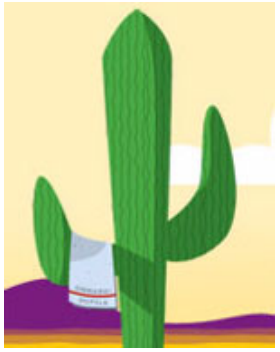
IBM Research - Tokyo

Adaptive Multi-Level Compilation in a Trace-based Java JIT Compiler



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

October 23, 2012 | SPLASH/OOPSLA 2012 at Tucson, Arizona

Goal

- Balancing steady-state performance and startup performance has been a challenging task for JIT compilers

Our Goal

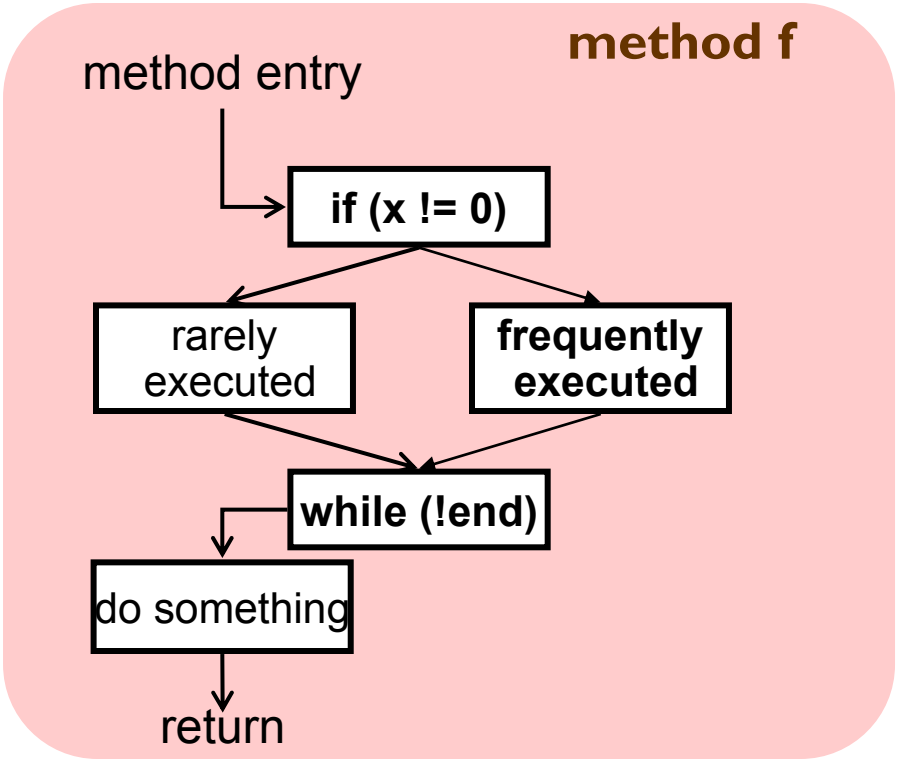
*to improve the steady-state performance without hurting startup performance in a **trace-based** Java JIT compiler*

Our contributions:

1. We show that adaptive multi-level compilation is a practical way to balance the startup time and steady-state performance in a trace-JIT
2. We develop a new technique to efficiently identify hot paths to recompile in a higher optimization level

