SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2006; **36**:615–650
Published online 14 February 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.712

Portable and accurate sampling profiling for Java

Walter Binder*,†

Ecole Polytechnique Fédérale de Lausanne (EPFL), Artificial Intelligence Laboratory, CH-1015 Lausanne, Switzerland

SUMMARY

This article presents a novel framework for the sampling-based profiling of Java programs, which relies on program transformation techniques. We exploit bytecode instruction counting to regularly activate a user-defined profiling agent, which processes the current call stack. This approach has several advantages, such as making the instrumentation entirely portable, generating reproducible profiles, and enabling a fine-grained adjustment of the sampling rate. Our framework offers a flexible API to write portable profiling agents in pure Java. While the overhead due to our profiling scheme is comparable to the overhead caused by prevailing, timing-based profilers, the resulting profiles are much more accurate. Copyright © 2006 John Wiley & Sons, Ltd.

KEY WORDS: Java; JVM; sampling profiling; program transformations; bytecode instrumentation

1. INTRODUCTION

Profiling enables a detailed performance analysis of programs. It helps to detect hot spots and performance bottlenecks, guiding the developer in which parts of a program optimizations may pay off. As Java [1] and the Java Virtual Machine (JVM) [2] are a preferred programming language and deployment platform for many application and middleware developers, there is a need for accurate and efficient Java profiling tools. The JVM Profiling Interface (JVMPI) [3,4] provides a set of hooks to the JVM to signal interesting events, such as thread start and object allocations. Its successor, the JVM Tool Interface (JVMTI) [5], provides additional facilities for bytecode instrumentation. Many profilers based on the JVMPI or JVMTI can operate in two modes: in the *exact profiling mode*, they track each method invocation, whereas in the *sampling mode*, the profiler spends most of the time sleeping and periodically (e.g. every few milliseconds) wakes up to register the stack traces of running threads.

Profilers based on the JVMPI or JVMTI interfaces implement profiling agents to intercept various events, such as method invocations. Unfortunately, these profiling agents have to be written in

*Correspondence to: Walter Binder, EPFL IC IIF LIA, INR 241 (Bâtiment IN), Station 14, CH-1015 Lausanne, Switzerland.
†E-mail: walter.binder@epfl.ch

Received 12 May 2005
Revised 11 July 2005
Accepted 11 July 2005

platform-dependent native code, contradicting the Java motto 'write once and run anywhere'. As exact profiling based on these APIs may cause an enormous overhead (in some cases we experienced a slowdown of factor 4000 and more), developers frequently resort to more efficient sampling-based profilers to analyze the performance of complex systems, such as application servers. Many prevailing sampling profilers for Java use an external timer to trigger the sampling, resulting in a non-deterministic behaviour: for the same program and input, the generated profiles may vary very much depending on the processor speed and the system load. In many cases the accuracy of the resulting profiles is so low that a reasonable performance analysis based on these profiles is not possible.

To solve these problems, we developed Komorium, a novel sampling profiler for Java that does not rely on either the JVMPI or on the JVMTI, but directly instruments the bytecode of Java programs in order to regularly generate samples of the call stack. Komorium transforms programs so that during execution they keep track of the number of executed bytecodes[‡]. If the number of executed bytecodes exceeds a certain adjustable threshold, a user-defined profiling agent is triggered to process a sample of the current call stack. Komorium reifies the call stack in order to make it accessible to the profiling agent, i.e. Komorium transforms programs to expose the call stack, which is normally 'hidden' in the implementation of the runtime system and not accessible to the programmer. To summarize, our approach offers the following advantages.

1. Custom profiling agents can be written in pure Java. Hence, profiling agents are portable and can be used in all Java environments.
2. Our approach does not rely on either a customized JVM or on native code. Based on bytecode instrumentation, our program transformation scheme is compatible with any standard Java runtime system, even if it does not offer any dedicated profiling support.
3. We use a deterministic sampling mechanism based on bytecode instruction counting. The number of issued bytecodes does not depend on machine characteristics, such as the processor speed, or on the JVM implementation, which may interpret the bytecode or run native code produced, e.g., by a just-in-time compiler. Hence, the number of executed bytecodes is a platform-independent dynamic metric [6]. Consequently, for deterministic programs, profiles are reproducible and directly comparable across different platforms (assuming the same Java class library is used).
4. While the overhead is comparable to prevailing, timing-based sampling profilers for Java, the profiles generated by our deterministic sampling mechanism are much more accurate.
5. The sampling rate can be dynamically tuned by the user-defined profiling agent. This allows the tradeoff between high accuracy and low overhead to be adjusted.
6. Stack reification ensures that the profiling agent receives the necessary information concerning the current call stack. For each invoked method, the class and method name, as well as the method signature are preserved. As the latter is currently not provided by standard Java APIs, many prevailing Java profilers cannot distinguish between overloaded methods (i.e. methods of the same class with the same name but different signatures). Stack reification solves this problem.

The main contribution of this article is an innovative program transformation scheme for platform-independent, deterministic sampling profiling. In contrast to prevailing sampling profilers, our profiling

[‡]In this article the term 'bytecode' stands for 'JVM bytecode instruction'.

framework is fully portable, it is compatible with any standard JVM and supports custom profiling agents written in pure Java. This is possible because we exploit bytecode instruction counting as platform-independent profiling metric. Another contribution of this article is a thorough evaluation, which shows that our approach yields highly accurate profiles and causes only moderate overhead, comparable with prevailing profilers.

The remainder of this article is structured as follows. Section 2 explains the rationale behind our deterministic sampling profiler. Section 3 presents the API to write custom profiling agents and the runtime classes needed for the profiling. Section 4 explicates our transformation scheme, the reification of the call stack and the periodic activation of a profiling agent based on bytecode instruction counting. We also discuss native code issues and illustrate the transformations with an example. In Section 5 we show two optimizations to reduce the profiling overhead. Section 6 explains how to implement efficient, custom profiling agents. Section 7 provides a detailed evaluation of the present profiling scheme. We analyze the accuracy and size of the generated profiles, as well as the profiling overhead. Moreover, we compare our approach with the 'hprof' profiling agent in its sampling mode, which is part of many standard Java Development Kits (JDKs). Section 8 discusses the benefits and limitations of our approach, whereas Section 9 compares our research with related work. Finally, Section 10 outlines our plans for future work and Section 11 provides the conclusions of this article.

2. DETERMINISTIC SAMPLING USING BYTECODE INSTRUCTION COUNTING

Komorium provides a simple but flexible API to write *custom profiling agents in pure Java*. It transforms the JVM bytecode so that each thread in the system periodically activates the user-defined profiling agent, providing a snapshot of the current call stack. The custom profiling agent maintains statistics of the stack snapshots. In general, there is a correlation between the number of times a certain call stack is reported to the profiling agent and the amount of processing spent in the corresponding calling context. Typically, the profiling agent counts the number of occurrences of the different samples (call stacks). The relative frequency of a certain call stack S (i.e. number of occurrences of S/total number of samples) approximates the proportion of processing spent in the corresponding calling context.

Many sampling profilers for Java are timing-based, i.e. the activation of the profiling agent is triggered by a timer. Timing-based sampling profiles can be used to estimate the relative CPU time spent in the various methods. However, this approach has several shortcomings: achieving a high sampling rate may require platform-specific features (such as access to special timers) and hence hamper portability, results may not be reproducible (the CPU time may depend on factors such as system load), and the sampling profiling itself may change the execution behaviour of the program (measurement perturbation). The last point is particularly true on JVMs where the use of the JVMPI disables just-in-time compilation. Another disadvantage of timing-based sampling is that the profiling agent, which is triggered by an external event (timer) and thus executes asynchronously, may need to interrupt the running threads in order to take snapshots of their call stacks. In Section 7.5 we show that the standard 'hprof' profiling agent in its sampling mode (timing-based) does not produce accurate execution profiles.

