

A Quantitative Evaluation of the Contribution of Native Code to Java Workloads

Walter Binder

University of Lugano
Switzerland

walter.binder@unisi.ch

Jarle Hulaas, Philippe Moret

EPFL
Switzerland

{jarle.hulaas,philippe.moret}@epfl.ch

Overview

- **Motivation:**
Platform-independent profiling using dynamic bytecode metrics
- **Profiling tool to measure native code execution**
- **Evaluation**

Profiling Support in Java

- **JVMPI**
 - ✗ Experimental interface
 - ✗ Inflexible, limited set of events
- **JVMTI**
 - ✗ Standard interface since JDK 1.5
 - ✗ Improved flexibility
- **Limitations of both interfaces**
 - ✗ Native profiling agents, limited portability
 - ✗ Prevailing profilers are very slow
 - ✗ Measurement perturbation

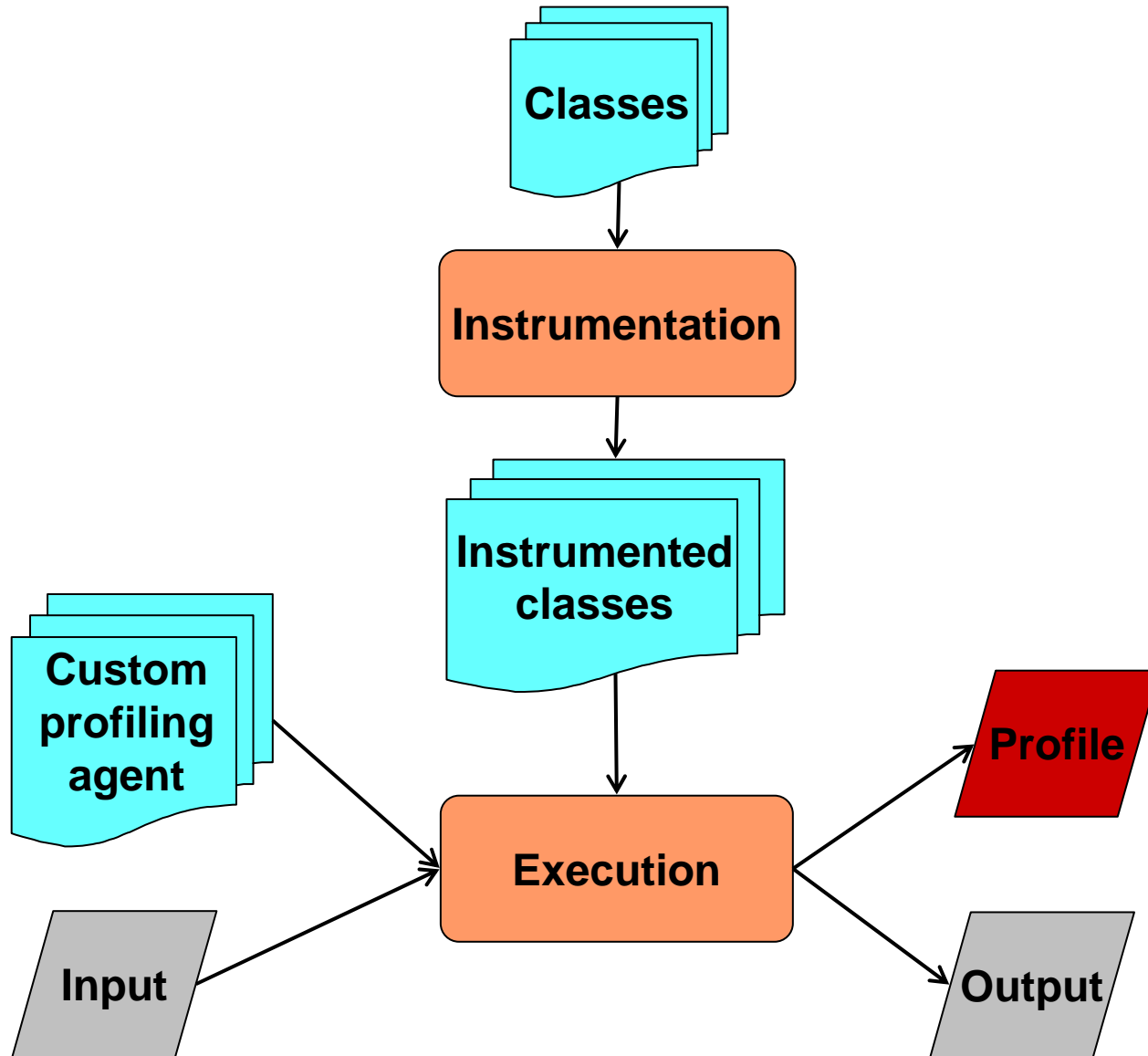
Motivating Scenario

- **Platform for service-oriented computing**
- **Many complex services (directories, composition engines, etc.)**
- **Heterogeneous environment**
- **Dynamic service deployment**
- **Requirements**
 - ✗ **Cross profiling**
 - ✗ **Platform independence**
 - ✗ **Efficient algorithms**

Our Approach

- **Portable profiling framework based on bytecode instrumentation**
- **Computation of (partial) Calling Context Tree**
- **Number of executed bytecode instructions as metric**
- **Periodic activation of pure Java profiling agent**

Profiling via Bytecode Instrumentation



Bytecode Metric

- **Platform-independent metric (in contrast to CPU second)**
