IBM Research - Tokyo

Identifying the Sources of Cache Misses in Java Programs Without Relying on Hardware Counters

Hiroshi Inoue and Toshio Nakatani
IBM Research - Tokyo

Motivation and Goal

- Cache miss information from Hardware Performance Monitor (HPM) is useful for runtime optimizations
  - Prefetch injection: Adl-Tabatabai *et al.* 2004
  - Object placement optimization: Schneider *et al.* 2007, etc..

☹ HPM is difficult to use in the real world

- HPM may require a special device driver and root privilege
- Only one process can use hardware at a time

Our Goal

- *To enable runtime optimization by identifying the sources of cache misses without using HPM*

Presentation Overview

- **Motivation and Goal**

➔**Analysis**
  - Key Observation
  - Our Technique
  - Evaluation in Coverages

- **Optimization**
  - Object Alignment and Collocation Optimizations
  - Our Techniques
  - Performance Evaluation

- **Summary**

Key Observation

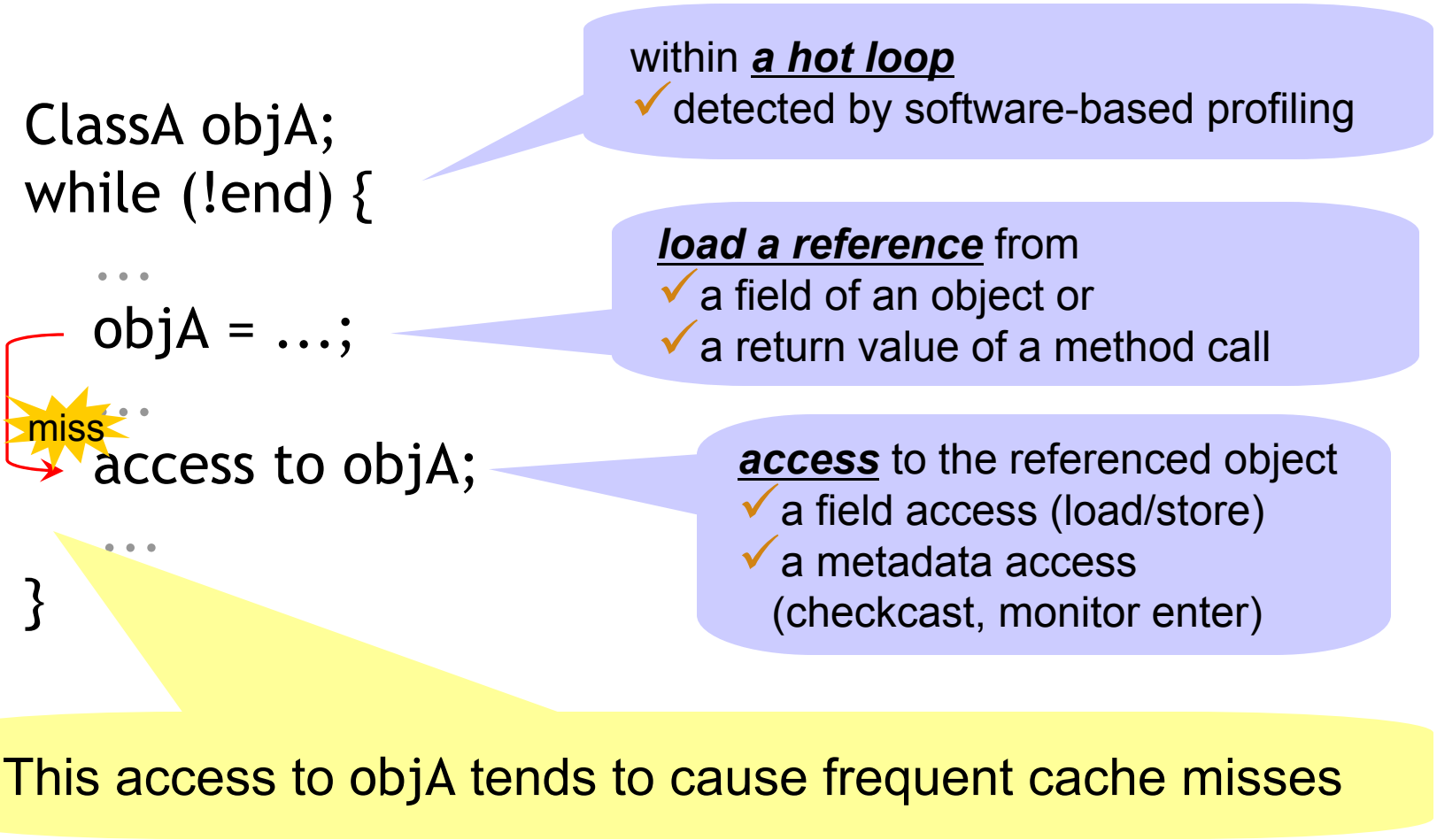
- Many cache misses in Java programs are caused in a simple idiomatic code pattern

