IBM Research - Tokyo

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

Hiroshi Inoue[†], Hiroshige Hayashizaki[†], Peng Wu[‡] and Toshio Nakatani[†]
† IBM Research – Tokyo
‡ IBM Research – T.J. Watson Research Center

Goal and Motivation

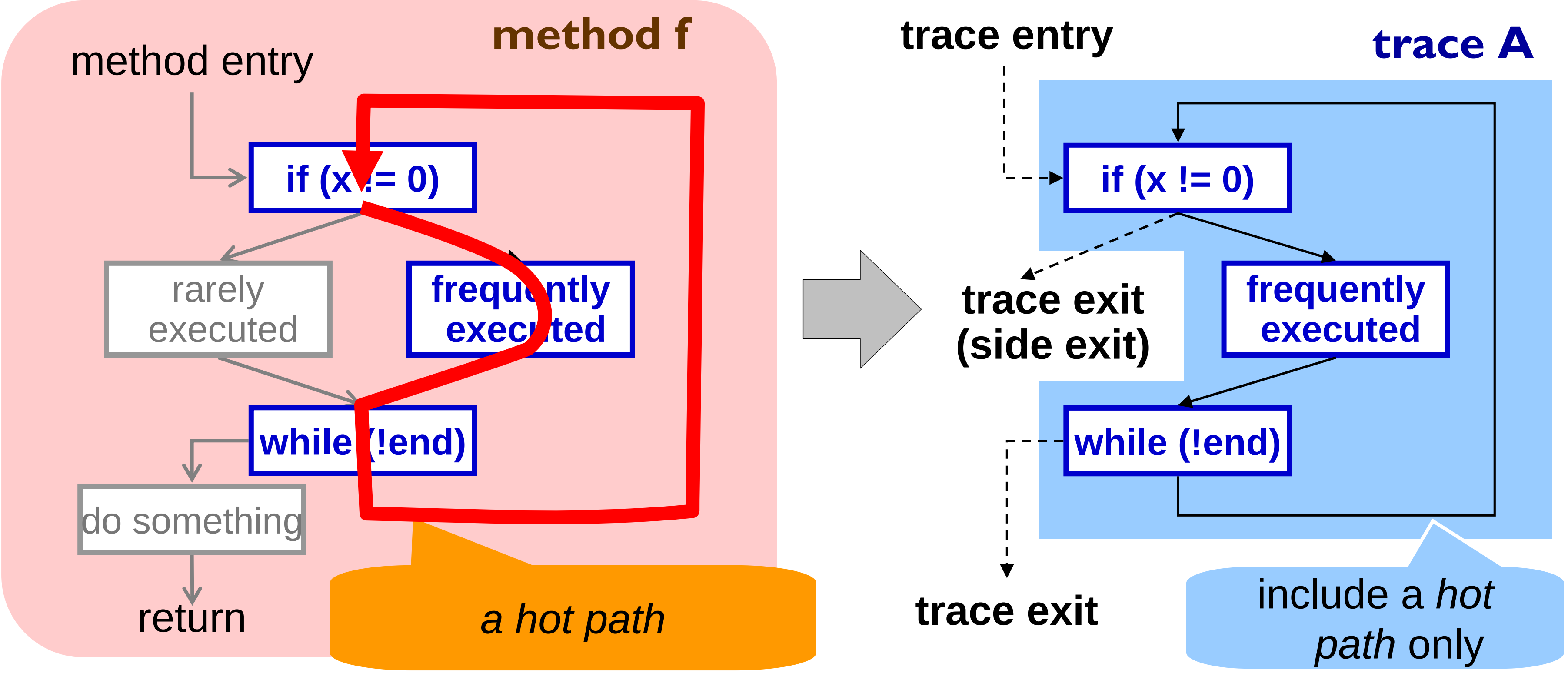
- Goals
  1. develop an efficient *trace-based Java JIT compiler (trace-JIT)* based on existing mature *method-based JIT compiler (method-JIT)*
  2. understand the benefits and drawbacks of the trace-JIT against the method-JIT

- 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?

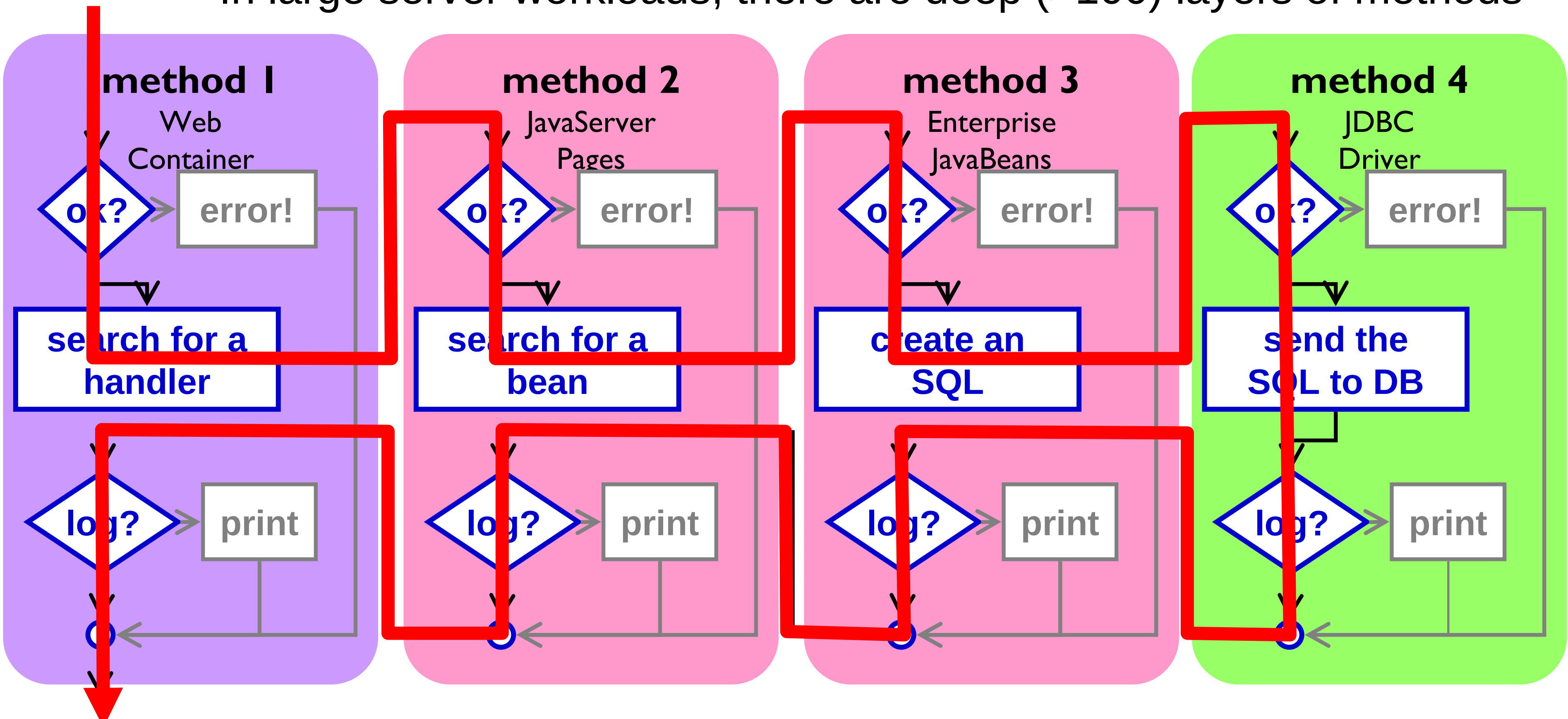
Background: Trace-based Compilation

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



Motivating Example

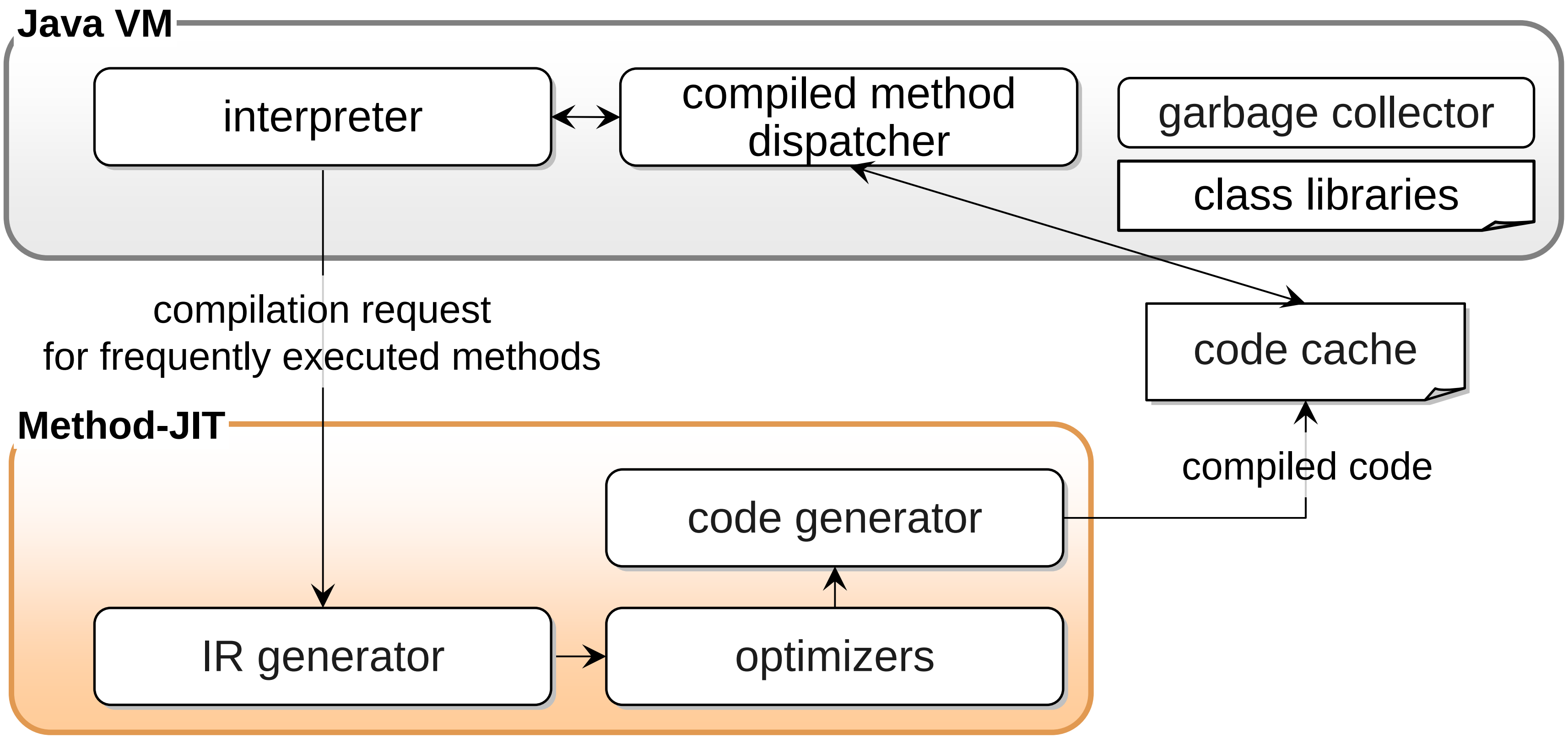
- A trace can span multiple methods
  - Free from method boundaries
  - In large server workloads, there are deep (>100) layers of methods