- **Reproducible profiles**
- **Reduced measurement perturbation**
- **Directly computed from program, no particular OS functionality needed**
- **Profilers can be written in pure Java**
- **Fully compatible with standard JVMs**
- **Does not prevent JVM optimizations**

Per-Thread Profiling

- **Each thread has its own profiling structures**
 - ✗ **No synchronization needed**
- **Periodically, each thread invokes a user-defined profiling agent**
 - ✗ **Aggregation**
 - ✗ **Continuous metrics**
- **Periodic activation of profiling agent enforced by bytecode counting**
- **Profiling agent dynamically adapts frequency of its invocation**

Platform-Independent Profiling Details

- **Portable and Accurate Sampling Profiling for Java.**
Software: Practice & Experience 36(6).
- **A Portable and Customizable Profiling Framework for Java Based on Bytecode Instruction Counting.**
APLAS 2005.
- **Flexible and Efficient Measurement of Dynamic Bytecode Metrics.**
GPCE 2006.

Coverage of Bytecode Metrics

- **Native code = code that has no corresponding bytecode representation**
- **Execution of native code not represented in profiles**
- **How much does native code contribute to overall execution time?**
- **How to measure native code execution accurately, efficiently, and in a portable way?**

Measuring Native Code Execution

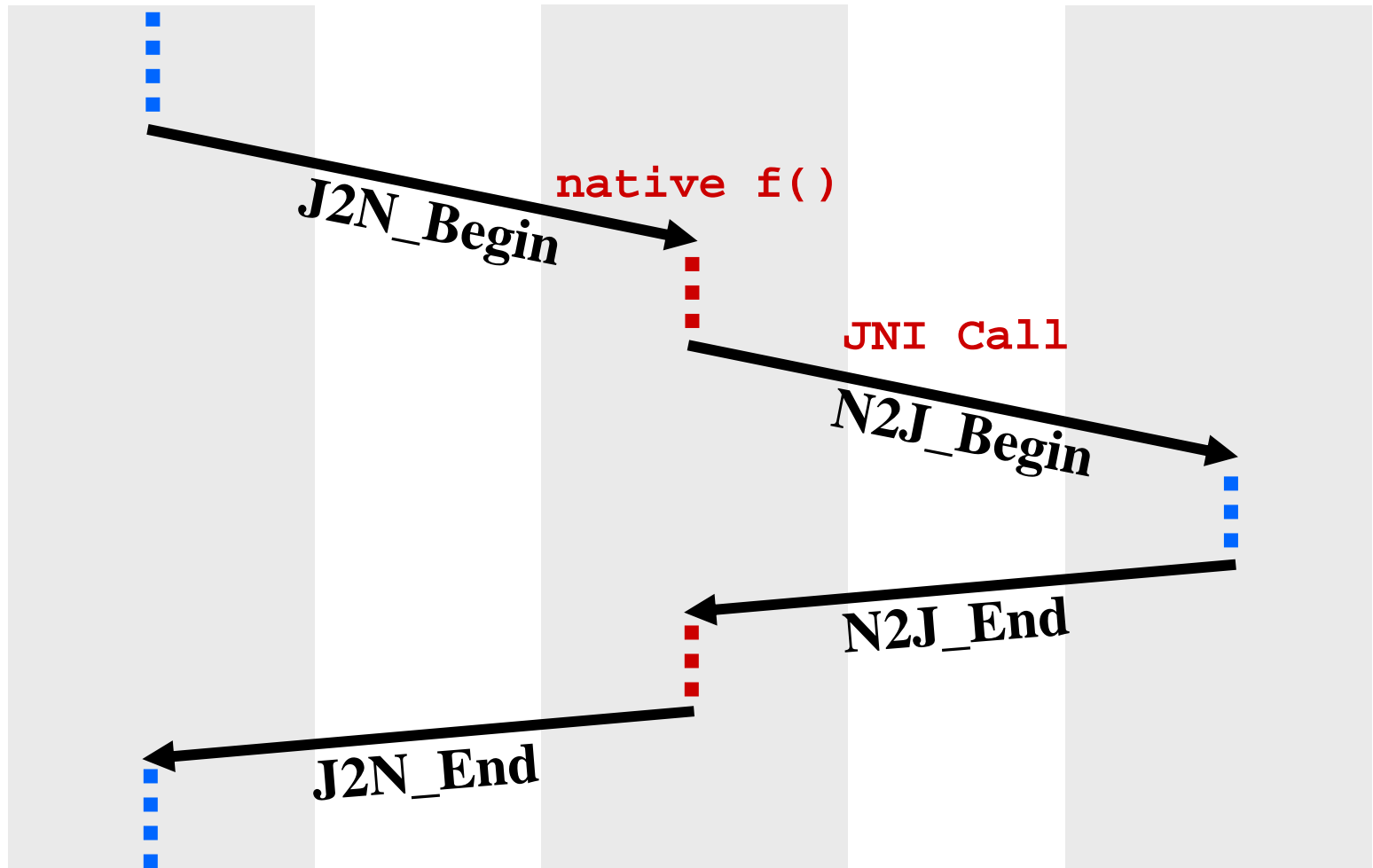
- **Intercept transitions between bytecode and native code**
- **Measurement upon transition**

