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

Walter Binder
Faculty of Informatics
University of Lugano
Lugano, Switzerland
Email: walter.binder@unisi.ch

Jarle Hulaas and Philippe Moret
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL)
CH–1015 Lausanne, Switzerland
Email: {jarle.hulaas, philippe.moret}@epfl.ch

***Abstract*— Many performance analysis tools for Java focus on tracking executed bytecodes, but provide little support in determining the specific contribution of native code libraries. This paper introduces and assesses a portable approach for characterizing the amount of native code executed by Java applications. A profiling agent based on the JVM Tool Interface (JVMTI) accurately keeps track of all runtime transitions between bytecode and native code. It relies on a combination of JVMTI events, Java Native Interface (JNI) function interception, bytecode instrumentation, and hardware performance counters.**[1]

I. Introduction

The Java Virtual Machine (JVM) promotes the development of portable software, since applications are represented as platform-independent bytecode. However, the JVM also supports the integration of platform-specific, native code, which does not have a corresponding bytecode representation. E.g., many functions of the Java Development Kit (JDK) are implemented in native code, sometimes in order to increase performance, but more often in order to get access to otherwise unavailable lower-level functionality.

Profiling and resource monitoring tools that are fundamentally based on analyzing and/or instrumenting Java bytecode offer many benefits, such as platform-independence and low measurement overhead [1]–[5]. However, because such tools do not track the execution of native code, the measurement results are only meaningful insofar as the measured application does not spend significant time in native code. Moreover, in prevailing profilers, such as the 'hprof' profiling agent that is included in standard JDKs, there is no support for segregating execution time spent in native code from time spent in interpreted or dynamically compiled bytecode, although this information is important in order to understand the program under development and to determine which parts of it can be further improved. When speaking about native code, we refer exclusively to code which has no corresponding bytecode representation. Note that this definition of native code does not cover code that is dynamically generated by a just-in-time (JIT) compiler, since then there is a corresponding bytecode representation (the JIT compiler input). This distinction provides a very valuable measure of the effective potential of bytecode instrumentation tools at transforming a sufficiently representative amount of the overall program code.

The general intuition that execution of native code contributes only a minor fraction of the total execution time is not easy to confirm in practice, because it is difficult to perform the required fine-grained measurements without introducing significant perturbations. Also, until recently, JVMs did not offer the APIs needed to fulfil this task, forcing researchers to modify virtual machines at the source code level [6]–[8]. Therefore, a way had to be found to measure native code execution that would by design be both *portable* and with *moderate overhead*, in order to minimize measurement perturbations.

In this paper we introduce a new methodology to characterize the execution of native code in Java applications. We present a portable and non-intrusive profiling agent, based on the JVM Tool Interface (JVMTI), in order to measure native code execution. Our profiling agent relies on a combination of JVMTI functions and bytecode instrumentation in order to keep track of transitions between bytecode and native code execution. The exploitation of hardware performance counters ensures high measurement accuracy. A performance evaluation shows that our profiling agent causes only moderate overhead.

Our tool presently provides a summary of the number of invocations to Java, respectively native methods, and of total CPU time spent in those two kinds of methods; it thus demonstrates the feasibility of a portable technique for gathering such low-level information, a feature until now restricted to system-specific profilers only. Moreover, a possible extension of our technique is to make it track complete call chains including a mix of Java and native methods; this is not possible with current profilers, since they are either Java-only or system-specific, and are therefore not aware of the frames of both Java and native C-language execution stacks.

This paper is structured as follows. Section II presents the basic APIs and libraries our approach depends on. Section III presents a first, unoptimized profiling agent, and Section IV shows a second, greatly improved agent. Section V evaluates our approach. Section VI presents related work, before concluding.

[1]This work was conducted while the first author was affiliated with the Artificial Intelligence Laboratory of the Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland.

II. JNI, JVMTI, AND PCL

The methodology presented here relies on the Java Native Interface (JNI), the JVM Tool Interface (JVMTI), and the Performance Counter Library (PCL), as explained in the following.

A. JNI

The Java Native Interface (JNI) [9] is a native programming interface. It allows bytecode executing inside the JVM to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly.

Bytecode may invoke native code through methods that are declared as `native`. I.e., native methods mark transitions from bytecode into native code. Native code libraries are loaded with the `System.loadLibrary(String)` method, which is typically called within the class initialization method. The JNI defines a resolution strategy to map native methods to the corresponding native code library functions.