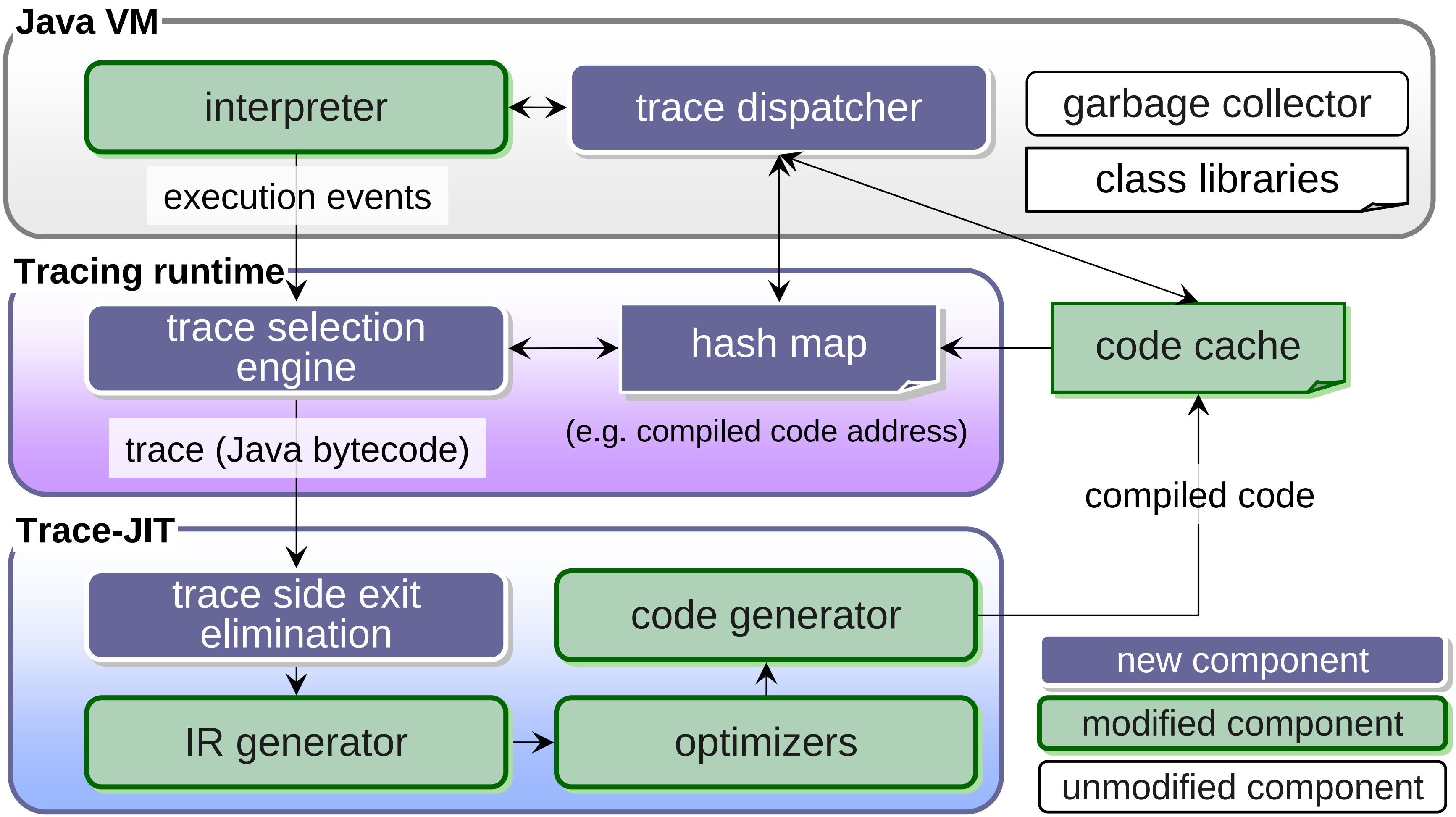
Outline

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

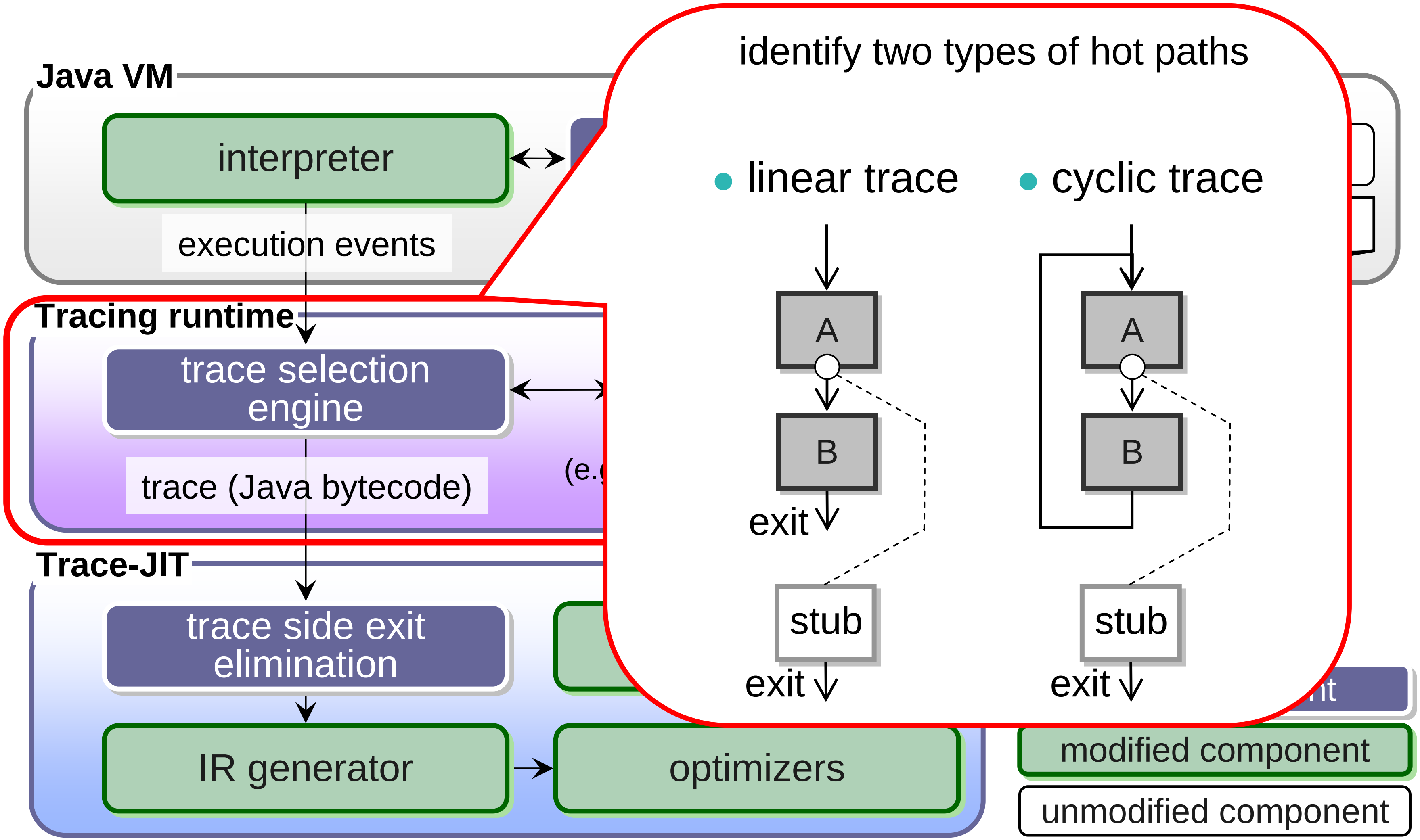
Baseline Method-JIT Components



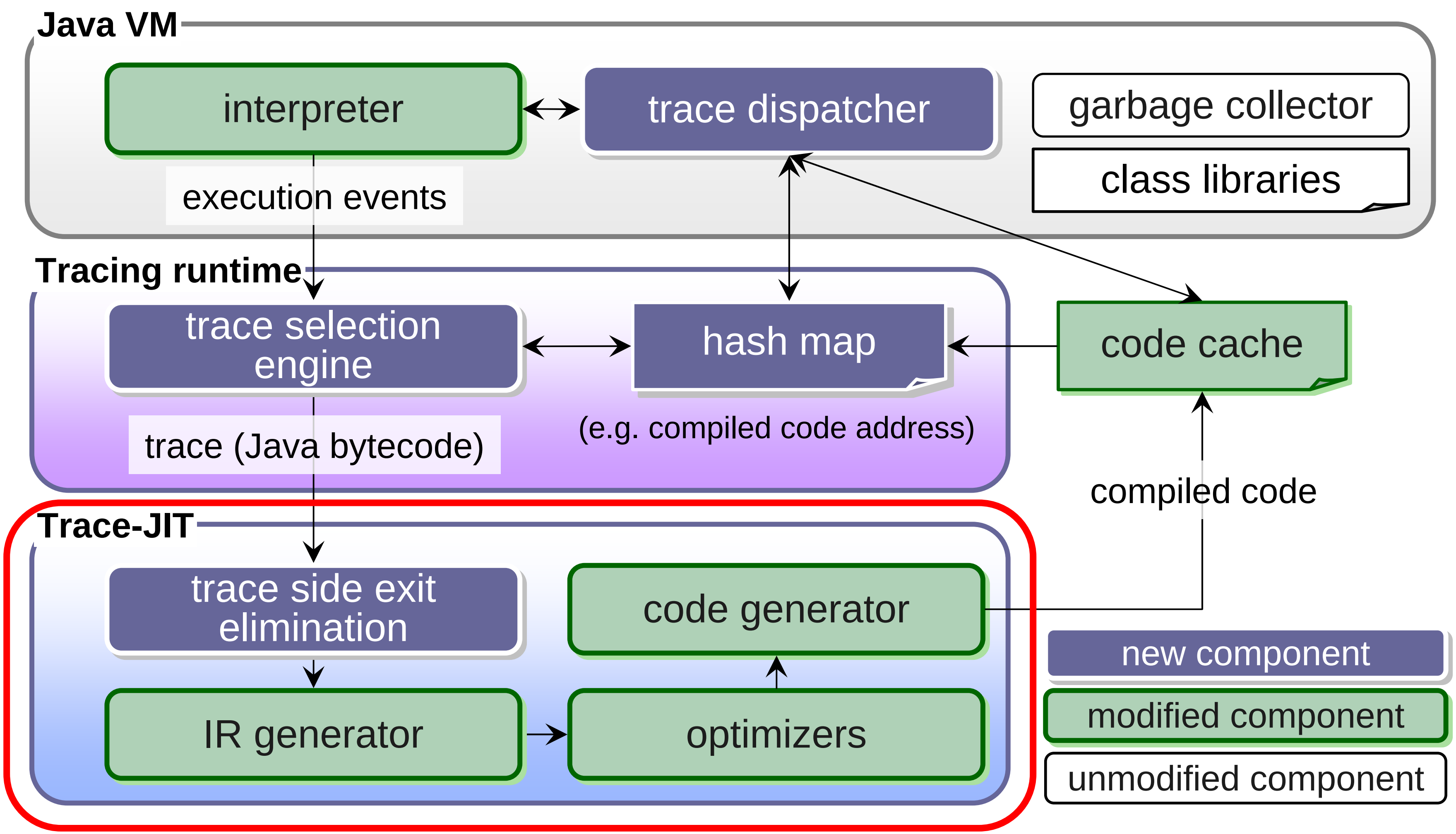
Our Trace-JIT Architecture



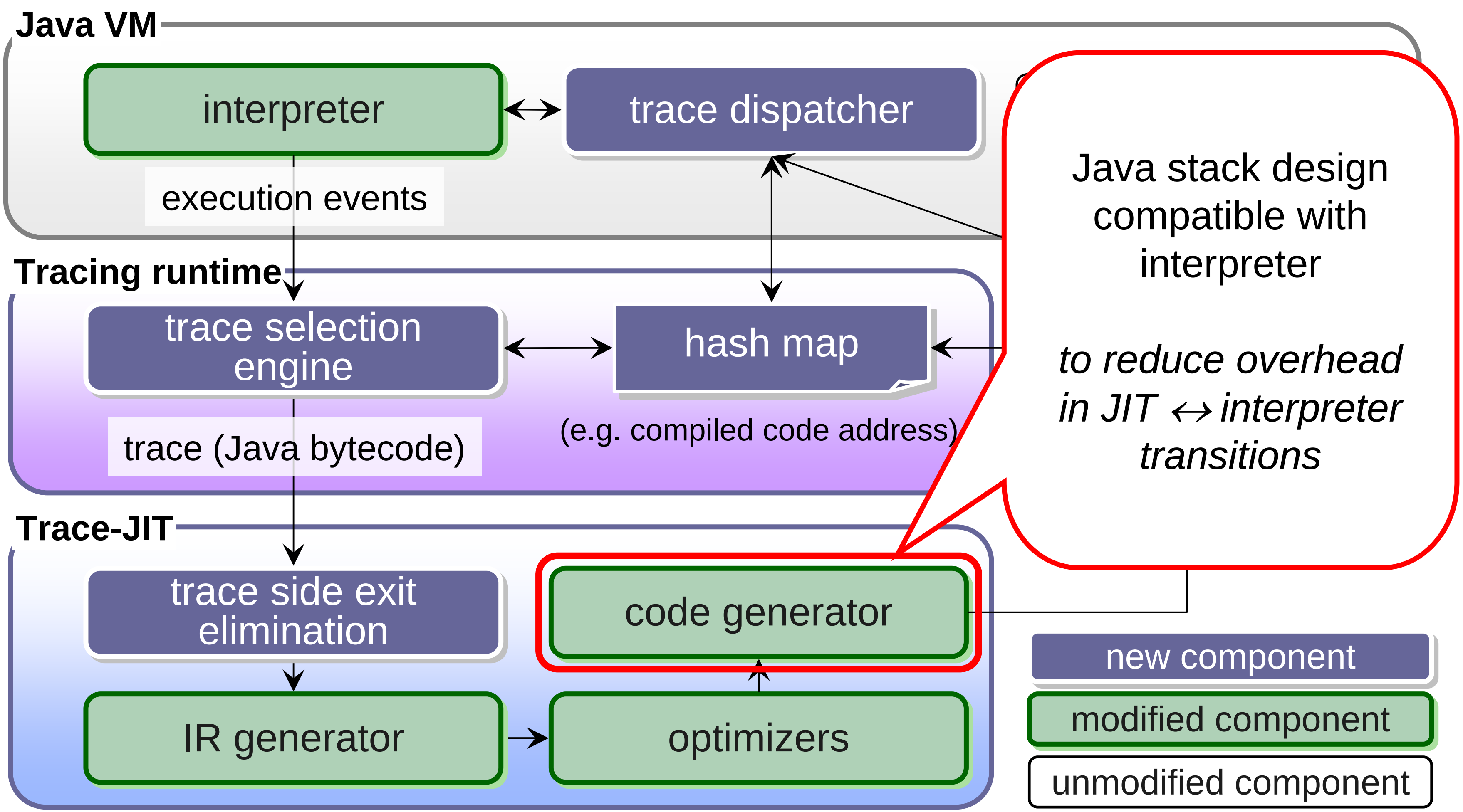
