

Dynamic Profile-Guided Optimization in the BEA JRockit* JVM

Shirish Aundhe
Greg Eastman
Robert Kasten
Robert Knight

Introduction

- ◆ Itanium® Processor Family (IPF) provides HW support for dynamic execution monitoring
- ◆ HW feedback can be used to create execution profiles that enable and improve JIT optimizations
- ◆ In this talk we describe how the BEA JRockit Java* VM uses IPF HW feedback for JIT code optimizations

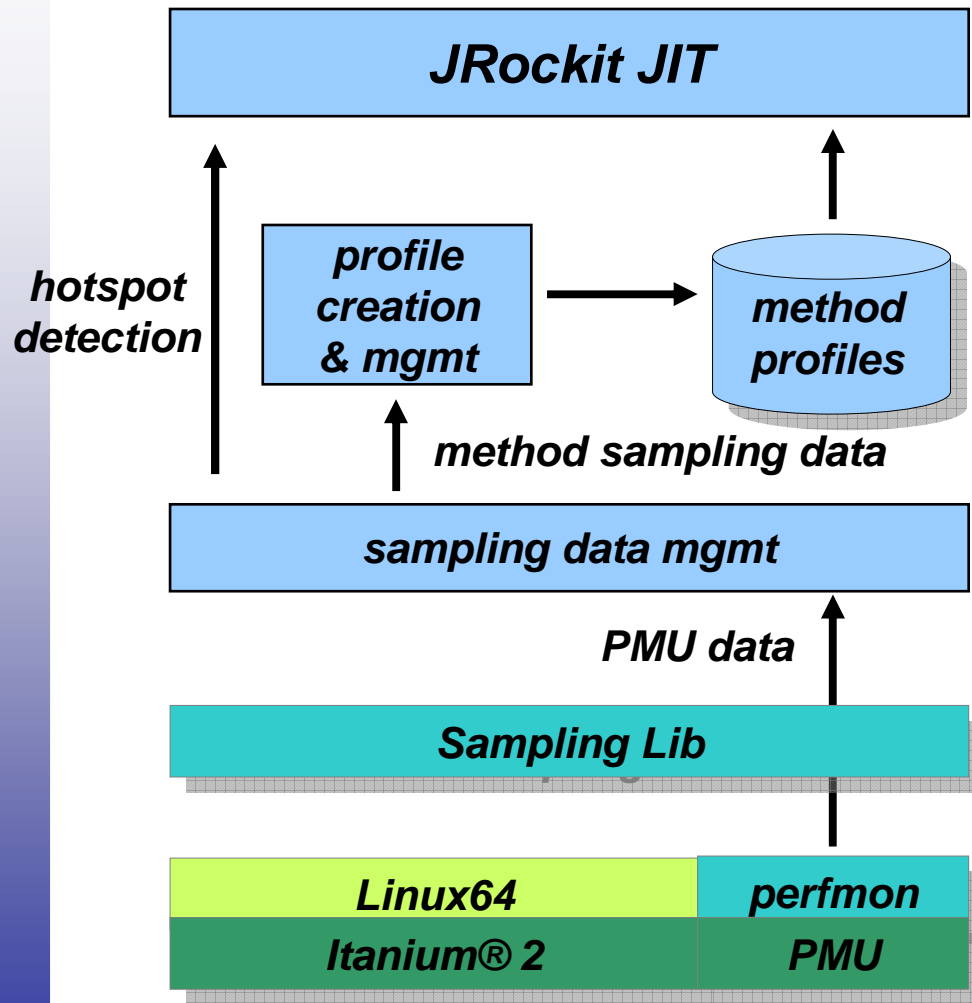
Outline

- ◆ Overview
- ◆ Dynamic Profile-Guided Optimization (DPGO)
- ◆ DPGO Profiles
- ◆ Performance
- ◆ Conclusions