Pattern

- *load a reference and touch the referenced object in a hot loop*

➔We can heuristically identify instructions and classes that may cause frequent cache misses by *matching hot loops with the idiomatic pattern*

So Simple Basic Code Pattern Tends to Cause Frequent Cache Misses



Anti-Pattern That Rarely Causes Cache Misses

```
ClassA objA;
while (!end) {
    ...
    objA = ...;
    ...
    access to objA;
    ...
}
```

within ***a hot loop***
- ✓ detected by software-based profiling

if the first load is ***loop invariant***

access to the referenced object
- ✓ a field access (load/store)
- ✓ a metadata access (checkcast, monitor enter)

This access to objA does *NOT* cause frequent cache misses

Implementation

- We implemented the analysis in 32-bit IBM J9/TR JVM Java 6 SR2
- We execute pattern matching in JIT compiler
  - after applying optimizations including method inlining and loop-invariant code motion
  - using execution frequency information obtained by software-based profiling to identify the hot loops
  - only for hot methods that are recompiled with higher optimization levels than the initial level

Evaluation

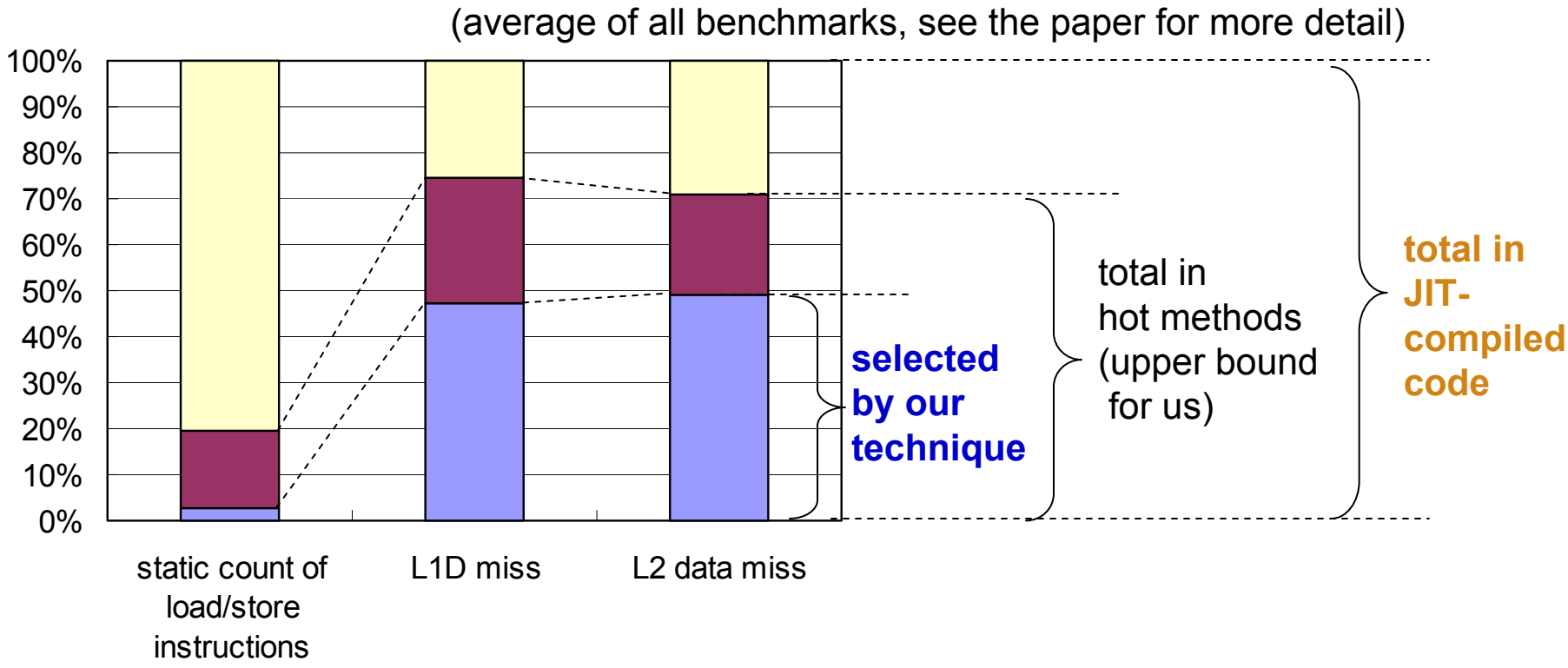
We show ***coverages*** by instructions identified by our technique for