Our Trace-JIT Architecture



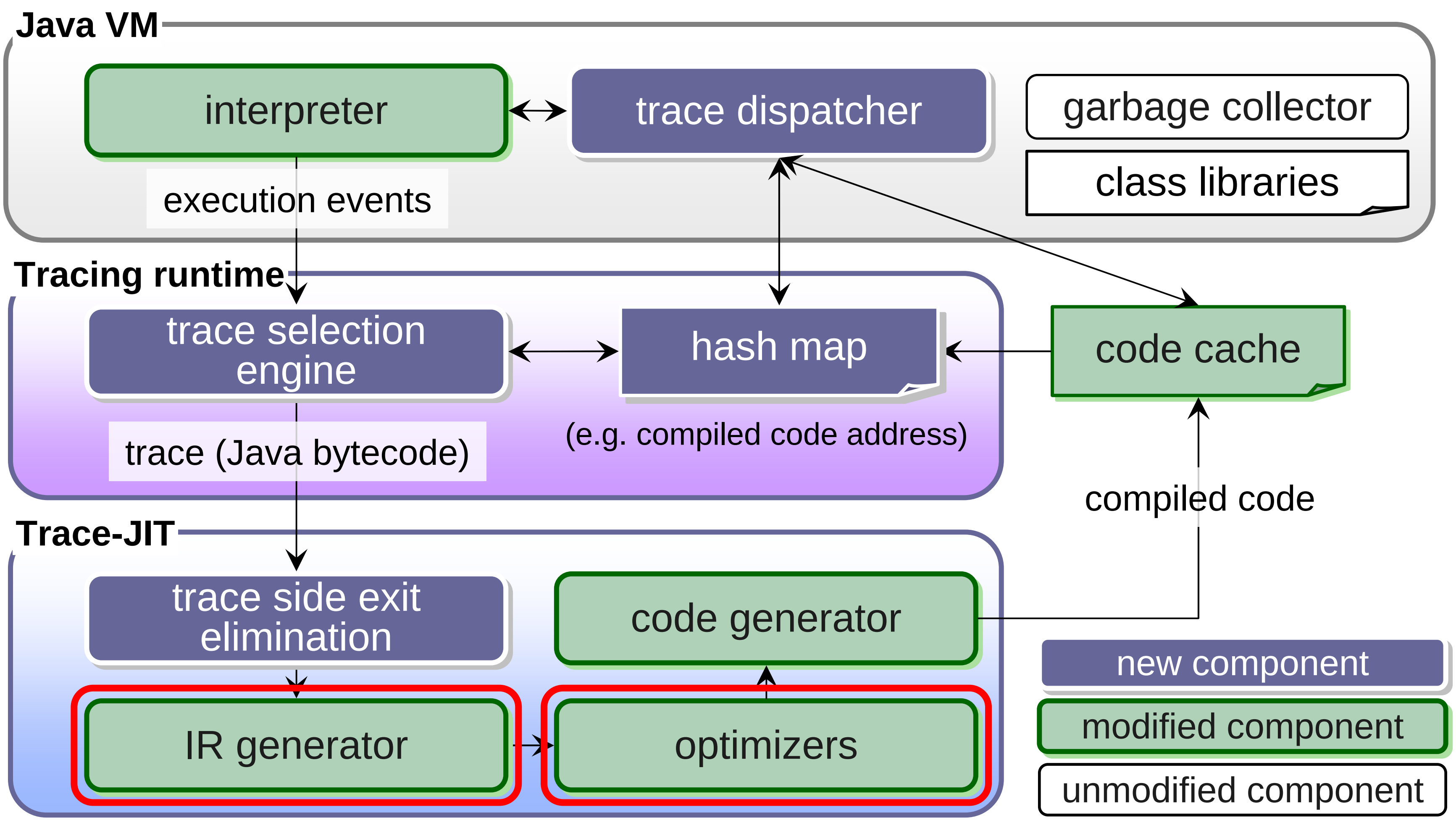
Our Trace-JIT Architecture



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

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

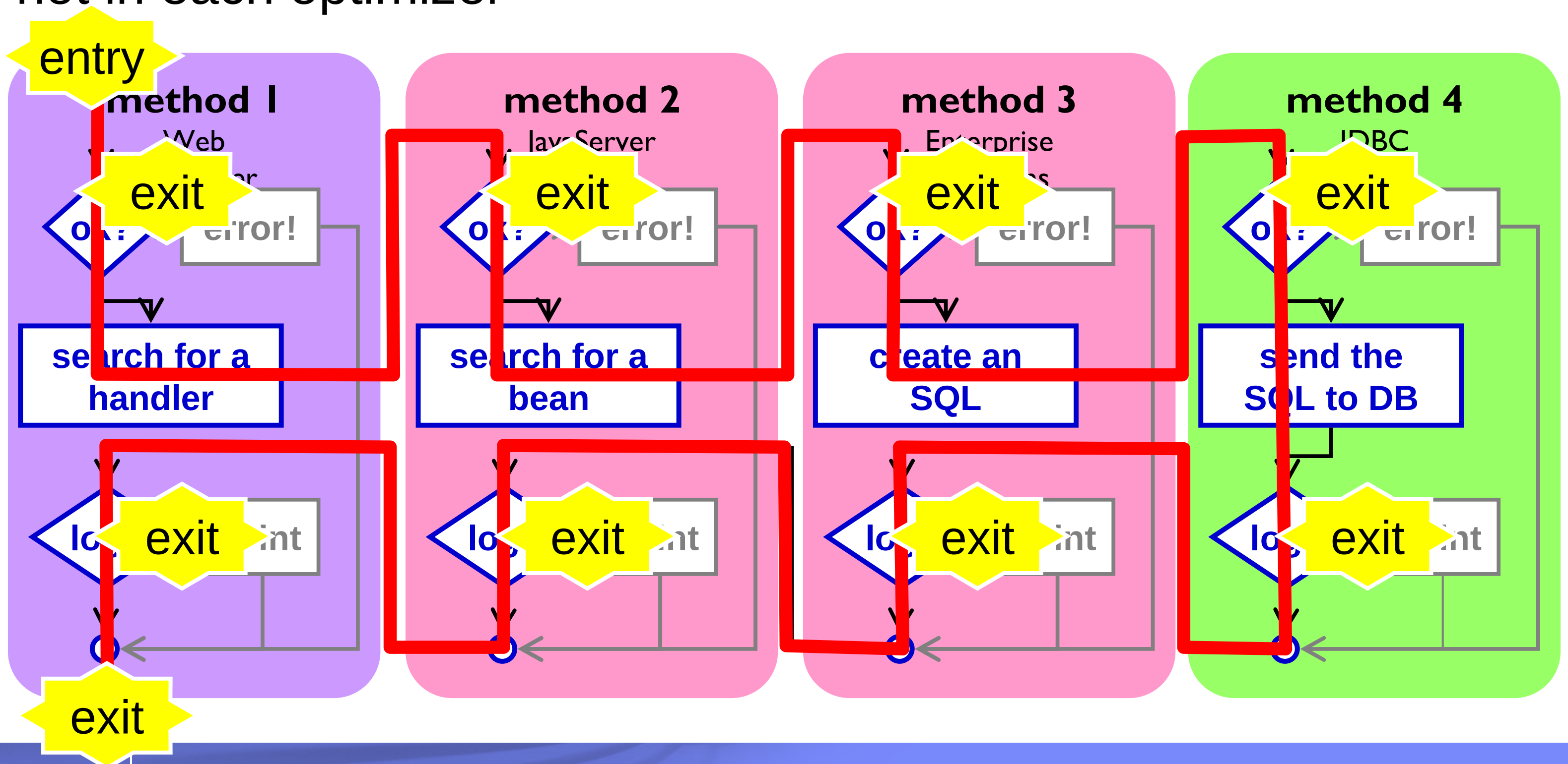
```
void prepend(e) {
    p = head;