JRokit Overview

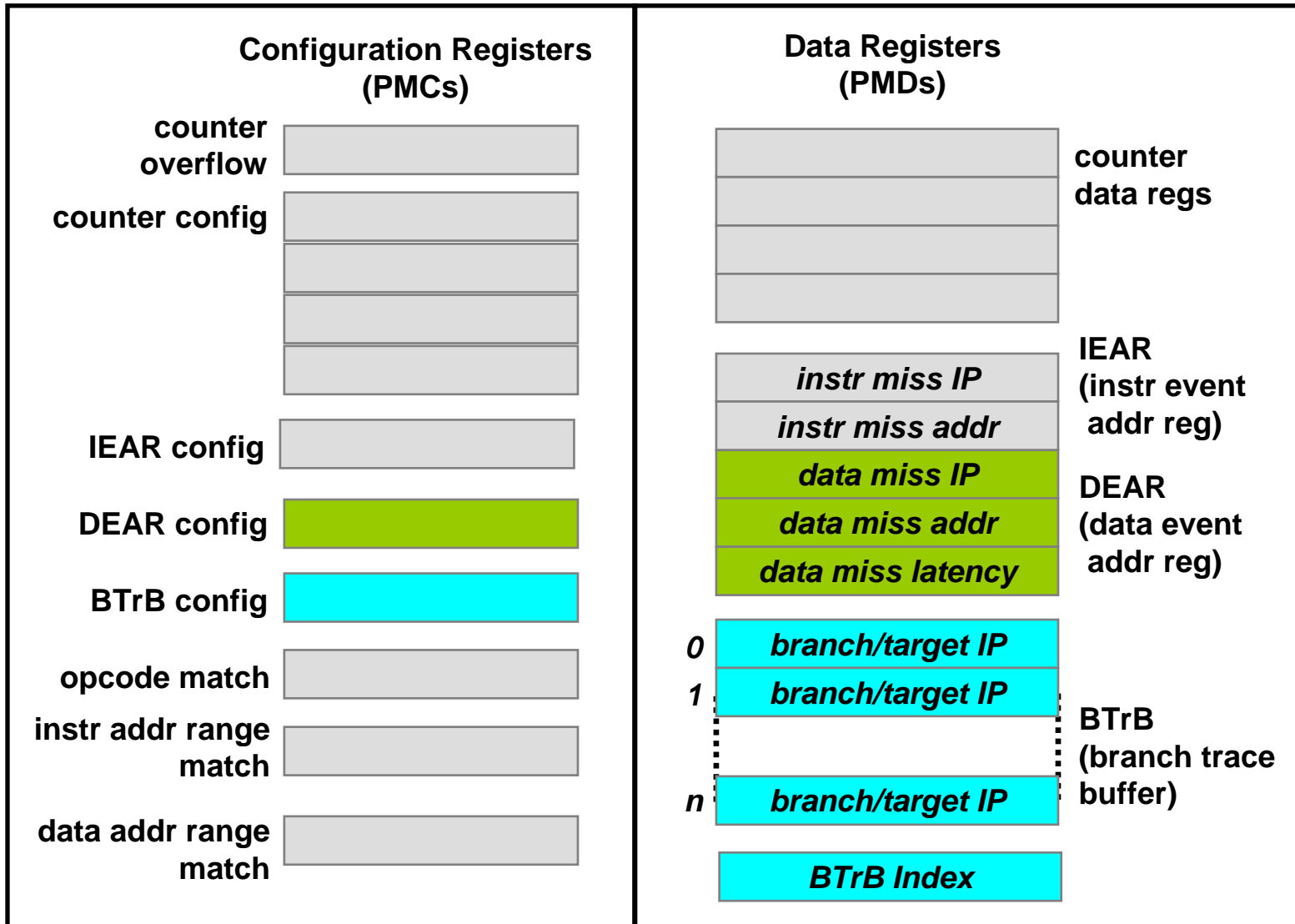
- ◆ JRokit – BEA’s enterprise-class Java VM & JIT
- ◆ JRokit JIT
 - Dynamic, optimizing compiler
 - No interpreter – all methods compiled
 - Optimizes frequently-executed methods
 - Hot methods detected by SW IP sampling
- ◆ JRokit with DPGO
 - IPF Linux64
 - DPGO sampling infrastructure
 - HW sampling data collection, profile creation
 - DPGO “starter set” optimizations
 - HW-based hotspot detection
 - Profile-guided inlining, prefetch
 - Available in Weblogic JRokit* 5.0 (<http://www.bea.com>)

