# Experiences Designing a Robust and Scalable Interpreter Profiling Framework

Ian Gartley, Nikola Grcevski, Marius Pirvu, Vijay Sundaresan

IBM Canada

© 2013 IBM Corporation

# Motivations

© 2013 IBM Corporation

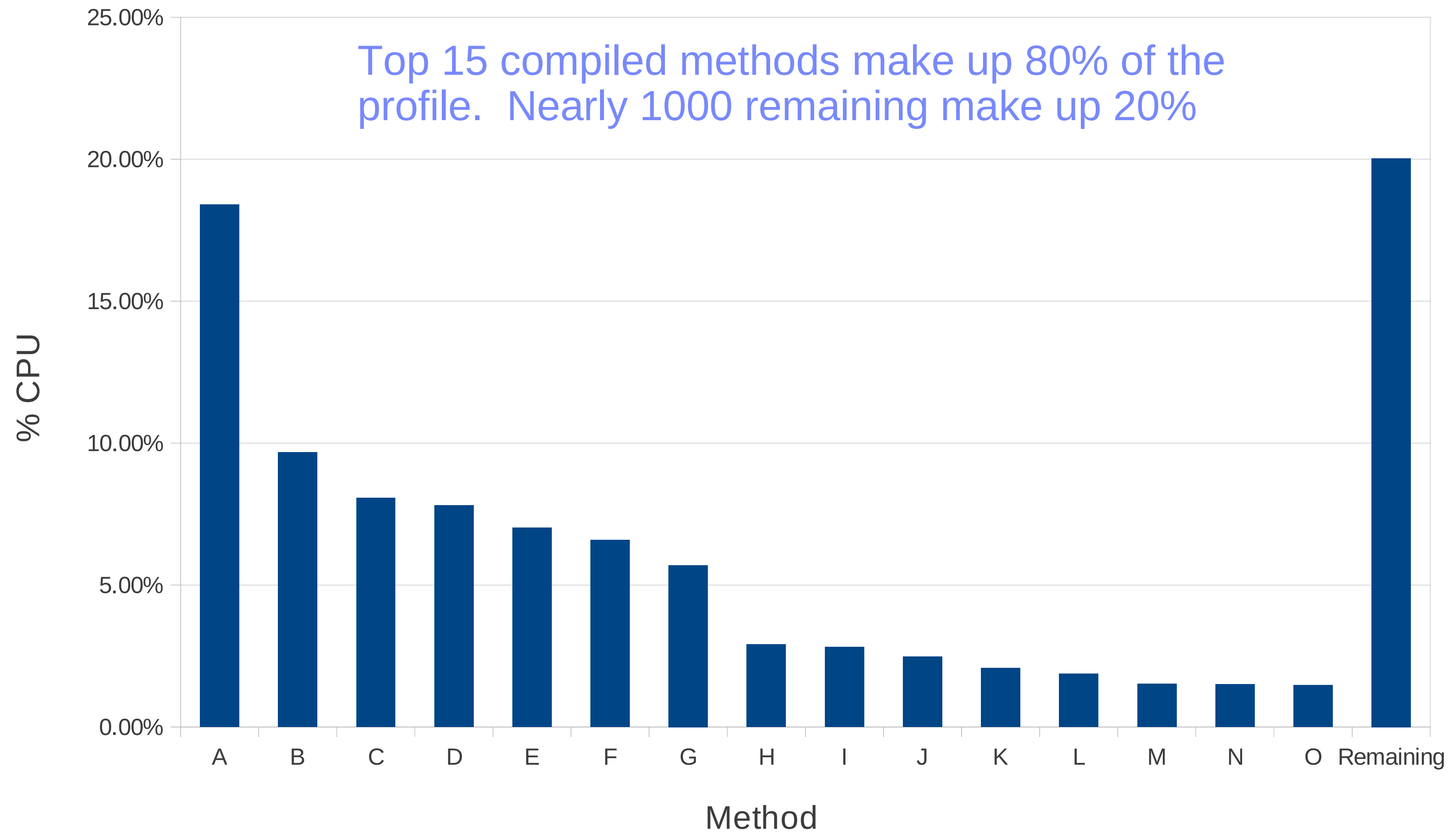
# Motivation : Profile Driven Feedback (PDF)

- Drastically improves performance in dynamic languages
  - Have seen 60% on some typical Java benchmarks
- Transparent to the user in dynamic languages
  - Static languages (generally) have to do a training run
- Allows for greatly increased performance of some optimizations:
  - Devirtualization
  - Inlining
  - Block ordering
  - Other value profiling based optimizations

- Profiling can be done by instrumenting compiled code or in the interpreter
  - Called JIT profiling if done in compiled code
  - Called interpreter profiling if done in the interpreter