- ✓ Static count of load and store instructions
- ✓ L1D cache misses
- ✓ L2 cache data misses
- ✓ Memory accesses

Environment

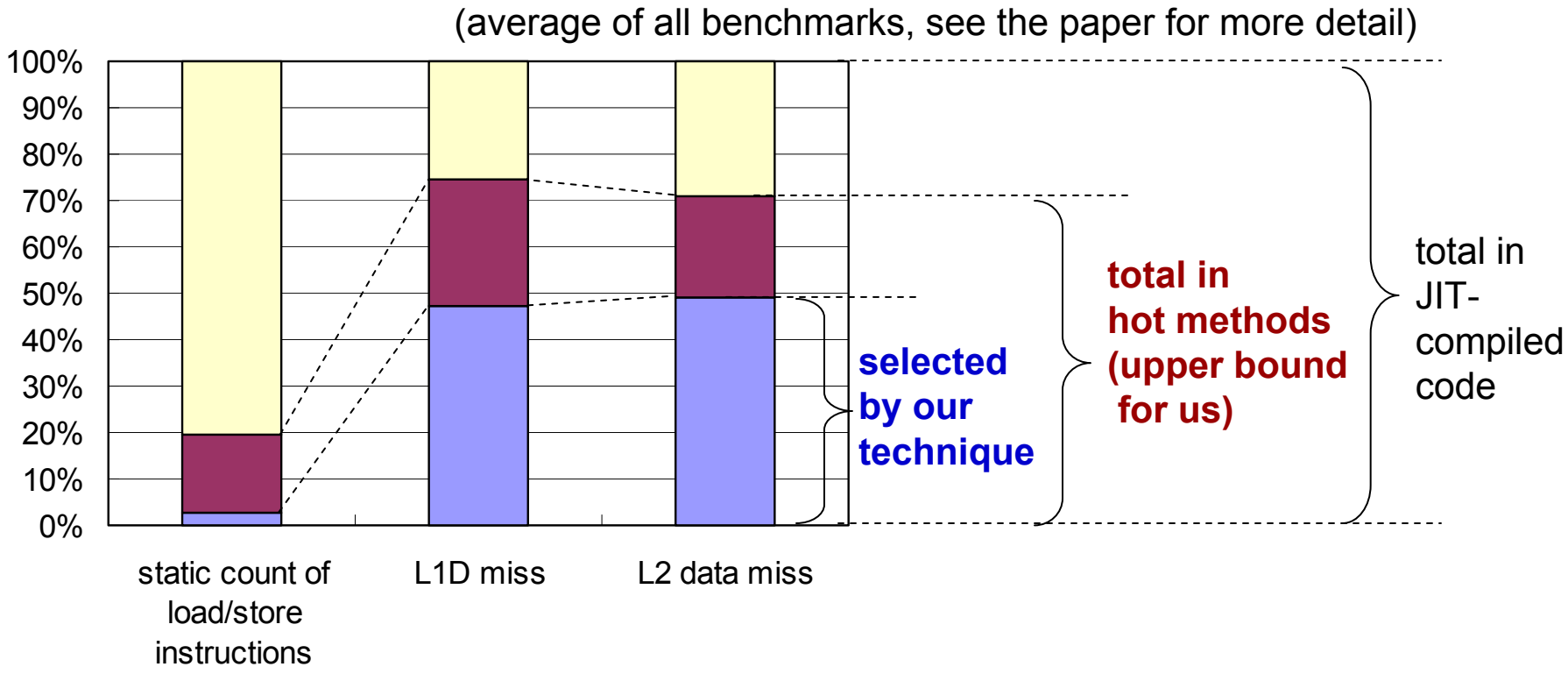
- Processor: POWER6 4.0GHz
  - 64-KB L1D cache, 64-KB L1I cache
  - 4-MB unified L2 cache
  - 128-byte cache line for both L1 and L2 cache
- OS: RedHat Enterprise Linux 5.2
- Benchmark: SPECpower_ssj2008, SPECjbb2005, SPECjvm2008, DaCapo-9.12