The JNI also allows native code to access Java classes and objects. E.g., there are JNI functions to invoke Java instance methods or static methods; these JNI functions mark transitions from native code into bytecode.

B. JVMTI

The JVM Tool Interface (JVMTI) [10], introduced in JDK 1.5, is a native programming interface for use by tools. It provides both a way to inspect the state and to control the execution of applications running in the JVM. As a standard interface, the JVMTI enables the development of portable profiling and debugging tools.

In the following we briefly summarize the JVMTI features our profiling agents rely on. For the sake of easy readability, in this paper we adopt an abstract, Java-like pseudo-code notation and omit low-level implementation details.

a) Events: The JVMTI can signal a variety of profiling events. In this paper we make use of the following events:

- `ThreadStart(Thread t)`: Generated by a new thread `t` before its initial method executes.
- `ThreadEnd(Thread t)`: Generated by a terminating thread `t` after its initial method has finished execution.
- `VMDeath()`: Notifies the agent of the termination of the JVM. No JVMTI events will occur afterwards.
- `MethodEntry(Thread t, Method m)`: Generated upon entry of Java programming language methods, including native methods; thread `t` invokes method `m`. We use `m.isNative()` to check whether `m` is a native method.
- `MethodExit(Thread t, Method m)`: Generated upon exit from Java programming language methods, including native methods; thread `t` exits method `m`, either by executing a `return` instruction or by throwing an exception.

b) Thread-local Storage: Thread-local storage allows to associate a datastructure with each thread. Our profiling agents keep the profiling statistics for each thread in thread-local storage, which enables efficient update without synchronization needs. Upon thread start, we initialize the thread-local storage, and upon thread termination, we extract the profiling statistics from the thread-local storage.

In this paper, we represent thread-local storage as a map, which is accessed via the static methods `ThreadLocalStorage.put(Thread, Object)` and `ThreadLocalStorage.get(Thread)`.

c) Raw Monitor: A raw monitor is a synchronization aid. We use a raw monitor to synchronize access to global data, i.e., the overall profiling statistics, which are updated upon thread termination.

d) JNI Function Interception: The JNI maintains a table of functions that can be invoked through the JNI. The JVMTI allows us to change that table in order to intercept calls to certain functions.

e) Native Method Prefixing: Native method prefixing was introduced in the JVMTI, version 1.1 [11], which is part of JDK 1.6.[2] It modifies the failure handling of native method resolution by retrying with a prefix prepended to the method name.

C. PCL

Standard Java APIs do not provide the level of resolution required for measuring the time spent by a given thread inside native code methods.[3] Therefore, we decided to exploit the added precision provided by processor cycle counters, as found in *hardware performance counter* (HPC) enabled processors (such as the Intel Pentium 4).

We use the Performance Counter Library (PCL) [12], which is a lightweight HPC library with C, C++, Fortran, and Java APIs. PCL supports per-thread cycle counters (requiring an OS kernel patch on some operating systems) and enables our time measurement scheme, and hence the profiling agent, to become portable across a wide variety of operating systems and architectures. Note, however, that we only require the cycle counter functionality of PCL, something which can often directly be obtained from the underlying system, although not in a standardized way.

In the pseudo-code notation of this paper, we introduce a fictive static method `PCL.getTimestamp(Thread)` in order to read the cycle counter for a given thread.

III. SIMPLE PROFILING AGENT (SPA)

In this Section we describe our initial, simple profiling agent (SPA) to compute the percentage of native code execution in Java applications. SPA is based on the JVMTI, version 1.0 [10], which was introduced in JDK 1.5. As the JVMTI features used by SPA also exist in the now outdated

[2] At the time of writing this paper, JDK 1.6 is still in a pre-release stage.