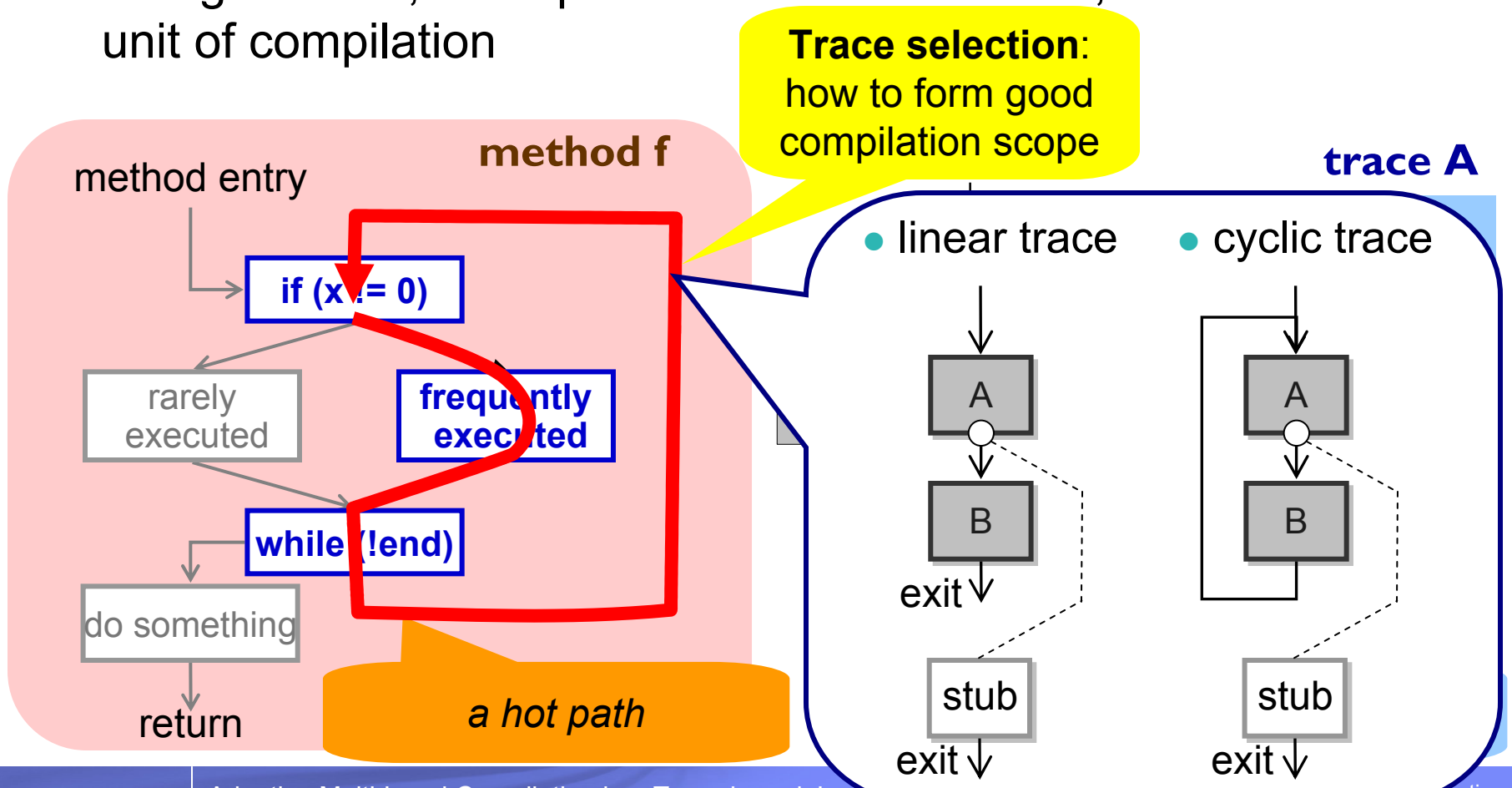
Background: Trace-based Compilation

- 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



Background: Trace Selection and Performance

- Steps of trace selection
  1. Identify a hot trace head
     - a target of a backward branch, 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, reaches pre-defined maximum length

- Generating longer traces (compilation scopes):
  - ☺ yields **better steady-state performance**
    - more optimization opportunities for compilers
    - smaller trace transitioning overhead
  - ☹ but it **hurts startup performance**
    - longer compilation time
    - more duplicated code among traces

Our Approach: Adaptive multi-level compilation for trace-JIT

1. Interpreted execution

   ⇩ *identify hot paths by counting the execution count for each potential trace head (e.g. loop backedge)*