Coverages by Instructions Selected by Our Technique



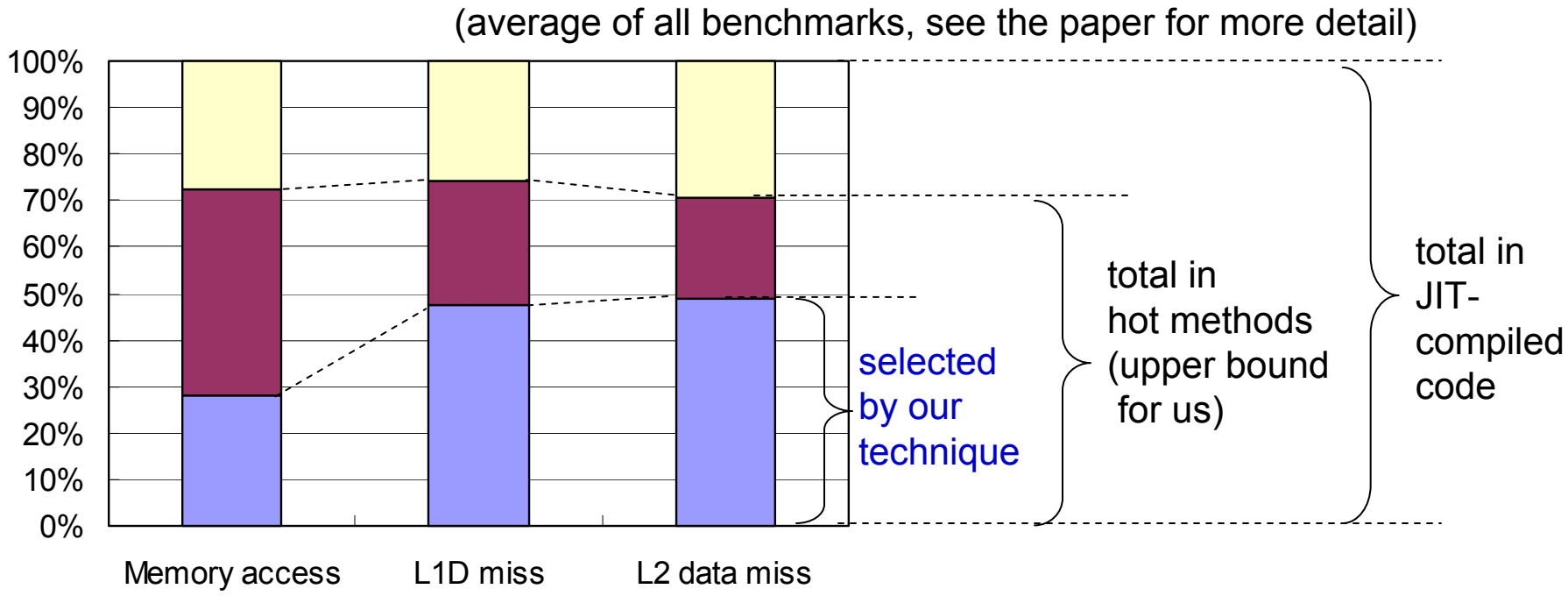
☺ Our technique selects only **2.8%** of load and store instructions and they cover about **50%** of the total cache misses compared to total in JIT-compiled code