Bytecode / Native Code Transitions

Bytecode

Native code

Bytecode



Implementation Techniques


- **JVMTI**
 - ✗ **Events: ThreadStart, ThreadEnd, VMDeath**
 - ✗ **JNI function interception**
 - ✗ **Native method prefixing**
- **Bytecode instrumentation**
 - ✗ **Wrappers for native methods**
- **Performance Counter Library (PCL)**
 - ✗ **Precise measurements**

Per-Thread Profiling

- **Minimizing synchronization**
- **Thread-local storage**
 - ✗ **Timestamp of last measurement**
 - ✗ **Bytecode execution time**
 - ✗ **Native code execution time**
- **ThreadStart**
 - ✗ **Initialize thread-local storage**
- **ThreadEnd**
 - ✗ **Update global state (synchronized)**

J2N Transitions

- Wrapper for each native method
- Static bytecode instrumentation
- JVMTI native method prefixing

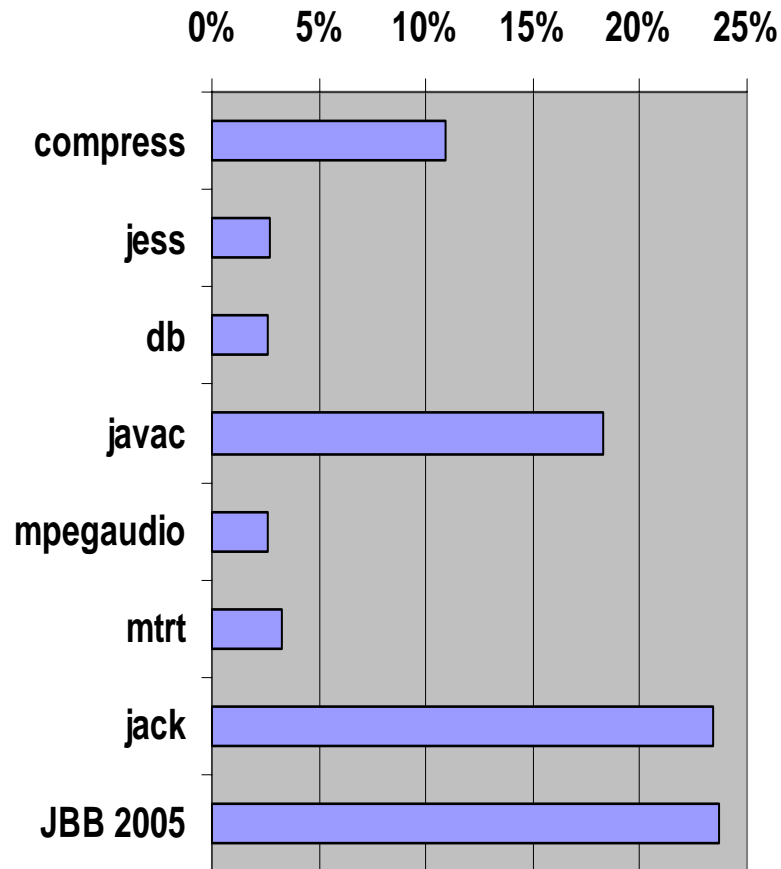
```
native int f();  int f() {  
    Profiler.J2N_Begin();  
    try {  
        return _prefixed_f();  
    }  
    finally {  
        Profiler.J2N_End();  
    }  
}  
  
native int _prefixed_f();
```

