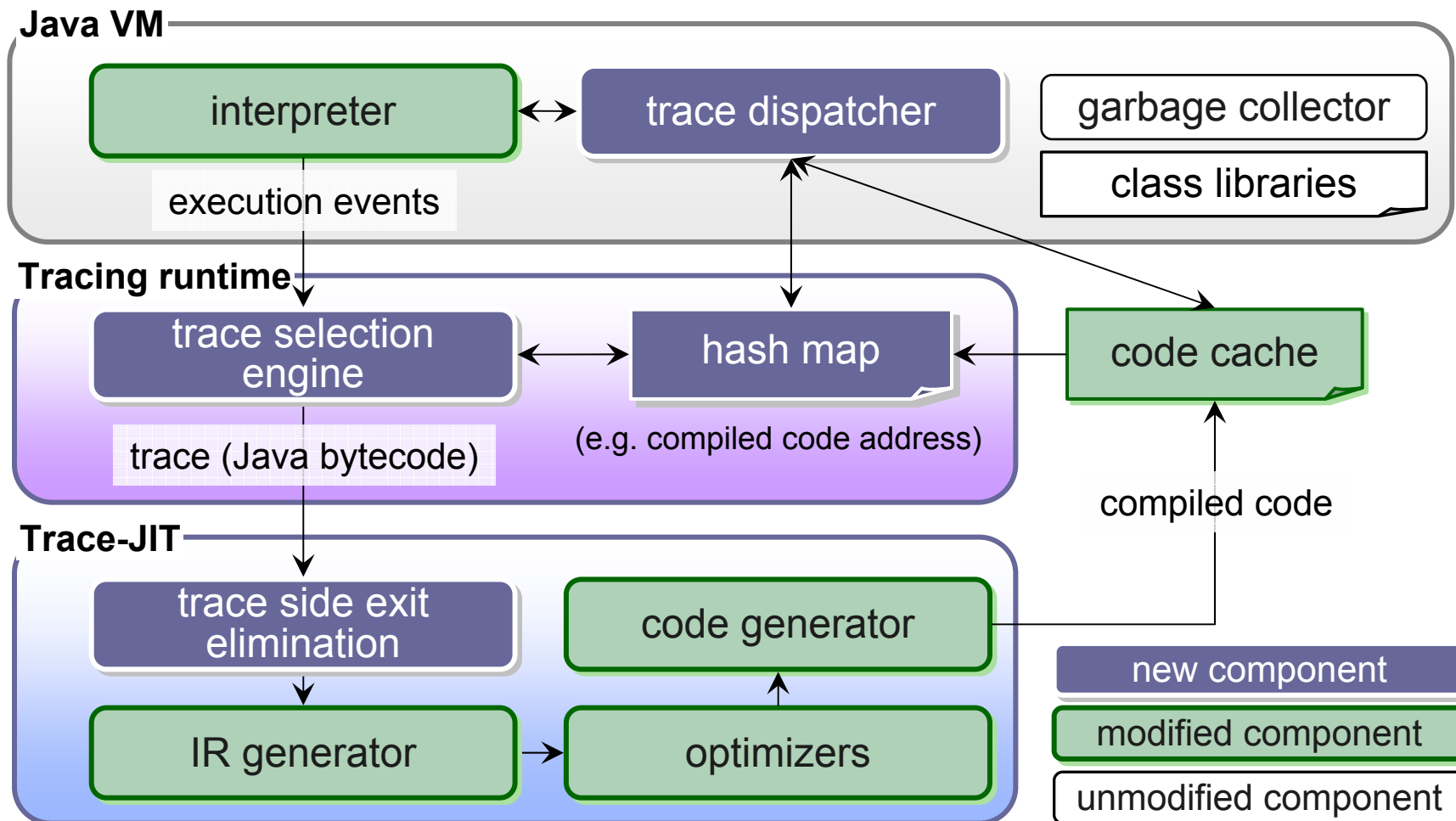
# Outline

- Motivation
- Background
- **Trace-JIT Architecture**
- Performance Evaluation
- Future work and Summary