For these reasons, we follow a different approach, using *bytecode instruction counting* to trigger the sampling. Each thread keeps track of the number of bytecodes it has executed. If this number

exceeds a threshold defined by the custom profiling agent which we call *sampling granularity*, the thread *synchronously* invokes the profiling agent to take a sample of the call stack, i.e. each thread in the system periodically invokes the profiling agent, which captures only the stack of the calling thread. There is no need to interrupt other threads in the system. For deterministic programs, this approach enables reproducible results: for a given sampling granularity, the profiling agent is always triggered exactly at the same program location, independent of the hardware characteristics of the execution machine and independent of the system load.

A sampling profile based on bytecode instruction counting does not directly relate to the relative CPU time spent in the different methods, but rather approximates the relative number of bytecodes executed by the various methods (which in turn may be seen as a coarse approximation of the relative CPU time). In previous work we established bytecode instruction counting as platform-independent metric for resource accounting (e.g. production-time monitoring of server environments) and resource control (e.g. prevention of denial-of-service attacks in mobile code systems) [7–9].

Due to its platform independence [6], the number of executed bytecodes is a useful metric for the analysis of algorithm complexity. The reproducibility of profiles (for deterministic programs) is convenient for the developer, since it is not necessary to ensure particular system conditions for benchmarking and profiling. As factors such as the system load do not affect the profiling results, the profiler may be executed as a background process on the developer's machine. This increases productivity, as there is no need to set up and maintain a dedicated, 'standardized' profiling environment. Moreover, multiple developers may concurrently profile an application under various conditions (e.g. different inputs) on different machines. Even though the characteristics of these machines may differ significantly, the collected profiles are directly comparable.

In contrast, using traditional, timing-based profiling techniques, the developer has to profile an application on the same type of system (hardware and JVM) where it is intended to be deployed later, in order to obtain relevant measurements[§]. In particular for server applications, this is often not possible, as the developer may not always have access to the concrete target platform. Furthermore, the profiles obtained by timing-based profilers frequently suffer from significant measurement perturbation due to the profiling, which may mislead the developer who searches for performance bottlenecks.

While bytecode instruction counting as profiling metric does not directly relate to CPU time on a particular system, it indicates the developer in which parts of an application the algorithmic complexity is high (which manifests in a high number of issued bytecodes). Hence, bytecode instruction counting is not a complete replacement for CPU time profiling, but it places a new kind of tool at the developer's disposal, which is uncomplicated in its application, can be easily integrated early in the development cycle, and helps to discover algorithmic inefficiencies. The latter argument is supported by [10], where the authors show that program optimizations using profiles based on bytecode instruction counting result in performance improvements of the optimized programs.

It is possible to assign distinct weights to different (sequences of) bytecode instructions, according to their complexity on a particular system (hardware and JVM). Such an approach would allow CPU time to be estimated in certain environments. While an analysis of the correlation of the CPU time and bytecode metrics is not in the scope of this article, we discuss some initial ideas in Section 10.

[§]Even on the same hardware, the CPU time for executing a given program may vary significantly depending on manufacturer and version of Java runtime system used.

Another advantage of bytecode instruction counting is that it enables fully portable profiling tools. This helps to reduce the development and maintenance costs for profiling tools, as a single version of a profiler can be compatible with any kind of virtual machine. This is in contrast to prevailing profilers, which exploit low-level, platform-dependent features (e.g. to obtain the CPU time of a thread from the underlying operating system) and require profiling agents to be written in native code.

Our deterministic sampling mechanism also exposes the (approximate) number of executed bytecodes to the user-defined profiling agent. While the deterministic sampling allows the estimation of the relative number of bytecodes executed in a calling context C, keeping track of the total number of executed bytecodes also enables the estimation of the absolute number of executed bytecodes in C (by multiplying the estimated relative number of bytecodes executed in C with the total number of executed bytecodes).

In contrast to timing-based sampling, which may be affected by the measurement granularity, deterministic sampling using bytecode instruction counting enables profiling at an extremely fine-grained level, which is also applicable for programs with a very short execution time. In Section 7 we show that our approach yields high accuracy and that the performance is comparable with the standard 'hprof' profiling agent in its sampling mode, even though the effective sampling rate is significantly higher in our approach.

3. RUNTIME CLASSES AND API

In this section we give an overview of the classes needed to run programs that have been transformed by Komorium and we present our simple but flexible API to write custom profiling agents. In Section 4 we will explain the details of our program transformation scheme.

3.1. Method identifier

Komorium reifies the call stack as an array of method identifiers, which are instances of `MID` (see Figure 1). `MID` is a marker interface. The default implementation, `MIDImpl` (see Figure 1), represents a method by its class name, method name, and method signature. It does not keep a reference to the class in order not to prevent the class from being reclaimed by the garbage collector. While Komorium actually supports custom implementations of `MID` through a factory mechanism, in this article we assume that each method identifier is an instance of `MIDImpl` in order to simplify the presentation.

3.2. Thread context

Komorium relies on the periodic activation of a user-defined profiling agent to process samples of the reified call stack. Each thread in the system periodically activates the profiling agent depending on the number of bytecodes it has executed. Therefore, each thread maintains a *bytecode instruction counter* in its *thread context*.

Each thread has an associated thread context, which is an instance of `TC`. Part of the `TC` implementation is shown in Figure 2. The `instrCounter` field is the bytecode instruction counter. During its execution, a thread is always associated with the same `TC` instance. This association may


```
// Method identifier (marker interface).
public interface MID {}


// Default method identifier implementation.
public final class MIDImpl implements MID {

   // Name of class where method to be identified is defined.
   final String className;

   // Method name and signature.
   final String  methNameSig;


   public MIDImpl(Class c, String methNameSig) {
      className = c.getName();
      this.methNameSig = methNameSig;
   }

   public String getClassName() { return className; }

   public String getMethodNameSig() { return methNameSig; }

}
```

Figure 1. `MID` and `MIDImpl`.

be implemented with the aid of a thread-local variable[¶], as shown in Figure 2. The static method `getTC()` returns the thread context of the current thread. If it does not already exist, it is created.

For each thread, its associated `TC` instance manages a pool of reified call stacks. `newStack()` provides a reified call stack and `releaseStack(MID[])` returns the reified stack into the pool. The size of a reified call stack is defined by `stackSize`, a configuration parameter. The pool of reified call stacks is managed as a stack with at most `maxStacks` entries (another configuration parameter). The fields `stacks` and `nStacks` implement the pool (stack) of reified call stacks.

Normally, a new thread obtains a reified call stack once and then uses it until it terminates. However, if the thread calls a native method, the reified stack is not available until the native method returns (native code is not modified by Komorium, hence the reified stack is not passed to the native code). If the native method calls a Java method, that method has to obtain a new reified stack. For this reason, a thread may need more than one reified stack. In order to avoid the repeated allocation of reified stacks, each thread context maintains a (typically very small) pool of reified stacks.

3.3. Profiling agent

The `TC` implementation provides the method `triggerSampling(MID[], int)` to invoke the user-defined profiling agent. A reference to the profiling agent is stored in the `profiler` field.