JRockit DPGO Architecture



- ◆ DPGO JIT infrastructure
 - Manages sampling data
 - Detects hotspots
 - Creates edge & dcache-miss profiles
 - Decorates IR with profiles
 - Dedicated thread processes sampling data
- ◆ Sampling library
 - Event-based sampling
 - Per-thread data collection
 - Delivers data to client
- ◆ IPF Performance Monitoring Unit (PMU)
 - Branch trace data
 - Cache-miss data

IPF Performance Monitoring Unit



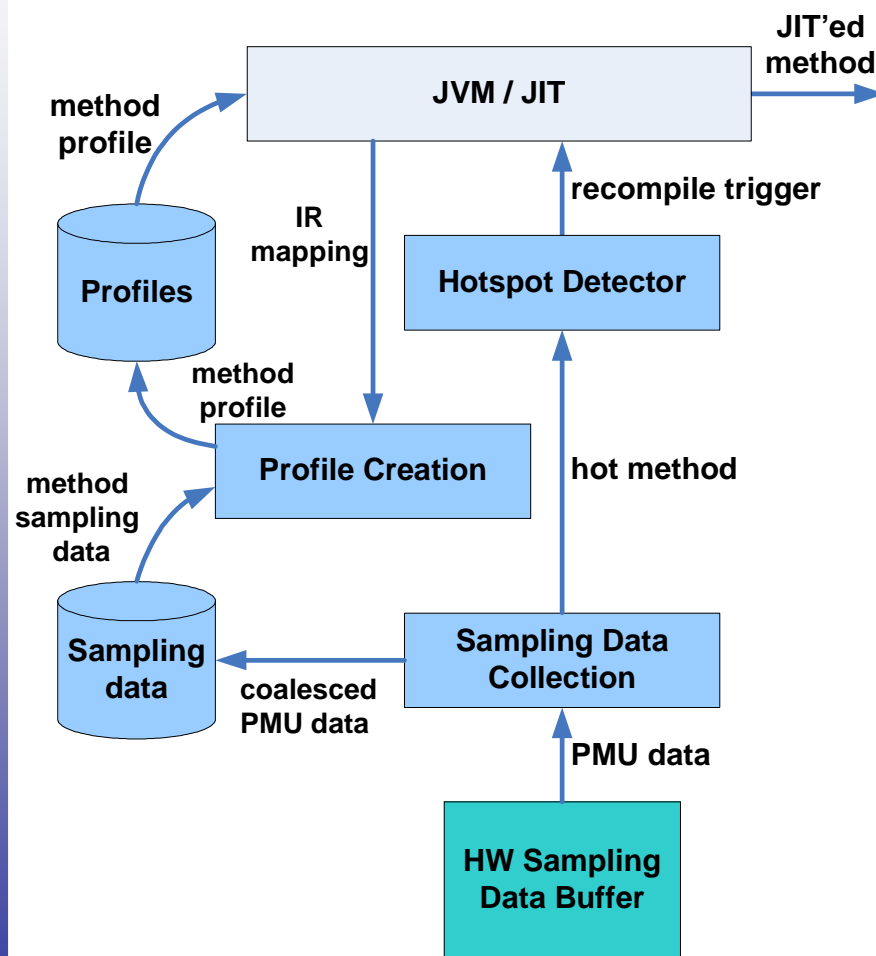
PMU Profiling Using Event-Based Sampling (EBS)

- ◆ Program event counter to count a hardware event
 - Instructions retired, branch instructions retired, etc.
- ◆ Program overflow value for event counter
 - “Sample-after value”
- ◆ Event counter overflow triggers an interrupt
- ◆ Collect sampling data during interrupt
 - BTrB data (branch trace), DEAR data (dcache miss)

HW Sampling Library

- ◆ User-mode library exports thread-centric, EBS services
- ◆ Supports multiple profiling schemes
 - **BTrB profiling**
 - count instruction retired events
 - collect branch trace samples
 - ~1 sample / ms
 - **DEAR profiling**
 - count dcache miss events
 - collect dcache miss samples
 - ~10 samples / ms
 - Sample-after values computed dynamically based on processor frequency
- ◆ Per-thread data collection
- ◆ Library buffers sampled data & delivers buffers to client listener
- ◆ Two usage models
 - Profiling
 - Event counting