Coverages by Instructions Selected by Our Technique



☺ Our technique selects **14%** of load and store instructions and they cover **64%** in L1 miss and **69%** in L2 miss compared to total in hot methods

Not Only Accessed Frequently



☺ Instructions selected by our technique causes
about 2x more cache misses per execution
than the instructions not selected

Presentation Overview

- **Motivation and Goal**

- **Analysis**
  - Key Observation
  - Our Technique
  - Evaluation in Coverages

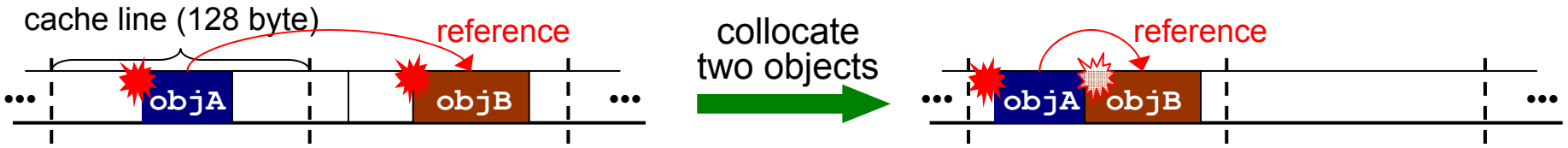
➔**Optimization**
  - Object Alignment and Collocation Optimizations
  - Our Techniques
  - Performance Evaluation

- **Summary**

Application in Runtime Optimization

- We implemented two object placement optimizations to reduce cache misses
  - Object alignment

  - Object collocation



➔ Both techniques can reduce cache misses from two to one

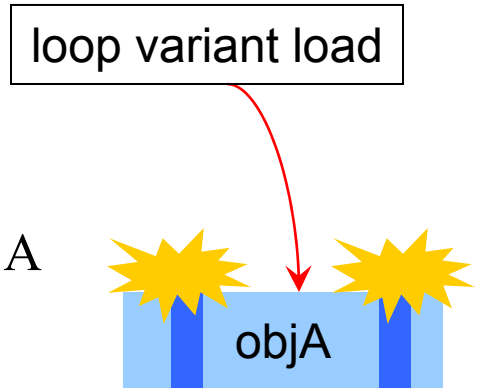
Our Approach for Optimizations

- We identify target **classes** for optimization based on our analysis in JIT compiler
  - If two accesses to distinct fields of a object are selected in a hot loop
    - ➔select the class for target of alignment optimization
  - If two accesses to objects, where one has a reference to another, are selected in a hot loop
    - ➔select the pair of classes for targets of collocation optimization
- We do optimizations both in garbage collector (for objects in tenure space) or at allocation time (for objects in nursery space)