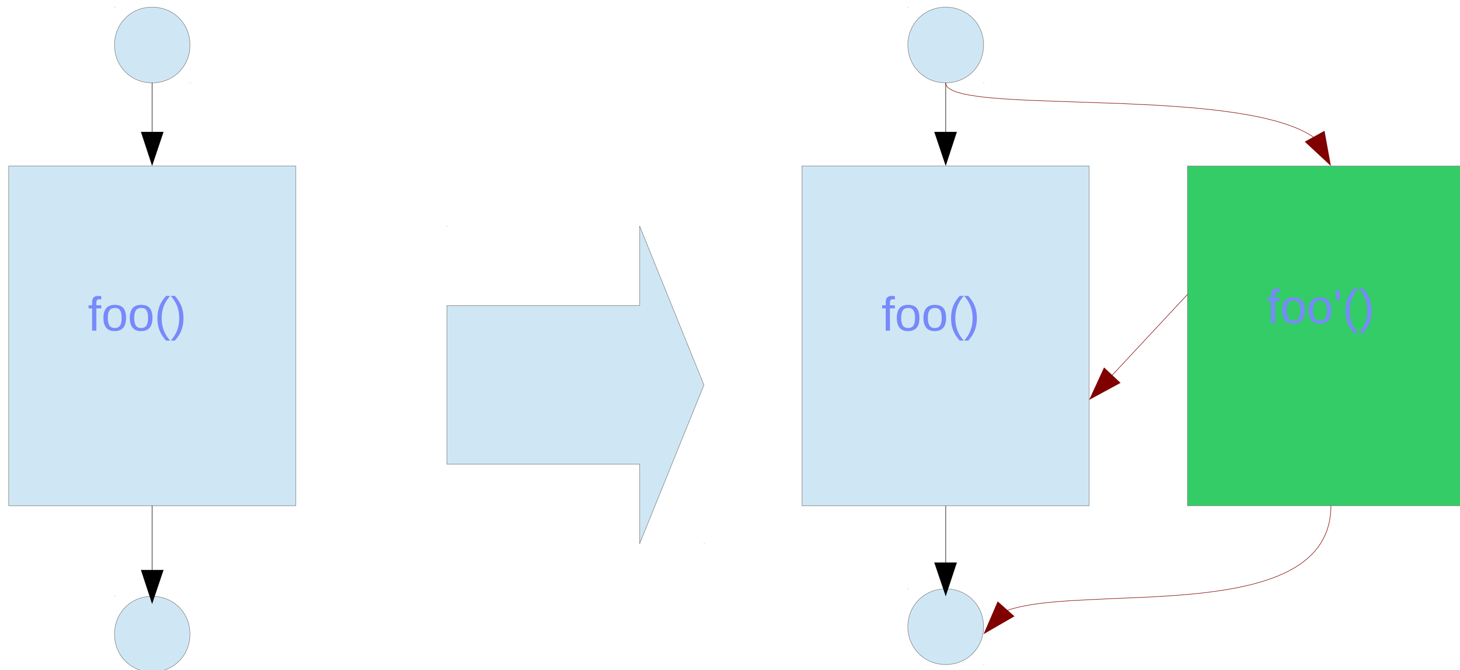
© 2013 IBM Corporation

# A Typical Benchmark Profile - SPECjbb2005




© 2013 IBM Corporation

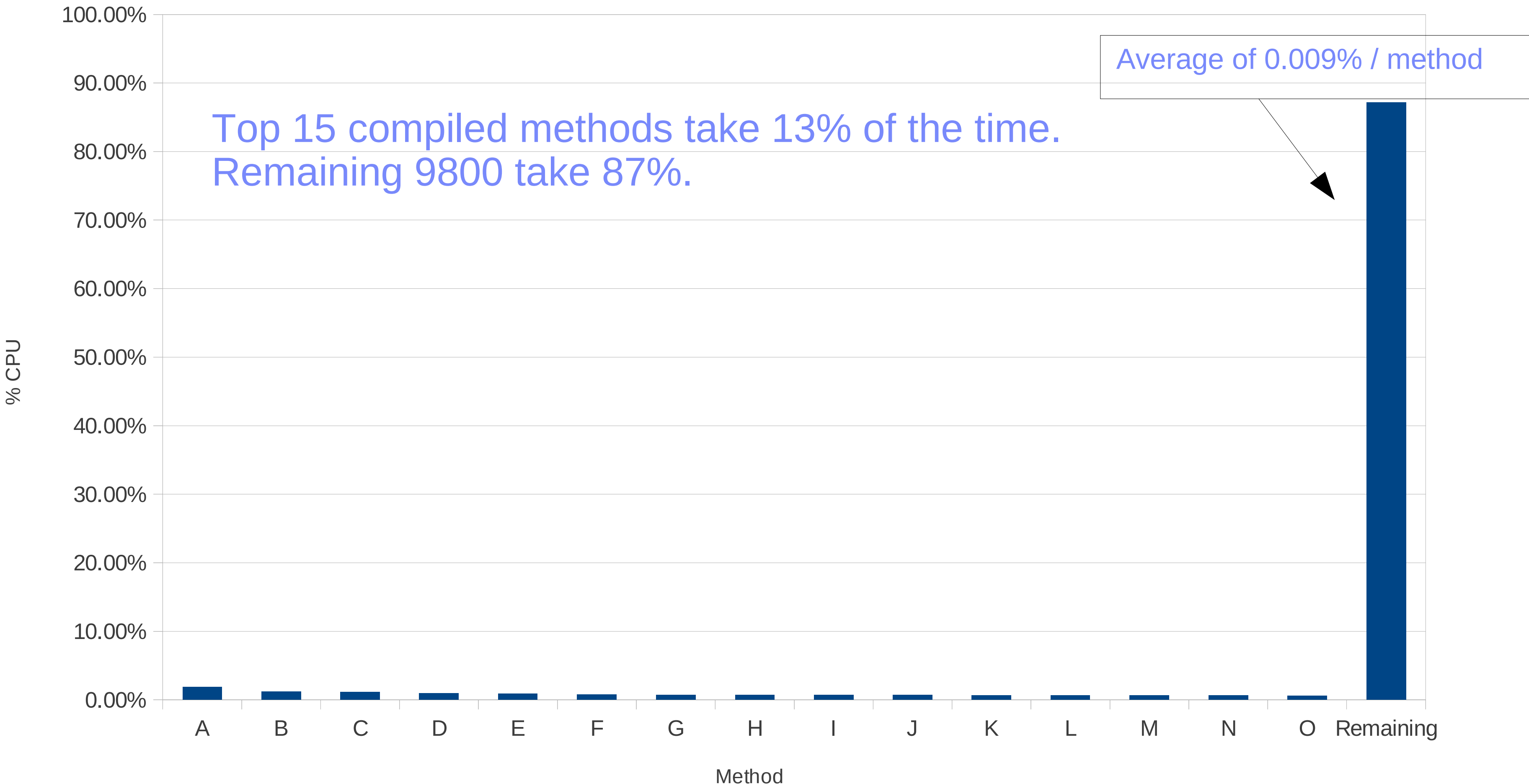
# J9 JIT Profiling




- IBM's J9 VM implements profiling as described in Arnold and Ryder's paper “A Framework for Reducing Cost of Instrumented Code”
  - Replicate method body, instrument it, with various control flow points back to original body
- Excellent for capturing peaked profiles
- Recompilation is necessary!