JRockit DPGO Details

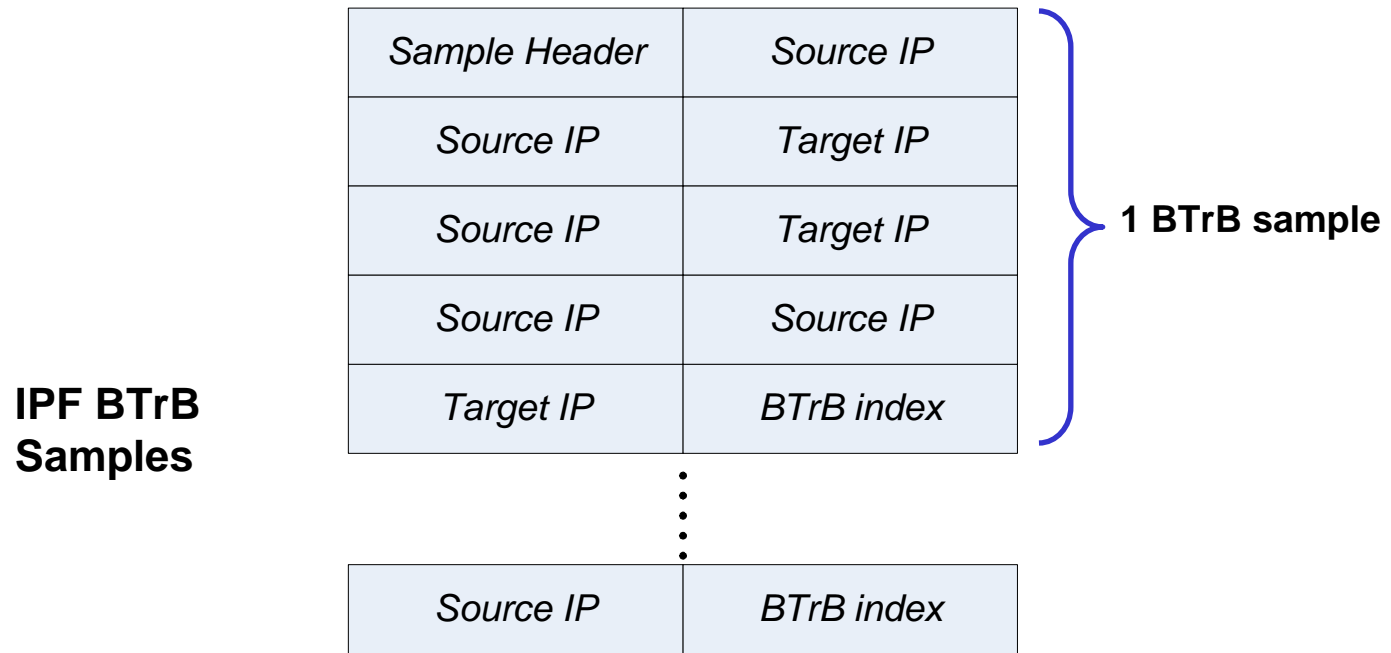


- ◆ HW sampling library delivers data buffers to sampling data collector
- ◆ Sampling data collector bins data by method & coalesces samples by count
- ◆ When method sampling threshold reached, Sampling data collector signals hot method detected to Hotspot detector
- ◆ Hotspot detector triggers recompilation by JRockit JIT
- ◆ JIT triggers on-demand profile creation
- ◆ Profiles are created from sampling data. Address-to-IR mapping provided by JRockit.
- ◆ JIT decorates IR with profiles
- ◆ Profiles are used by JIT during optimization

DPGO Profiles

- ◆ Edge profile
 - Profile of branch IPs/targets
 - Created with BTrB data
 - source IP, target IP, prediction info
 - Profile includes:
 - source IR location
 - target IR locations
 - mispredicts, probabilities, execution frequencies
 - Used for hot path optimizations, inlining, etc.
- ◆ Load miss profile
 - Profile of high-latency load instructions
 - Created with DEAR data
 - load IP, miss address, latency
 - Profile includes:
 - load IR location
 - average load latency
 - Used to guide data prefetching