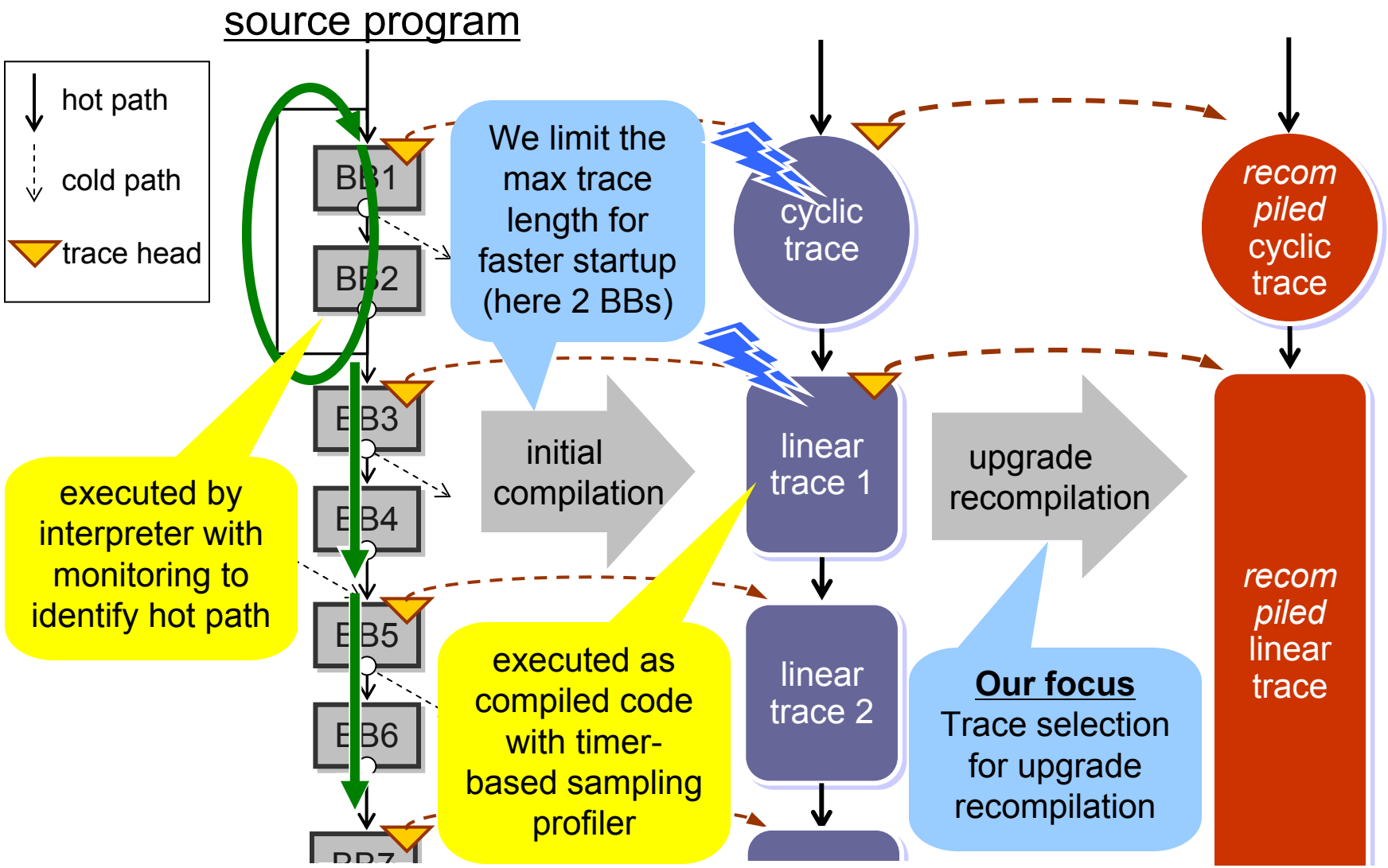
2. Initial compilation for faster startup
   - smaller compilation scope
   - lower optimization level

   ⇩ *identify really hot paths by using timer-based sampling profiler*

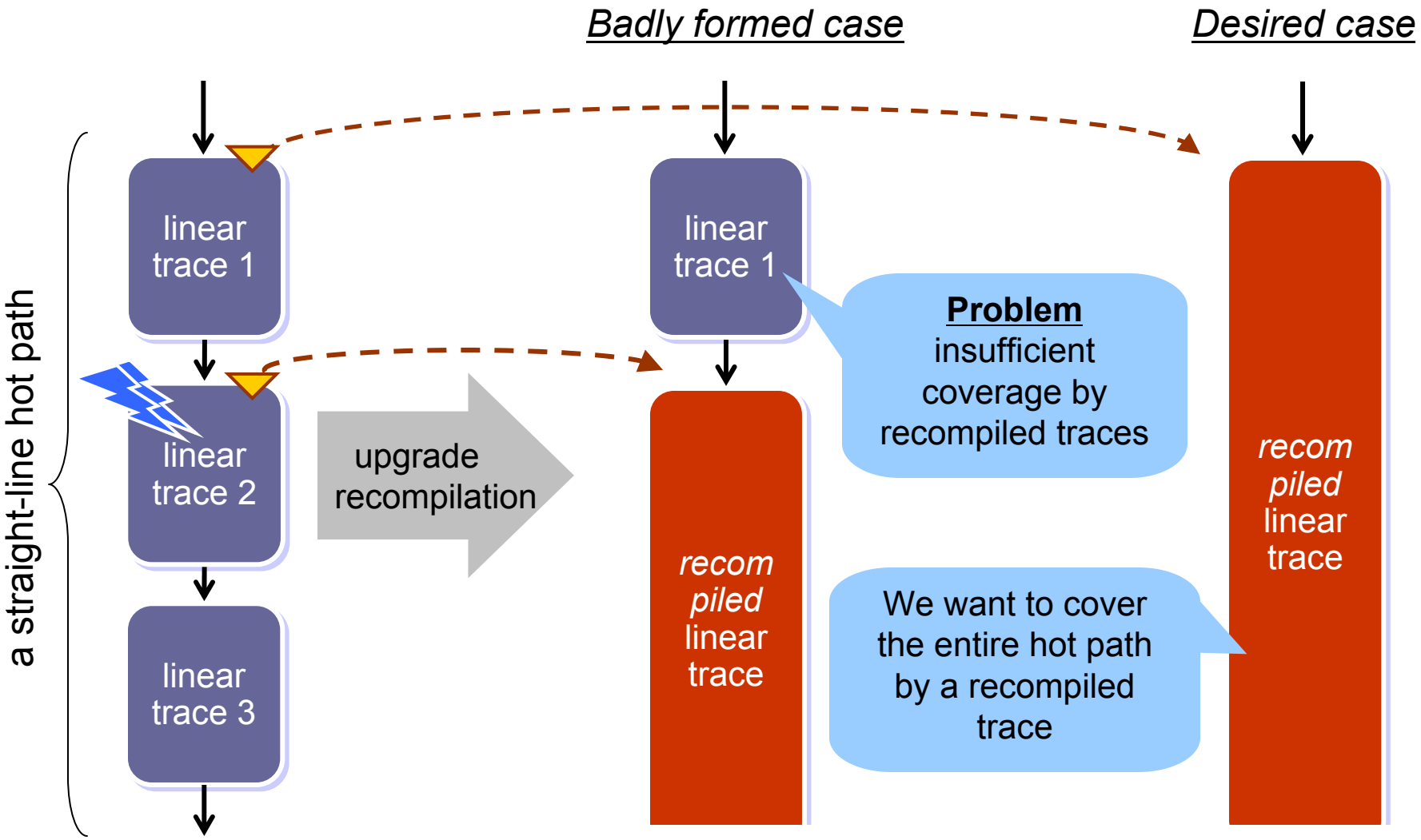
3. Upgrade recompilation for higher steady-state performance
   - larger compilation scope
   - higher optimization level