[¶]Thread-local variables are instances of `java.lang.ThreadLocal`. Each thread has its own instance of a thread-local variable.

```
public final class TC {

   // Associates a TC instance with each thread.
   static ThreadLocal threadContext = new ThreadLocal();

   static SamplingProfiler profiler; // Custom profiling agent.

   static int stackSize; // Maximum supported size of reified stack (e.g., 1000).

   static int maxStacks; // Maximum number of reified stacks kept for each thread (e.g., 5).

   public int instrCounter; // Counter to trigger activation of profiling agent.

   int gran; // Current sampling granularity.

   // Pool of reified stacks, implemented as a stack of reified stacks.
   MID[][] stacks = new MID[maxStacks][];

   int nStacks = 0; // Number of reified stacks currently in the pool.


   // Returns the TC instance of the current thread.
   public static TC getTC() {
      TC tc = (TC)threadContext.get();
      if (tc == null) { tc = new TC(); threadContext.set(tc); tc.initialize(); }
      return tc;
   }

   // Registers the current thread with the profiling agent.
   void initialize() {
      instrCounter = gran = INTEGER.MAX_VALUE;
      if (profiler != null) instrCounter = gran = profiler.register(Thread.currentThread());
   }

   // Returns a reified stack. If possible, the reified stack is taken from the pool.
   public MID[] newStack() { return (nStacks == 0) ? new MID[stackSize] : stacks[--nStacks]; }

   // Inserts a reified stack that is not used anymore into the pool, unless the pool is full.
   public void releaseStack(MID[] stack) {
      if (nStacks < maxStacks) stacks[nStacks++] = stack;
   }

   // Invokes the profiling agent.
   public void triggerSampling(MID[] stack, int sp) {
      if (profiler == null) instrCounter = gran = INTEGER.MAX_VALUE;
      else instrCounter = gran = profiler.processSample(stack, sp, gran - instrCounter);
   }

   ...
}
```

Figure 2. Part of the TC implementation.


```
public interface SamplingProfiler {

   /*
     Registers a new thread t with the profiling agent. The return value
     is the new sampling granularity for t, i.e., the (approximate) number
     of bytecodes that t will execute before invoking
     processSample(MID[], int, int) for the first time.
   */
   public int register(Thread t);

   /*
     Processes a sample of the current reified call stack. stack[i]
     represents the ith method frame on the reified stack (0 ≤ i < sp). stack[0]
     is the bottom of the reified stack, while stack[sp-1] corresponds to the
     currently executing method of the program being profiled. bytecodes is
     the approximate number of executed bytecodes since the last invocation
     of the profiling agent by the calling thread. The return value is the
     new sampling granularity for the calling thread, i.e., the
     (approximate) number of bytecodes the calling thread will execute
     before invoking processSample(MID[], int, int) again.
   */
   public int processSample(MID[] stack, int sp, int bytecodes);

}
```

Figure 3. `SamplingProfiler` interface.

In this article we do not show the details of how the user-defined profiling agent is initially loaded. We use a similar loading mechanism to that described in [8].

The custom profiling agent is an implementation of the `SamplingProfiler` interface (see Figure 3). The profiling agent has to provide two methods: `register(Thread)` and `processSample(MID[], int, int)`. When a new thread is associated with its `TC` instance, the method `initialize()` (see Figure 2) invokes the profiling agent's `register(Thread)` method. The profiling agent may use this method to allocate data structures for the profiling information generated by the corresponding thread.

The method `processSample(MID[], int, int)` is periodically invoked by each thread in the system. The first two arguments represent the current reified stack of the calling thread (the first argument is a reference to the reified call stack, the second is the current stack pointer). The third argument is the (approximate) number of bytecodes the calling thread executed since the previous invocation of the profiling agent, enabling the profiling agent to keep track of the (approximate) number of executed bytecodes for each thread. Whenever `processSample(MID[], int, int)` is called, the profiling agent will update the execution statistics to include the passed reified call stack. Preserving the (approximate) number of executed bytecodes allows the profiling agent to estimate the absolute number of bytecodes executed in a calling context. Without this information, only the relative number of executed bytecodes could be estimated (i.e. the relative frequency of a certain reified call stack).

Both methods `register(Thread)` and `processSample(MID[], int, int)` return an `int` value representing the new *sampling granularity*. This value indicates after how many executed bytecodes the current thread will invoke `processSample(MID, int, int)`. Hence, the sampling granularity indirectly defines the sampling rate (samples per second): a low sampling

granularity results in a high sampling rate, a high sampling granularity implies a low sampling rate. In the simplest case, the sampling granularity is constant. However, the profiling agent may use different sampling granularities for distinct threads, or it may change the sampling granularity after each invocation of `processSample(MID[], int, int)`. In Section 7.1 we show that slightly varying the sampling granularity in a randomized way helps to improve the accuracy of generated profiles.

4. PROGRAM TRANSFORMATION SCHEME

In the following we explain the details of our program transformations. Section 4.1 discusses the reification of the call stack, Section 4.2 explicates the periodic activation of a custom profiling agent based on bytecode instruction counting, Section 4.3 addresses issues related to native code, which cannot be modified by Komorium, and Section 4.4 illustrates our transformations with a concrete example.

4.1. Stack reification

Profilers based on sampling techniques periodically take a snapshot of the current call stack. As our goal is to write portable profiling agents in pure Java, we need a mechanism to obtain the current call stack of a thread. The only standard API for this purpose is the `Throwable` class: using '`new Throwable().getStackTrace()`', a thread can obtain a trace of its call stack. Unfortunately, the stack trace provides only class and method name for each stack frame, but not the method signature. As the stack trace may also include the name of the Java source file and a line number for each stack frame, it may be feasible to obtain the method signature from the source file. However, the source file may not always be available (e.g. consider the use of libraries that are distributed only as archives of compiled classes), i.e. in general it may not be possible to distinguish overloaded methods (methods of the same class with the same name but different signatures). Moreover, stack traces may be incomplete on certain JVMs, since the `Throwable` specification gives a lot of flexibility to the JVM implementor.

In order to avoid these shortcomings, Komorium does not rely on the `Throwable` API but reifies the call stack, i.e. Komorium transforms programs so that each thread maintains a representation of its call stack. The reified stack consists of an array of `MID` instances (`MID[] stack`) and a stack pointer (`int sp`), which is the index of the next free element on the reified stack. `stack[`i`]` is the ith method identifier on the reified stack ($0 \leq i <$ `sp`). `stack[0]` is the bottom of the reified stack, while `stack[sp-1]` corresponds to the currently executing method.

For each Java method (and constructor), Komorium adds a static field to hold the corresponding `MID` instance, which is allocated by the static initializer. The method is rewritten in order to pass the reified stack as two extra arguments, `stack` and `sp`. In the beginning of each rewritten Java method, the corresponding method identifier is pushed onto the reified stack and `sp` is incremented. As `sp` is an `int`, it is always passed by value, i.e. callees receive (a copy of) the new value of `sp` and may increment it, which does not affect the value of `sp` in the caller. If a method `m()` invokes first `a()` and then `b()`, both `a()` and `b()` will receive the same `stack` reference and the same value of `sp` as extra arguments. `b()` will overwrite the method identifiers that were pushed onto the reified stack during the execution of `a()`.

4.2. Periodic sampling

In order to schedule the regular activation of the custom profiling agent, each thread has a counter `instrCounter` of the number of executed bytecodes since the last invocation of the `processSample(MID[], int, int)` method. The counter is kept within the thread context. In order to make this counter accessible within each method (without resorting to thread-local variables, which would cause high overhead), we pass the thread context as a third additional argument to all invocations of non-native methods/constructors. If `instrCounter` exceeds the sampling granularity, the thread calls `triggerSampling(MID[], int)`, which in turn invokes the profiling agent (see Figure 2).