Edge Profiling Example – IPF Branch Trace Buffer Samples



- ◆ HW sampling layer delivers raw BTrB data to JRockit
 - Taken branches: source IP + target IP
 - Not-taken branches: source IP only
 - Branch misprediction encoded in source IP

Edge Profiling Example – Method Data

**Method
Sampling
Data**

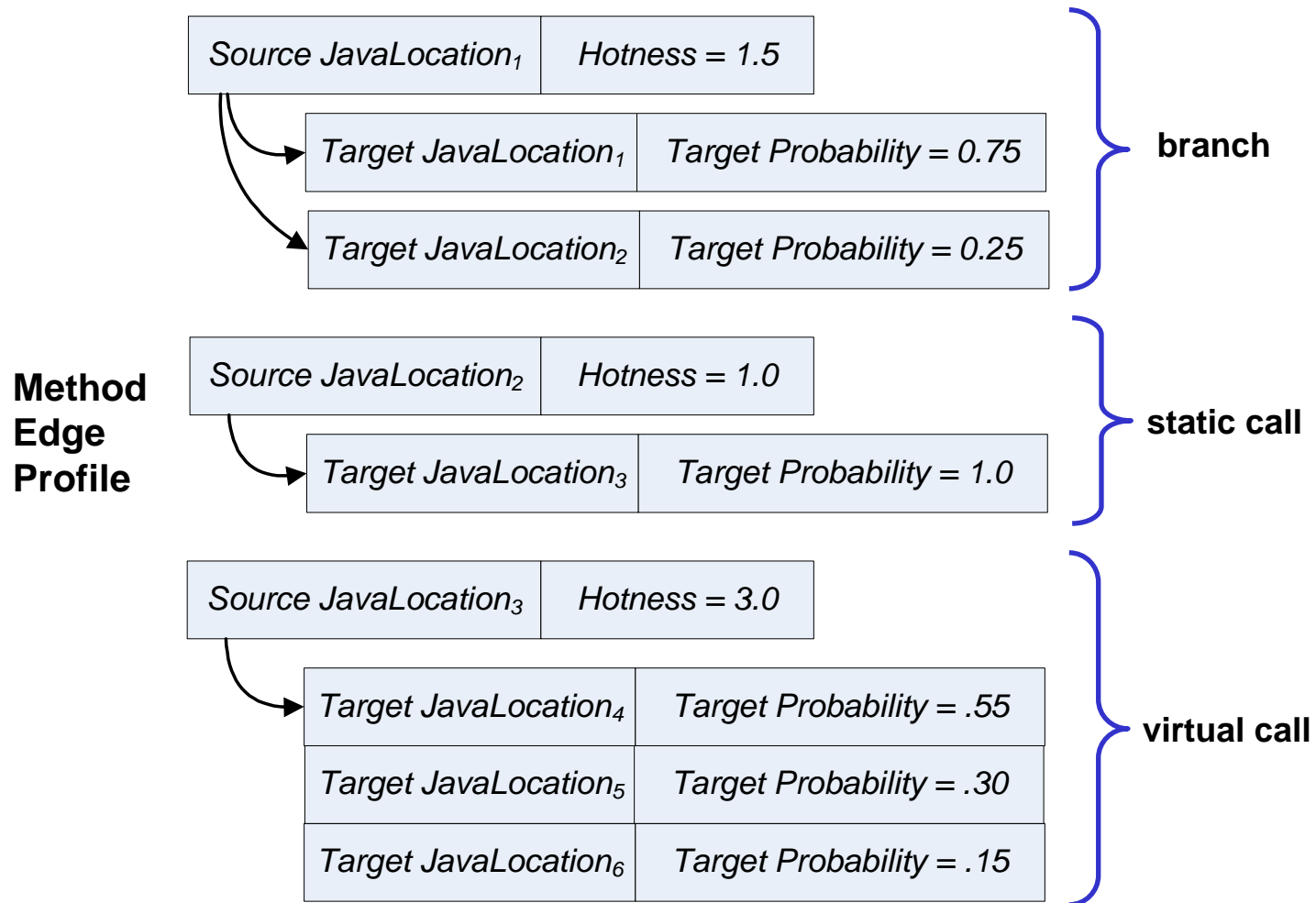
<i>Source IP₁</i>	<i>Target IP₁</i>	<i>Count</i>	<i>Mispredict Count</i>
	<i>Target IP₂</i>	<i>Count</i>	<i>Mispredict Count</i>
	<i>Target IP₃</i>	<i>Count</i>	<i>Mispredict Count</i>
<i>Source IP₂</i>	<i>Target IP₄</i>	<i>Count</i>	<i>Mispredict Count</i>