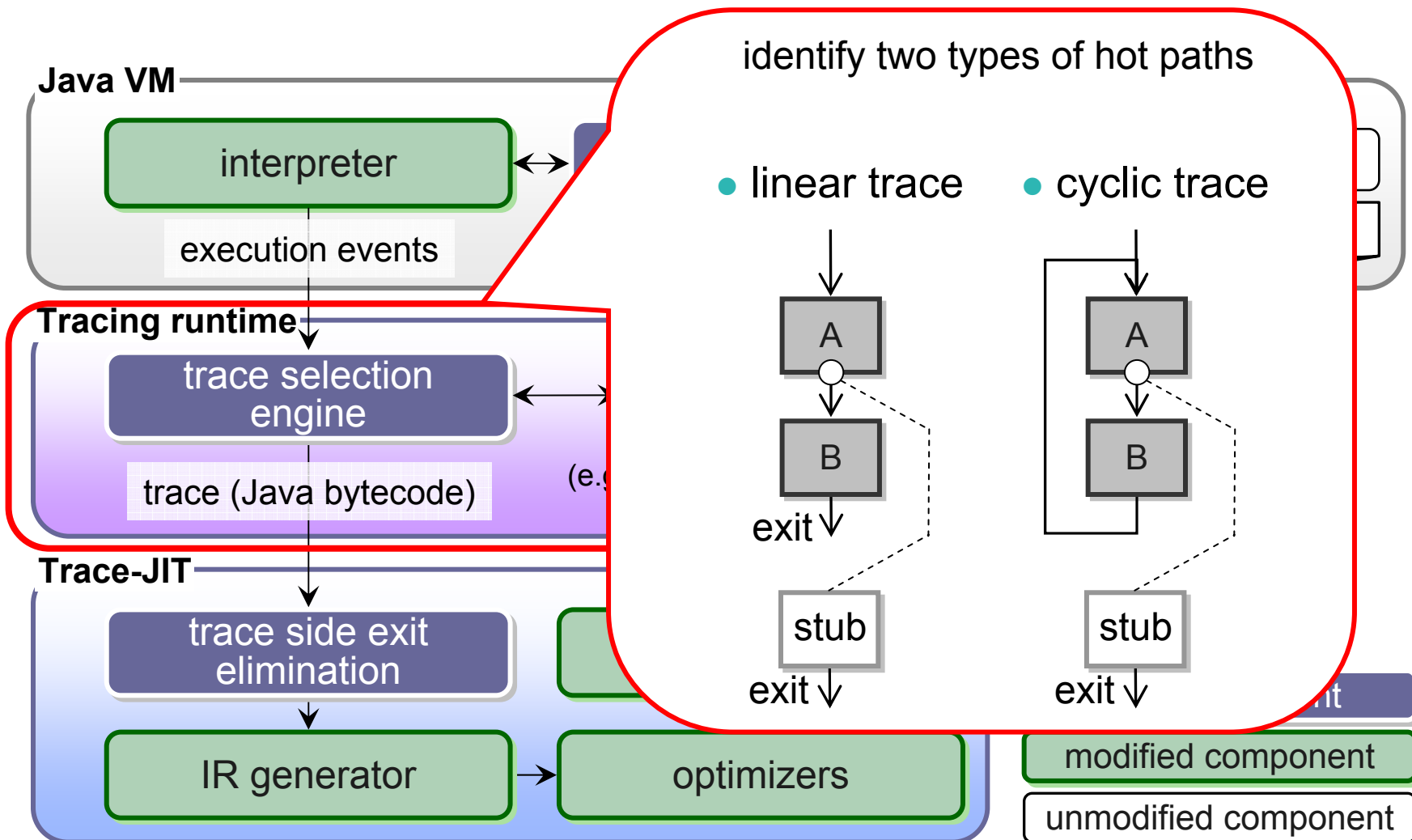
# Baseline Method-JIT Components



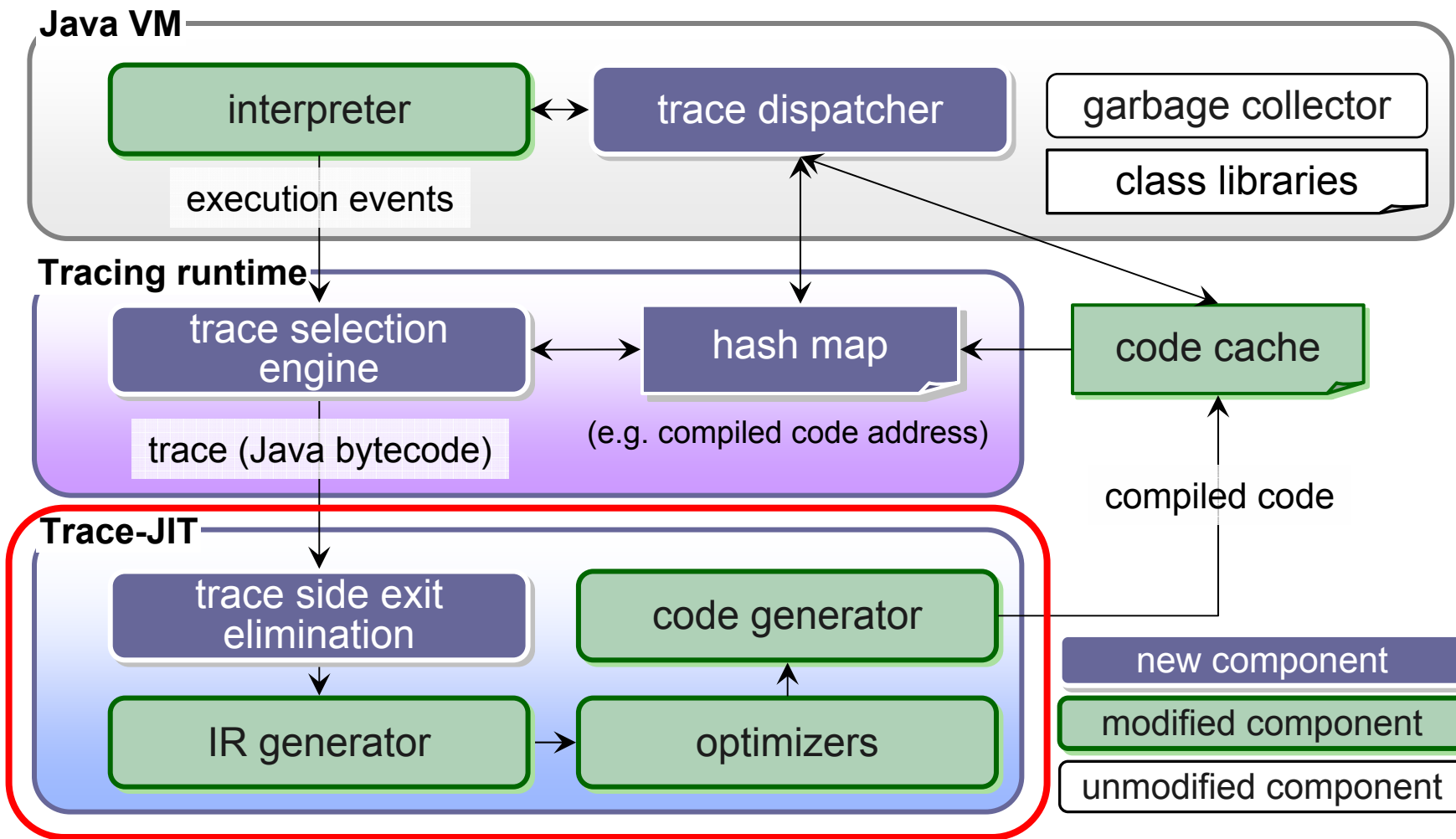
# Our Trace-JIT Architecture



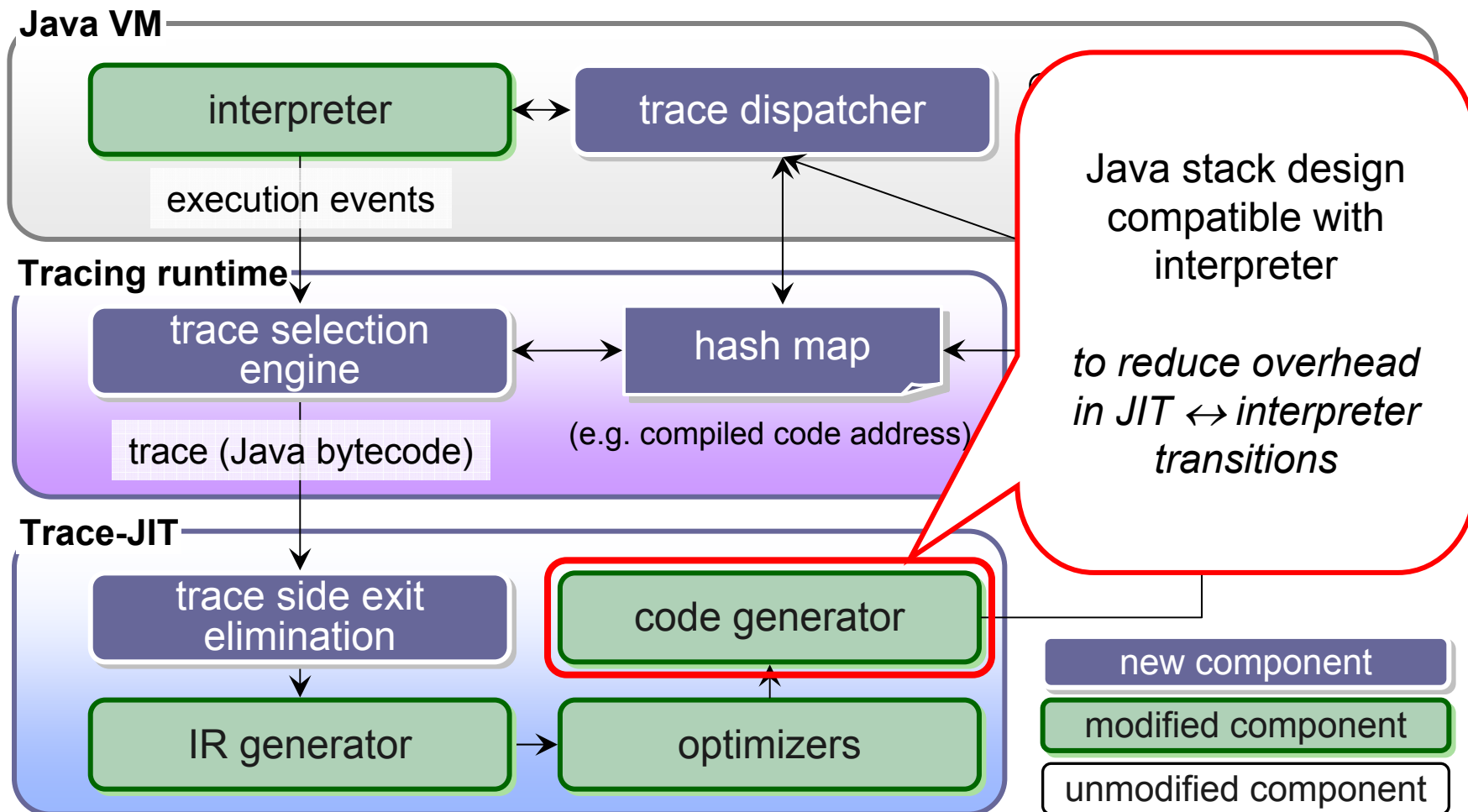
# Our Trace-JIT Architecture



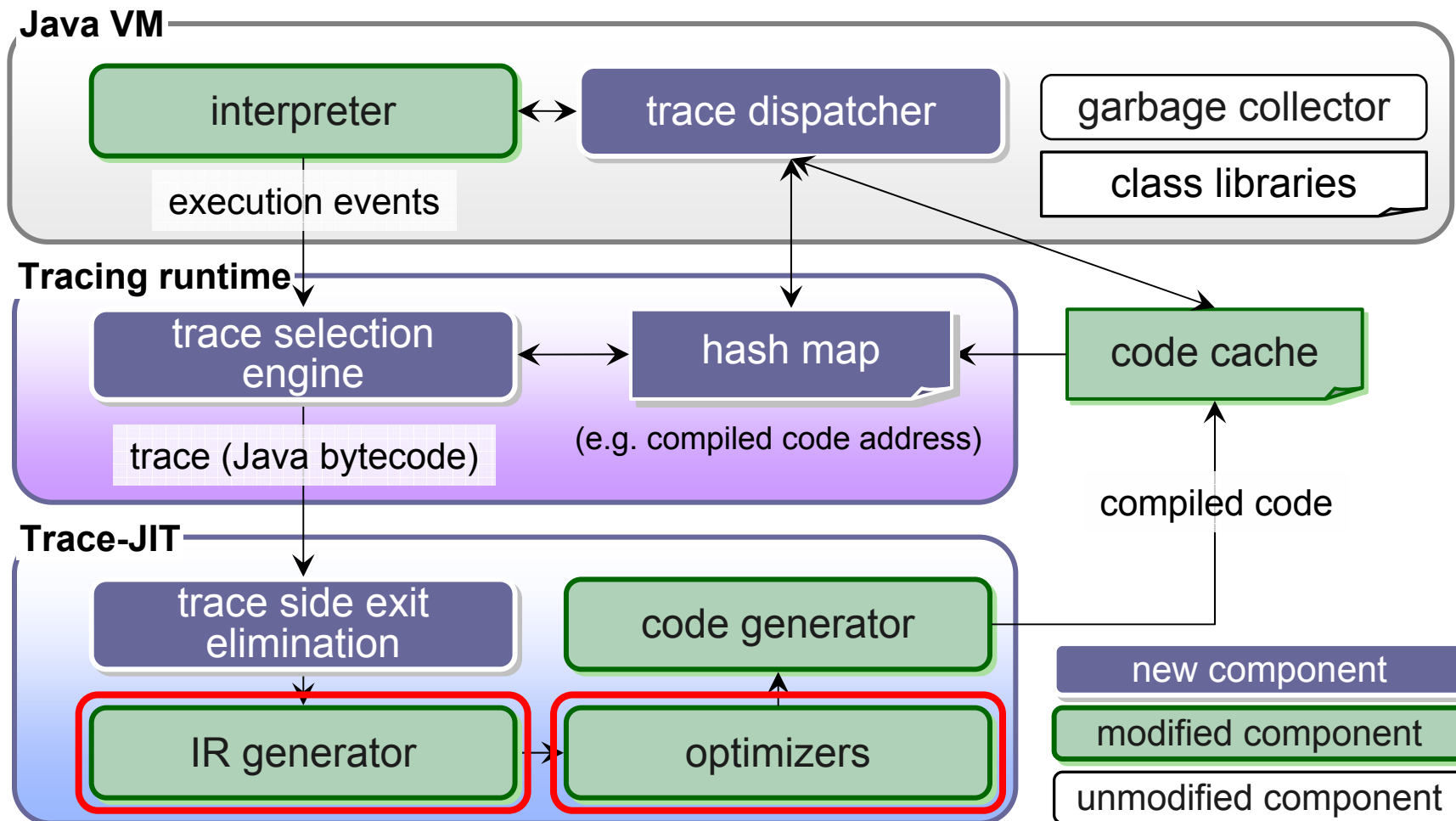
# Our Trace-JIT Architecture



# Our Trace-JIT Architecture



# Our Trace-JIT Architecture



# Technical challenge in reusing a method-based compiler for trace-JIT

## Scope mismatch problem

- In method-JIT,
    - local variables **must be dead** at the start and the end of compilation scope
  - In trace-JIT
    - local variables **may live** at the start and the end of compilation scope
- ➔ Live range of local variables does not match with compilation scope in trace-JIT

# Solving the scope mismatch problem

- *dead store elimination (DSE)* as an example

```

void prepend(e) {
  p = head;
  do {
    tail = p;
    p = p->next;
  } while (p != NULL);
  tail->next = e;
  e->next = NULL;
}

```

Is this dead store?  
(no use in the trace)