[3] On our test machines, we experienced resolutions severely out of scale with the speed at which GHz-class CPUs execute native code.

```
class TC_SPA { // Thread Context
   long timestamp, timeBytecode = 0, timeNative = 0;
   boolean[] stack = new boolean[MAX_STACK_SIZE]; int sp = 0;

   TC_SPA(Thread t) { timestamp = PCL.getTimestamp(t); }
}


class SPA {
   long totalTimeBytecode = 0, totalTimeNative = 0;

   // Initialize SPA;
   // Enable the events ThreadStart, ThreadEnd, MethodEntry, MethodExit, and VMDeath;
   SPA() { ... }


// JVMTI events:

   void ThreadStart(Thread t) { ThreadLocalStorage.put(t, new TC_SPA(t)); }

   void ThreadEnd(Thread t) {
      TC_SPA tc = GetThreadLocalStorage(t);
      boolean inNative = true; if (tc.sp > 0) inNative = tc.stack[tc.sp - 1];
      long delta = PCL.getTimestamp(t) - tc.timestamp;
      if (inNative) tc.timeNative += delta; else tc.timeBytecode += delta;
      synchronized (this) {
        totalTimeBytecode += tc.timeBytecode; totalTimeNative += tc.timeNative;
      }
   }

   void MethodEntry(Thread t, Method m) {
      TC_SPA tc = GetThreadLocalStorage(t);
      boolean isNativeM = m.isNative();
      boolean isNativeCaller = true; if (tc.sp > 0) isNativeCaller = tc.stack[tc.sp - 1];
      if (isNativeM != isNativeCaller) {
        long currentTime = PCL.getTimestamp(t); long delta = currentTime - tc.timestamp;
        if (isNativeCaller) tc.timeNative += delta; else tc.timeBytecode += delta;
        tc.timestamp = currentTime;
      }
      tc.stack[tc.sp++] = isNativeM;
   }

   void MethodExit(Thread t, Method m) {
      TC_SPA tc = GetThreadLocalStorage(t);
      tc.sp--;
      boolean isNativeM = tc.stack[tc.sp]; // method being left (== m.isNative())
      boolean isNativeCaller = true; if (tc.sp > 0) isNativeCaller = tc.stack[tc.sp - 1];
      if (isNativeM != isNativeCaller) {
         long currentTime = PCL.getTimestamp(t); long delta = currentTime - tc.timestamp;
         if (isNativeM) tc.timeNative += delta; else tc.timeBytecode += delta;
         tc.timestamp = currentTime;
      }
   }

   void VMDeath() { ... } // Print statistics (totalTimeBytecode, totalTimeNative);


// Helper routines:

   static TC_SPA GetThreadLocalStorage(Thread t) {
      TC_SPA tc = (TC_SPA)ThreadLocalStorage.get(t);
      if (t == null) { tc = new TC_SPA(t); ThreadLocalStorage.put(t, tc); }
      return tc;
   }
}
```

Fig. 1. SPA pseudo-code.

```
native int foo(int a);            -->        int foo(int a) {
                                                IPA.J2N_Begin();
                                                try {
                                                   return _prefixed_foo(a);
                                                }
                                                finally {
                                                   IPA.J2N_End();
                                                }
                                             }

                                             native int _prefixed_foo(int a);
```

Fig. 2. Example wrapper for a native method, created by static bytecode instrumentation.

JVMPI [13], [14], SPA could be easily ported to run in older Java environments that only support the JVMPI.

Figure 1 presents SPA as Java-based pseudo-code. Our actual implementation is coded in C. SPA maintains a thread-local data structure, the thread context `TC_SPA`, for each thread in the system. Whenever a thread is started (the `ThreadStart` event), a `TC_SPA` instance is allocated and associated with the new thread. The thread context includes counters for the cycles spent by the thread executing (possibly compiled) bytecode (`timeBytecode`) respectively native code (`timeNative`). It also stores the thread's most recent timestamp obtained via PCL. Moreover, we reify the execution stack of each thread (`stack` and `sp`) in order to keep track whether stack frames correspond to Java methods or to native methods.

Upon method entry (the `MethodEntry` event), the implementation-type (native or not) of the callee is pushed onto the reified stack. If the implementation-type of caller and callee differ, a transition between bytecode and native code has been detected and the thread-local execution statistics are updated accordingly. We assume that each thread initially executes native code when it is started.

Upon method exit (the `MethodExit` event), the implementation-type of the callee (the method being exited) is popped off the reified stack. In a similar way as for method entry, the thread-local execution statistics are updated, if the implementation-type of caller and callee differ. Note that without the reified stack, we would not be able to determine the implementation-type of the caller upon method exit.

Upon thread termination (the `ThreadEnd` event), the overall execution statistics (the counters `totalTimeBytecode` and `totalTimeNative`) are updated in a synchronized way, using a raw monitor of the JVMTI. Finally, upon termination of the JVM (the `VMDeath` event), the overall statistics are printed out.

Note that in the code of the `ThreadEnd`, `MethodEntry`, and `MethodExit` events, we do not assume that the thread context has been created before. The helper method `GetThreadLocalStorage(Thread)` allocates the thread context on demand. This is necessary because the JVMTI does not signal the `ThreadStart` event for the bootstrapping thread. Hence, the initial execution of the bootstrapping thread (e.g., initialization of SPA) cannot be tracked by a JVMTI-based profiling agent.