As the JVM offers dedicated bytecodes for the comparison with zero, `instrCounter` runs from the sampling granularity down to zero, i.e. the following conditional, which we call *granularity check*, schedules the periodic activation of the profiling agent (`tc` represents the current thread context, `stack` and `sp` the current reified stack):

```
if (tc.instrCounter <= 0) tc.triggerSampling(stack, sp);
```

Komorium instruments the bytecode of methods in order to decrement the `instrCounter` according to the number of executed bytecodes. As `instrCounter` is directly decremented for performance reasons, it has been declared `public` in Figure 2. For each Java method, Komorium performs a basic block analysis to compute a control flow graph. In the beginning of each basic block it inserts a code sequence that decrements the `instrCounter` field by the number of bytecodes within the basic block.

The basic block analysis algorithm is not hard-coded in Komorium, via a system property the user can specify a custom analysis algorithm. In the default basic block analysis, only bytecodes that may change the control flow non-sequentially (i.e. jumps, branches, return of method or JVM subroutine, exception throwing) end a basic block. Method, constructor, or JVM subroutine invocations do not end basic blocks of code, because we assume that the execution will usually return after the call. Our definition of basic block corresponds to that used in [7] and is related to the factored control flow graph [11].

The advantage of this basic block definition is that it yields rather large basic blocks. Therefore, the number of locations is reduced where updates to `instrCounter` have to be inserted, resulting in a lower profiling overhead. As long as no exceptions are thrown, the bytecode instruction counting is precise. However, exceptions (e.g. an invoked method may terminate abnormally throwing an exception) may cause some (minor) imprecision, as we always count all bytecodes in a basic block, even though some bytecodes may not be executed in case of an exception, i.e. we may count more bytecodes than are actually executed.

The granularity check is inserted in each basic block after the update of `instrCounter`. In Section 5.2 we present an optimization to reduce the number of checks.

As the bytecode instrumentation, which may trigger the profiling agent, is limited to the beginning of each basic block, the presence of very large basic blocks[‖] may significantly delay the invocation

[‖] For instance, the 'mpegaudio' benchmark of the SPEC JVM98 benchmark suite [12] includes several classes with methods that contain basic blocks with more than 100 bytecodes.

of the profiling agent. To solve this issue, Komorium allows the length of non-instrumented execution paths to be limited. This limit, which we call *MaxPath* in this article, is specified through a system property. If $MaxPath > 0$, Komorium splits basic blocks, if their length exceeds *MaxPath*.

4.3. Native code and reflection issues

As native code is not changed by the rewriting, we add a wrapper method with the unmodified signature for each rewritten Java method. Therefore, native code is able to invoke Java methods with the unmodified signatures. A wrapper method first obtains the current thread context by calling the static method `TC.getTC()` (see Figure 2). Then it invokes `newStack()` on the thread context to acquire a reified stack. Next, the rewritten method is invoked, passing the thread context, the reified stack, and an initial stack pointer value of zero as extra arguments. Finally, the reified stack is released by invoking `releaseStack(MID[])`. More details on wrapper methods are discussed with a concrete example in Section 4.4.

Whenever native code calls a rewritten Java method, it will invoke the wrapper method with the unmodified signature, i.e. the caller's reified stack is not passed and consequently the information concerning the call stack is lost. However, because these callbacks from native code to rewritten Java code are not frequent, this inexactness is not a problem in practice.

For native methods, which we cannot rewrite, we add so-called 'reverse' wrappers which discard the extra arguments before invoking the native method. The 'reverse' wrappers allow rewritten code to invoke all methods with the additional arguments, no matter whether the callee is native or not. As Komorium does not modify native code, there will be no samples of native methods.

Our transformation scheme introduces a number of overloaded methods receiving the reified stack as extra arguments, which are visible through the reflection API and hence may change the program behaviour. The methods `getConstructors()`, `getDeclaredConstructors()`, `getMethods()`, and `getDeclaredMethods()` of `java.lang.Class` return arrays of reflection objects (i.e. instances of `java.lang.reflect.Constructor`, respectively `java.lang.reflect.Method`), representing wrapper methods (with the unmodified signature) as well as methods with extended signature. If an application selects a method from this array considering only the method name (but not the signature), it may try to invoke a method with extended signature, but fail to provide the extra argument, resulting in an `IllegalArgumentException`. For instance, we encountered this kind of problem with some versions of Tomcat. We solved this issue by patching the aforementioned methods of `java.lang.Class` to filter out the reflection objects that represent methods with extended signature. This modification of `java.lang.Class` was straightforward, because in standard JDKs these methods are implemented in Java (and not in native code).

4.4. Rewriting example

The example in Figure 4 illustrates our transformation scheme. To the left is the class `Foo` with methods `sq(int)` and `sqSum(int, int)` before rewriting, to the right is the rewritten version. For the sake of better readability, in this article we show all transformations on Java code, whereas Komorium works at the JVM bytecode level. The methods compute the following mathematical functions: $sq(x) = x^2$ and $sqSum(a, b) = \sum_{i=a}^{b} sq(i) = \sum_{i=a}^{b} i^2$. In `sqSum(int, int)` we use an infinite `while()`


```
class Foo {

   static int sq(int x) {

      return x * x;
   }

   static int sqSum(int from, int to) {

      int result = 0;
      while (true) {

         if (from > to) {

            return result;
         }

         result += sq(from);
         ++from;
      }
   }

}
```

```
class Foo {
   private static final MID mid_sq, mid_sqSum;
   static {
      Class c   = Class.forName("Foo");
      mid_sq    = new MIDImpl(c, "sq(I)I");
      mid_sqSum = new MIDImpl(c, "sqSum(II)I");
   }

   static int sq(int x, TC tc, MID[] stack, int sp) {
      stack[sp++] = mid_sq;
      if ((tc.instrCounter -= 4) <= 0)
         tc.triggerSampling(stack, sp);
      return x * x;
   }
   static int sq(int x) {
      TC tc = TC.getTC();
      MID[] stack = tc.newStack();
      int result = sq(x, tc, stack, 0);
      tc.releaseStack(stack);
      return result;
   }

   static int sqSum(int from, int to,
                    TC tc, MID[] stack, int sp) {
      stack[sp++] = mid_sqSum;
      if ((tc.instrCounter -= 2) <= 0)
         tc.triggerSampling(stack, sp);
      int result = 0;
      while (true) {
         if ((tc.instrCounter -= 3) <= 0)
            tc.triggerSampling(stack, sp);
         if (from > to) {
            if ((tc.instrCounter -= 2) <= 0)
               tc.triggerSampling(stack, sp);
            return result;
         }
         if ((tc.instrCounter -= 7) <= 0)
            tc.triggerSampling(stack, sp);
         result += sq(from, tc, stack, sp);
         ++from;
      }
   }
   static int sqSum(int from, int to) {
      TC tc = TC.getTC();
      MID[] stack = tc.newStack();
      int result = sqSum(from, to, tc, stack, 0);
      tc.releaseStack(stack);
      return result;
   }
}
```

Figure 4. Rewriting example.

loop with an explicit conditional to end the loop instead of a `for()` loop that the reader might expect, in order to better reflect the basic block structure of the compiled JVM bytecode.