    do {
        tail = p;
        p = p->next;
    } while (p != NULL);
    tail->next = e;
    e->next = NULL;
}
```

compilation scope
(= trace)

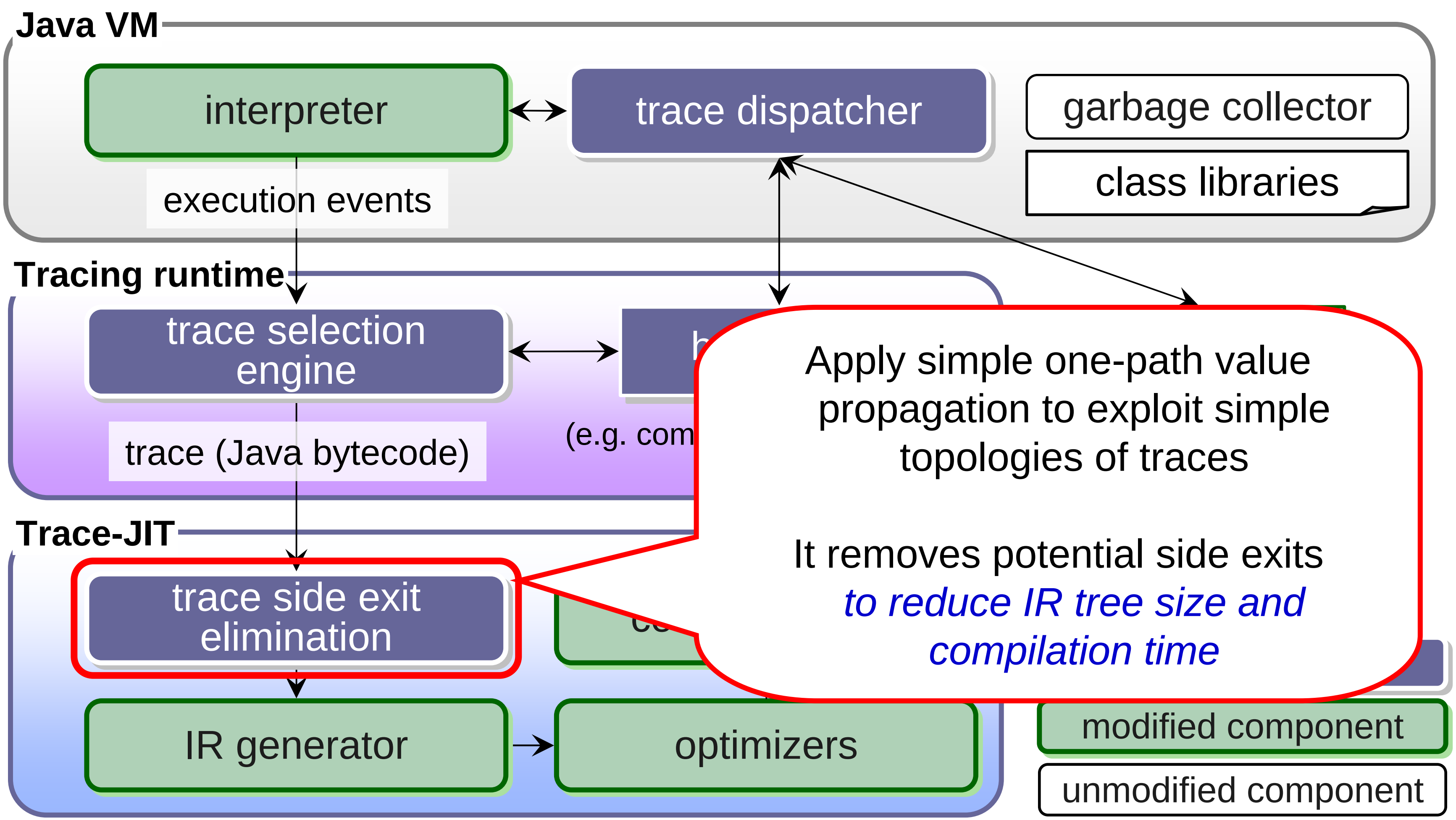
we analyze *outside* the compilation scope to identify liveness at the end of compilation scope