We designed SPA in such a way that the execution of measurement code (access to PCL) is minimized. SPA does not require a timestamp upon each method entry/exit event, but only upon transitions between bytecode and native code. Moreover, using a thread-local datastructure, we can avoid possibly costly synchronization. Synchronization is only needed upon thread termination in order to update the overall execution statistics.

SPA meets our first design goal of portability, because it is programmed against a standard interface and does not require any modifications to the JVM. However, SPA completely fails to meet our second design goal of low overhead and measurement perturbation. Enabling the `MethodEntry` and `MethodExit` events prevents JIT compilation. Hence, SPA causes excessive overhead (see Section V) and consequently also significant measurement perturbation.

IV. Improved Profiling Agent (IPA)

In this Section we present our improved profiling agent (IPA) to compute the percentage of native code execution in Java applications. IPA is based on the JVMTI, version 1.1 [11]. In contrast to SPA, IPA does not rely on the `MethodEntry` and `MethodExit` events that prevent JIT compilation. IPA executes measurement code only upon transitions between bytecode and native code. In the following, N2J refers to a native code to bytecode transition (native code calling a JNI method invocation function), whereas J2N relates to a bytecode to native code transition (invocation of a native method).

In order to intercept N2J transitions, IPA exploits the JVMTI feature called *JNI function interception*. IPA registers wrappers for all JNI functions that are used to invoke methods: `Call`⟨*type*⟩`Method`⟨*style*⟩`()`, `CallStatic`⟨*type*⟩`Method`⟨*style*⟩`()`, as well as `CallNonvirtual`⟨*type*⟩`Method`⟨*style*⟩`()`. The ⟨*type*⟩ notation specifies the return type and may be `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, `Double`, or `Void`. ⟨*style*⟩ selects one of 3 possible ways of parameter passing. I.e., in total $3 * 10 * 3 = 90$ wrappers have to be registered. Each wrapper first signals a `N2J_Begin()` transition, then invokes the original JNI function (resulting in

```
class TC_IPA { // Thread Context
   long timestamp, timeBytecode = 0, timeNative = 0;
   boolean inNative = true;

   TC_IPA(Thread t) { timestamp = PCL.getTimestamp(t); }
}


class IPA {
   long totalTimeBytecode = 0, totalTimeNative = 0;

   // Initialize IPA;
   // Enable the events ThreadStart, ThreadEnd, and VMDeath;
   // Install wrappers for JNI method invocation functions (JNI function interception);
   // Enable native method prefixing;
   IPA() { ... }


// JVMTI events:

   void ThreadStart(Thread t) { TC_IPA tc = new TC_IPA(t); ThreadLocalStorage.put(t, tc); }

   void ThreadEnd(Thread t) {
      TC_IPA tc = GetThreadLocalStorage(t);
      long delta = PCL.getTimestamp(t) - tc.timestamp;
      if (tc.inNative) tc.timeNative += delta; else tc.timeBytecode += delta;
      synchronized (this) {
         totalTimeBytecode += tc.timeBytecode; totalTimeNative += tc.timeNative;
      }
   }

   void VMDeath() { ... } // Print statistics (totalTimeBytecode, totalTimeNative);


// IPA transitions:

   static void N2J_Begin() {
      Thread t = Thread.currentThread(); TC_IPA tc = GetThreadLocalStorage(t);
      long currentTime = PCL.getTimestamp(t);
      tc.timeNative += (currentTime - tc.timestamp); tc.timestamp = currentTime;
      tc.inNative = false;
   }

   static void N2J_End() { J2N_Begin(); }

   static void J2N_Begin() {
      Thread t = Thread.currentThread(); TC_IPA tc = GetThreadLocalStorage(t);
      long currentTime = PCL.getTimestamp(t);
      tc.timeBytecode += (currentTime - tc.timestamp); tc.timestamp = currentTime;
      tc.inNative = true;
   }

   static void J2N_End() { N2J_Begin(); }


// Helper routines:

   static TC_IPA GetThreadLocalStorage(Thread t) {
      TC_IPA tc = (TC_IPA)ThreadLocalStorage.get(t);
      if (t == null) { tc = new TC_IPA(t); ThreadLocalStorage.put(t, tc); }
      return tc;
   }
}
```

Fig. 3. IPA pseudo-code.

a Java method invocation), and finally signals a `N2J_End()` transition.