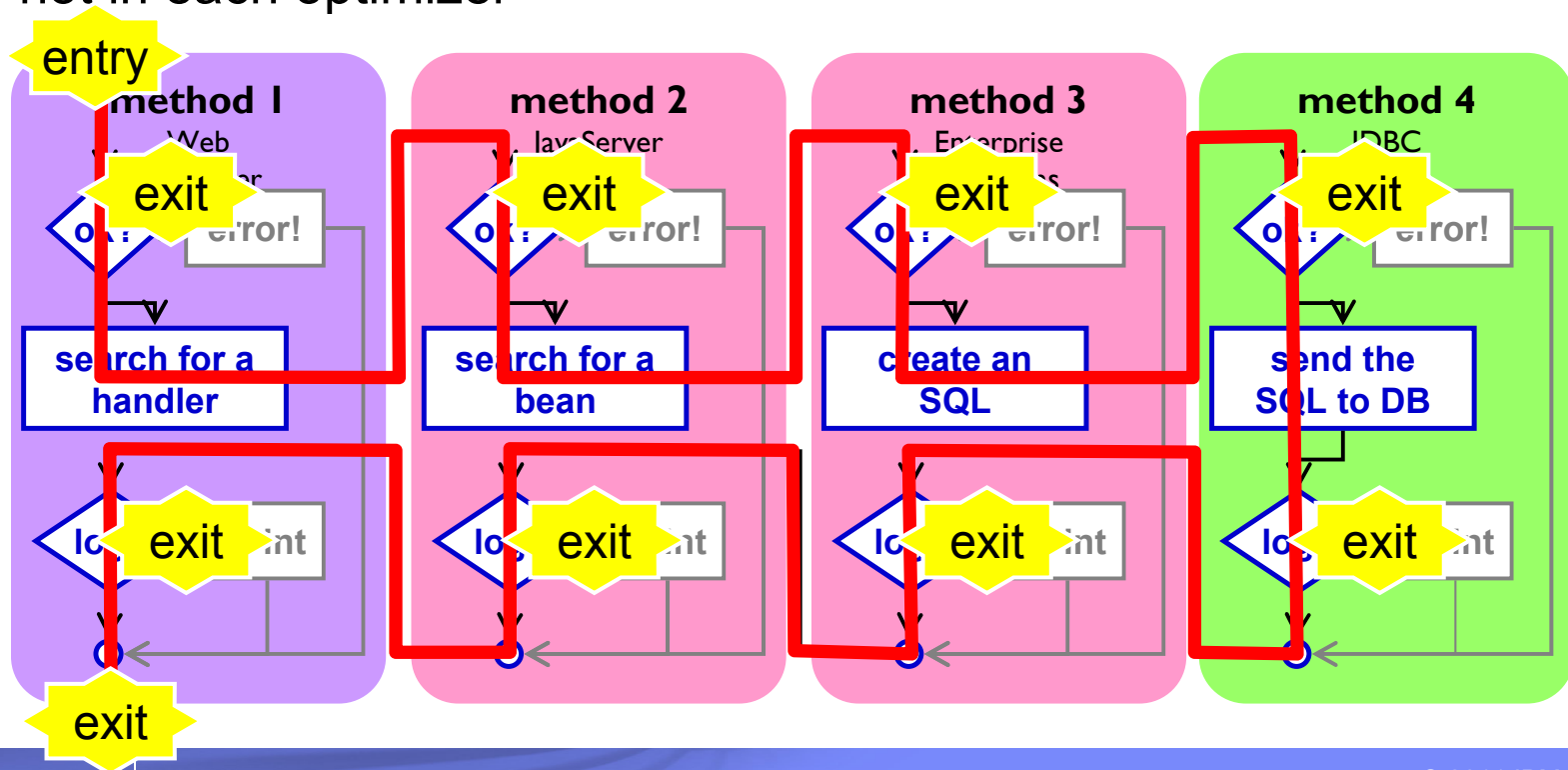
↓  
No!

↑  
↓  
compilation scope  
(= trace)

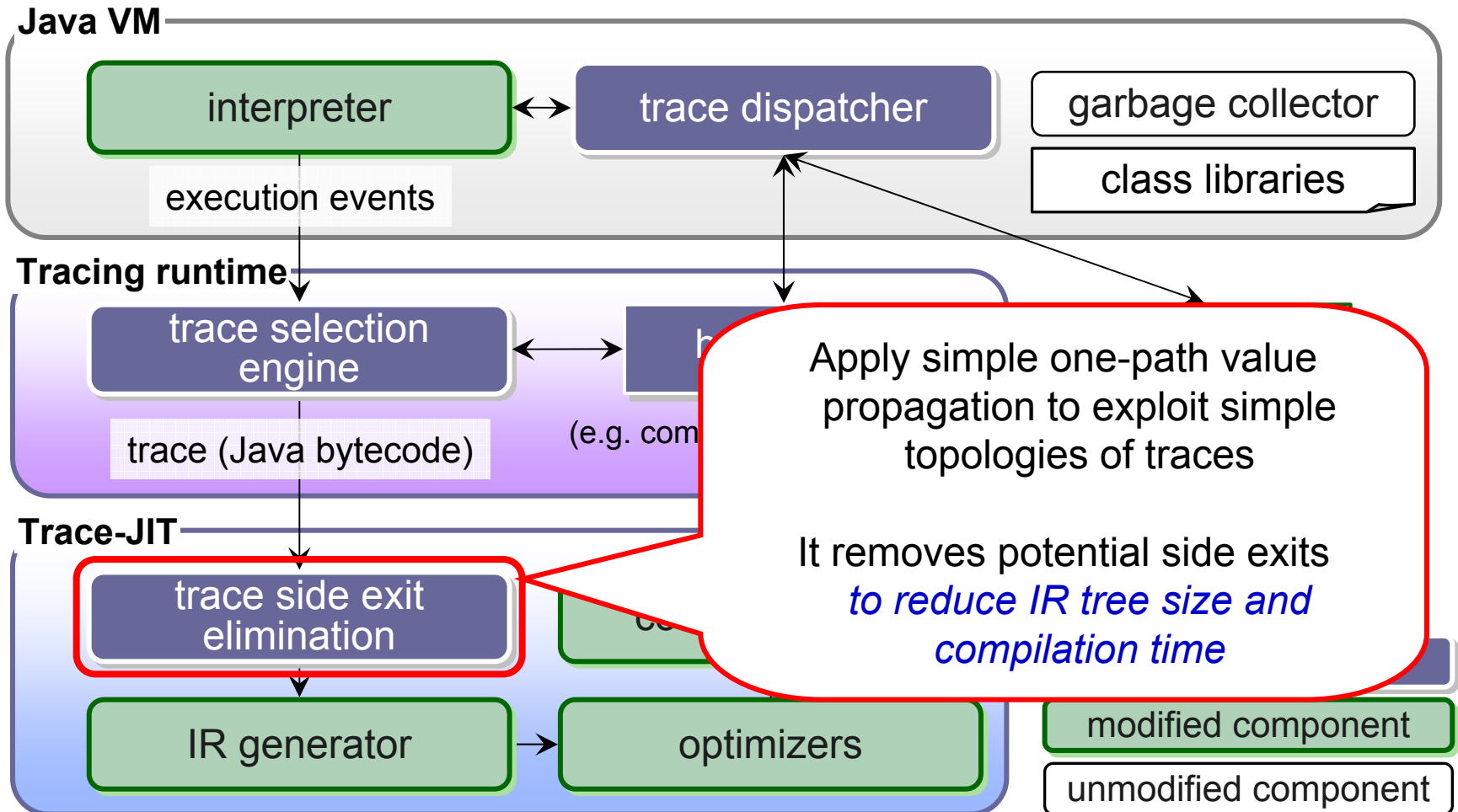
we analyze *outside* the  
compilation scope to identify  
liveness at the end of  
compilation scope