© 2013 IBM Corporation

# Profile for WebSphere Application Server 8.5 running DayTrader




© 2013 IBM Corporation

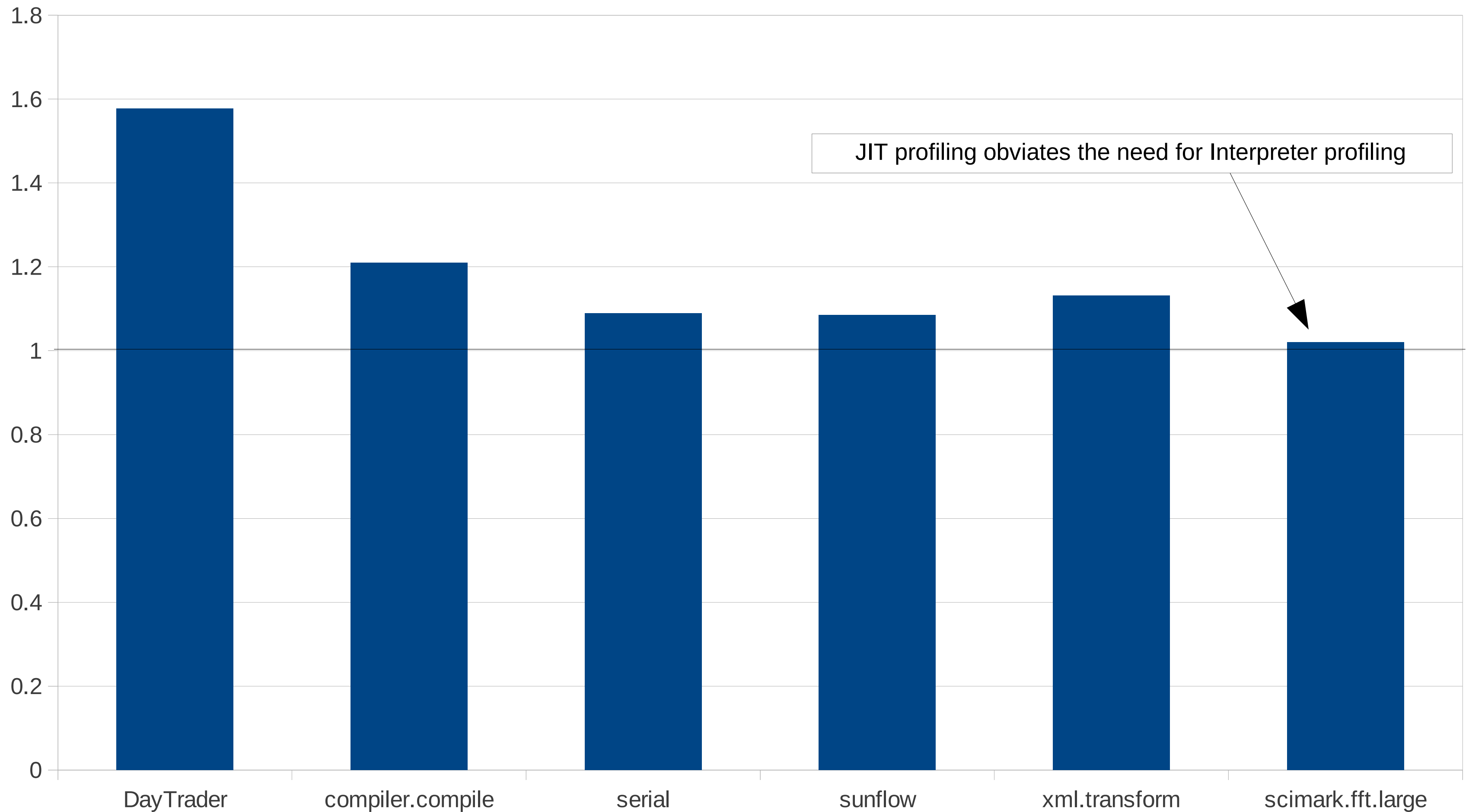
# The solution: Interpreter Profiling

- J9 has mixed-mode execution
  - Methods begin interpreted
  - Frequently executed methods are compiled
- Collects profiling information without resorting to “mass” recompilation

- Introduces a new problem: interpreter profiling overhead during JVM start-up
- Many customers care about startup time!

