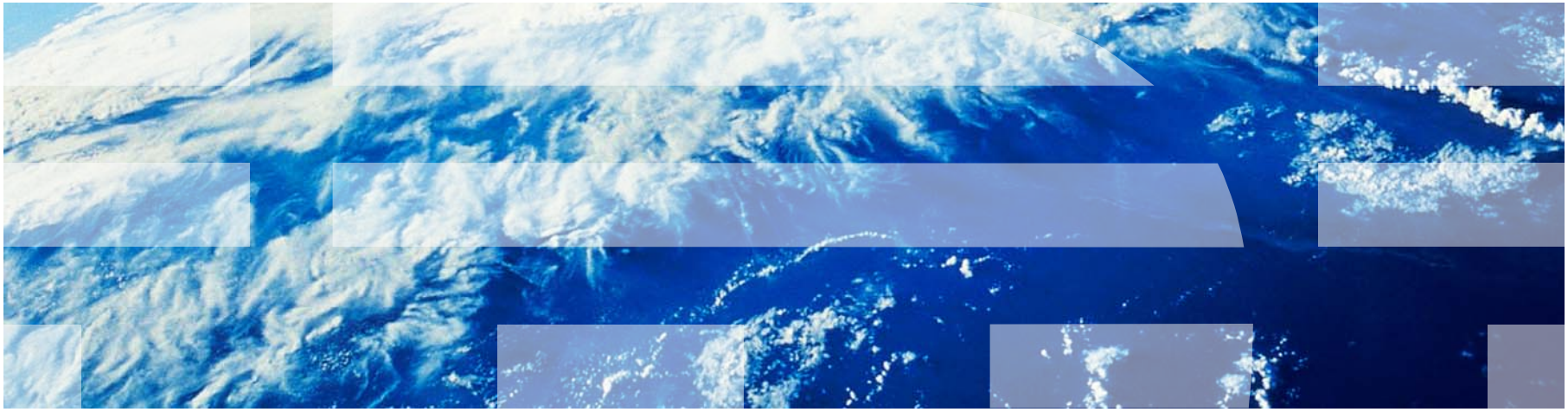
IBM

Understanding the Building Blocks of Trace Selection

Peng Wu, Hiroshige Hayashizaki, and Hiroshi Inoue

IBM Research



Overview of Trace-based Compilation

- Trace-based compilation uses traces as the basic unit for compilation
  - A trace is a sequence of instructions collected at runtime
    - Simple topology: typically single-entry, multiple-exit
    - Dynamic: based on runtime execution
    - Non-canonical: may start or end at arbitrary points in a method

- A brief history of trace compilation
  - Initially used in binary translator (zPDT, DynamoRIO, Transitive)
  - First demonstrated optimization benefit by Dynamo (PLDI'00)
    - To improve binaries compiled at low-opt levels
  - Later explored in embedded Java (HotpathVM and YETI)
  - Recently experienced a boom in compiling dynamic scripting languages
    - Javascirpt: TraceMonkey (Mozilla) and SPUR (MS research)
    - Python: PyPy and HotPy
    - Lua: LuaJIT

Trace Compilation for Java On Top of J9/Testarossa

Challenges

1. Java compilation has matured, why do we need trace compilation?
2. Trace compilation is good for dynamic scripting languages, but not for Java
3. Trace compilation is the same as partial inlining
4. Do you really believe the trace compiler can outperform Testarossa?