## Analyze outside the compilation scope

- We identify all live variables at each compilation scope boundary point
  - trace head, trace exit points
- For each boundary point, we analyze the method that includes the point
  - mostly in *live range analyzer* and *use-def analyzer* in the framework, not in each optimizer



# Our Trace-JIT Architecture



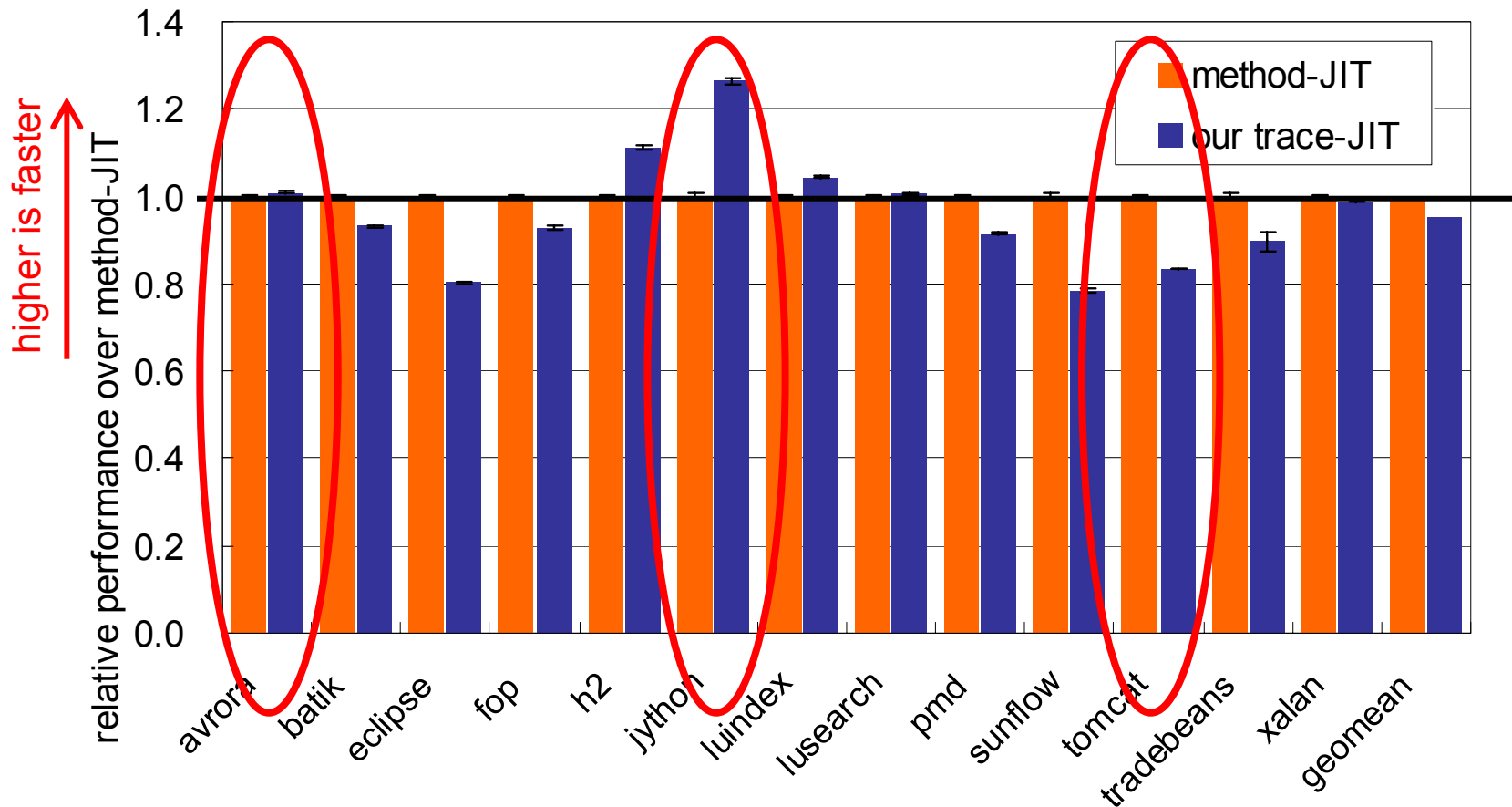
# Outline

- Motivation
- Background
- Trace-JIT Architecture
- **Performance Evaluation**
- Future work and Summary