While `sq(int)` consists only of a single basic block of code, `sqSum(int, int)` has four blocks: The first (two bytecodes) initializes the local variable `result` with zero, the second (three bytecodes) compares the values of the local variables `from` and `to` and branches, the third (two bytecodes) returns the value of the local variable `result`, and the fourth block (seven bytecodes) adds the return value of `sq(from)` to the local variable `result`, increments the local variable `from`, and jumps to the begin of the loop. We assume that *MaxPath* ≥ 7 so that none of these blocks is split.

In the rewritten code, the static initializer allocates the method identifiers `mid_sq` (respectively `mid_sqSum`) to represent invocations of `sq(int)` (respectively `sqSum(int, int)`) on the reified stack. The rewritten methods receive three extra arguments, the thread context (`tc`), the reified stack (`stack`), and the stack pointer (`sp`). The stack pointer is the index of the callee method in the reified stack. First, the rewritten methods update the stack, i.e. they push their respective method identifier on top of the reified stack. The stack pointer is incremented to point to the next free entry.

In the beginning of each basic block of code, the bytecode instruction counter of the current thread (`tc.instrCounter`) is decremented by the number of bytecodes in the block. If the new value of the bytecode instruction counter is equal to or smaller than zero, the reified stack is to be processed by the profiling agent (`triggerSampling(stack, sp)`). Method invocations (e.g. `sq(from)`) are rewritten in order to pass the thread context, the reified stack, and the stack pointer as extra arguments.

Wrapper methods with the unmodified signature are added to allow native code, which is not aware of the additional arguments, to invoke the rewritten methods. The wrapper methods obtain the current thread context (`TC.getTC()`) and a reified stack (`tc.newStack()`) before invoking the rewritten method. Initially, the stack pointer is zero. If the method returns normally, the reified stack is released (`tc.releaseStack(stack)`) and the result is returned. The methods `newStack()` and `releaseStack(MID[])` of class `TC` (see Figure 2) maintain a (usually small) pool of reified stacks for each thread, in order to reduce the number of allocations of reified stacks. For example, if a native method repeatedly invokes the wrapper of a rewritten Java method, the same reified stack instance may be reused, assuming that the rewritten method returns normally. If the rewritten method terminates abnormally (i.e. throwing an exception), `releaseStack(MID[])` will not be called. In this case, the reified stack may be later reclaimed by the garbage collector. Nonetheless, in general this scheme reduces the number of allocations of reified stacks and therefore also the workload for the garbage collector.

5. OPTIMIZATIONS

In order to reduce the profiling overhead, we implemented two optimizations, which are presented below.

5.1. Leaf methods

The following optimization reduces the overhead of maintaining the reified stack in leaf methods. As leaf methods do not invoke any methods, there is no need to pass an updated reified stack to other methods. In leaf methods, the reified stack is used only as argument to `triggerSampling(MID[], int)`, i.e. it is needed only if the bytecode instruction counter


```
public final class TC {
   ...

   // Invokes the profiling agent in leaf methods.
   public void triggerSamplingLeaf(MID[] stack, int sp, MID mid_caller) {
      if (profiler == null) instrCounter = gran = INTEGER.MAX_VALUE;
      else {
         stack[sp] = mid_caller;
         instrCounter = gran = profiler.processSample(stack, sp + 1, gran - instrCounter);
      }
   }

   ...
}
```

Figure 5. Part of the `TC` implementation: optimization for leaf methods.

```
static int sq(int x, TC tc,
              MID[] stack, int sp) {
   stack[sp++] = mid_sq;
   if ((tc.instrCounter -= 4) <= 0)
      tc.triggerSampling(stack, sp);
   return x * x;
}
```

```
static int sq(int x, TC tc,
              MID[] stack, int sp) {

   if ((tc.instrCounter -= 4) <= 0)
      tc.triggerSamplingLeaf(stack, sp, mid_sq);
   return x * x;
}
```

Figure 6. Rewriting example: optimization for leaf methods.

exceeds the sampling granularity. Therefore, it is possible to move the update of the reified stack from the beginning of the method into the conditional that performs the granularity check. The class `TC` offers a dedicated method `triggerSamplingLeaf(MID[], int, MID)` for this purpose (see Figure 5).

Figure 6 illustrates this optimization on the leaf method `sq(int)` of Figure 4. To the left is the rewritten method without optimization, to the right is the optimized version. In general, this optimization significantly reduces the number of updates of the reified stack in leaf methods. This optimization may only cause extra overhead in the rare case of a complex leaf method (e.g. a leaf method with a loop) that triggers the sampling profiler more than once during one execution. In this case, the update of the reified stack (which is an idempotent operation) is repeated by multiple invocations of `triggerSamplingLeaf(MID[], int, MID)`.

5.2. Granularity checks

The following optimization aims at reducing the number of granularity checks. According to the description in Section 4.2, the granularity check is inserted in the beginning of each basic block of code, directly after the update of `instrCounter`. The regularity of the granularity check is essential

to achieve a high accuracy of profiles. For example, if the bytecode instruction counter exceeds the sampling granularity while executing method `m()`, but the granularity check is performed in a callee method of `m()`, the sample will be wrongly attributed to the callee. Hence, the granularity check has to be frequent in order not to create a bias in the profiling samples. Nonetheless, as the granularity check causes significant overhead, we try to reduce the number of checks, inserting them only in the following locations.

- Basic blocks that terminate (i.e. return or throw an exception). This ensures that a sample is not wrongly attributed to the caller.
- Basic blocks that contain a method invocation. This ensures that a sample is not wrongly attributed to a callee.
- Basic blocks that are the beginning of loops, to ensure that the check is present in iterative computations.

After the insertion of the granularity check in these locations, we examine whether there are execution paths longer than *MaxPath* which do not execute any granularity check. If this is the case, we insert additional granularity checks (as few as possible), in order to grantee that at most *MaxPath* bytecodes (of the original program) are executed between two subsequent granularity checks.

In the example in Figure 4, this optimization allows to remove the granularity check from the first block of the method `sqSum(int, int)`. In the other three blocks, the granularity check is not removed, since the blocks represent the beginning of a loop, return, or invoke another method.

6. EXAMPLE PROFILING AGENT

In the following we show the sampling profiling agent (SA) which we employed for the evaluation in Section 7. SA is used in a similar way as the standard profilers included in many JDKs, such as those invoked by the '`-Xrunhprof:cpu=samples`' or '`-agentlib:hprof=cpu=samples`' (the latter since JDK 1.5.0) command line options: it collects execution statistics which are written to a file after completion of the profiled program. Existing tools, such as prophIt (http://prophit.westslopesoftware.com/), may be used to visualize the output.

In Section 6.1 we discuss a simple implementation of SA to illustrate how a typical profiling agent may process the samples. In Section 6.2 we explain an important optimization that significantly reduces the overhead for low sampling granularities (i.e. for high sampling rates).

6.1. Processing samples

Part of the (simplified) SA implementation is depicted in Figures 7 and 8. SA implements the `SamplingProfiler` interface. It uses a constant sampling granularity. The configuration of SA (e.g. sampling granularity, output file, etc.), which is not shown here, is based on a set of system properties.