Motivation

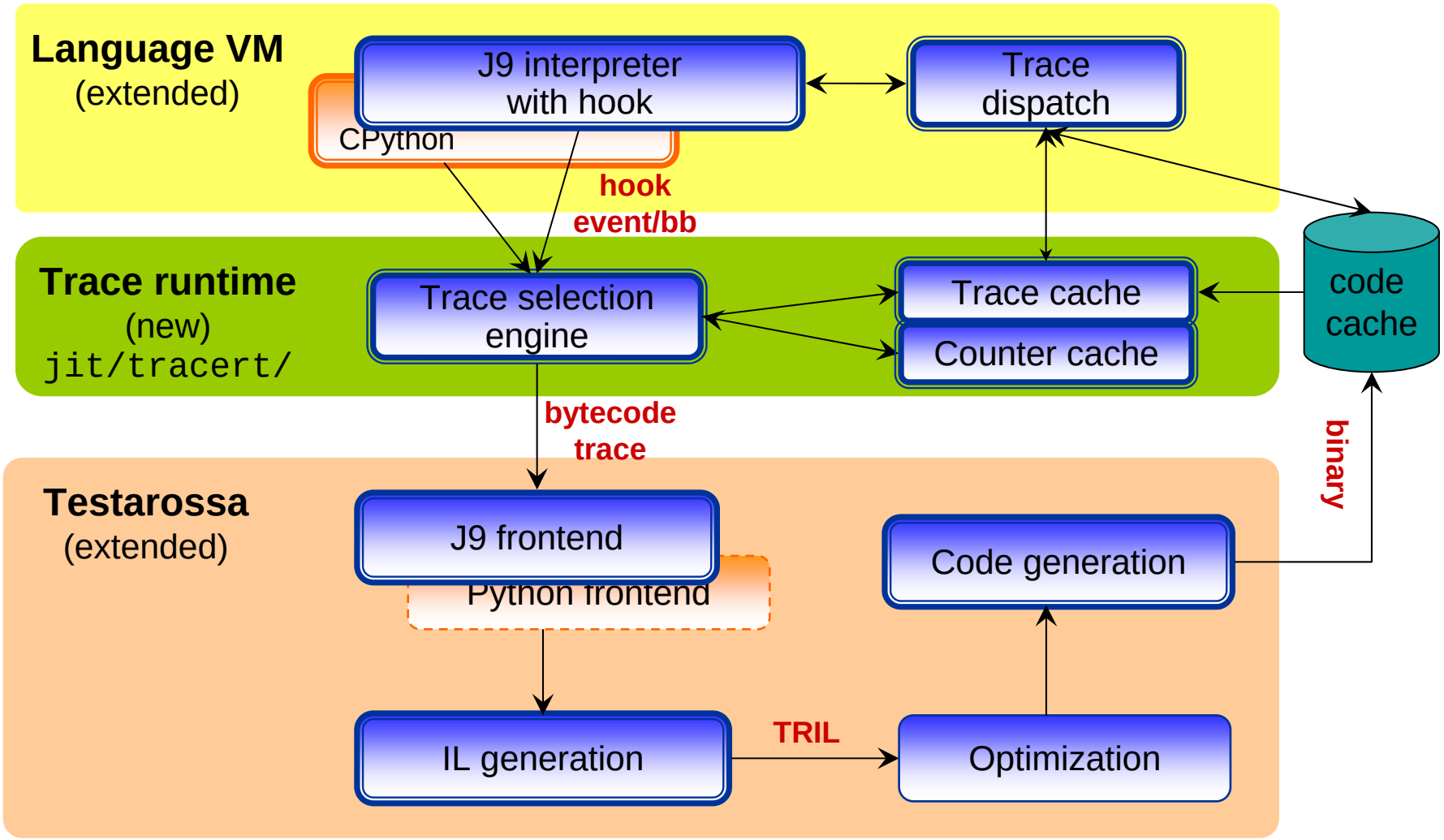
- Limitation of method-base compilation
  - Workloads with flat profile are hard to optimize due to limited inlining
- Trace compilation is *perceived* as not limited by method boundaries
  - Can it be used to break the "*method wall*" of traditional JIT?

Our approach

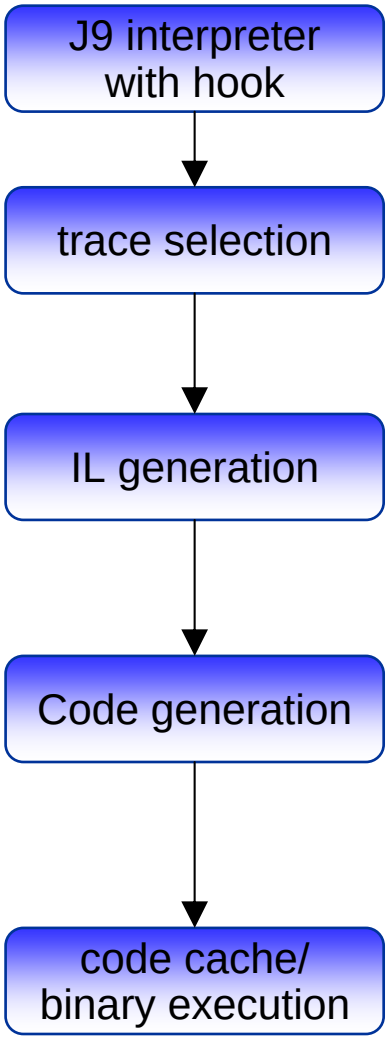
- Use trace selection to drive better region selection
- Reuse Testarossa as much as possible as optimization and codegen engine

Open minded: thorough design space exploration, if we fail, we want to know why

Trace JIT Overview



Example of Trace Lifecycle



trace selection engine is driven by a sequence of events from interpreter.

```
trace
JBiadd     java/lang/String.indexOf (bcIndex=7)
JBistore   java/lang/String.indexOf (bcIndex=8)
...
JBreturn
```

trace selection engine forms a trace. each BC is labeled by the method name and bcIndex to show its origin. *(starting from JBiadd is not possible in Java method compilation!)*

```
TR-IR
istore #5
   iadd
      iload (load from op stack inserted)
      iload (load from op stack inserted)
....
```

IL gen translates the trace into TR-IR. Implicit loads are inserted before the first bytecode if the operand stack is not empty upon entry to the trace.

```
binary code
lwz        gr5, [gr21, -32]
lwz        gr7, [gr21, -36]
add        gr6, gr5, gr7
stw        [gr21, -20], gr6
.....
```