Pattern for Alignment Optimization

- Derived from the basic pattern

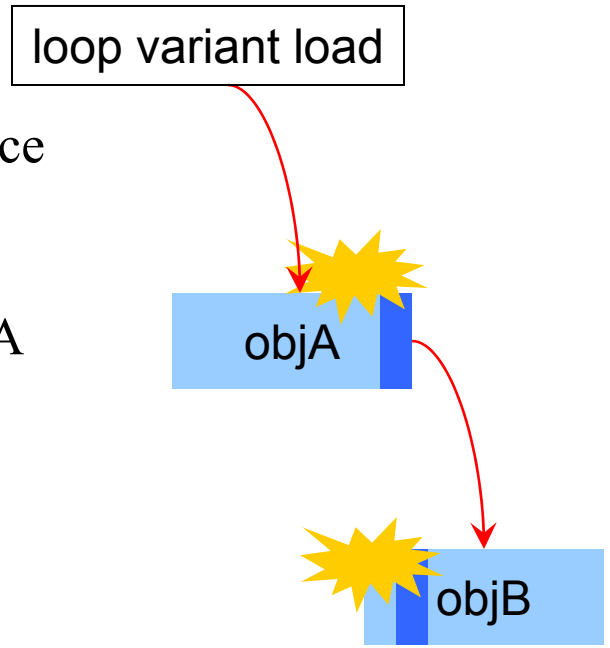
```
ClassA objA;
while (!end) { // in a hot loop
    …
    // 1) first, load a reference to a ClassA's instance
    objA = …;
    …
    // 2) then, access at least two different fields of objA
    access to objA.field1;
    …
    access to objA.field2;
    …
}
```



➔ **ClassA** is selected for target of alignment optimization

Pattern for Collocation Optimization

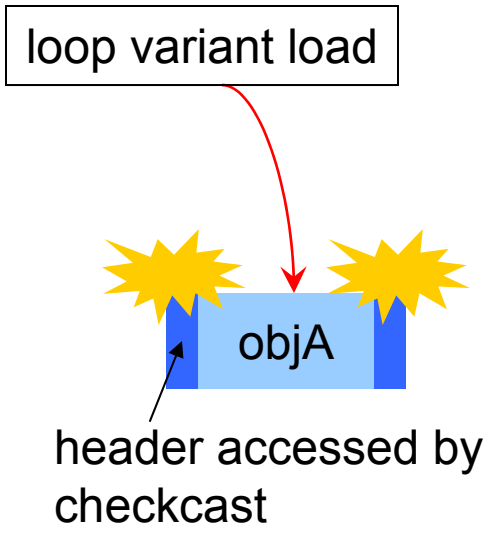
```
ClassA objA;
ClassB objB;
while (!end) { // in a hot loop
    …
    // 1) first, load a reference to a ClassA's instance
    objA = …;
    …
    // 2) next, load a reference of ClassB from objA
    objB = objA.referenceToClassB;
    …
    // 3) then, access at least one field of objB
    access to objB.field1;
    …
}
```



➔ pair of **ClassA** and **ClassB** is selected for target of collocation optimization