© 2013 IBM Corporation

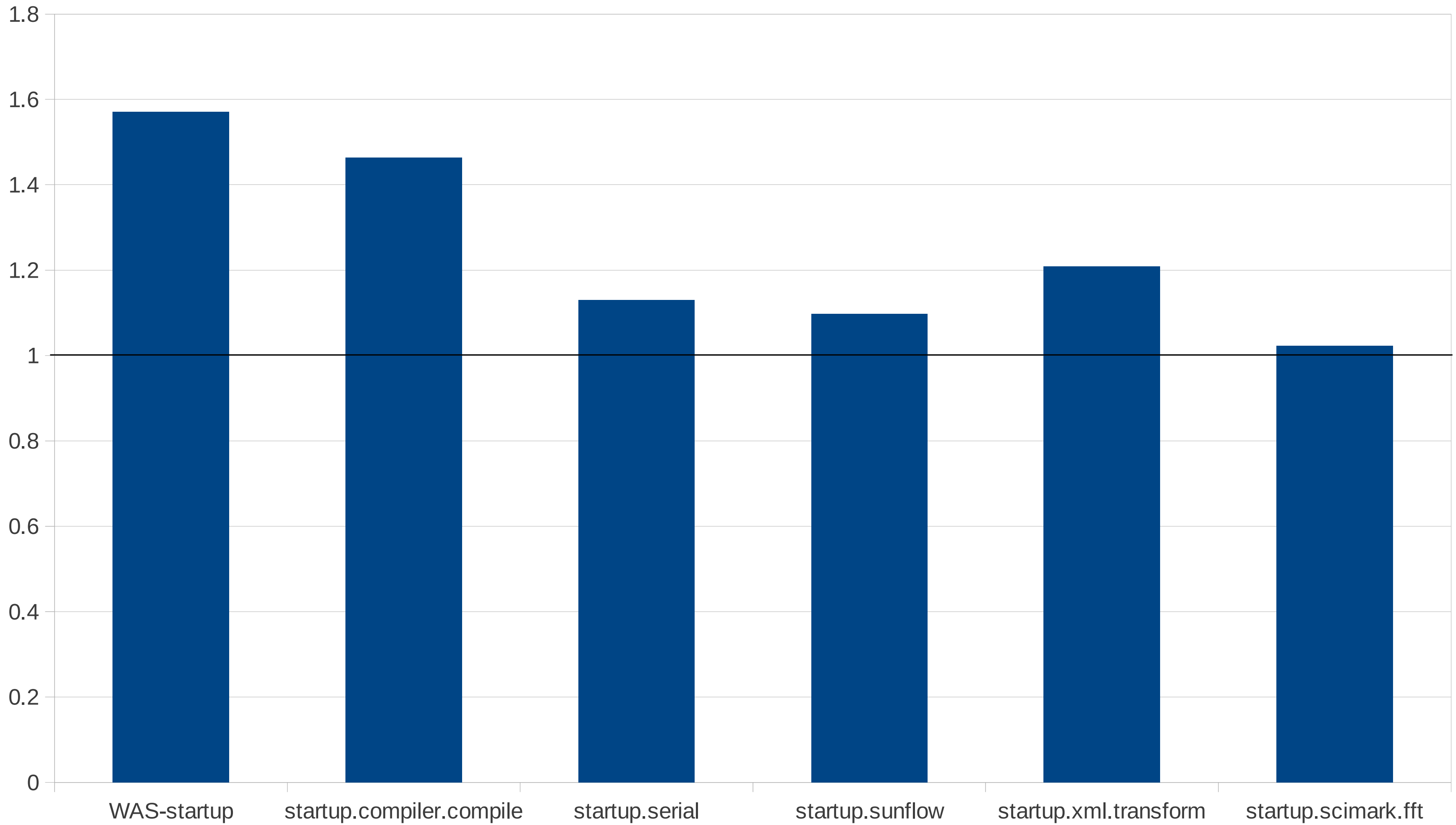
# Normalized Throughput Results with Initial Interpreter Profiling




(Higher is better)

© 2013 IBM Corporation

# Effect of Interpreter Profiling on Startup Time




(Lower is better)