# Performance Evaluation

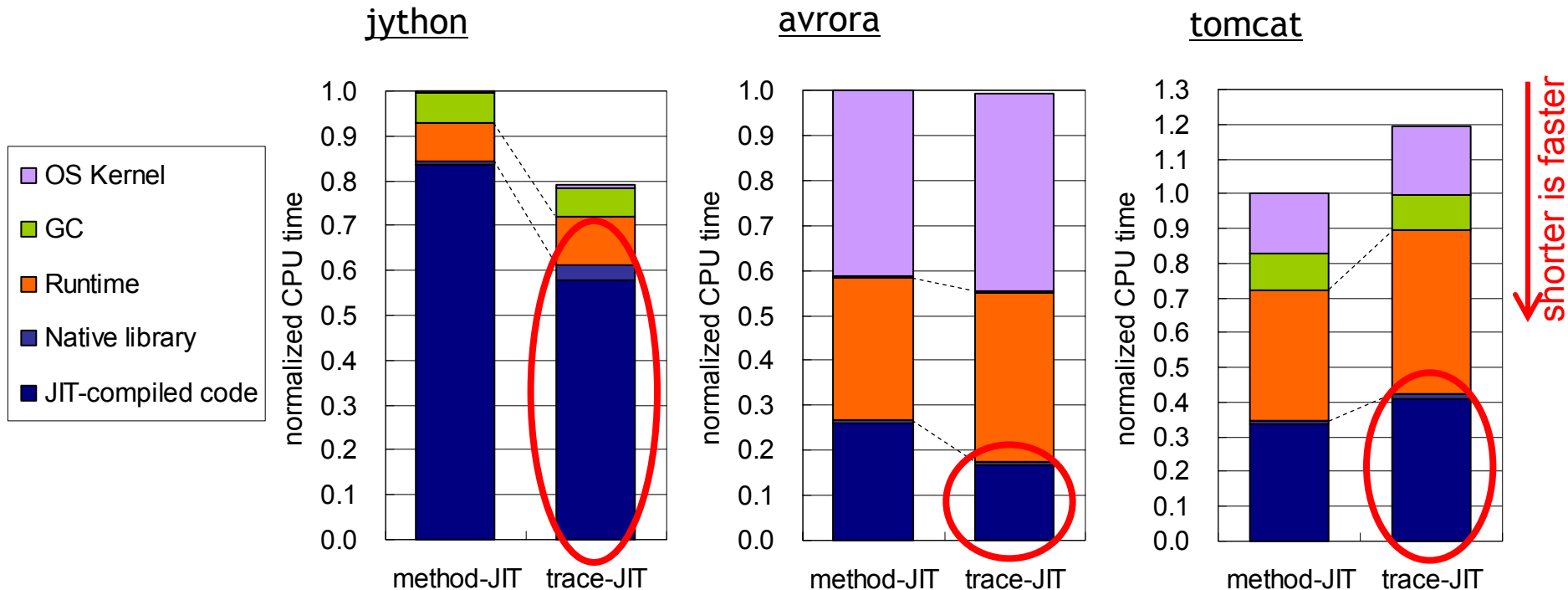
- Hardware: IBM BladeCenter JS22
  - 4 cores (8 SMT threads) of POWER6 4.0GHz
  - 16 GB system memory
- Software:
  - AIX 6.1
  - Method-JIT: IBM JDK for Java 6 (32 bit)
  - Trace-JIT: Our Trace-JIT based on the same IBM JDK
    - used only standard optimization level (-Xjit:optlevel=warm)
    - 512 MB Java heap with large page enabled
    - generational garbage collector (gencon)
- Benchmark:
  - DaCapo benchmark suite 9.12