SA collects the samples of all threads in a single structure, the *samples tree*. The tree nodes are of the type `InvocationNode` (see Figure 8). A given reified stack ($\texttt{stack}[i], 0 \leq i < \texttt{sp}, \texttt{sp} > 0$)


```
public class SA implements SamplingProfiler, Runnable {

   // Root of the samples tree.
   final InvocationNode root = new InvocationNode();

   // Approximate number of executed bytecodes.
   long totalBytecodes = 0;

   // Configuration parameters.
   int granularity; // sampling granularity
   ...


   public SA() {
      // get configuration parameters from system properties
      granularity = ...;
      ...

      // create shutdown hook
      Runtime.getRuntime().addShutdownHook(new Thread(this));
   }

   public int register(Thread t) { return granularity; }

   public synchronized int processSample(MID[] stack, int sp, int bytecodes) {
      totalBytecodes += bytecodes;
      InvocationNode n = root;
      for (int i = 0; i < sp; ++i) n = n.getOrCreateCallee(stack[i]);
      ++n.counter;
      return granularity;
   }

   // Executed by the shutdown hook.
   public synchronized void run() {
      // write profiling statistics (i.e., samples tree and
      // approximate number of executed bytecodes) into a file
      ...
   }

}
```

Figure 7. Sampling profiling agent.

corresponds to a path of length `sp` in the samples tree, starting from the `InvocationNode` `root`. Each node in the samples tree has a `counter`, which represents the number of samples with the same corresponding reified stack. Each node may have an arbitrary number of children maintained in the map `callees`, which maps method identifiers (instances of `MID`) to the children nodes. In order to simplify the presentation, in Figure 8 the map `callees` is of the type `java.util.HashMap`. Our actual implementation uses a special, optimized map implementation, because access is very frequent.

The `processSample(MID[], int, int)` method iterates through the passed reified stack and follows the corresponding path in the samples tree. If a node does not yet exist in the samples tree, it is created (`getOrCreateCallee(MID)`). When the top of the reified stack


```
public class InvocationNode {

   // Number of samples where this node corresponds to top of reified stack.
   public long counter = 0;

   // Maps method identifiers to InvocationNodes.
   HashMap callees = null;


   public InvocationNode getOrCreateCallee(MID mid) {
      if (callees == null) {
         callees = new HashMap(5);
         InvocationNode n = new InvocationNode();
         callees.put(mid, n);
         return n;
      }

      InvocationNode n = (InvocationNode)callees.get(mid);
      if (n == null) {
         n = new InvocationNode();
         callees.put(mid, n);
      }
      return n;
   }

   ...
}
```

Figure 8. Node in the samples tree.

is reached (i.e. `stack[sp-1]`), the `counter` in the corresponding node is incremented. `processSample(MID[], int, int)` is `synchronized`, since SA maintains a single samples tree for all threads in the system, which may invoke the method concurrently. SA also keeps track of the (approximate) total number of executed bytecodes in the field `totalBytecodes`**.

The method `register(Thread)` just returns the sampling granularity. If a profiling agent maintains separate statistics for each thread, `register(Thread)` may be used to allocate the necessary data structures for new threads.

SA employs a shutdown hook, a dedicated thread that starts running when the JVM is about to terminate, i.e. when the program exits normally (the last non-daemon thread has terminated or `System.exit(int)` is called), or when the JVM is terminated in response to a user interrupt. The shutdown hook writes the collected profiling information (i.e. the samples tree) into a file. As the writing of a tree structure in a serialized form is straightforward, we have omitted such details in Figure 7.

**If the profiling agent uses a high sampling granularity close to `Integer.MAX_VALUE`, the third argument to `processSample(MID[], int, int)` may be negative because of a possible overflow. In this case, the profiling agent has to be prepared to handle such an overflow.

6.2. Reducing the overhead for low sampling granularities

As shown in Figure 7, the method `processSample(MID[], int, int)` iterates through the reified stack and follows the corresponding path in the samples tree. For low sampling granularities (i.e. high sampling rates) and deep call stacks, this may cause excessive overhead.

However, if the sampling granularity is low, it is likely that for subsequent invocations of `processSample(MID[], int, int)` by the same thread, some elements on the bottom of the reified stacks are identical, i.e., upon subsequent invocations, `processSample(MID[], int, int)` follows paths in the samples tree that have the same prefix. Therefore, it would pay off to cache the recently followed path for each thread.

As the reified stack is read exclusively by SA, whereas the transformed program only writes to the reified stack, it is possible for SA to directly exploit the reified stack as a per-thread cache of the most recently followed path in the samples tree. It is sufficient to make `InvocationNode` implement the `MID` interface to store references to nodes of the samples tree in the reified stack. Hence, `processSample(MID[], int, int)` has to scan the reified stack from the top to the bottom in order to discover the topmost instance of `InvocationNode`. There is no need to process the stack below, because it has not been overwritten since the previous invocation of `processSample(MID[], int, int)`. Figure 9 shows the relevant aspects of the optimized version of SA.

For low sampling granularities (e.g. below 1000), the linear search for the topmost instance of `InvocationNode` on the reified stack takes only a few steps on average. For high sampling granularities, the linear search may require more steps, but the extra overhead is not noticeable, because SA is not invoked frequently anyway. Replacing the linear search with a binary search did not improve performance in our measurements.

7. EVALUATION

In this section we evaluate the accuracy of sampling profiles (Section 7.1), the accuracy of bytecode instruction counting (Section 7.2), the size of profiles (Section 7.3), as well as the profiling overhead for different sampling granularities (Section 7.4). We also compare our results with measurements of the standard 'hprof' profiling agent in its sampling mode (Section 7.5).

7.1. Accuracy of profiles

In order to assess the accuracy of sampling-based profiles, we compared them with perfect (exact) execution profiles. Relying on bytecode instruction counting, Komorium produces profiles that can be used to approximate the relative number of executed bytecodes for each calling context (call stack). Therefore, we need perfect profiles with the exact numbers of executed bytecodes as reference[††].

[††]A direct comparison of a profile produced by Komorium with one generated by a timing-based profiler would not be meaningful, since the number of executed bytecodes and CPU time are distinct metrics for different purpose. The analysis of the relationship between these two metrics is not in the scope of this article. Section 10 discusses some initial ideas for future work on CPU time prediction using profiles based on bytecode instruction counting.


```
public class SA implements SamplingProfiler, Runnable {
   ...

   public synchronized int processSample(MID[] stack, int sp, int bytecodes) {
      totalBytecodes += bytecodes;
      int i = sp - 1;  // sp > 0
      if (stack[i] instanceof InvocationNode) ++((InvocationNode)stack[i]).counter;
      else {
         InvocationNode n = null;
         for (; i > 0; --i) {
            if (stack[i - 1] instanceof InvocationNode) {
               n = (InvocationNode)stack[i - 1];
               break;
            }
         }
         if (i == 0) n = root;
         for (; i < sp; ++i) {
            n = n.getOrCreateCallee(stack[i]);
            stack[i] = n;
         }
         ++n.counter;
      }
      return granularity;
   }

   ...
}


public class InvocationNode implements MID {
   ...
}
```

Figure 9. Optimized sampling profiling agent.

We used our profiler JP [13] to generate exact profiles for comparison. JP creates a complete Method Call Tree (MCT), similar to a Calling Context Tree (CCT) with unbounded depth [14]. Each node in the MCT corresponds to a calling context. It stores the exact number of invocations of the corresponding method (with the same call stack) and the precise number of bytecodes executed in the calling context (excluding the number of bytecodes executed by callee methods). For comparison with the sampling profiles, we exploit the exact number of executed bytecodes, whereas the method invocation counters are ignored.

We use an *overlap percentage* metric as in [15,16] to compare profiles. Informally, the overlap represents the percentage of profiled information weighted by execution frequency that exists in both profiles. Two identical profiles have an overlap percentage of 100%. More formally, we define the overlap percentage as follows.