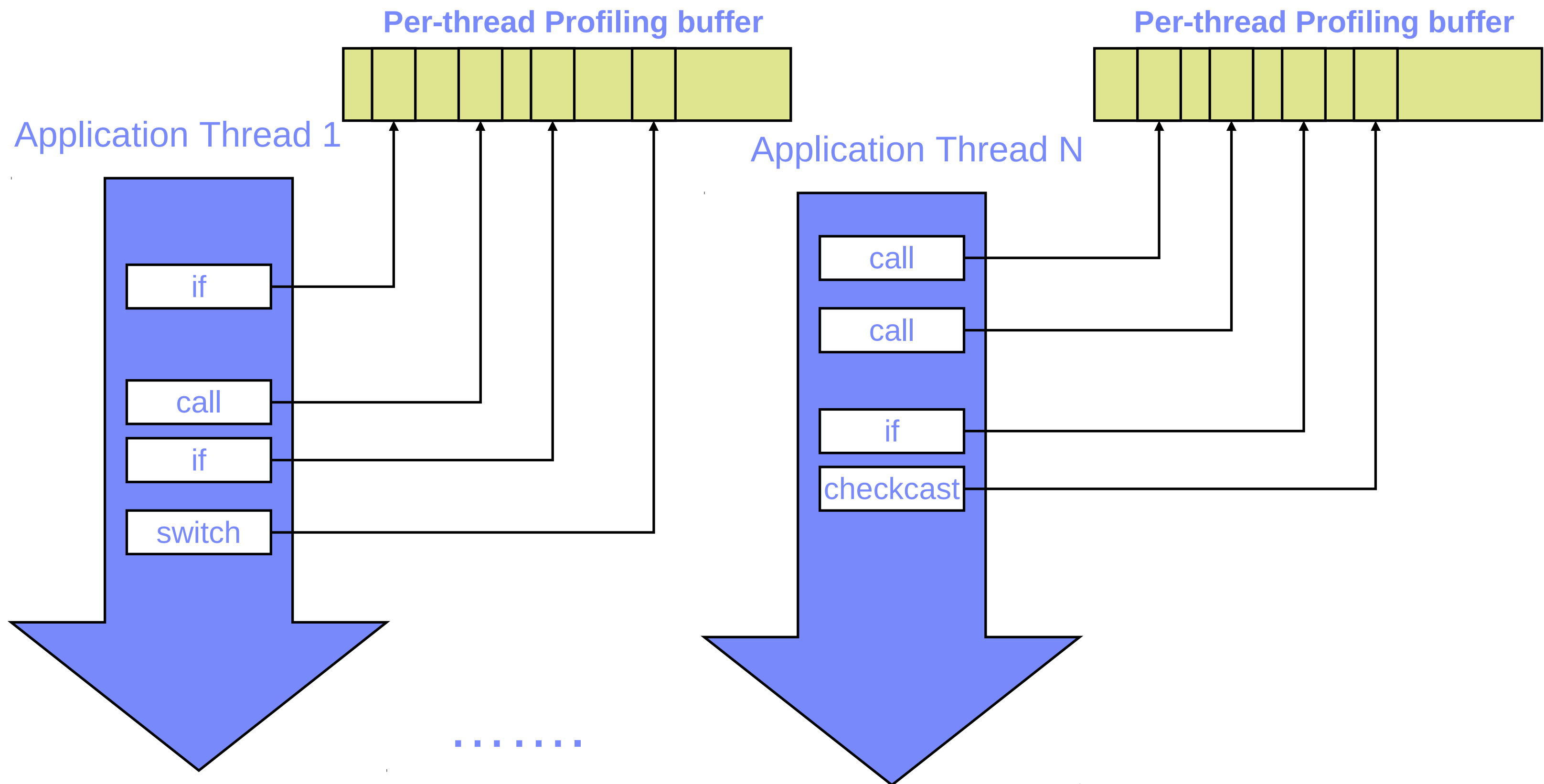
© 2013 IBM Corporation

# Interpreter Profiler Design

© 2013 IBM Corporation

# J9 Interpreter Profiler Design

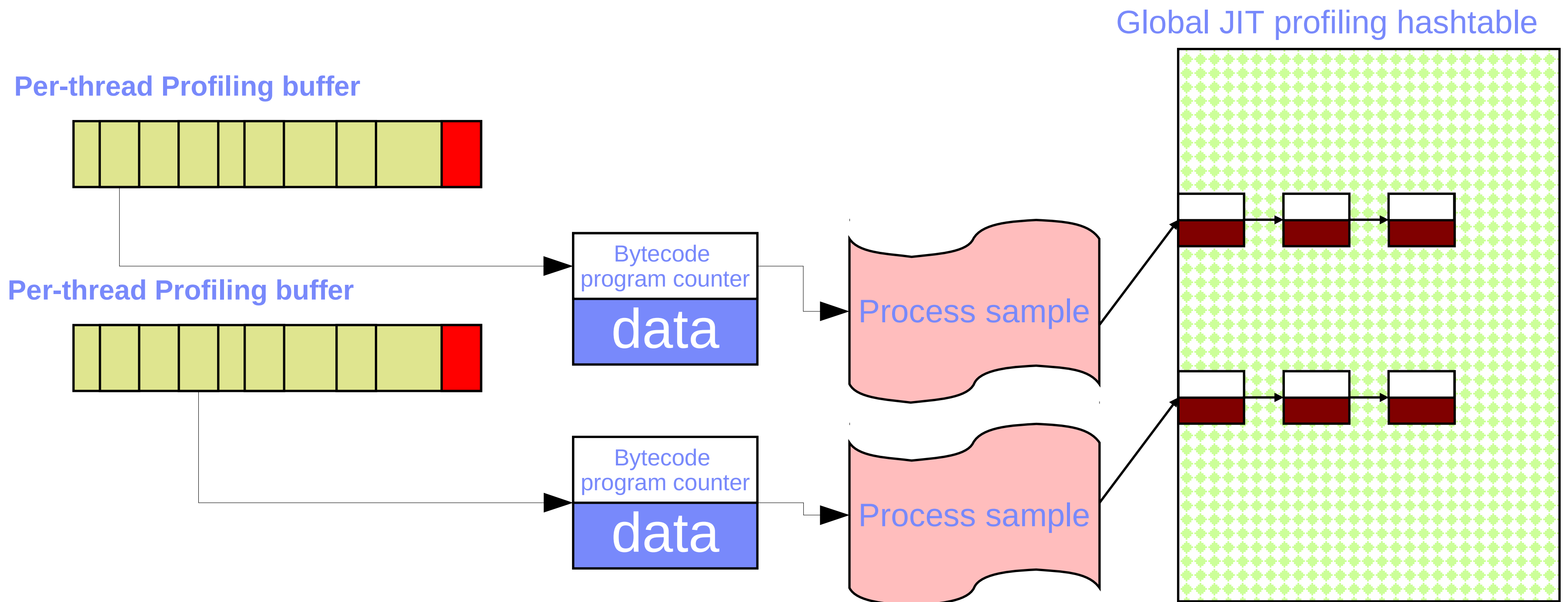
- Application threads use buffered approach to  collect data




© 2013 IBM Corporation

# J9 Interpreter Profiler Design

- The JIT runtime processes the data on the application threads and populates an internal global data structure
- Processing triggered by a buffer full event