# Steady state performance



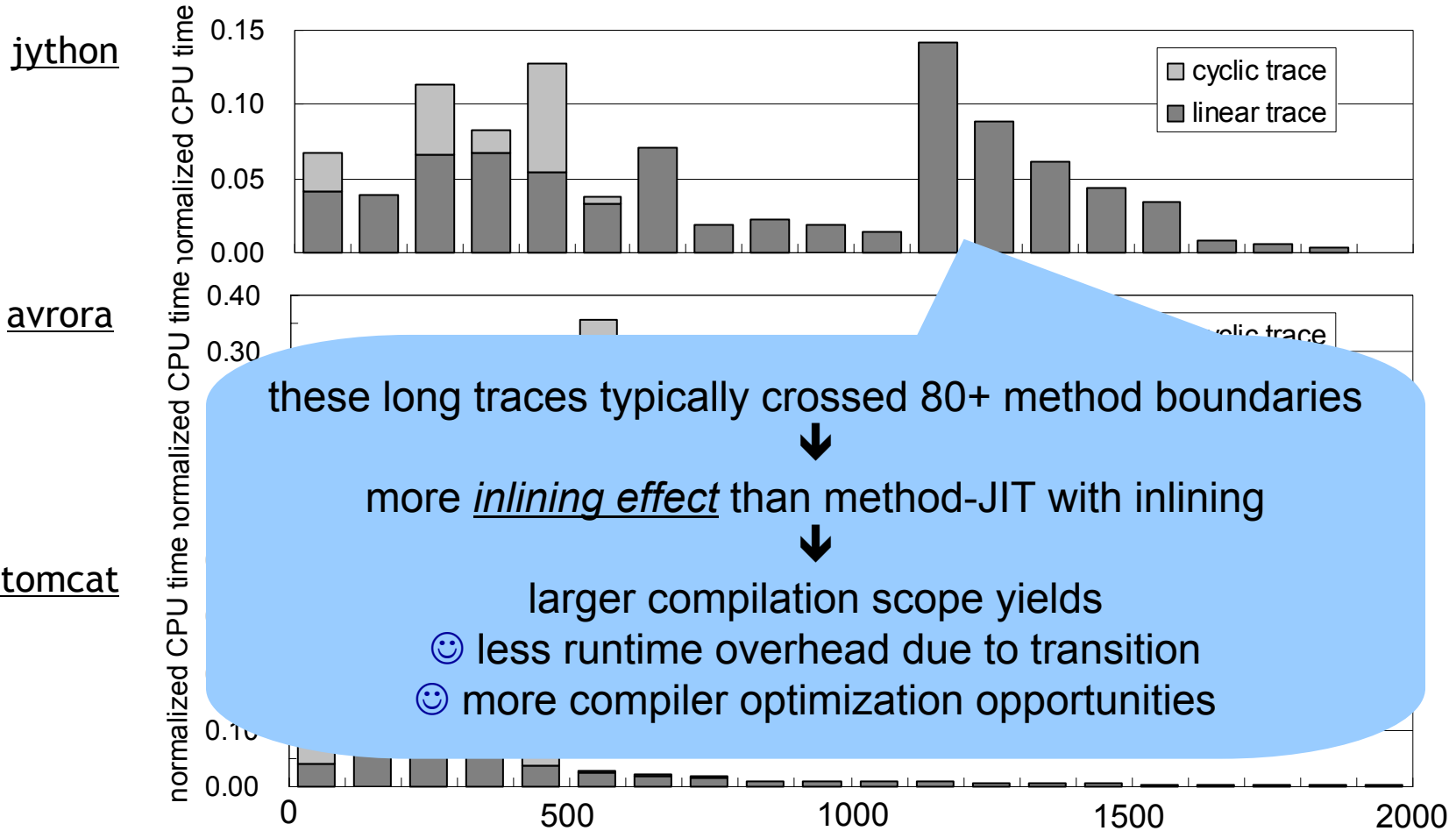
- Trace-JIT was 22% slower to 26% faster than method-JIT

# Execution time breakdown



- ☺ Trace-JIT often (not always) shows better JITted code performance (blue parts)
- ☹ Trace-JIT incurs larger **runtime overhead** (orange parts)

# Execution time breakdown by trace length



shorter trace ← trace length in number of Java bytecodes → longer trace

# Outline

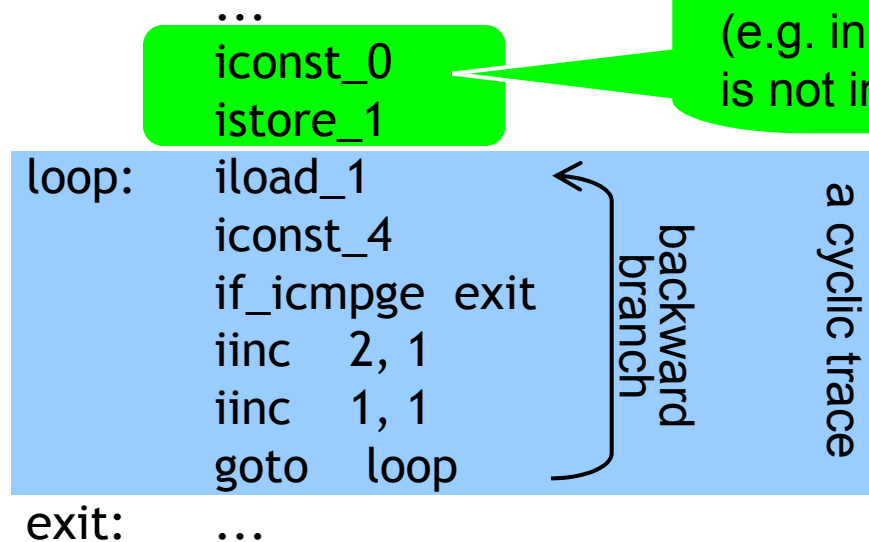
- Motivation
- Background
- Trace-JIT Architecture
- Performance Evaluation
- **Future work and Summary**

# Optimization opportunities and challenges

- Opportunities
  - Potentially larger compilation scope than method-JIT
  - Simple control flow
    - main path of a trace is a very large extended basic block
  - Explicit control flow
    - like method inlining
  - More specialization
    - type specialization, value specialization etc
  
- Challenges
  - Interaction between trace selection and optimizations
    - e.g. Loop optimizations

# Future work: Effective Loop Optimization in trace-JIT

- More loop optimizations in trace-JIT
  - backward-branch-based cyclic trace identification is not suitable for loop optimizations
- need to enhance trace selection algorithm to maximize the optimization opportunities



loop preheader  
(e.g. initialization of loop variable)  
is not included in a cyclic trace.

Java code:

```

for (int i=0; i<4; i++) {
    j ++;
}

```

# Summary

- We implemented trace-based Java JIT compiler based on the existing method-based JIT compiler
  - handling scope mismatch problem
  - reducing runtime overhead
- Our trace-JIT achieved almost comparable performance to mature method-JIT with almost same set of optimizations
  - better JITted code performance in trade for larger runtime overhead
  - generating longer trace is a key to superior performance

Refer to the paper for

- ✓ our new runtime overhead reduction techniques
- ✓ more detailed comparisons including code size, compilation time and so on