Our focus
Trace selection for upgrade recompilation

Steps of Our Multi-level Compilation

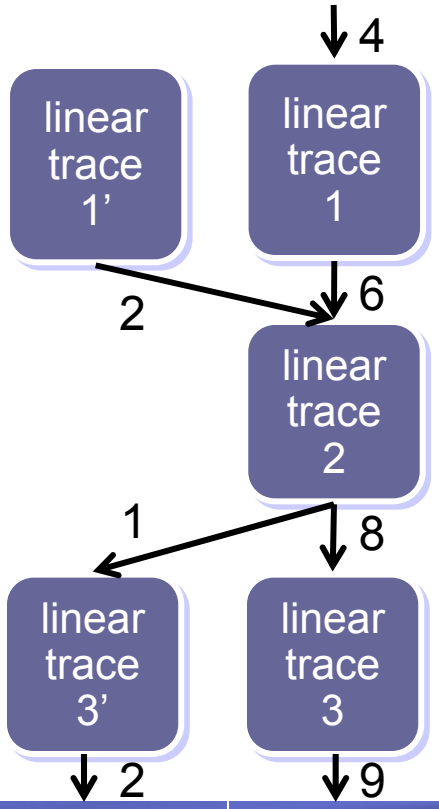


An Example of Inefficient Selection for Recompilation



Our Solution: Trace Transition Graph (*TTgraph*)

- *Trace-Transition Graph* (*TTgraph*) is a weighted directed graph representing the control flow among traces

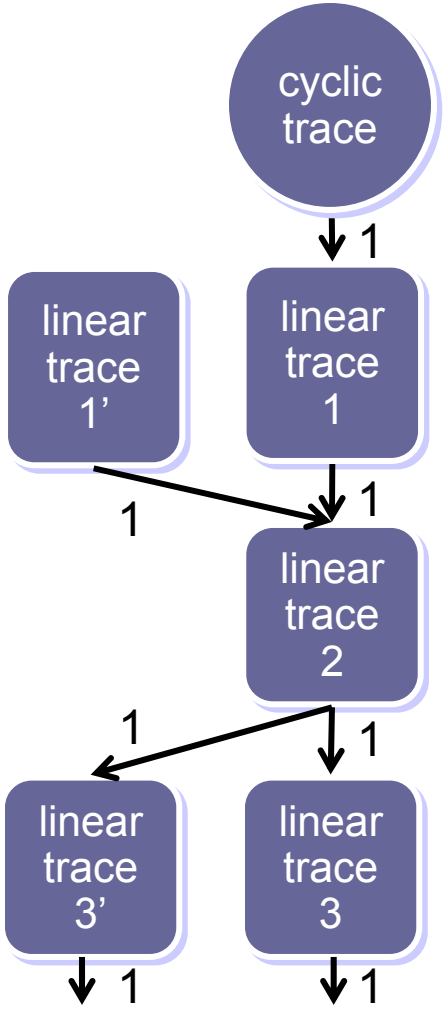


- Node: compiled trace
- Edge: transition between two traces
  - weight represents relative frequency