For J2N transitions, IPA relies on a new feature of JVMTI, version 1.1, called *native method prefixing*. It allows to introduce wrappers for methods declared as `native`. Using bytecode instrumentation, for each native method a corresponding Java method with the same name and signature (but without the `native` declaration) is added. The original native method is renamed, as we will explain later.

The added Java method acts as a wrapper for the native method (see Figure 2): First it calls IPA in order to signal the `J2N_Begin()` transition. Afterwards it invokes the renamed native method with the received arguments and returns the result. A `finally` clause ensures that before termination of the wrapper method, IPA is called again to signal the `J2N_End()` transition. The `finally` clause ensures that IPA is called also in case of an exception.

Native methods are renamed by prepending a well-chosen prefix that is announced to the JVM (the prefix should not occur in any method name). When linking native code libraries, the JVM is able to match method names declared with a prefix with unchanged method names in native code libraries.

We have considered two different approaches to instrument the bytecode:

1) Static instrumentation of all used classes (including the classes of the JDK) before the application to be profiled is executed.
2) Dynamic instrumentation whenever a class is loaded.

Static instrumentation has the advantage of causing less overhead and measurement perturbation at runtime, because the instrumentation happens before the profiling. Moreover, we can use one of many available Java-based bytecode instrumentation libraries, such as e.g. ASM [15], BCEL [16], Javassist [17], JOIE [18], BIT [19], JikesBT [20], or SERP [21]. The drawback of static instrumentation is that dynamically generated or loaded code is not instrumented. However, usually such code does not declare any native methods so that an instrumentation is not necessary.

Dynamic instrumentation delegates the instrumentation task to the profiling agent. Whenever a class is loaded[4], the profiling agent checks if the class contains a native method. If this is the case, the profiling agents performs the bytecode instrumentation. As bytecode instrumentation happens at runtime of the application being profiled, overhead and measurement perturbation increase. Furthermore, a Java-based bytecode instrumentation library cannot be used, because the instrumentation is already needed during the bootstrapping of the JVM. Trying to use a bytecode instrumentation library at such an early stage would disrupt the JVM's class-loading sequence, resulting in a crash. Therefore, the instrumentation has to be done either in a separate JVM process which is contacted through the Inter-Process-Communication facilities of the underlying operating system, or in native code. The first approach increases overhead and measurement perturbation, while the second one results in increased development effort, because to the best of our knowledge, there is no complete bytecode instrumentation library implemented in native code publicly available.[5]

For these reasons, we resort to static instrumentation. Our bytecode instrumentation tool is based on ASM [15]. It processes individual class files or archives of class files. We also applied our instrumentation tool to the classes of the JDK, including the core classes within the library '`rt.jar`'. We use the JVM's '`-Xbootclasspath/p:`' option in order to load the instrumented classes at runtime.

Figure 3 illustrates the pseudo-code of our profiling agent. The code executed upon `ThreadStart` and `ThreadEnd` events is similar to the SPA presented in the previous Section. The `MethodEntry` and `MethodExit` events are disabled. Instead of handling these events, the JNI method invocation wrappers (implemented in C) and the native method wrappers (provided as bytecode by the static instrumentation) invoke IPA's transition routines only upon transitions between native code and bytecode. In order to enable native method wrappers to call these transition routines from bytecode, we created a Java class corresponding to IPA which declares the four corresponding static methods as `native` (this special class is excluded from instrumentation).

Note that invocations of IPA's transition routines are not JVMTI events; these routines are called during the normal control flow. While in the pseudo-code we use the static method `Thread.currentThread()` to obtain the current thread, the implementation avoids to explicitly obtain a reference to the current thread, because the JVMTI functions to access the thread-local storage accept a `null` value to represent the current thread.

In order to simplify the presentation, we assumed `N2J_End()` and `J2N_Begin()` (resp. `J2N_End()` and `N2J_Begin()`) to execute the same profiling code. However, in our implementation we adjust the timestamp obtained from PCL in order to compensate for the average execution time of the corresponding wrapper. I.e., we aim at excluding the wrappers' execution time from the profiling statistics in order to reduce measurement perturbations.

V. Evaluation

In this Section we first evaluate the overhead caused by SPA and IPA, and second present some profiling statistics generated by IPA.

Our evaluation is based on the SPEC JVM98 benchmark suite [22] (problem size 100), which consists of 7 benchmarks ('compress', 'jess', 'db', 'javac', 'mpegaudio', 'mtrt', 'jack'), as well as the SPEC JBB2005 benchmark [23] (warehouse sequence 1, 2, 3, 4) on a Linux Fedora Core 2 computer (Intel Pentium 4, 2.66 GHz, 1024 MB RAM). The metric used by SPEC JVM98 is the execution time in seconds,

[4]The JVMTI supports a `ClassFileLoadHook` event allowing a profiling agent to change the bytecode of classes before they are linked into the JVM.