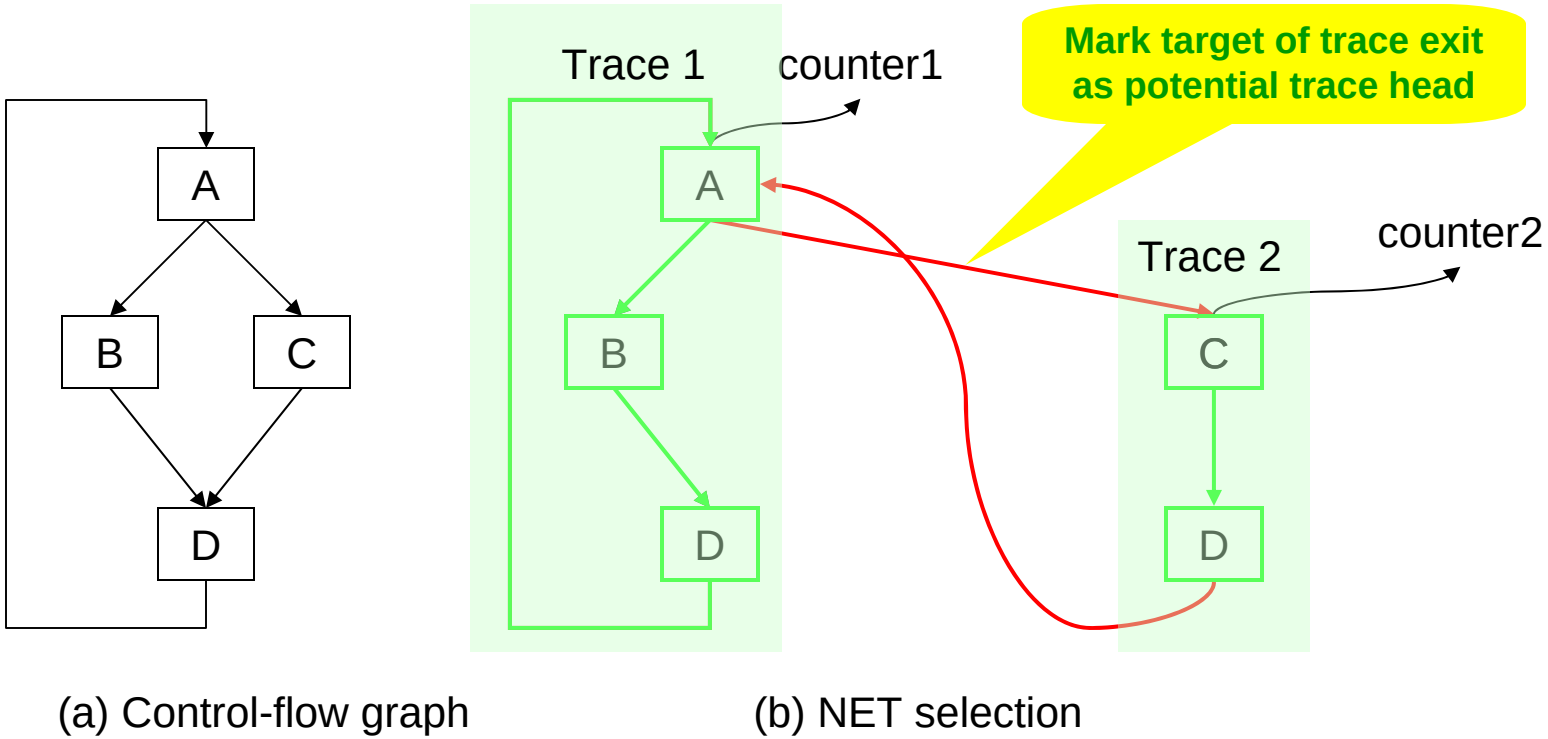
Code generator emits binary codes.

Trace Selection

- ***Trace selection*** forms traces out of executed instructions at runtime
  - An active area of research as it is at the heart of any trace compilation system

- Dynamo pioneered a form of two-step trace selection, called ***next-executing-tail*** (NET)
  1. Trace head selection: identify starting point of a trace by frequency-profiling a pool of potential trace heads
     A. Targets of backward branches (i.e., loop headers), or
     B. Instructions immediately following the exit point of a trace (exit-heads)
  2. Trace recording: record a trace from the selected trace head until meeting one of the trace termination conditions, e.g.,
     A. when encountering the head of an already formed trace
     B. when detecting a likely cycle in the recorded trace
     C. when the trace recording buffer overflows
     D. …

Next-Executing-Tail (NET) Selection



(a) Control-flow graph

(b) NET selection

- Traces are initially built from targets of backward branches
- They gradually grow out of side-branches (side-exits) of existing traces
- Trace size is determined by the termination conditions used in trace recording