➔ We identify the hot paths to recompile using the TTgraph

  - to capture the entire hot path by traversing the edges backward

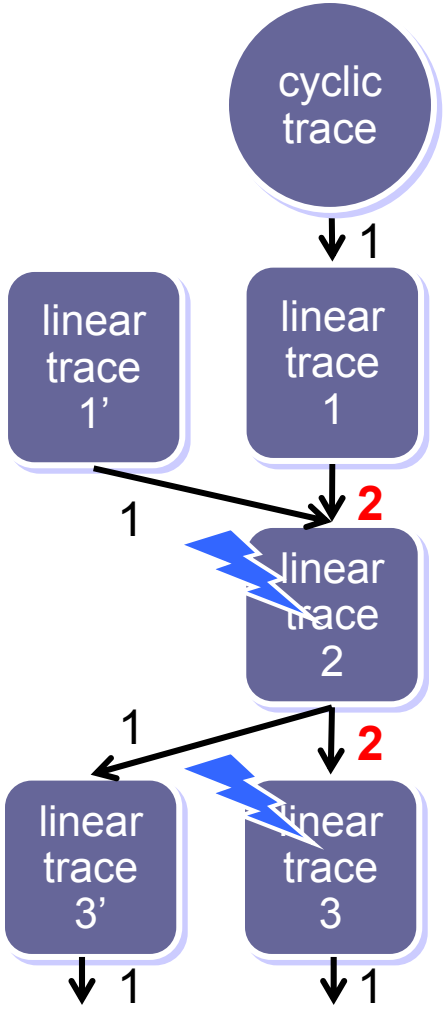
Steps to Build TTgraph



1. We add an node when a trace is compiled
2. We add an edge between two traces when we link them (trace linking)
3. We increment weight of an edge by timer-based sampling profiler

See the paper for more detail (e.g. how we profile transition and how we employ bursty tracing)

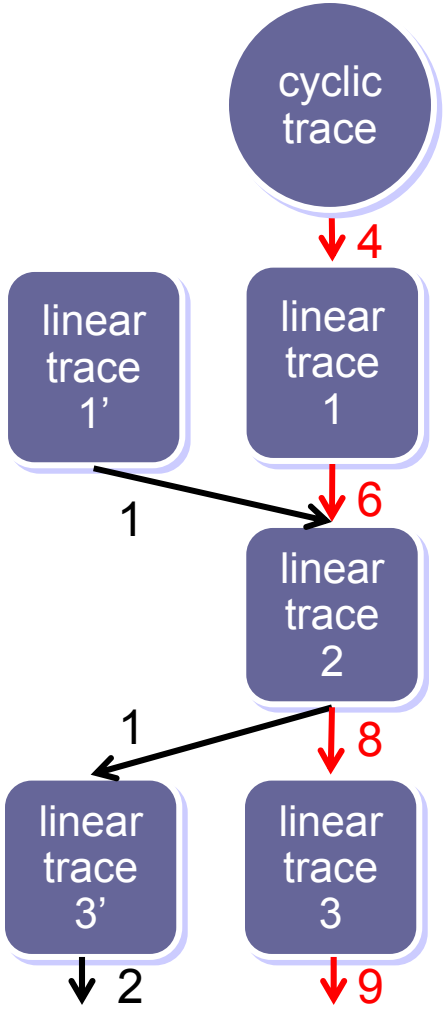
Steps to Build TTgraph



1. We add an node when a trace is compiled
2. We add an edge between two traces when we link them (trace linking)
3. We increment weight of an edge by timer-based sampling profiler

See the paper for more detail (e.g. how we profile transition and how we employ bursty tracing)