[5]The JVMTI demo distributed with Sun's JDK covers some bytecode instrumentation implemented in native code, but this cannot be considered a general-purpose bytecode instrumentation library.

TABLE I

EXECUTION TIME AND PROFILING OVERHEAD FOR SPA AND IPA USING THE JVM98 AND JBB2005 BENCHMARKS.

benchmark	time original [s]	time SPA [s]	time IPA [s]	overhead SPA	overhead IPA
compress	5.74	445.86	6.38	7 667.60%	11.15%
jess	1.49	237.20	1.53	15 819.46%	2.68%
db	14.25	231.88	14.35	1 527.23%	0.70%
javac	3.80	224.73	4.32	5 813.95%	13.68%
mpegaudio	2.54	251.50	2.65	9 801.57%	4.33%
mtrt	1.16	485.75	1.16	41 775.00%	0.00%
jack	3.47	123.12	4.17	3 448.13%	20.17%
geom. mean	3.35	261.17	3.59	7 696.25%	7.31%

benchmark	throughput	throughout SPA	throughput IPA	overhead SPA	overhead IPA
JBB2005	7 251	66.4	6 021	10 820.18%	20.43%

TABLE II

PROFILING STATISTICS FOR THE JVM98 (15 RUNS) AND JBB2005 (WAREHOUSE SEQUENCE 1, 2, 3, 4) BENCHMARKS.

benchmark	% native execution	JNI calls	native method calls
compress	4.54%	1 538	45 858
jess	5.38%	918	492 762
db	0.84%	512	595 849
javac	16.82%	25 633	3 701 694
mpegaudio	0.95%	571	106 117
mtrt	1.62%	513	73 357
jack	20.26%	1 308	4 991 615
JBB2005	12.19%	770	123 199 879

whereas SPEC JBB2005 measures the throughput in operations/second. All benchmarks were run in single-user mode (no networking) and we removed background processes as much as possible in order to obtain reproducible results. We present our measurements obtained with Sun JDK 1.6.0-rc-b97 in its 'server' mode, which enables the optimizing Hotspot Server JIT compiler. Our statistics are to be considered preliminary, because at the time of writing this paper JDK 1.6 still is in a pre-release stage.

A. Overhead

Table I shows the execution time and profiling overhead for both SPA and IPA. For each setting and each benchmark, we took the median of 15 runs. For the SPEC JVM98 suite, we also computed the geometric mean of the 7 benchmarks. For the SPEC JVM98 benchmarks, the overhead is computed as $\left(\frac{\text{execution time with profiling}}{\text{execution time without profiling}} - 1\right)$, while for the SPEC JBB2005 benchmark, the overhead is calculated as $\left(\frac{\text{operations/second without profiling}}{\text{operations/second with profiling}} - 1\right)$.

The overhead due to SPA is between 1 500% and 42 000%, whereas IPA causes an overhead of only 0–20%. The excessive overhead for SPA is caused by the `MethodEntry` and `MethodExit` events, which prevent JIT compilation. Therefore, SPA cannot be used to accurately measure the fraction of native code execution in a Java workload, as the enormous overhead results in serious measurement perturbation: The performance characteristics of the profiled application is significantly different from executing the application without profiling. Moreover, because of the high overhead, SPA is not suited for long-running applications. In contrast to SPA, IPA is much less intrusive. Except for transitions between bytecode and native code, there is no overhead for method invocation and termination. Thanks to its moderate overhead, IPA can be used to profile long-running applications.

B. Profiling Statistics

Table II shows the profiling statistics generated by IPA for the previously mentioned benchmarks. For each benchmark, we present the percentage of execution time spent in native code, as well as the number of intercepted JNI calls (native to bytecode transitions) and the number of native method invocations (from bytecode). These results correspond to 15 runs of the SPEC JVM98 benchmarks, respectively to a warehouse sequence 1, 2, 3, 4 in the case of SPEC JBB2005.

The most interesting result is that the execution time spent in native code is within 20% for all benchmarks. Several benchmarks, such as '`compress`', '`db`', '`mpegaudio`', and '`mtrt`' (which is the most object-oriented benchmark in the SPEC JVM98 suite according to [24]), spend less than 5% of their execution in native code. In all benchmarks we measured, the execution time spent in bytecode is significantly higher than the time spent in native code. Therefore, we conclude that techniques for the platform-independent performance analysis of Java applications [1]–[5], which focus on the bytecode execution and may neglect the execution of native code, can be appropriate for many Java workloads.