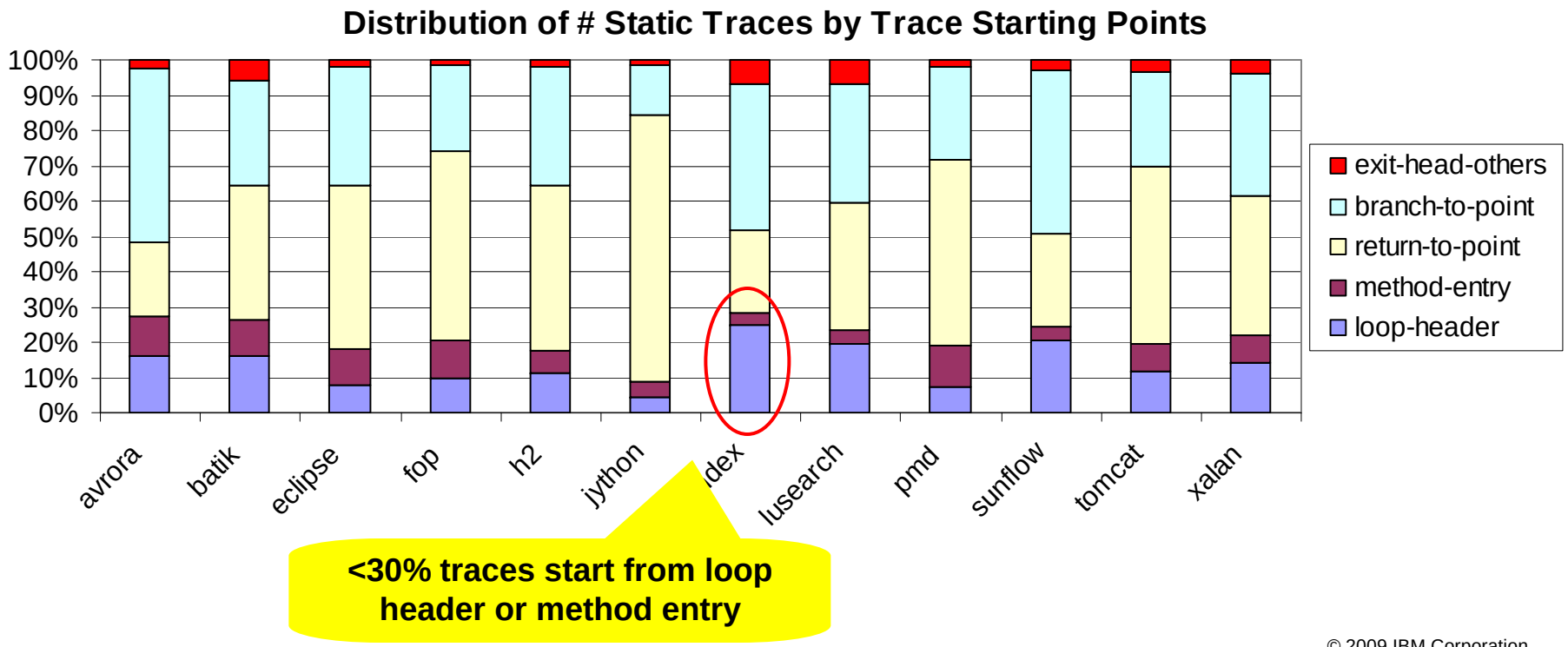
Trace Compilation vs Partial Inlining (I)

Partially inlined regions

- Inline *a complete path* of a callee method into the caller (space efficiency)
- Start and end at method boundaries with potential side-exits

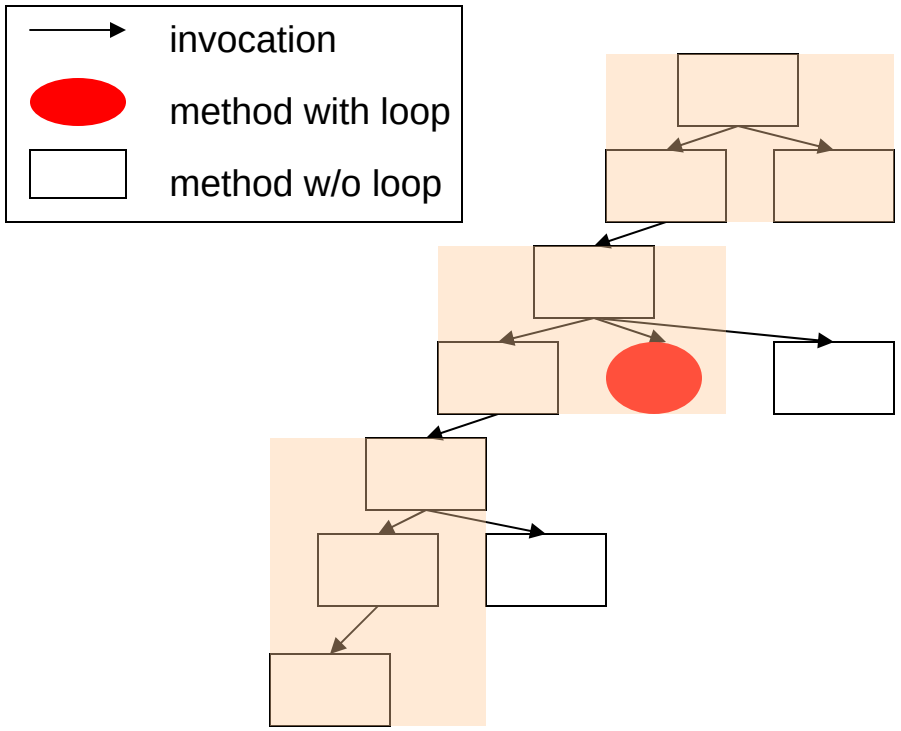
Trace regions

- Formed out of runtime execution paths
- Partial inlining naturally occurs in traces
- Most trace regions start at non canonical program boundaries