- A profile X is a set of calling contexts C_{X_i}. Each C_{X_i} is identified by its call stack, $stack(C_{X_i})$, and has an associated counter, $counter(C_{X_i})$. For a perfect profile P, $counter(C_{P_i})$ is the number of bytecodes executed in C_{P_i}. For a sampling profile S, $counter(C_{S_i})$ is the number of samples collected in C_{S_i}.

- $rce(C_{X_i}, X)$ is the relative computation effort spent in C_{X_i}:

$$rce(C_{X_i}, X) = \frac{counter(C_{X_i})}{\sum_{C_{X_k} \in X} counter(C_{X_k})}$$

- The overlap percentage $overlap(A, B)$ of two profiles A and B is computed as follows:

$$overlap(A, B) = \sum_{\substack{C_{A_i} \in A, \\ C_{B_j} \in B, \\ stack(C_{A_i}) = stack(C_{B_j})}} \min(rce(C_{A_i}, A), rce(C_{B_j}, B))$$

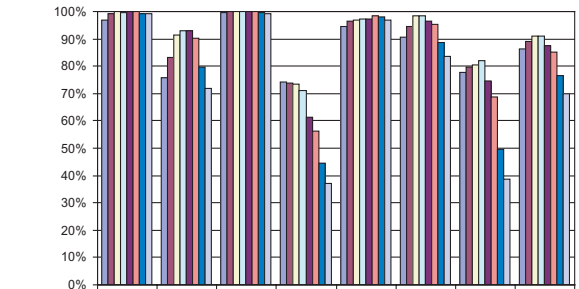
In Figure 10(a) we show the accuracy of sampling profiles (i.e. the overlap percentage of a sampling profile with the perfect one) obtained with different, constant sampling granularities in the range of 500–1 000 000 for the SPEC JVM98 benchmarks [12] with problem size 100. We applied both optimizations presented in Section 5. For all measurements, *MaxPath* was set to 50. The column 'geo. mean' gives the geometric mean of the accuracy for the different benchmark programs (average accuracy). We executed the benchmarks with the Sun JDK 1.5.0 Server VM on a Windows XP machine. As our profiling is based on bytecode instruction counting, the resulting profiles are not influenced by the processor speed, the system load, etc. With different JVMs, there may be differences in the execution of methods in JDK classes (because of different class libraries). However, as this does not perceivably affect the resulting accuracy, we only show the results obtained with the Sun JDK 1.5.0 Server VM here.

On average, the obtainable accuracy with a profiling agent that uses a constant sampling granularity (e.g. using the profiling agent shown in Figure 7) is 91%. For 'compress', 'db', 'mpegaudio', and 'mtrt', the accuracy is up to 98–100%. We observed the lowest accuracies with 'javac' (maximum 74%), 'jack' (maximum 82%), and 'jess' (maximum 93%). These are also the benchmarks with the largest perfect profiles (e.g. deep call stacks, many short methods, etc.).

The *MaxPath* setting was important to yield a high profile accuracy for 'mpegaudio': Without limiting the length of non-instrumented execution paths, the accuracy was about 5% lower in most settings ('mpegaudio' executes several methods with large basic blocks of code). For all other benchmarks, the setting of *MaxPath* did not affect the resulting profile accuracy.

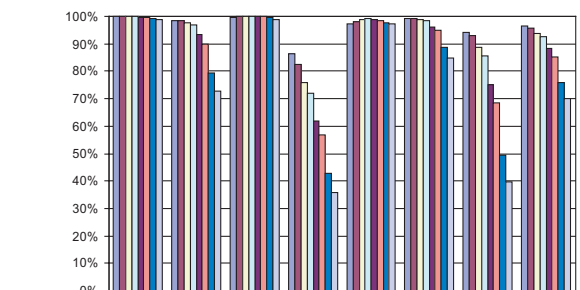
The best results are obtained with a sampling granularity of 5000–10 000. As one could expect, the accuracy decreases with increasing sampling granularity (i.e. fewer samples). Surprisingly, also for lower granularities (500–1000), the accuracy decreases. As shown in Figure 10(a), for most benchmarks there is a sampling granularity so that a smaller value does not further improve the accuracy. This is because program behaviour can correlate with a deterministic sampling mechanism, reducing the accuracy of profiles [15].

The accuracy can be improved by adding a small random integer r to the sampling granularity [17]. For our measurements, r was uniformly distributed in the range $0 \leq r < 100$. As illustrated in Figure 10(b), this randomization increases the overlap percentage for lower sampling granularities. With randomization, a lower sampling granularity (i.e. more samples) generally results in a higher accuracy. A sampling granularity of 500 yields an average profile accuracy of more than 96%. In order to enable reproducible results despite the randomization, the profiling agent may use a separate pseudo-random number generator for each thread and initialize it with a constant seed.



Sampling Granularity	com-press	jess	db	javac	mpeg-audio	mtrt	jack	geo. mean
500	97.06%	75.74%	99.72%	74.08%	94.35%	90.63%	77.54%	86.42%
1000	99.14%	83.08%	99.83%	73.81%	96.30%	94.52%	79.68%	88.94%
5000	99.89%	91.58%	99.97%	73.44%	96.78%	98.41%	80.39%	90.94%
10000	99.70%	93.10%	99.97%	71.01%	97.38%	98.34%	81.91%	91.00%
50000	99.90%	92.78%	99.95%	61.35%	97.35%	96.31%	74.78%	87.68%
100000	99.81%	90.18%	99.90%	56.10%	98.39%	95.22%	68.88%	85.19%
500000	99.05%	79.71%	99.53%	44.69%	97.87%	88.72%	49.73%	76.38%
1000000	99.16%	71.96%	99.08%	36.96%	96.96%	83.73%	38.53%	69.92%

(a)



Sampling Granularity	com-press	jess	db	javac	mpeg-audio	mtrt	jack	geo. mean
500+r	99.83%	98.45%	99.76%	86.47%	97.45%	99.17%	93.97%	96.33%
1000+r	99.93%	98.32%	99.87%	82.41%	98.21%	99.37%	93.02%	95.68%
5000+r	99.91%	97.54%	99.96%	75.71%	98.92%	98.84%	88.74%	93.82%
10000+r	99.86%	96.96%	99.97%	72.17%	99.17%	98.36%	85.75%	92.61%
50000+r	99.67%	93.45%	99.92%	62.02%	98.95%	96.30%	75.28%	88.16%
100000+r	99.73%	90.01%	99.88%	56.92%	98.63%	94.79%	68.67%	85.27%
500000+r	99.40%	79.57%	99.52%	42.82%	97.80%	88.72%	49.28%	75.83%
1000000+r	99.00%	72.89%	98.87%	35.90%	97.22%	84.87%	39.68%	70.18%

(b)

Figure 10. Profile accuracy (overlap percentage) for different sampling granularities. (a) Constant sampling granularity. (b) Randomized sampling granularity, r is a uniformly distributed random integer ($0 \leq r < 100$).