VI. Related Work

The present work concentrates on dynamic metrics, i.e., metrics that have to be gathered at runtime, as opposed to static metrics, which are based on off-line code analysis. In [25] the authors present a variety of dynamic metrics for Java programs. They introduce a tool called *J [26] for the metrics computation. *J relies on the JVMPI [13], [14], a former profiling interface for the JVM, which is known to cause very high measurement overhead. Hence, *J is only applicable to programs with short execution time. Dynamic metrics concerning native code execution are not addressed in [25], [26].

The number of actual invocations to native methods is a dynamic metric that can be obtained by incrementing a counter at runtime. The work presented in reference [6] uses an instrumented version of the Kaffe virtual machine [27] without JIT compilation (purely interpreted mode) in order to gather this metric. Thus, that approach is not portable and provides only a very coarse-grained view of where the CPU is actually spent.

Some researchers have managed to provide a detailed breakdown of where CPU time is spent in Java workloads [7], [8]. However, they also had to modify a JVM and thus did not provide a portable profiling tool. To increase measurement accuracy, some authors advocate the incorporation of per-thread hardware performance counters directly inside processors, e.g., processor cycle (also called timestamp) counters, in order to avoid the overhead of virtualizing such counters in software [7]. However, this software virtualization is well accepted, since integrated into widely used operating systems such as Microsoft Windows, Sun Solaris, and IBM AIX, whereas Linux is an exception which currently requires a kernel patch.[6]

Our goal is to realize accurate profiling, as opposed to sampling-based profilers (e.g., IBM `tprof`) that are able to calculate the time spent in native code very efficiently, but at the expense of a slight loss of accuracy. These profilers work by periodically sampling the PC, and comparing this value to a map of active code modules, such as the native code libraries loaded by a JVM, a technique which is inherently system-dependent. In contrast to our approach, such tools are not able to construct accurate counts of the number or frequency of JNI calls, nor do they have the potential of exposing the details of mixed Java/native call chains.

In contrast to related work that relies on an instrumented JVM, our Improved Profiling Agent (IPA) is portable to any JVM that supports the JVMTI (since version 1.1 [11]), and to any hardware platform that supports hardware performance counters compatible with PCL. To our best knowledge, the work presented here is the first portable methodology which allows to measure how much of their CPU time Java programs spend executing native code.

VII. Conclusion

This paper has presented a portable approach for characterizing the amount of native code executed by Java applications. It consists of a profiling agent based on the JVM Tool Interface (JVMTI), and accurately keeps track of all runtime transitions between bytecode and native code. Our methodology relies on a combination of JVMTI events, Java Native Interface (JNI) function interception, bytecode instrumentation, and hardware performance counters. Our performance evaluation shows that this approach causes moderate overhead of 0–20%. Using our profiling agent, we were able to report that for SPEC JVM98 and SPEC JBB2005 run on Sun JDK 1.6 (beta version), native code contributes to 1–20% of total execution time. Consequently, we may conclude that, on this first platform, bytecode instrumentation is an effective technique which provides a good coverage of actually executed code in Java programs.

Our tool presently provides a summary of the number of invocations to Java, respectively native methods, and of total CPU time spent in those two kinds of methods; it thus demonstrates the feasibility of a portable technique for gathering such low-level information, a feature until now restricted to system-specific profilers only. Moreover, we are currently working on an extension which consists in tracking complete call chains including a mix of Java and native methods, a capability which opens up new debugging and profiling perspectives; this would not be possible with current profilers, since they are either Java-only or system-specific, and are therefore not aware of the frames of both Java and native C-language execution stacks.

[6] See the web sites of the Performance Counter Library (PCL) (`http://www.fz-juelich.de/zam/PCL/`) and the Performance Application Programming Interface (PAPI) [28] (`http://icl.cs.utk.edu/papi/`).

References

[1] W. Binder, "A portable and customizable profiling framework for Java based on bytecode instruction counting," in *Third Asian Symposium on Programming Languages and Systems (APLAS 2005)*, ser. Lecture Notes in Computer Science, vol. 3780. Tsukuba, Japan: Springer Verlag, Nov. 2005, pp. 178–194.

[2] W. Binder, "Portable and accurate sampling profiling for Java," *Software: Practice and Experience*, vol. 36, no. 6, pp. 615–650, 2006.