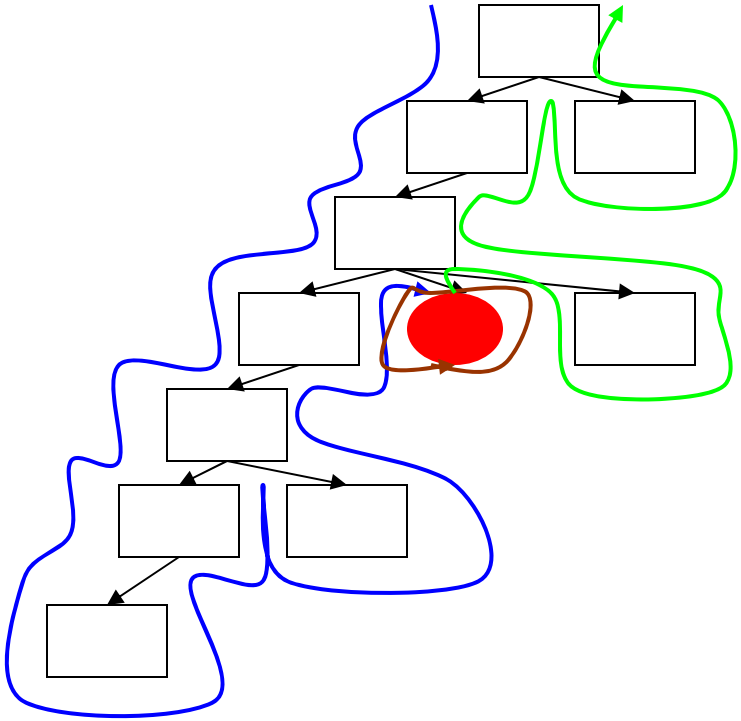


Trace Selection vs. Method Inlining (II)

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



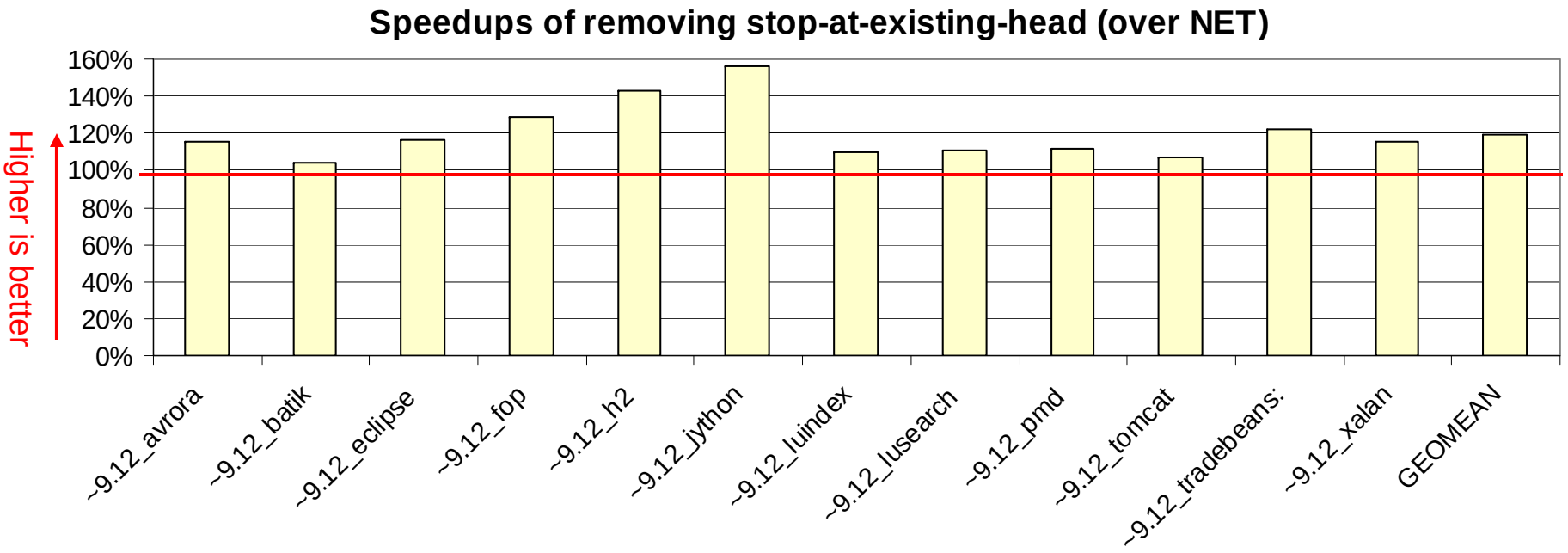
Method (partial) inlining forms hierarchical regions