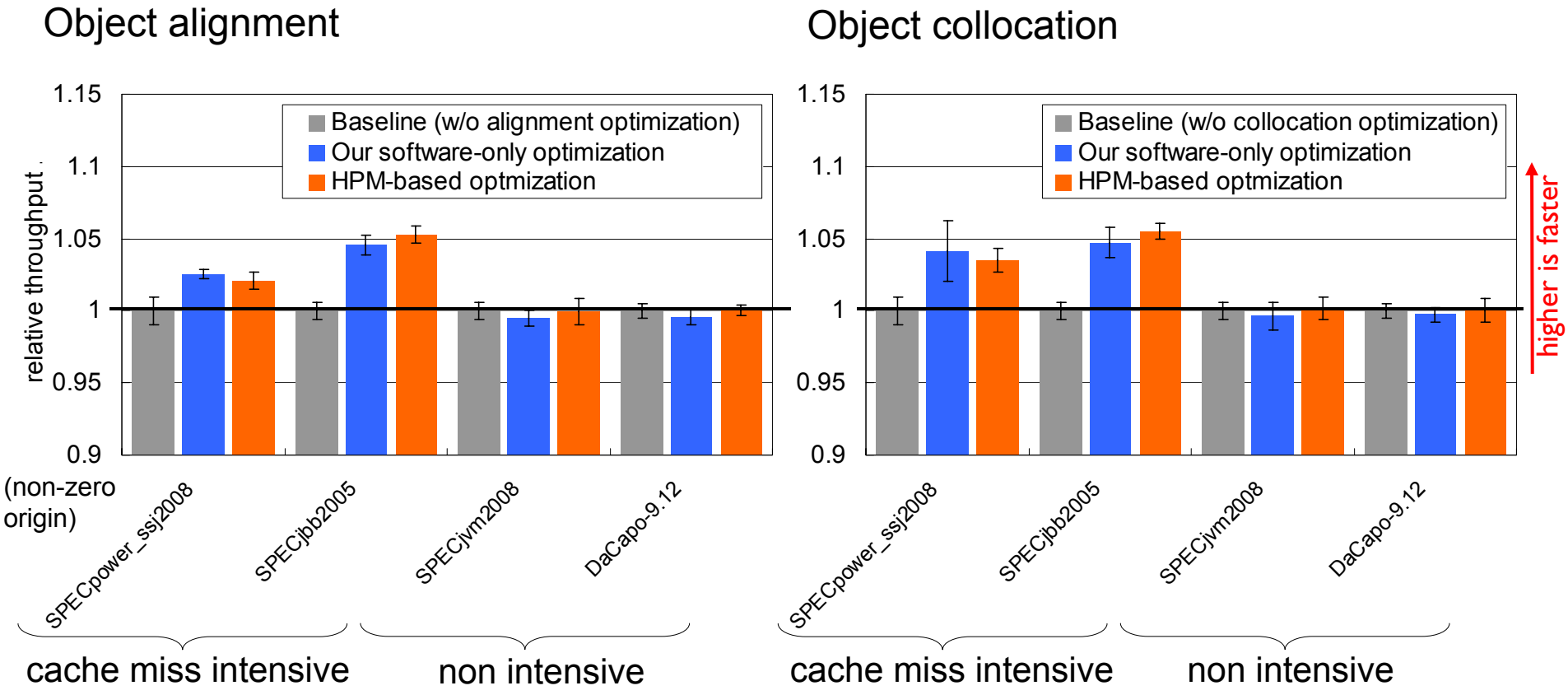
Special Handling For `checkcast` Operation

```
ClassA objA;
ClassS objS;  // ClassS is a super class of ClassA
while (!end) { // in a hot loop
    …
    // 1) first, load a reference of a super class of ClassA
    objS = …;
    …
    // 2) next, cast objC to ClassA (access to object header)
    objA = (ClassA) objS;
    …
    // 3) then, access at least one field of objA
    access to objA.field1;
    …
}
```



➔ **ClassA** (not ClassS) is selected for target of alignment optimization
➔ Common pattern in HashMap or TreeMap accesses in Java

Performance Improvements by Optimizations



➔Our technique achieved comparable performance gains in cache-miss-intensive programs *without relying on the hardware help*

Remaining Challenges

- For Optimizations
  - Pattern matching in compiler cannot tell us the location of the objects in the Java heap (e.g. tenure or nursery)
  - An instance of a subclass of the identified target may cause cache misses

  ➔More detailed software-based profiling can help (in trade for additional overhead)

- For Analysis
  - Current pattern matching cannot identify frequent cache misses caused by conflicting writes from multiple thread

  ➔Different patterns and profiling information is required to achieve higher coverage

Summary

- We present a technique to identify the instructions and objects that frequently cause cache misses in Java programs without relying on the HPM
  - ➔Matching hot loops with simple idiomatic patterns worked very well for many Java programs
- We showed the effectiveness of our approach using two types of optimizations

backup

Coverage for each benchmark (L1D cache misses)