[3] M. Dmitriev, "Profiling Java applications using code hotswapping and dynamic call graph revelation," in *WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance*. ACM Press, 2004, pp. 139–150.

[4] J. Hulaas and W. Binder, "Program transformations for portable CPU accounting and control in Java," in *Proceedings of PEPM'04 (2004 ACM SIGPLAN Symposium on Partial Evaluation & Program Manipulation)*, Verona, Italy, August 24–25 2004, pp. 169–177.

[5] J. D. Turner, "A dynamic prediction and monitoring framework for distributed applications," PhD Thesis, Department of Computer Science, University of Warwick, UK, May 2003.

[6] D. Gregg, J. F. Power, and J. Waldron, "A method-level comparison of the Java Grande and SPEC JVM98 benchmark suites," *Concurrency and Computation: Practice and Experience*, 2005.

[7] N. M. Hanish and W. E. Cohen, "Hardware support for profiling java programs," in *Workshop on Hardware Support for Objects And Microarchitectures for Java*, October 10 1999.

[8] G. Lashari and S. Srinivas, "Characterizing Java application performance," in *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 138.1.

[9] Sun Microsystems, Inc., "Java Native Interface (JNI)," Web pages at http://java.sun.com/javase/6/docs/technotes/guides/jni/index.html.

[10] Sun Microsystems, Inc., "JVM Tool Interface (JVMTI), Version 1.0," Web pages at http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/.

[11] Sun Microsystems, Inc., "JVM Tool Interface (JVMTI) version 1.1," Web pages at http://java.sun.com/javase/6/docs/technotes/guides/jvmti/index.html.

[12] R. Berrendorf and H. Ziegler, "PCL – The Performance Counter Library: A common interface to access hardware performance counters on microprocessors (version 2.2)," Central Institute for Applied Mathematics, Research Centre Juelich GmbH, Tech. Rep., 2003, http://www.fz-juelich.de/zam/PCL/.

[13] S. Liang and D. Viswanathan, "Comprehensive profiling support in the Java virtual machine," in *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS-99)*. Berkeley, CA: USENIX Association, May 3–7 1999, pp. 229–240.

[14] Sun Microsystems, Inc., "Java Virtual Machine Profiler Interface (JVMPI)," Web pages at http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/.

[15] ObjectWeb, "ASM," Web pages at http://asm.objectweb.org/.

[16] M. Dahm, "Byte code engineering," in *Java-Information-Tage 1999 (JIT'99)*, Sept. 1999, http://jakarta.apache.org/bcel/.

[17] S. Chiba, "Load-time structural reflection in Java," in *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, ser. Lecture Notes in Computer Science. Cannes, France: Springer Verlag, June 2000, vol. 1850, pp. 313–336.

[18] G. Cohen, J. Chase, and D. Kaminsky, "Automatic program transformation with JOIE," in *1998 USENIX Annual Technical Symposium*, 1998, pp. 167–178. [Online]. Available at http://www.cs.duke.edu/~gac/joie/joie.ps

[19] H. B. Lee and B. G. Zorn, "BIT: A tool for instrumenting Java bytecodes," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems (ITS-97)*. Berkeley: USENIX Association, Dec. 8–11 1997, pp. 73–82.

[20] IBM, "Jikes Bytecode Toolkit," Web pages at http://www.alphaworks.ibm.com/tech/jikesbt.

[21] BEA, "Serp," Web pages at http://serp.sourceforge.net/.

[22] The Standard Performance Evaluation Corporation, "SPEC JVM98 Benchmarks," Web pages at http://www.spec.org/osg/jvm98/.

[23] The Standard Performance Evaluation Corporation, "SPEC JBB2005 (Java Business Benchmark)," Web pages at http://www.spec.org/osg/jbb2005/.

[24] J. Dujmovic and C. Herder, "Visualization of Java workloads using ternary diagrams," *Software Engineering Notes*, vol. 29, no. 1, pp. 261–265, 2004.

[25] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge, "Dynamic metrics for Java," *ACM SIGPLAN Notices*, vol. 38, no. 11, pp. 149–168, Nov. 2003.

[26] B. Dufour, L. Hendren, and C. Verbrugge, "*J: A tool for dynamic analysis of Java programs," in *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM Press, 2003, pp. 306–307.

[27] T. Wilkinson, "Kaffe - a Java virtual machine," Web pages at http://www.kaffe.org/.

[28] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *The International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, Fall 2000. Available at http://citeseer.ist.psu.edu/browne00portable.html