- ◆ JRocket bins samples by method and coalesces samples by edge

Mapping Addresses to IR

- ◆ Addresses in sampling data must be mapped locations in JIT IR
- ◆ JRockit provides capability for address/IR mapping
 - `JavaLocation getLocationForIP(void* address)`
- ◆ `JavaLocation = { method*, byte code index }`
- ◆ `JavaLocation` is a “sufficiently unique” way to locate any point in a Java program
 - Addresses in profiles are mapped to `JavaLocation`
 - Every instruction in IR has a `JavaLocation`
 - Profile “decoration” phase correlates profiles with IR instructions using `JavaLocation`

Edge Profiling Example – Creating Edge Profiles from Method Data



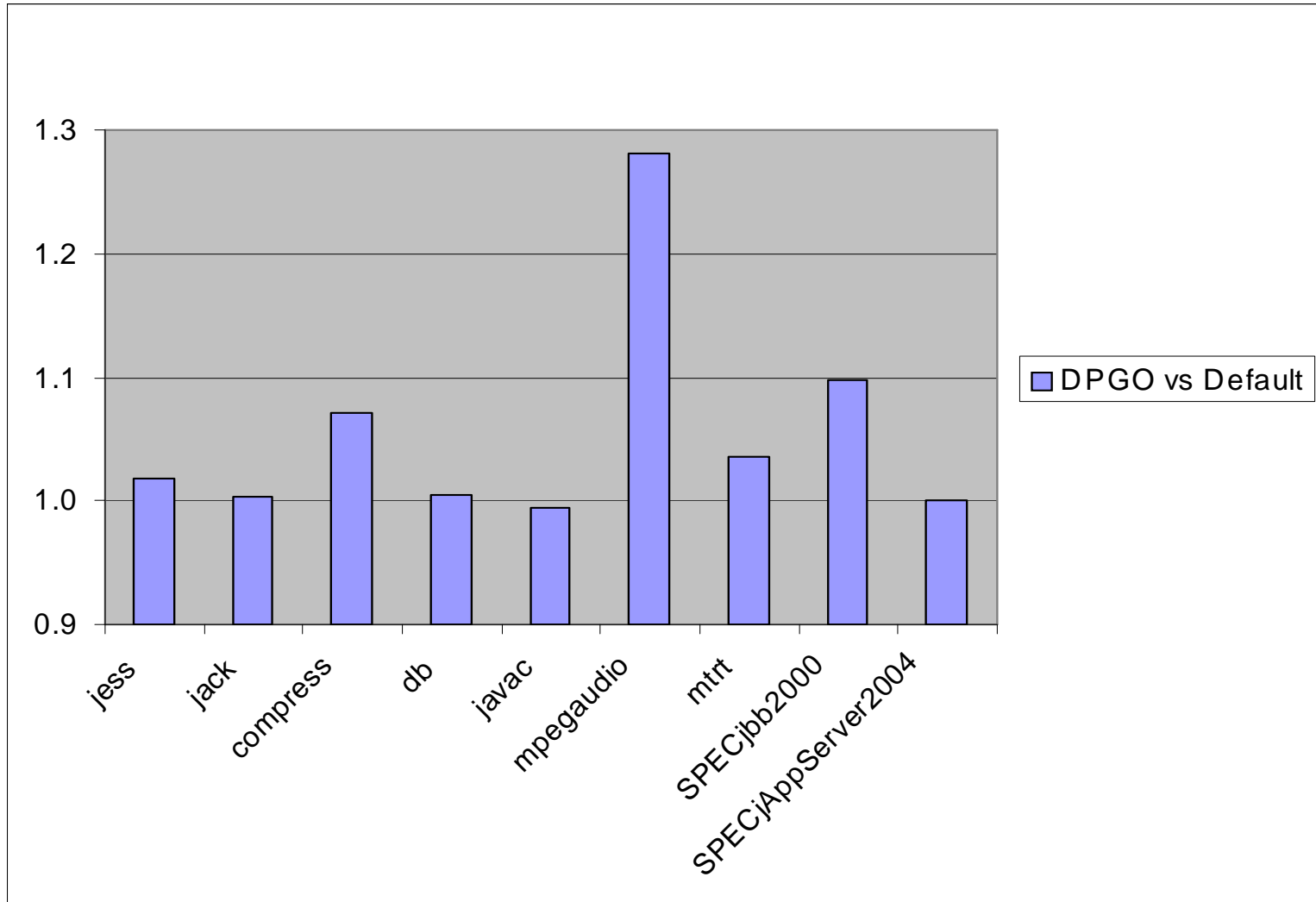
Method Inlining with Edge Profiles

- ◆ DPGO provides profile-derived measure of “hotness” when evaluating call-sites as candidates for inlining
- ◆ *Collection rate* quantifies hotness of caller method:
 - Dynamically measures how quickly method hotness threshold was reached
 - $0 < \text{collection rate} \leq 1$
- ◆ *Instruction execution frequency* quantifies how frequently the call instruction is executed relative to the caller:
 - $\text{Frequency}(\text{instr}) = \text{callee count} / \text{caller return count}$
- ◆ Call-site hotness =
 $f(\text{collection rate}, \text{frequency}(\text{call instr}))$

Profile-Guided Prefetch Optimization

- ◆ Goal is to minimize stalls due to sequential high-latency load misses induced by pointer chasing
 - Optimization uses heuristic to prefetch cache lines containing dependent objects
 - Leverages spatial co-allocation
- ◆ Identifies hot high-latency loads using load miss profile
- ◆ Discovers chain of dependent hot loads
 - Special cases for some operations, such as calls with NULL check, type-checks, etc.
- ◆ Inserts prefetches at the definition of the “root” load