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



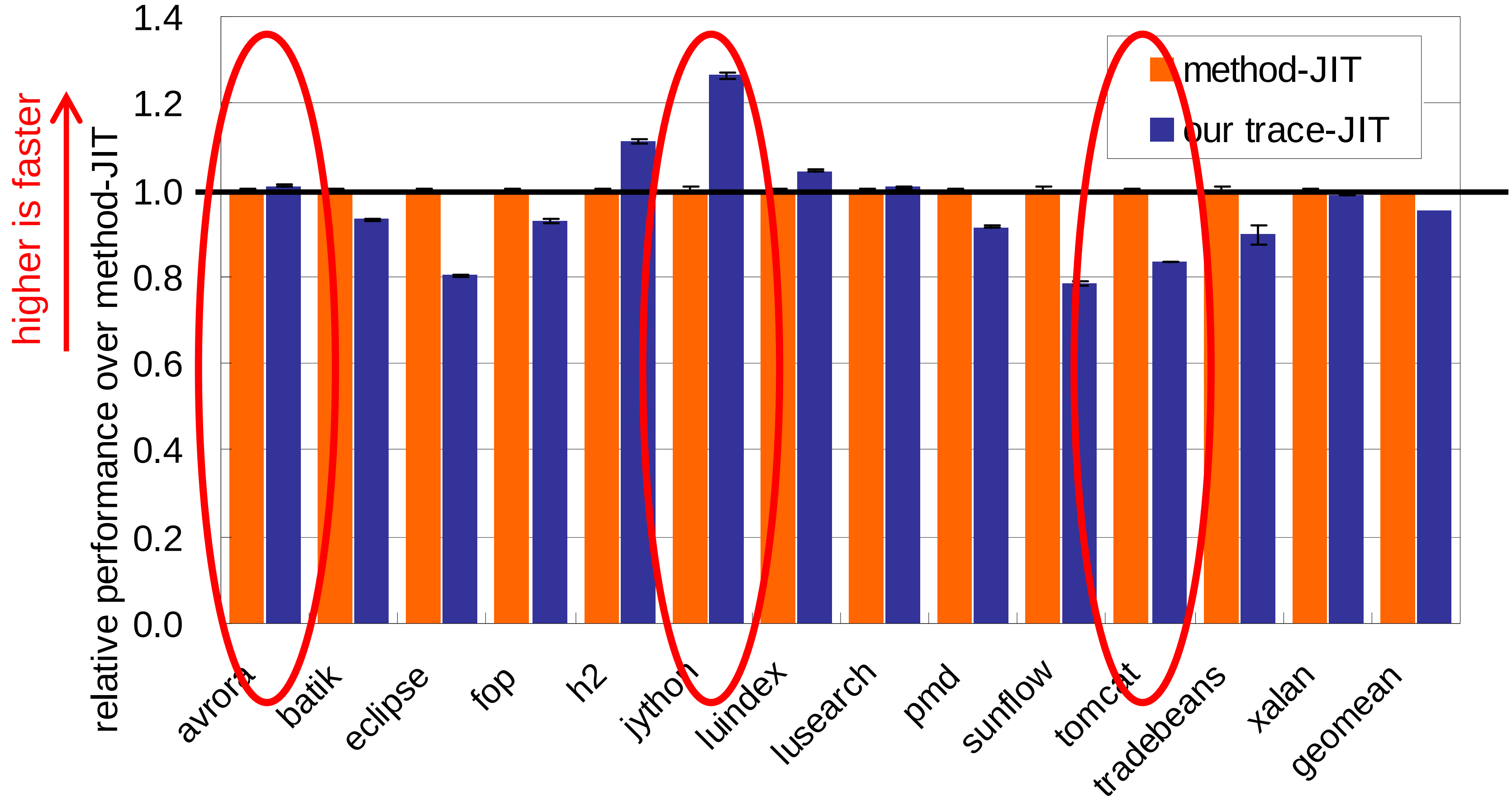
Outline

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

Performance Evaluation

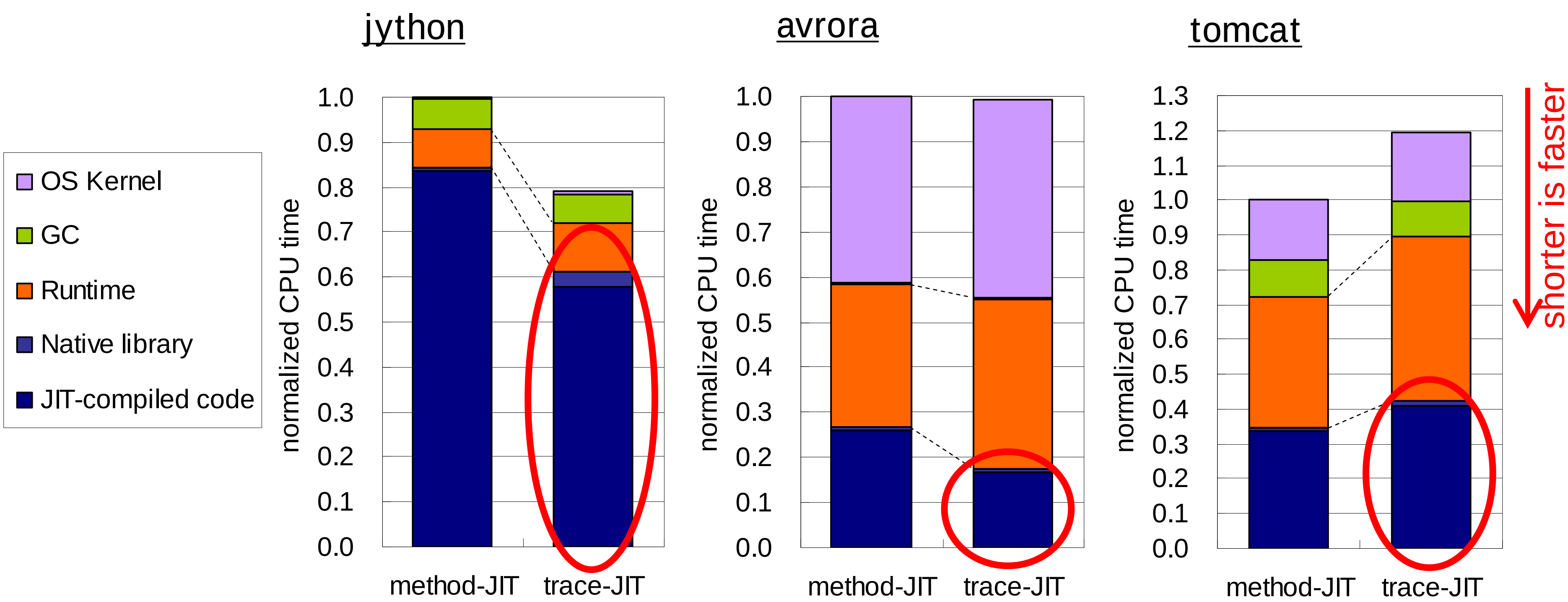
- Hardware: IBM BladeCenter JS22
  - 4 cores (8 SMT threads) of POWER6 4.0GHz
  - 16 GB system memory
- Software:
  - AIX 6.1
  - Method-JIT: IBM JDK for Java 6 (32 bit)
  - Trace-JIT: Our Trace-JIT based on the same IBM JDK
    - used only standard optimization level (-Xjit:optlevel=warm)
    - 512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

Steady state performance



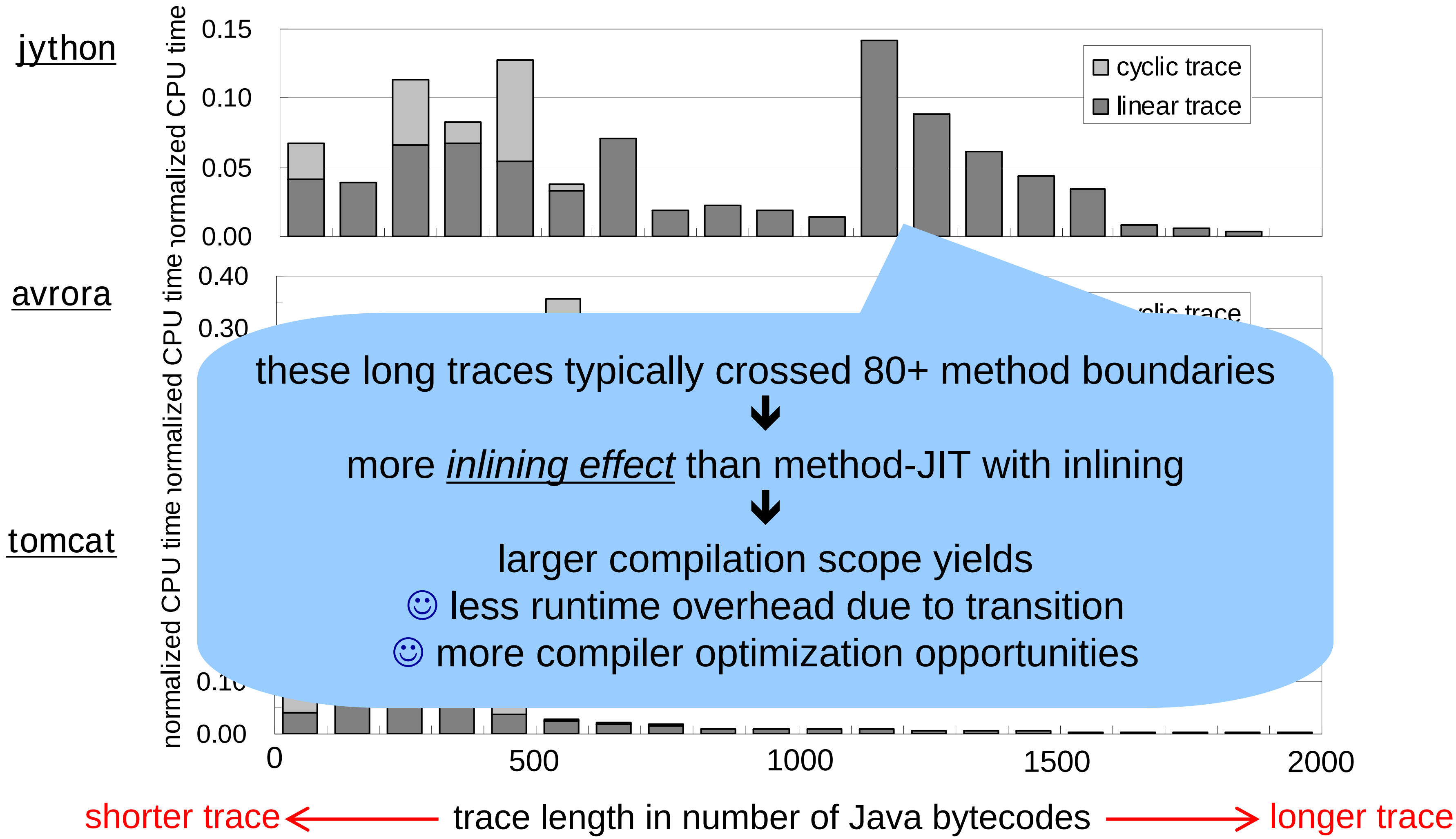
- Trace-JIT was 22% slower to 26% faster than method-JIT