Trace selection forms contiguous regions

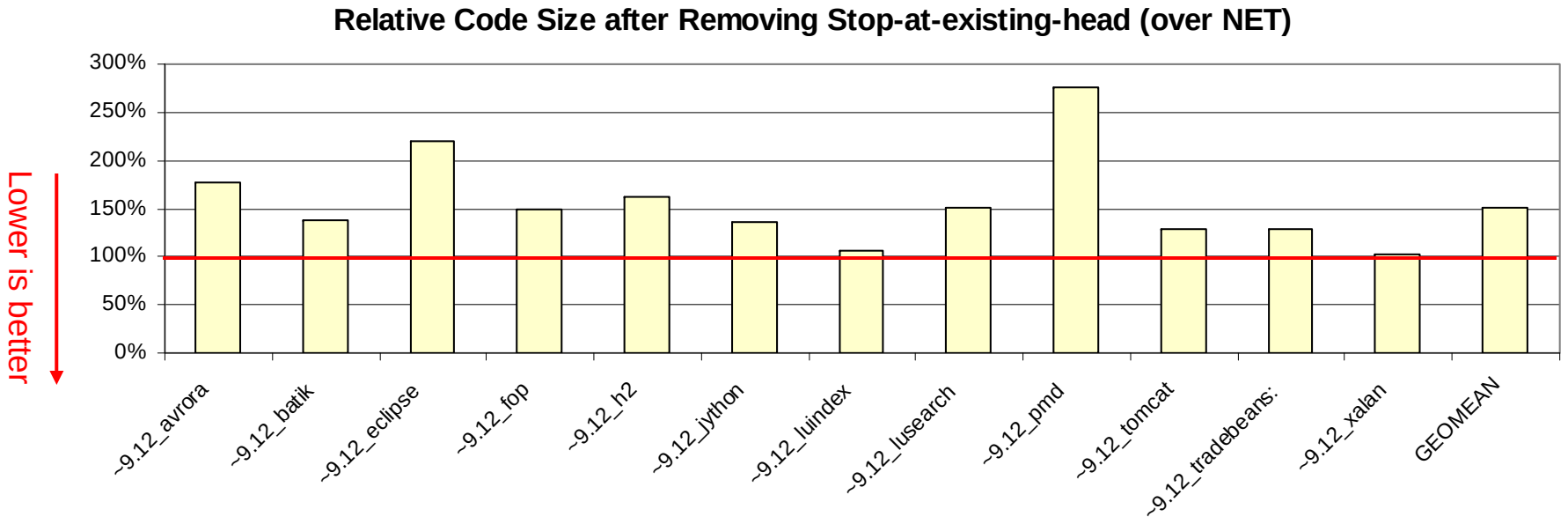
- blue, brown, green

Performance Impact of Trace Selection



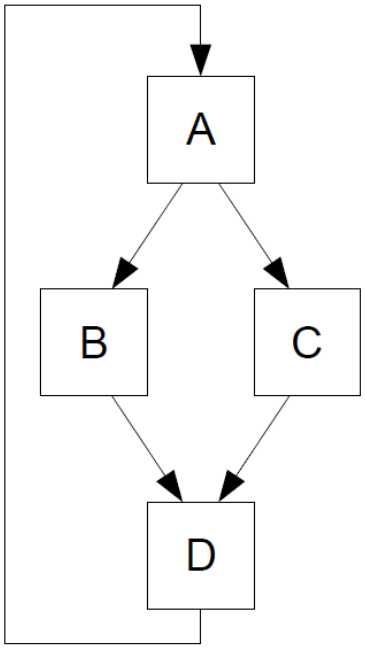
- Better trace selection contribute to 50% speedups over NET selection using the same JIT and runtime
  - Mainly by relaxing trace termination conditions to form longer traces
    - Removing stop-at-existing-head termination condition
    - And other techniques to relax termination conditions (ASPLOS 2011, under submission CGO2011)