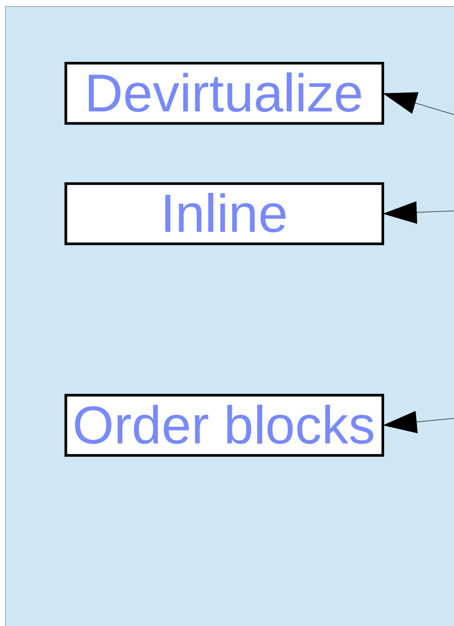



© 2013 IBM Corporation

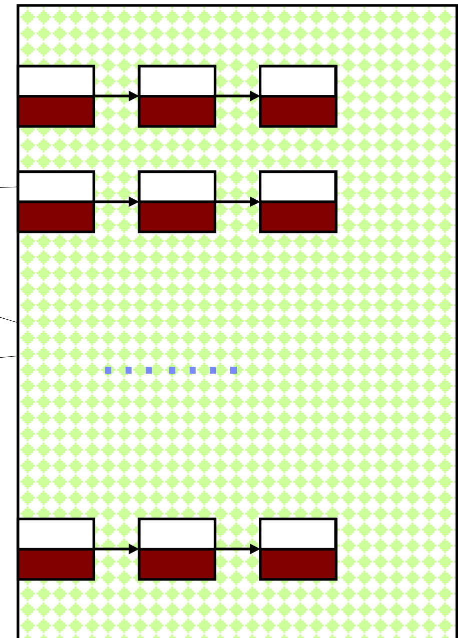
# J9 Interpreter Profiler Design

- JIT compiler consults the profiling hashtable in various stages of compilation

Compilation Thread




Global JIT profiling hashtable



© 2013 IBM Corporation

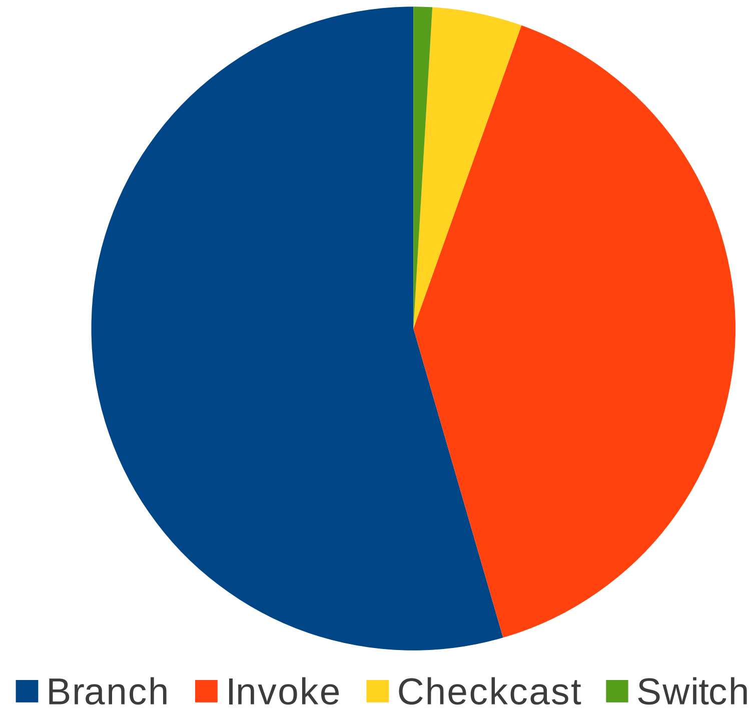
# J9 Interpreter Profiler Implementation Details

- Data collected:
  - Branch direction
  - Virtual call targets
  - Switch statements
  - Instanceof and checkcast runtime types

- Sources of overhead:
  - Populate the per-thread interpreter profiler buffer ( 4% )
  - Scanning the buffer (4%)
  - Process and populate the global hash table (92%)
  - Compilation thread consulting the global hash table (negligible)

© 2013 IBM Corporation

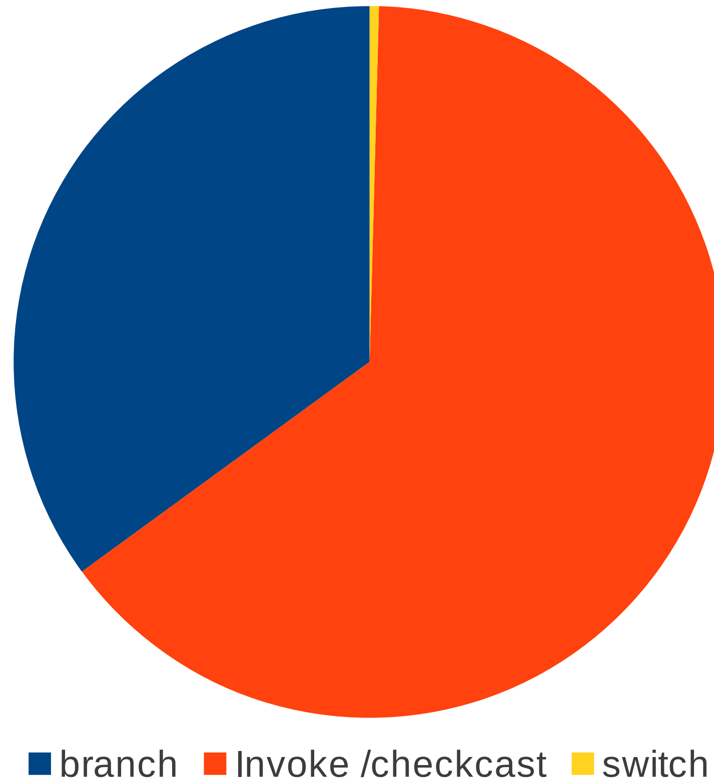
# Distribution of Raw Interpreter Profiling Data




- Raw data consists of over 50% branches

© 2013 IBM Corporation

# Distribution of Processed Interpreter Profiling Data




- Processed data is dominated by invoke and checkcast information
- Suggests branches are re-executed more times than calls

© 2013 IBM Corporation

# Optimizations

© 2013 IBM Corporation

# Problem 1: Synchronization overhead on global hash table

- Global hash table is accessed and updated by multiple threads
- Common Solutions to ensure correctness:
  - Lock / mutex
  - Read/Write lock
  - Lock striping (ex: java/util/ConcurrentHashMap)
  - Lock free (non-blocking) hash table
- Problem: All these solutions rely on (expensive) hardware atomic operations