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



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*

```
         ...
         iconst_0
         istore_1
loop:    iload_1
         iconst_4
         if_icmpge  exit
         iinc    2, 1
         iinc    1, 1
         goto    loop
exit:    ...
```

backward branch

a cyclic trace

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 ++;
}
```

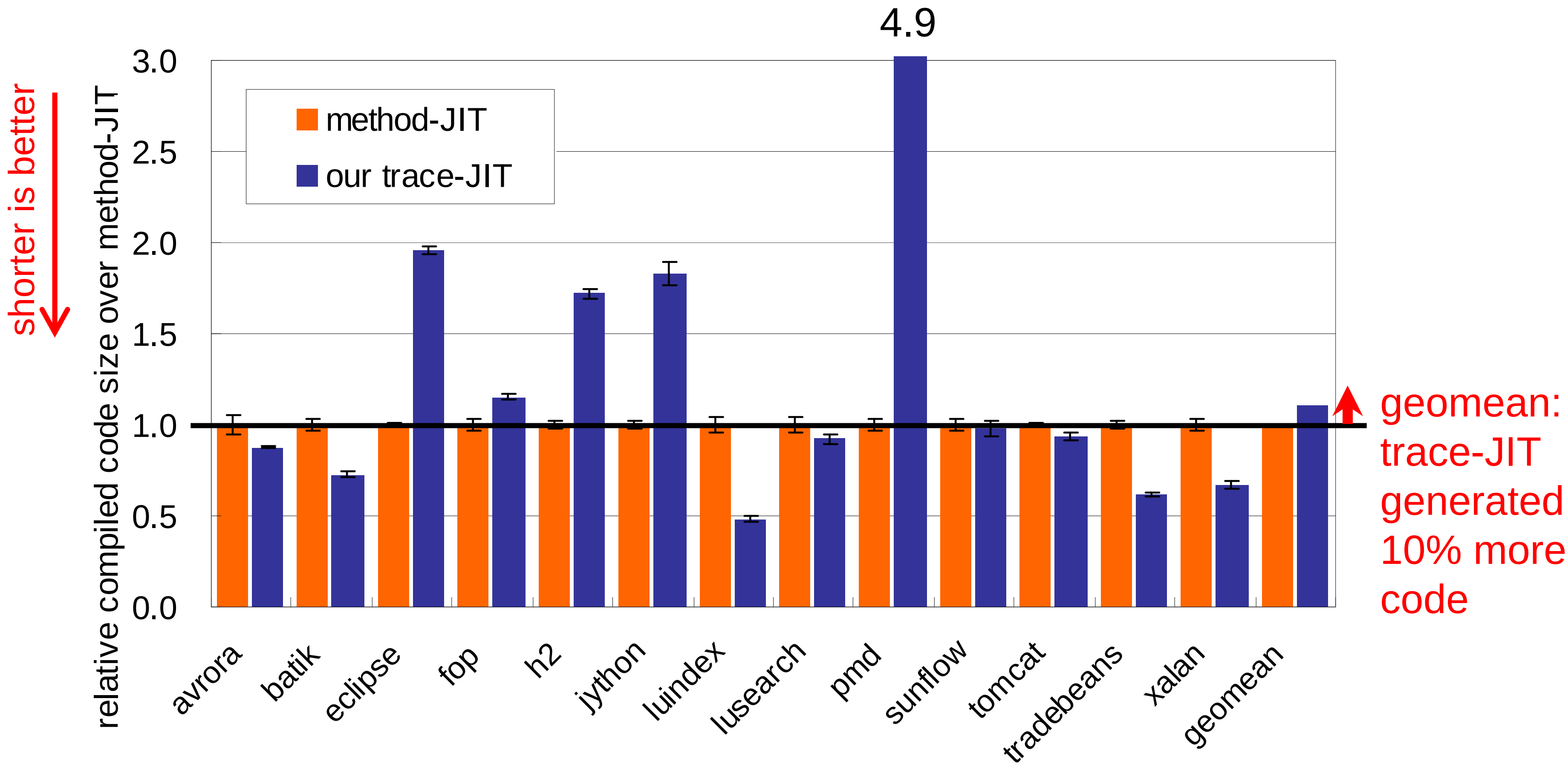
Summary

- 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 opt 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

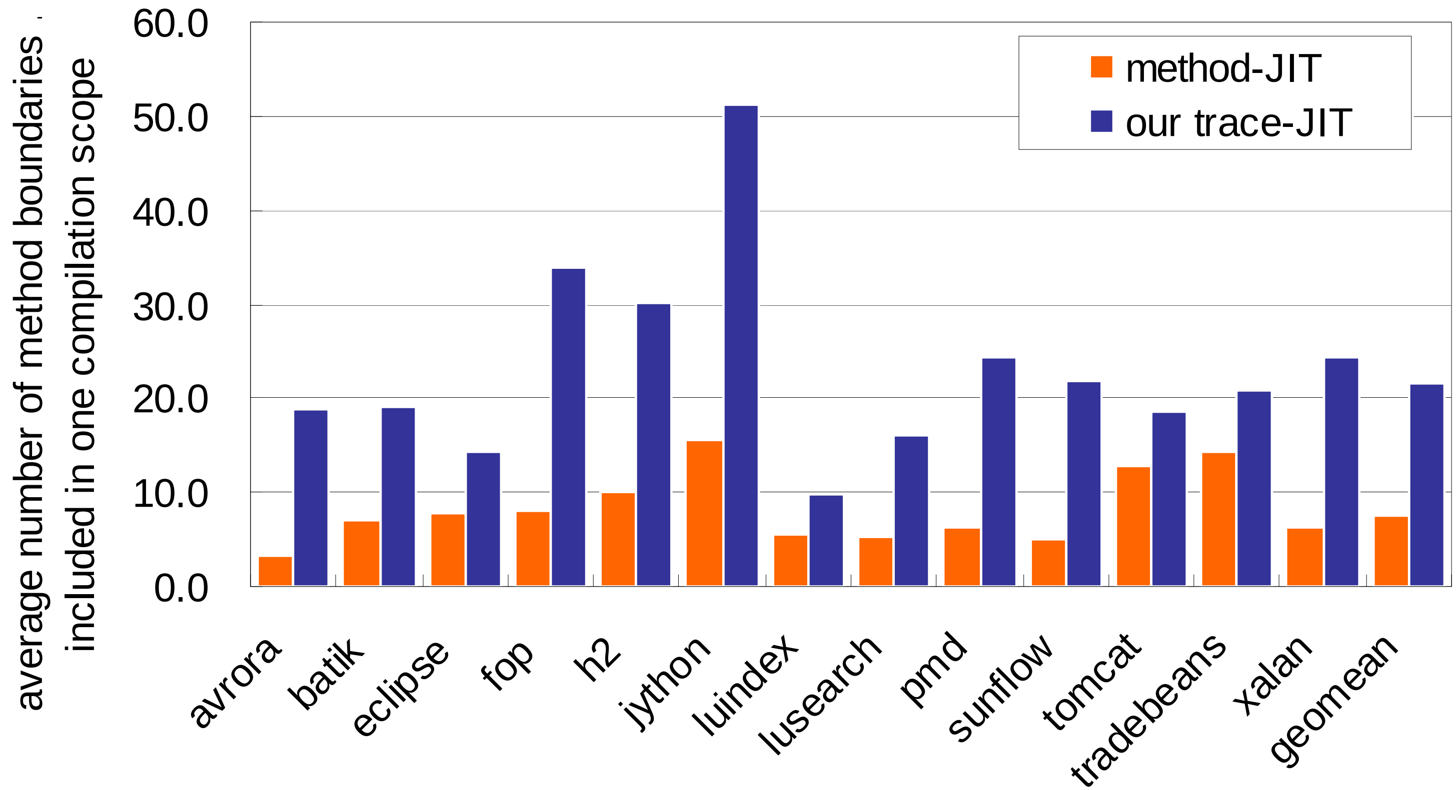
backup

Compiled code size



- ☹ The larger code size was mainly caused by the duplicated codes among traces
- ☺ On the positive side, traces include only frequently executed code sequence

Inlining effect of trace JIT
(number of method boundaries included in one trace)



Trace-JIT provides larger compilation scope than method-JIT with inlining

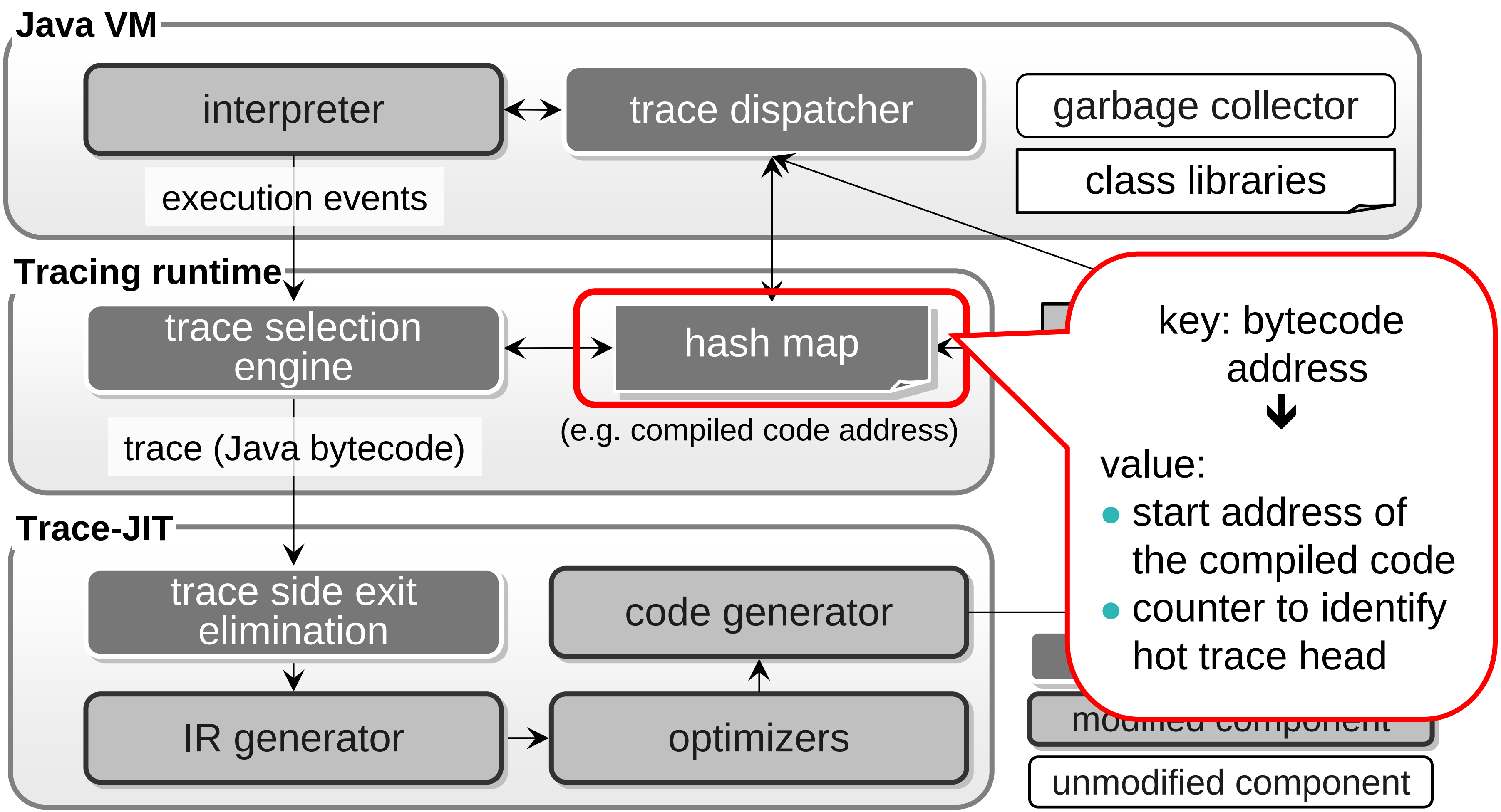
☺ Less method invocation overhead, more compiler optimization opportunities

☹ Potentially larger JITted code size due to duplicated code among traces

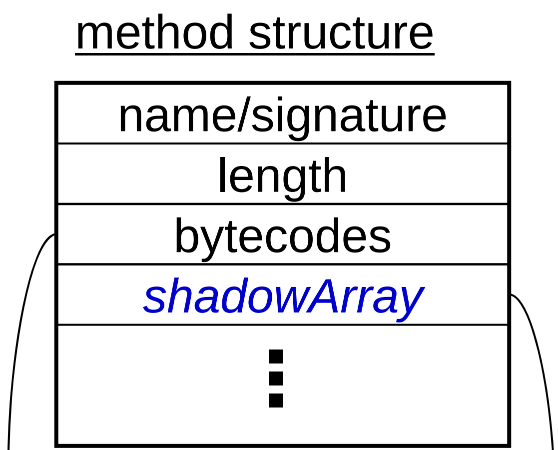
Our techniques to reduce overhead

- Hash Lookup Reduction Using a Shadow Array