Impact of Trace Selection to Jitted Code Size

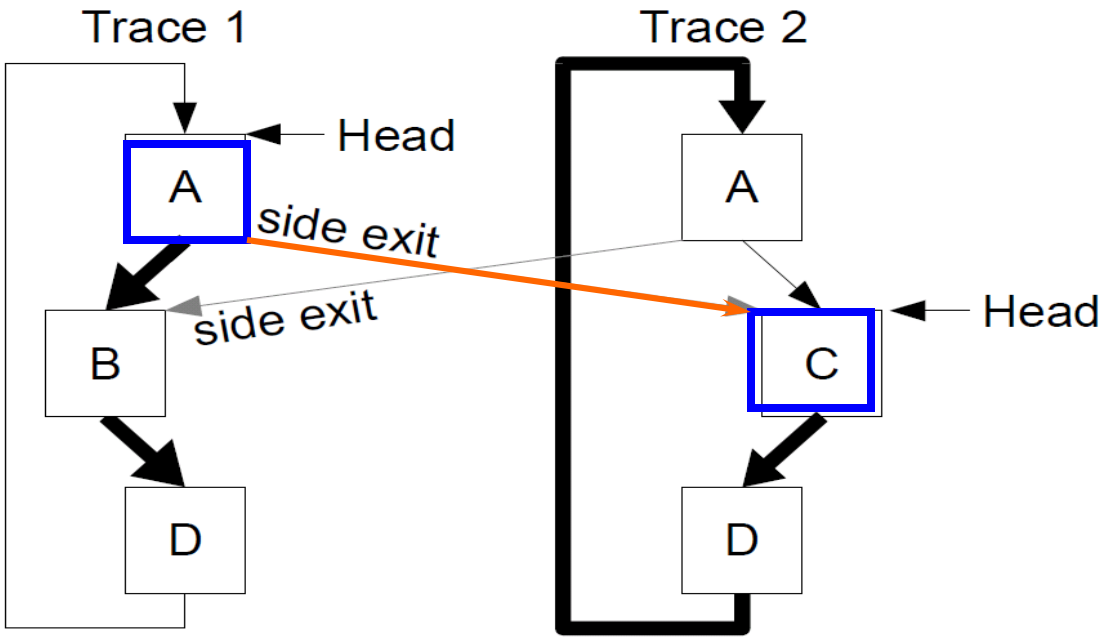


- Increasing trace size typically increases code size (>2X)
- Challenge of trace selection is to control code size without limiting trace scope

Limitation of Stop-at-exiting-Head: Cyclic Trace



(a) control-flow graph



(b) without stop-at-existing-head

Limitation of stop-at-existing-head termination conditions:

- Limit the ability to capture cyclic paths: one cyclic path per loop
- Limit the ability to “inline”: when a method entry becomes a trace head, no subsequence trace can “inline” this method

Can Trace Compilation Offer New Value to Java?

Where are we now?

- Built a robust trace JIT based on J9/Testarossa
  - multithreading, monitor, GC, exception, async compilation, JNI, trace linking, …
- Enabled most warm-level optimizers except for some loop opts and escape analysis
- Thorough design space exploration on trace selection, new algorithms
- Significant efforts to reduce trace runtime overhead
- Missing features: re-compilation, trace cache flush, interpreter profiling

Compare to default TR JIT (pap3260)?

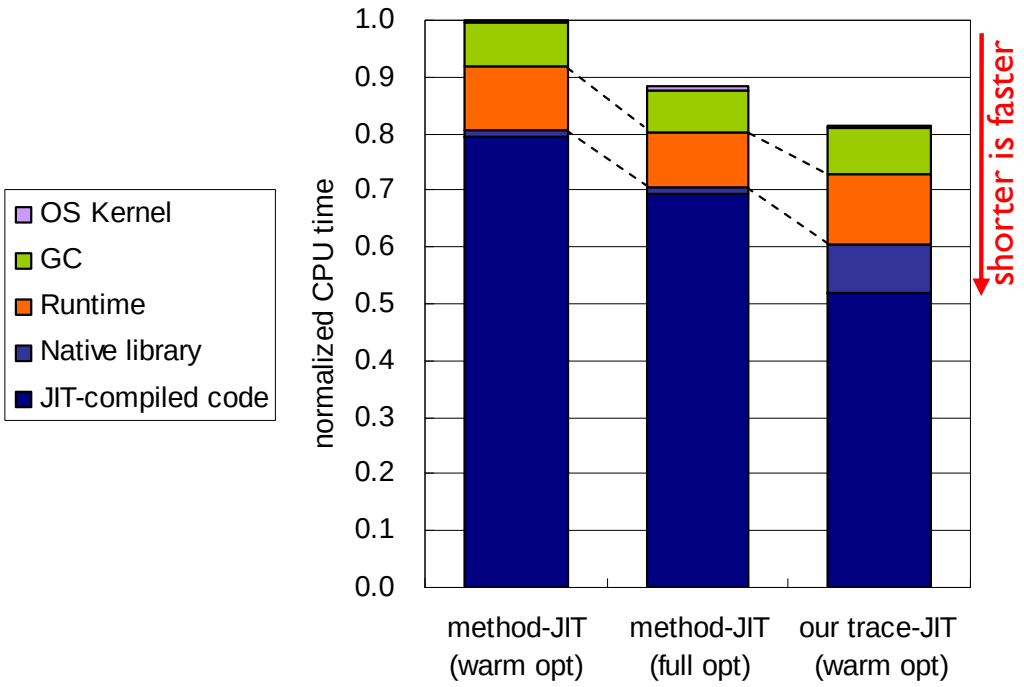
- *Almost* comparable performance to TR method-JIT
  - 6% slower than default TR JIT on DaCapo (new)
  - 50% more jitted code size

Where do we see potentials?

- In some benchmarks, trace JIT outperforms method JIT (6%~36%)
- In more cases, trace JIT produces better jitted codes but suffers from more runtime overheads
- Observed very long traces through many method layers

Steady-state CPU Time Breakdown for trace-JIT and method-JIT

❑ **Jython (DaCapo 9.12)**



- ❑ Flat profile
  - max weight of single trace is <4%
  - top 1670 traces (out of 8800) cover 90% execution
- ❑ Many very long traces
  - 25% time spent on traces of 256 BB (max trace size)
- ❑ Many method boundary crossing
  - among the top 1000 traces, single trace contains an average of 51 invoke bytecodes
- ❑ (Partial) inlining effect on traces
  - 44% of bytecodes executed on inlined portions of traces

- ❑ Trace 6788 (rank#3): 1.98% time spent, cyclic trace of 57BB, with 18 invocations, 99% utilized (5 side-exits BBs)
- ❑ Trace 7870 (rank#6): 1.17% time spent, linear trace of 256BB, with 56 invocations, 2 side-exit, 75% utilized (2 side-exit BBs)

Concluding Remarks

Trace compilation is an exciting new approach to dynamic compilation, but it is still at its early age of explorations

Our surprise findings

- Trace regions are truly different from method regions
- A method-based optimizer can be retrofitted to optimize (non-canonical) traces and be quite effective
- Extending trace scopes matters a lot in performance
- Increasing trace lengths are easy, controlling code size is tricky
- Linear traces are not necessarily inferior to structured traces

Open research questions:

- Lack of deep understanding for trace compilation
- When can trace compilation outperform method compilation and why?