# Optimization: Remove synchronization

- Constraints:
  - Insertion involves carefully crafted sequence of operations
  - No deletions
  - Do not re-size
- Implications:
  - Contention between threads when adding entry could result in lost entry

© 2013 IBM Corporation

## Problem 2: Buffer processing done on application threads

- Drawback: Application thread pauses while buffer processing occurs

## Optimization: Dedicated interpreter profiling processing thread pool

- Benefits:
  - Asynchronous model: threads do not pause for processing
  - Hide overhead on multi-cpu machines
- More opportunities for tuning:
  - Thread pool size
  - Drop buffers
  - Option to delegate buffer processing back to application threads

© 2013 IBM Corporation

## Problem 3: Over abundance of branch entries in buffers

- Branch bias is the important piece of data to drive block ordering
- Branch frequency of lesser importance

## Optimization: Skip processing branch buffer records in random fashion

- Benefit:
  - Reduces time spent processing entries
  - Skipping raw record in a random fashion ensures branch bias is not affected
- Does not improve footprint

© 2013 IBM Corporation

# Problem 4: Not all profiling information is used

- Some methods are not executed frequently enough to warrant compilation
- Each compiled method has an excess of profiling information

# Optimization: reduce time spent profiling a method

- Profile only last 'N' invocations of a method
- 'N' tuned based on application characteristics
- Benefits:
  - Reduces time spent processing entries by reducing number of entries generated
  - Footprint savings of around 50% by reducing little-used entries
    - Resizing of global hash table is less important
  - Bias profiling information to just before compilation

© 2013 IBM Corporation

# Problem 5: Expensive validity check caused by class unloading

- Class unloading in Java:
  - An optimization to help reduce memory use
  - A class may be unloaded if its defining class loader may be reclaimed by the garbage collector
  - Memory used by class is freed (including loaded bytecodes of the methods)
- Possible interactions with interpreter profiling:
  - Processed profiling data may come from an unloaded class
  - Processed profiling data may refer to an unloaded class
- Problem: each entry needs to be checked for validity before being accessed
  - This validity check is very expensive!

# Optimization: reduce times validity check needs to be performed

- Idea: Can avoid validity check if no class unloading has happened since last validity check.
  - Add a version ID to each hash table entry
  - Compare entry version ID to current global version ID
  - Store global version ID in entry version ID when entry is validated

© 2013 IBM Corporation

smarter software for a smarter planet


# Problem 6: Continuous profiling

- Some applications continuously generate new profiling information
- Common in many flat-profile applications:
  - Example: rules based application that uses reflection API
- Results in a throughput penalty at steady state

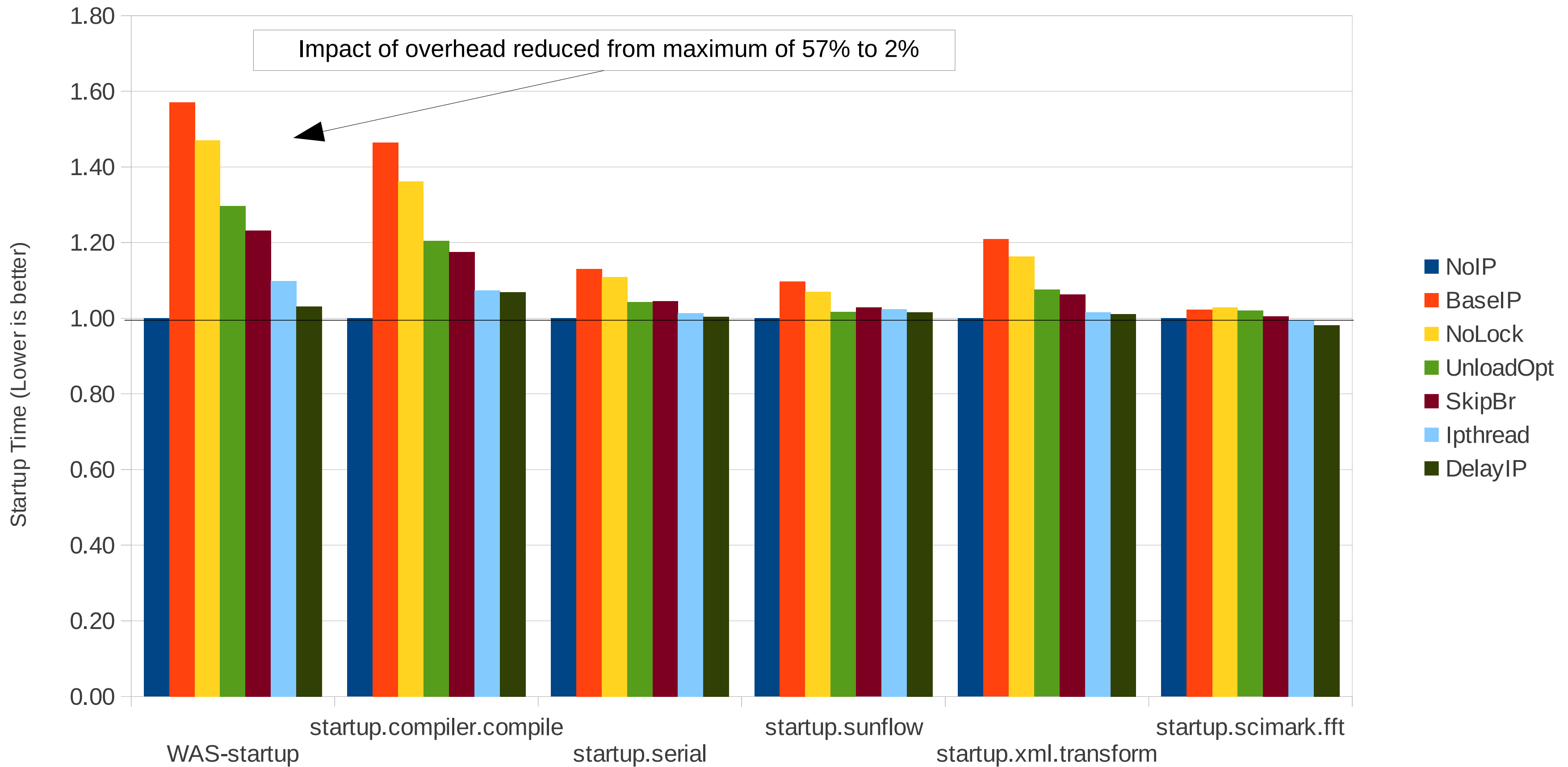
## Optimization : stop profiling when benefit is small