  - Allocate a shadow array for each method to store information corresponding to each bytecode (e.g. start address of compiled trace starting from that bytecode)
  - Lookup the shadow array instead of slow global hash map
- JNI inclusion
  - Include certain JNI methods into traces and call JNIs from traces directly
  - Reduced trace enter/exit overhead
  - Some recognized JNI methods are further optimized (inlined)

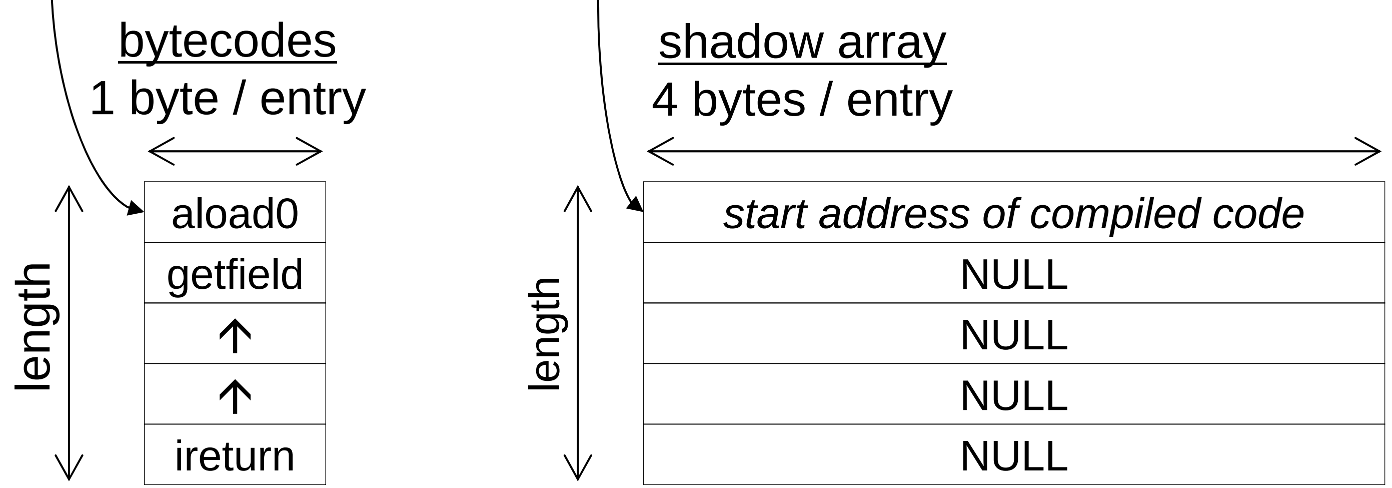
Our Trace-JIT Architecture



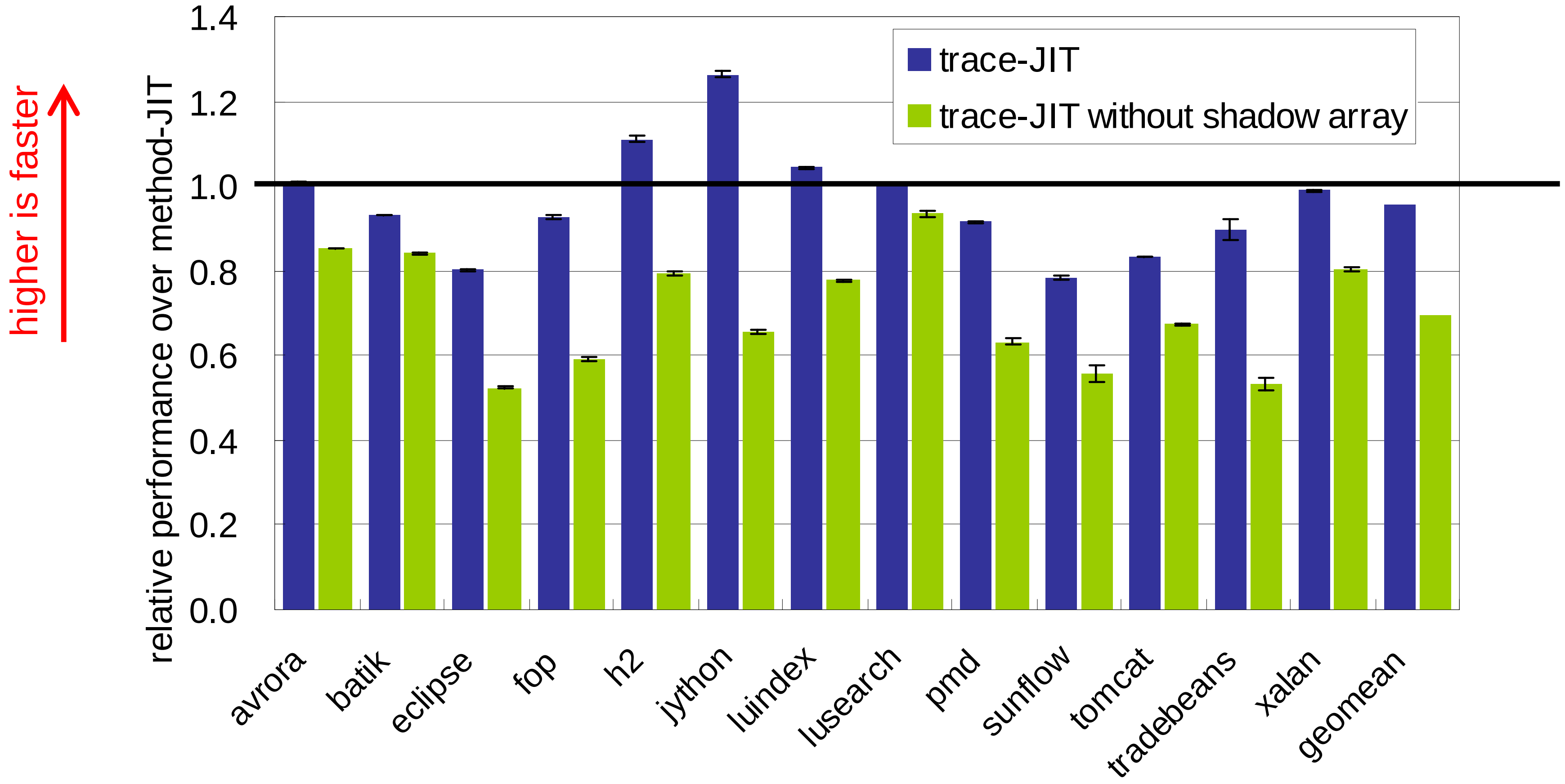
Shadow Array



- shadow array is allocated for each method lazily (we allocate it on demand)
- the shadow array entry corresponding to a bytecode index can be easily found without hash calculation.

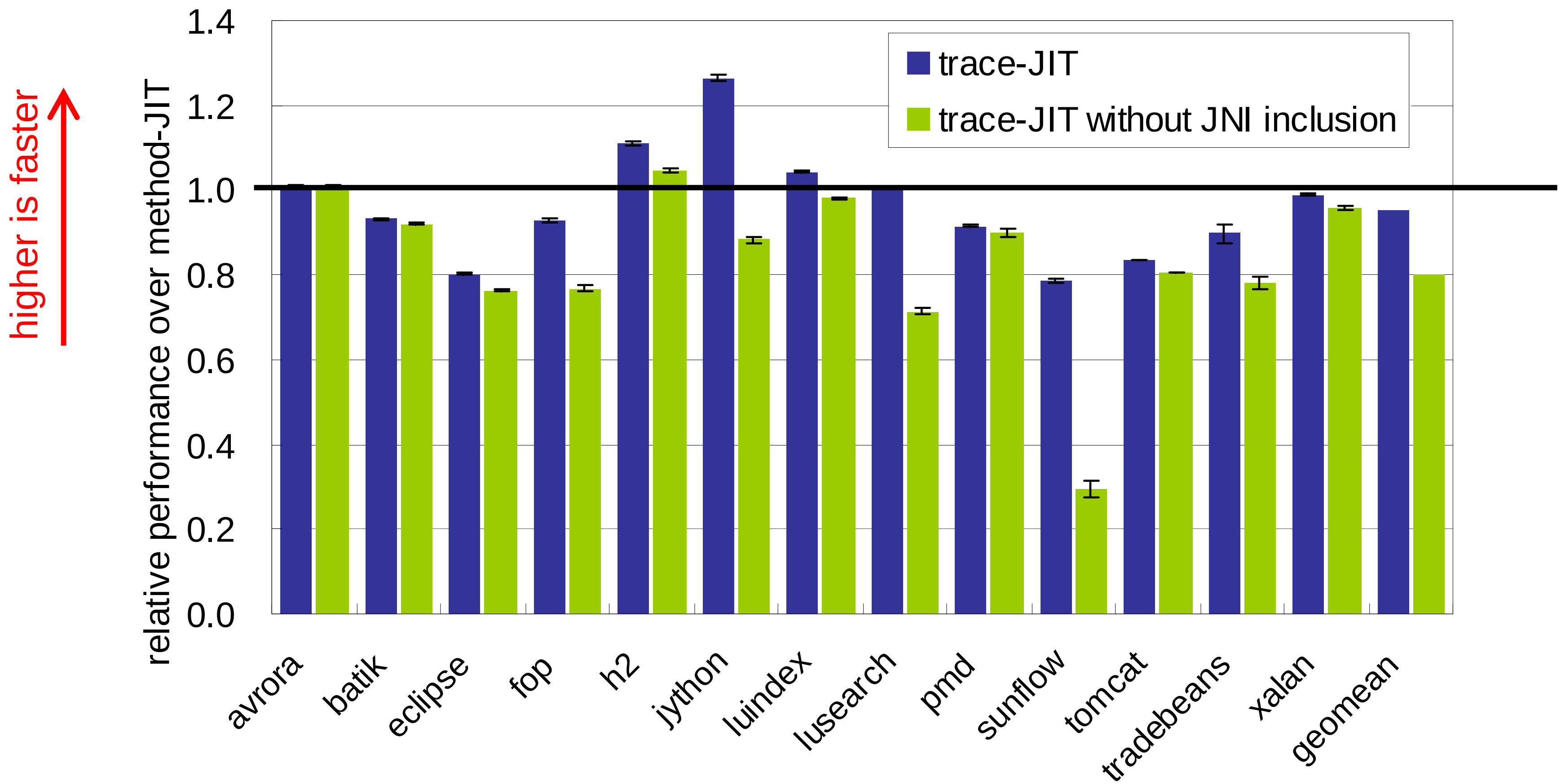


Effect of hash lookup reduction using a shadow array



- improved performance by 27.4% on average
- using additional memory space: 1.3 MB on average and up to 6.8 MB (tomcat)

Effect of JNI inclusion



- improved performance by up to 2.7x (for sunflow) and about 15% on average

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 or recursion)
   - executes a backward branch
   - calls or returns to a JNI (native) method
   - throws an exception
   - reaches pre-defined maximum length (128 basic blocks)