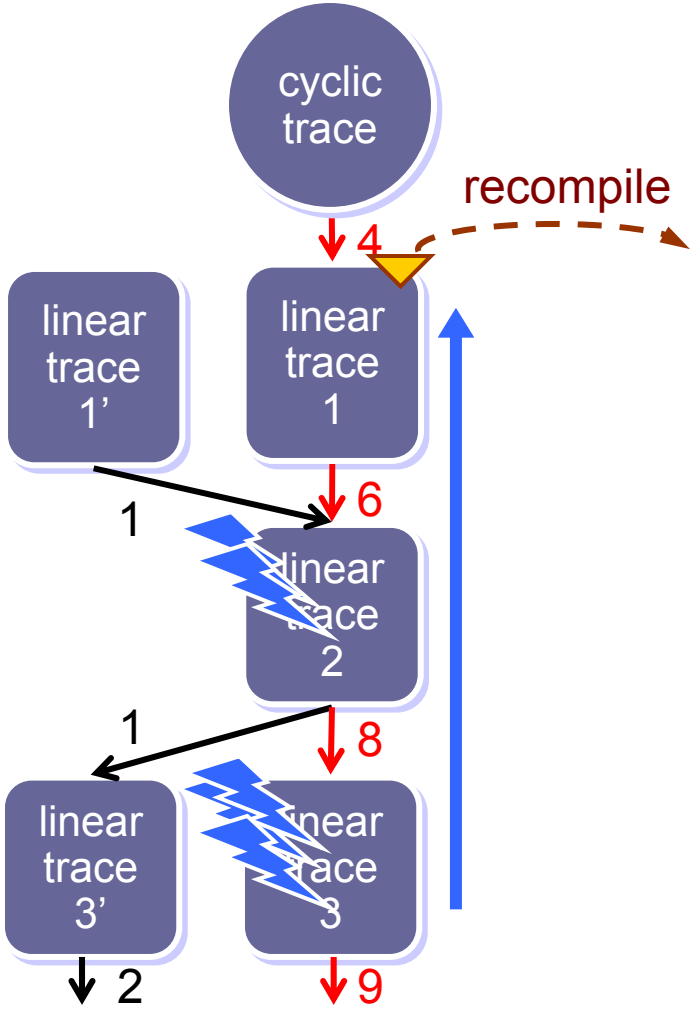
Steps to Build TTgraph



1. We add an node when a trace is compiled
2. We add an edge between two traces when we link them (trace linking)
3. We increment weight of an edge by timer-based sampling profiler

See the paper for more detail (e.g. how we profile transition and how we employ bursty tracing)

Recompilation Based on TTgraph



We traverse the hot edges backward to identify the new trace head for recompilation

- we stop the backward traversal when hitting a cyclic trace (our system does not capture cyclic and linear path in one trace)
- If there are multiple hot incoming edges for a node, we track both of them