- Stop profiling when interpreter's share of time is below certain threshold
- Phase change detection necessary to re-enable profiling
- Applications discussed in this presentation do not exhibit this behaviour

© 2013 IBM Corporation

# Normalized Startup Results

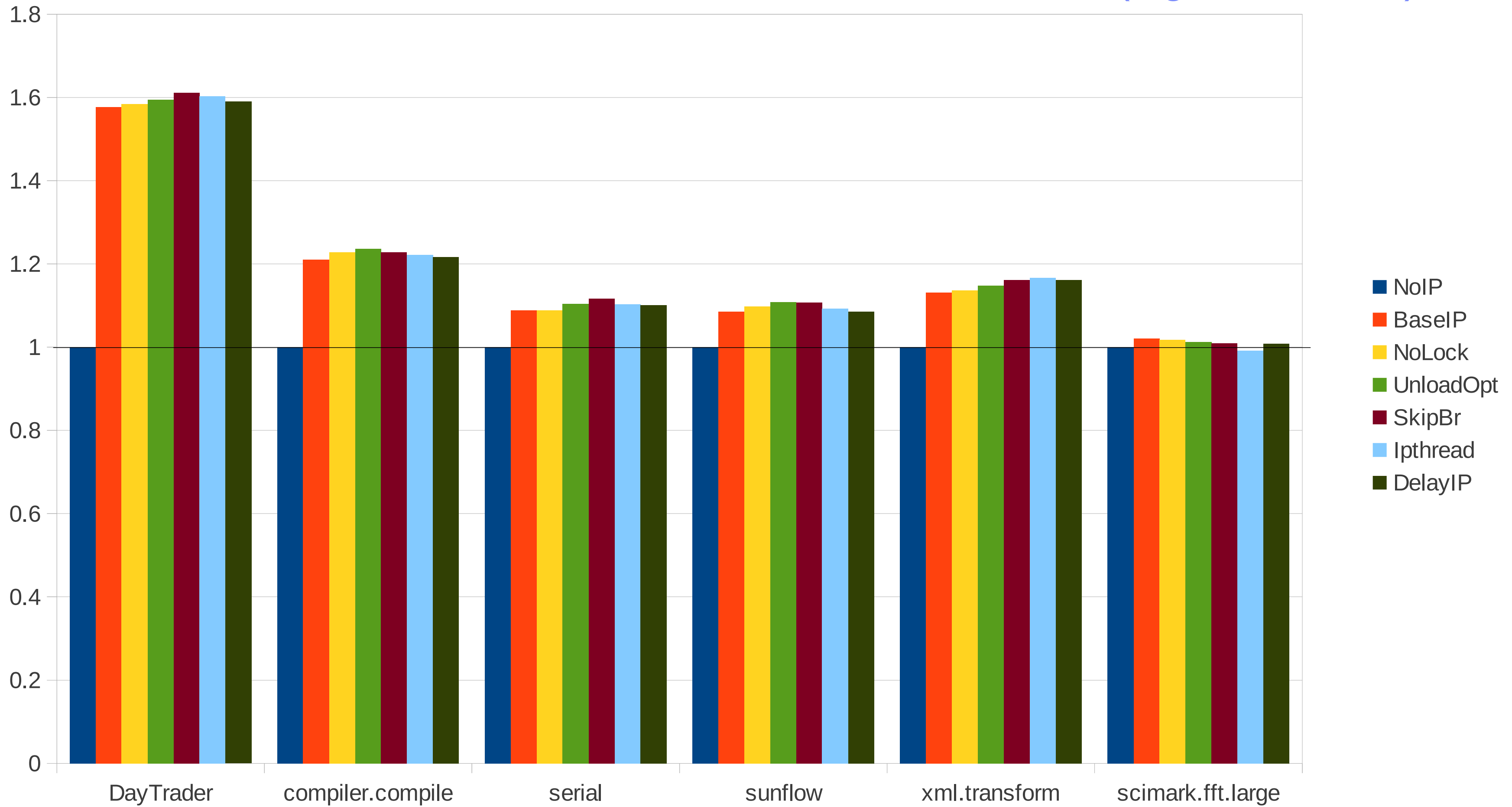
(lower is better)




- For enterprise-class applications: class unload optimization has biggest impact, followed by eliminating synchronization
- For general applications: eliminating synchronization has biggest impact

© 2013 IBM Corporation

# Normalized Throughput Results




- Throughput improves by 58% running Websphere Application Server

© 2013 IBM Corporation

# Summary

- Profile Driven Feedback is a necessity in any production JVM
  - Throughput performance benefits up to 58%
- Unique characteristics of flat profile applications make interpreter profiling an especially effective solution to generate profiling data
- Interpreter profiling overhead can degrade application startup time significantly
- Optimizations reduce interpreter profiling overhead:
  - No synchronization on global profiling data structure
  - Separate thread for processing buffers
  - Skip processing branches in buffers
  - Only profile last 'N' invocations of a method
  - Class unload optimization
  - Turning off interpreter profiling once steady state is reached
- Initial startup overhead of 57% can be reduced down to a few percent without any throughput loss

© 2013 IBM Corporation

# Questions?

© 2013 IBM Corporation