Trace exit handling

- We exit the current trace when:
  - the execution takes different path at a conditional branch or a switch
  - the target of a virtual method invocation is different from that of the recording time
  - the target of a method return is different from that of the recording time
  - an exception was thrown
- At a trace exit,
  - we assure the JVM state (e.g. Java stack, instruction pointer) is compatible with that of the interpreter
  - we fall back to interpreter or directly jump into the next compiled trace if already exists (trace linking)

Back ground: Existing Trace-based Compilers backup

- Binary translators and optimizers
  - Dinamo, Dinamo Rio, Strata

  ➔trace-JIT is used because no method structure is available

- JIT for dynamic scripting languages
  - TraceMonkey, SPUR, PyPy, Lua-JIT

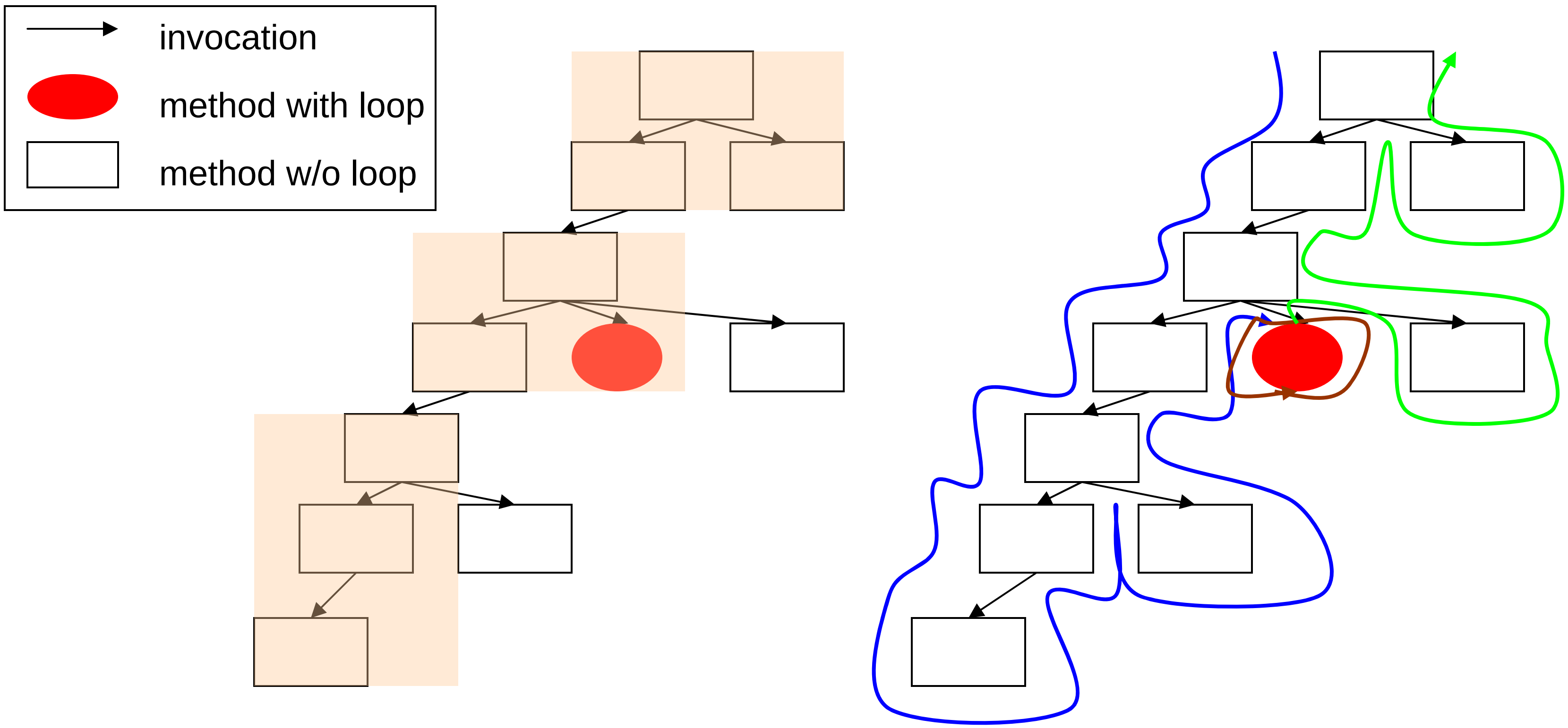
  ➔trace-JIT can provide more type specialization opportunity

- JIT for Java
  - Hotpath VM, YETI, Maxpath

  ➔smaller resource requirement, ease of JIT development

Trace Selection vs. Method Inlining

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



Method (partial) inlining forms hierarchical regions

Trace selection forms contiguous regions

– blue, brown, green

Our Trace-JIT Architecture