N2J Transitions

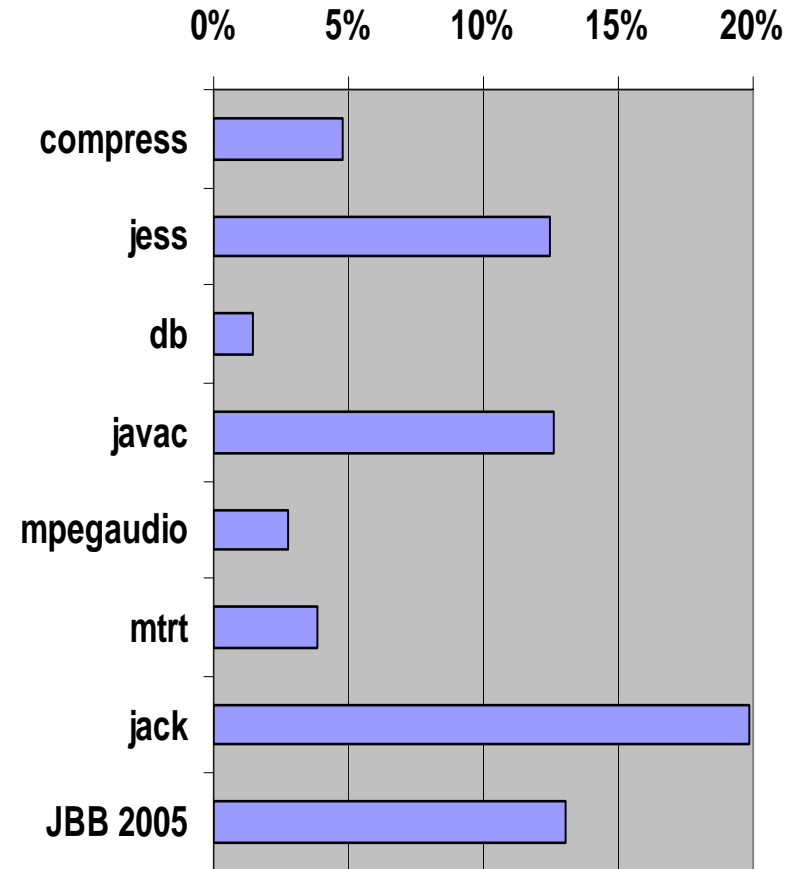
- **90 JNI functions for method invocation**
- **JVMTI enables JNI function interception**
- **90 hard-coded proxies**

Preliminary Results

Overhead Percentage



Native Code Percentage



JDK 1.6.0, Beta 2, Build 86, Hotspot Server VM

Limitations

- **Static instrumentation not applicable to dynamically generated/loaded code**
- **Normally, such code does not define native methods**

Future Work

- **Combination of instrumentation-based bytecode profiling with JVMTI-based native code profiling**
- **Explore detailed use of native code**
 - ✘ **Call stack upon J2N transitions**
 - ✘ **Execution time for individual native methods**

Conclusions

- **Platform-independent profiling**
 - ✗ **Dynamic bytecode metrics**
 - ✗ **Bytecode instrumentation**
 - ✗ **Portable**
 - ✗ **Efficient**
- **Contribution of native code to overall program execution time?**
 - ✗ **New profiling agent using JVMTI**
 - ✗ **Interception of bytecode / native code transitions**
 - ✗ **Overhead: < 25%**
 - ✗ **Native code: < 20%**