JRokit DPGO Performance



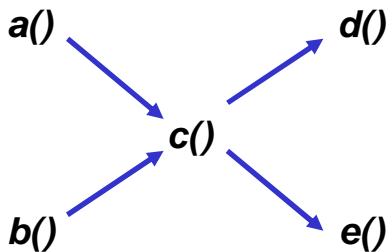
◆ Not tuned (yet) for SPECjAppServer

JRokit DPGO Benefits

- ◆ Low overhead
 - ~2.5% for sampling and profile creation
 - Memory requirements not excessive (< size of JIT'ed code)
- ◆ Fine-grained, accurate hotspot detection
 - On SPECjAppServer2002, ~ performance achieved with fewer hot methods detected (~750 vs. ~1100)
- ◆ Micro-architecture profiles
 - Cache, TLB misses
 - Branch mispredictions
- ◆ Provides better quantitative measure of hotness
 - Instantaneous & accurate hotness ratings

Limitations to HW Profiling

- ◆ HW profile data alone is not sufficient for some optimizations
 - Guarded devirtualization example:
 - `Object.hashCode()` is often used but rarely redefined
 - Suppose `MyClass` does not redefine `hashCode()`, and `MyClass.hashCode()` is frequently executed at some call site
 - HW profiling would identify `Object.hashCode()` as dominant target of calls to `MyClass.hashCode()`
 - Using only HW profile data, guarded call devirtualization would produce a class-check guard for `Object` instead of `MyClass`
- ◆ HW-created edge profile lacks path context



- Example:
 - Suppose hot paths are `a() → c() → d()` and `b() → c() → e()`
 - Inlining could produce `a() β c() β d() β e()` and `b() β c() β d() β e()` using edge profile, since all edges are hot
- Knowing paths would enable better inlining decisions

Conclusions

- ◆ IPF PMU provides rich, accurate execution feedback with minimal overhead
 - Inexpensive to use dynamically
 - Provides data that cannot be replicated by SW instrumentation
- ◆ A single profiling infrastructure enables multiple optimizations
 - Hotspot detection
 - Control-flow optimizations
 - Data optimizations
- ◆ Good initial results with small set of optimizations
- ◆ We are just getting started!