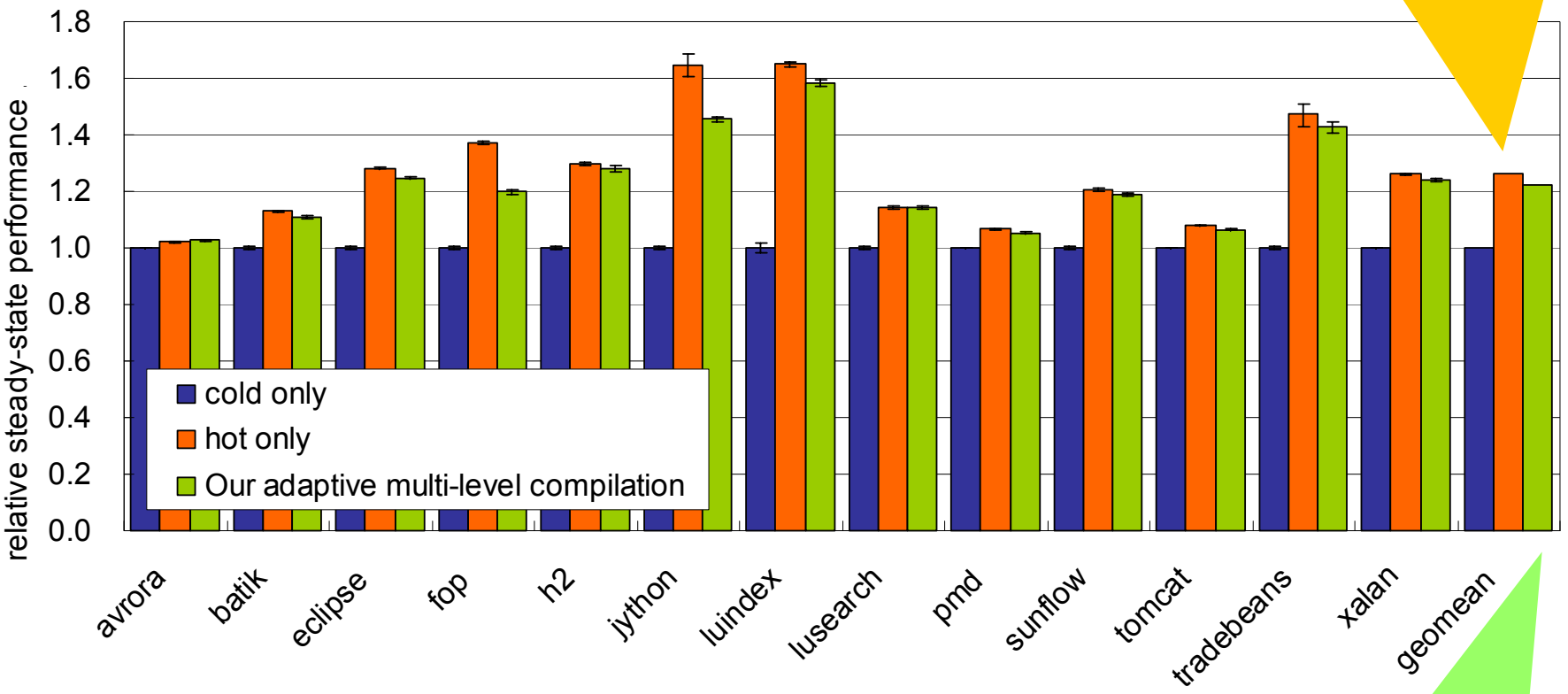
See the paper for more detail

Performance Evaluation

- Hardware: IBM BladeCenter JS22
  - 4 cores (8 SMT threads) of POWER6 4.0GHz
  - 16 GB system memory
- Our trace-JIT:
  - Extended IBM JVM for Java 6 (32-bit) to support trace compilation
  - Two optimization levels (both trace selection and code generation), **cold** and **hot**
  - We compare cold only, hot only, and our adaptive multi-level compilation
- Benchmark:
  - DaCapo benchmark suite 9.12

Steady-state Performance

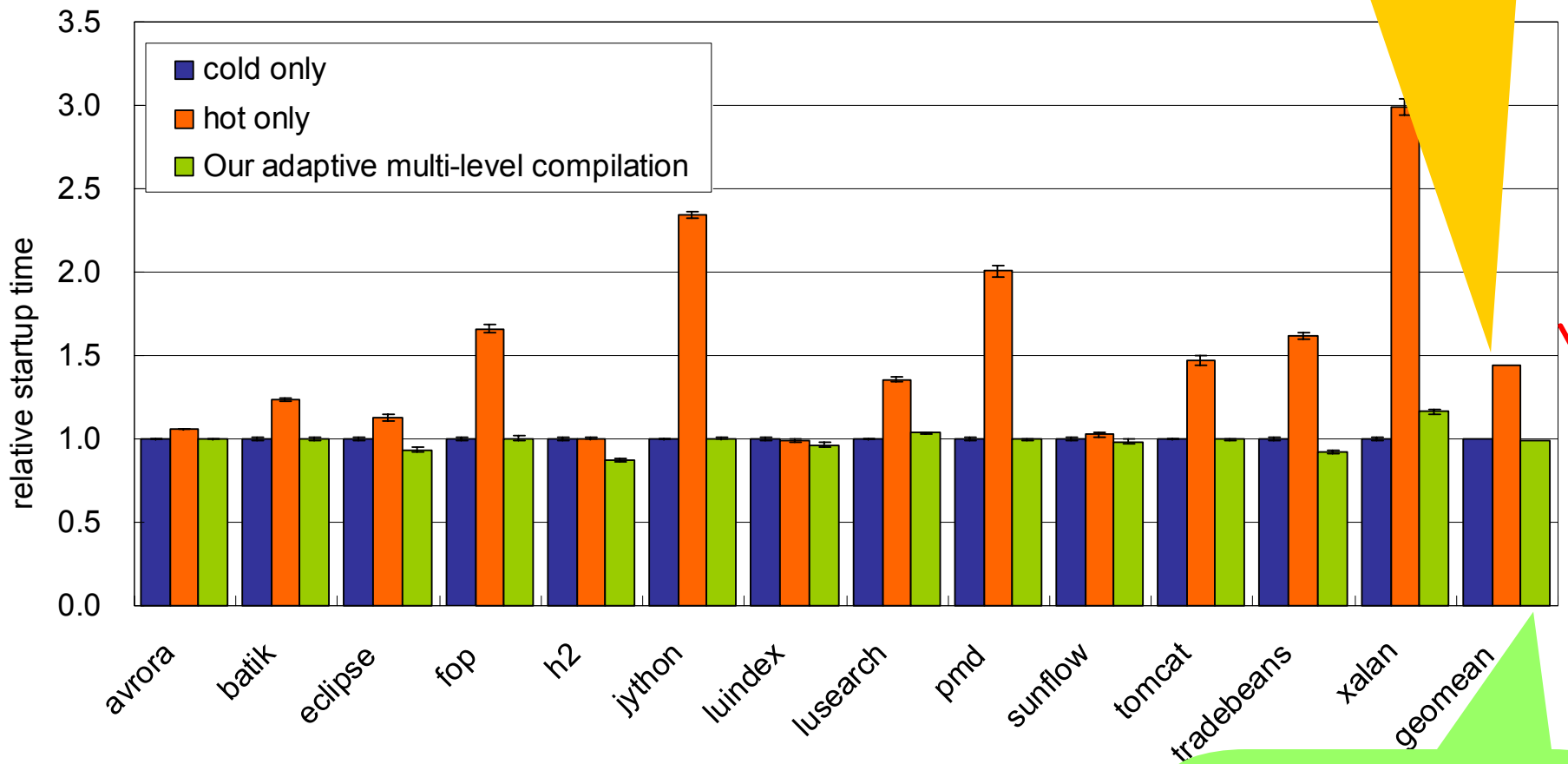
always using hot level gives 26% gain in steady state