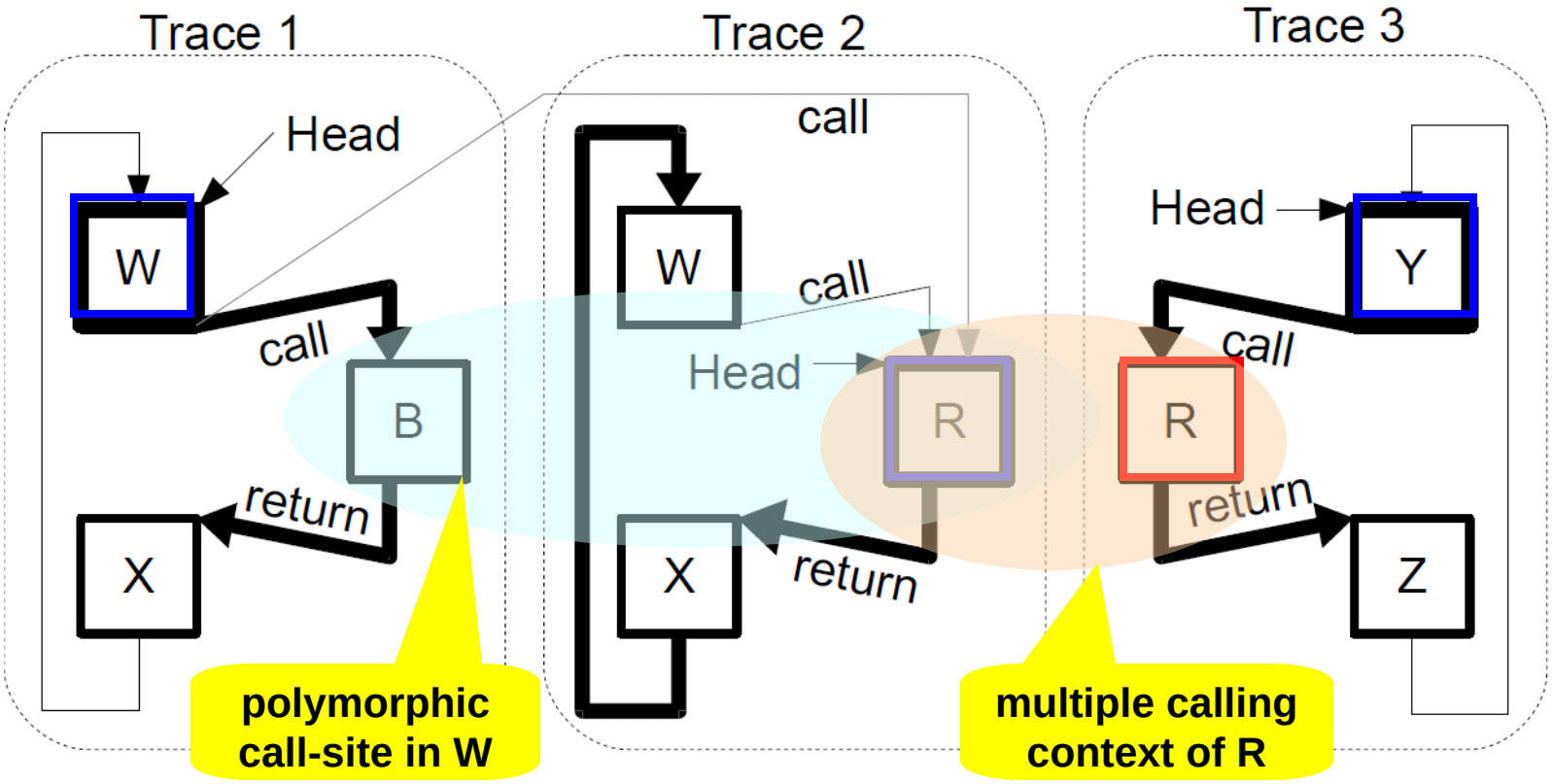
Our on-going explorations:

- Built some theoretical foundation on trace compilation
- Control code size without limiting trace scope
- Trace formation beyond linear traces
- Incorporate profiling into trace compilation
- Enable remaining Testarossa optimizers
- Explore trace compilation for Python

Our hunch: there is unlikely a one-size-fit-all approach, the end system would have a mixture of method- and trace-based compilation

BACK UP

Limitation of Stop-at-existing-head: “Inlining” Effect



Stop-at-existing head limits the “inlining” effect on traces

- when a trace is formed at the entry point of a method, the method cannot be “inlined” to any subsequent traces