Our technique achieved 22% gain in steady state

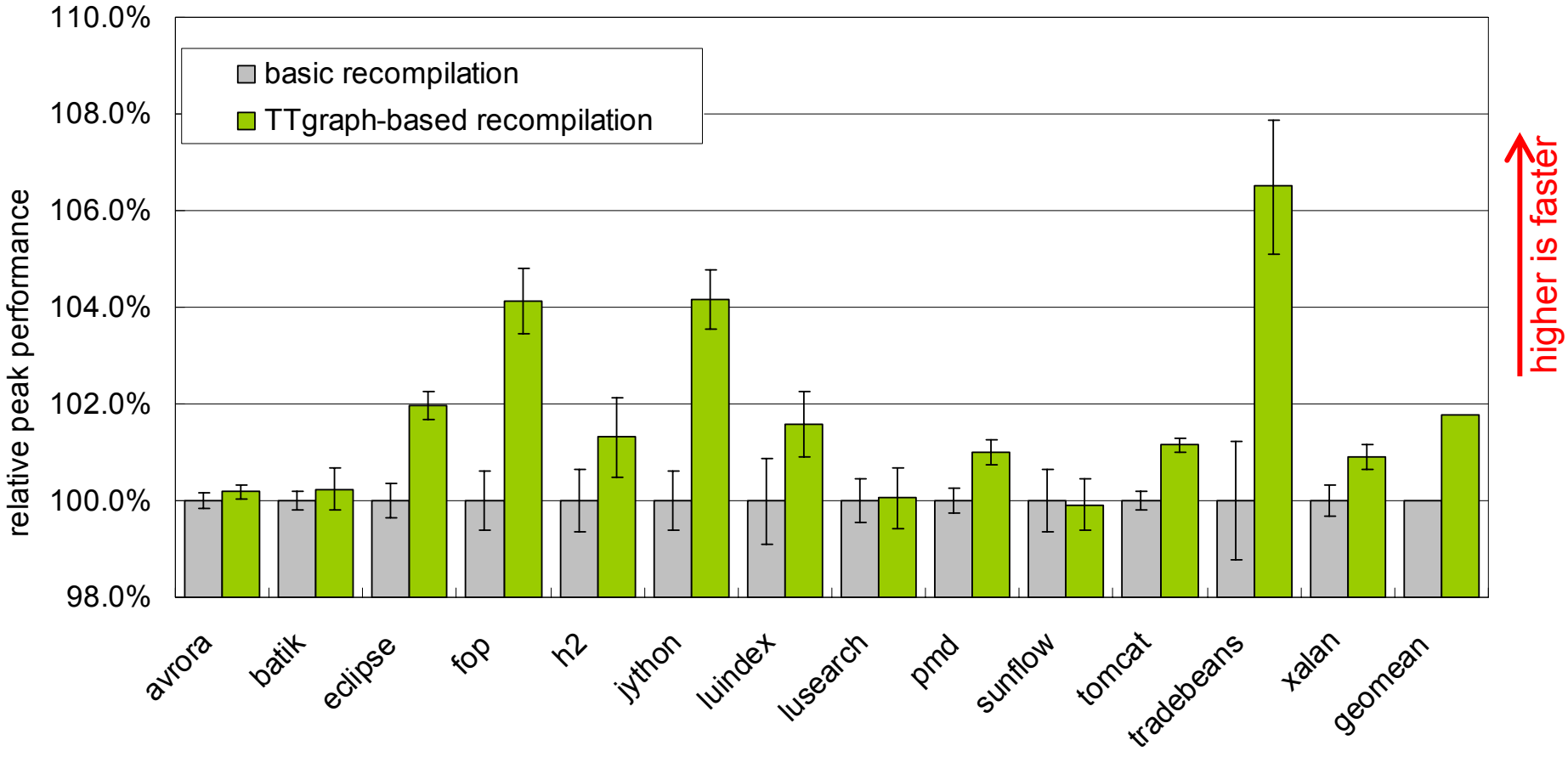
Startup Time (Execution time of fi

using hot level makes the startup 50% (up to 3x) slower!



Our technique does not hurt the startup performance!

Improvement by TTgraph-Based Recompilation



Summary

- We showed that adaptive multi-level compilation is a practical way to balance the startup time and steady-state performance in a trace-JIT
- We developed an efficient technique based on TTgraph to identify hot paths. We described the trace selection engine, the timer-based sampling profiler, and the code generator work together for effective recompilation.

Thank you!

Backup

CPU Time Breakdown