Table I. Relative error δb of the computed approximate number of executed bytecodes for different sampling granularities.

	compress	jess	db	javac	mpegaudio	mtrt	jack
500	-4.9×10^{-8}	-7.6×10^{-8}	-1.5×10^{-7}	-1.0×10^{-4}	-2.7×10^{-8}	-6.2×10^{-6}	6.4×10^{-3}
1000	-8.6×10^{-8}	-4.7×10^{-7}	-4.2×10^{-7}	-9.8×10^{-5}	-7.7×10^{-8}	-6.5×10^{-6}	6.4×10^{-3}
5000	-1.8×10^{-7}	-3.0×10^{-6}	-4.3×10^{-6}	-1.0×10^{-4}	-1.0×10^{-7}	-9.4×10^{-6}	6.4×10^{-3}
10 000	-2.6×10^{-8}	-2.0×10^{-7}	-2.9×10^{-6}	-9.4×10^{-5}	-4.7×10^{-7}	-1.4×10^{-5}	6.4×10^{-3}
50 000	-6.7×10^{-7}	-2.2×10^{-5}	-1.3×10^{-5}	-1.3×10^{-4}	-2.1×10^{-6}	-2.6×10^{-5}	6.4×10^{-3}
100 000	-3.1×10^{-6}	-5.8×10^{-5}	-1.1×10^{-5}	-2.0×10^{-4}	-8.4×10^{-6}	-8.6×10^{-5}	6.1×10^{-3}
500 000	-4.0×10^{-5}	-1.1×10^{-4}	-4.2×10^{-4}	-1.4×10^{-4}	-1.5×10^{-5}	-4.2×10^{-4}	5.3×10^{-3}
1 000 000	-5.3×10^{-5}	-1.2×10^{-4}	-8.9×10^{-4}	-1.5×10^{-4}	-4.5×10^{-5}	-4.2×10^{-4}	3.3×10^{-3}

7.2. Accuracy of bytecode instruction counting

Komorium allows profiling agents to keep track of the approximate number of executed bytecodes b_{approx}. Thus, the profiling agent may multiply the relative computation effort $rce(C_{X_i}, X)$ with b_{approx}, in order to estimate the absolute number of bytecodes executed in the calling context C_{X_i}. In this section we examine the accuracy of b_{approx}.

In Table I we show the relative error δb of the computed approximate number of executed bytecodes b_{approx} for different sampling granularities. δb is computed as follows (b_{exact} denotes the exact number of executed bytecodes computed from the exact profile):

$$\delta b = \left(\frac{b_{\text{approx}} - b_{\text{exact}}}{b_{\text{exact}}} \right) = \left(\frac{b_{\text{approx}}}{b_{\text{exact}}} - 1 \right)$$

A negative value of δb means that b_{approx} underestimates the number of effectively executed bytecodes, whereas a positive value of δb stands for an overestimation. There are two factors contributing to δb as follows.

1. As explained in Section 4.2, the bytecode instruction counter `instrCounter` is updated in the beginning of each basic block, reflecting the execution of all bytecodes in the block. In the case of an exception, the actual number of executed bytecodes may be lower, i.e. the number of executed bytecodes is overestimated. This factor is important for programs that have large basic blocks and throw many exceptions at runtime.
2. When a thread terminates, the number of bytecodes it has executed since the last invocation of `triggerSampling(MID[], int)` or `triggerSamplingLeaf(MID[], int, MID)` is not reported to the profiling agent, resulting in an underestimation of the number of actually executed bytecodes. The importance of this factor increases with the sampling granularity and with the number of threads. In particular, a high sampling granularity in conjunction with a large number of short-living threads (i.e. no thread pooling) may significantly distort the value of b_{approx}.

As illustrated in Table I, the absolute value of the relative error $|\delta b|$ is below 1% for all settings. For all benchmarks but 'jack', $|\delta b|$ is also below 0.1% for all measured profiling granularities. For these benchmarks, b_{approx} underestimates b_{exact}, which becomes more significant with increasing sampling granularity. An overestimation can be observed only for 'jack', which is known to be a particularly exception-intensive program [18,19].

7.3. Profile size

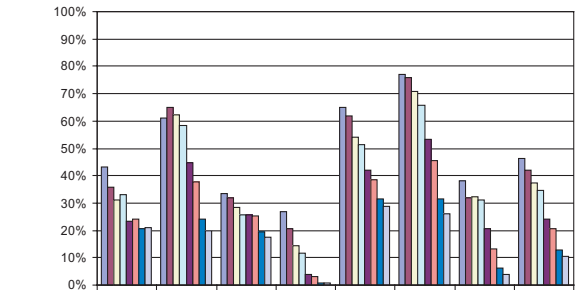
Figure 11 presents the relative size of sampling profiles in relation to the size of exact profiles for different sampling granularities. In order to measure the size of profiles, we stored them in a canonical representation. In this representation, an exact profile contains the number of executed bytecodes for each calling context, while a sampling profile includes the number of samples for each calling context. In general, a sampling profile is smaller than an exact profile, because the sampling profile may cover only a subset of the calling contexts in the exact profile. Moreover, for a calling context that exists in both profiles, the number of samples is usually smaller than the number of executed bytecodes.

Figure 11 confirms that in general, a higher sampling granularity (i.e. less samples) results in a smaller profile. This size reduction is particularly significant for 'javac' (down to 0.6%) and 'jack' (down to 3.73%), which are also the two benchmarks with the largest exact profiles. Comparing Figure 11 with Figure 10 reveals that this heavy reduction of the profile size correlates with a significant loss in profile accuracy for the benchmarks 'javac' and 'jack'.

For the results in Figure 11(a), the profiling agent used a constant sampling granularity, whereas for the results in Figure 11(b), the agent added a small random integer r, uniformly distributed in the range $0 \leq r < 100$, to the sampling granularity. For sampling granularities of 500–10 000, the profiles produced with a constant sampling granularity are consistently smaller than the profiles generated with a randomized sampling granularity. This is an indication that for lower sampling granularities, the deterministic sampling mechanism with a constant sampling granularity may systematically overlook certain call stacks. Figure 11(a) also shows a few cases where a higher sampling granularity surprisingly yields a larger profile than a lower sampling granularity (e.g. 'jess' with a sampling granularity of 500, respectively 1000), whereas such spurious phenomena cannot be found in Figure 11(b).

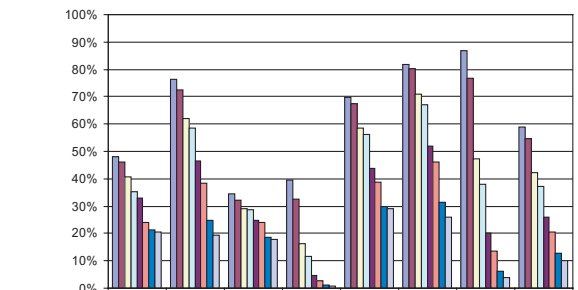
7.4. Overhead

To evaluate the overhead caused by our profiling scheme, we ran the SPEC JVM98 benchmark suite [12] with problem size 100 on a Windows XP computer (Intel Pentium 4, 2.4 GHz, 512 MB RAM). In order to obtain reproducible results, all benchmarks were run under the same conditions on a very lightly loaded system. For all settings, the entire JVM98 benchmark suite (consisting of several sub-tests) was run 15 times, and the final results were obtained by calculating the geometric mean of the median of each sub-test. We now present the measurements made with Sun JDK 1.5.0 Client VM, Sun JDK 1.5.0 Server VM, and IBM JDK 1.4.2. We started the Sun JDK 1.5.0 Client VM (respectively, Sun JDK 1.5.0 Server VM) with the '`-client`' (respectively, '`-server`') command line option. We did not use any other JVM-specific option.



Sampling Granularity	compress	jess	db	javac	mpeg-audio	mtrt	jack	geo. mean
500	43.00%	60.94%	33.51%	26.80%	64.91%	77.13%	38.06%	46.19%
1000	35.80%	65.11%	31.81%	20.66%	61.89%	75.99%	31.75%	41.95%
5000	31.23%	62.12%	28.38%	14.43%	54.24%	70.76%	32.19%	37.18%
10000	33.12%	58.42%	25.73%	11.80%	51.47%	65.72%	31.06%	34.79%
50000	23.26%	44.89%	25.66%	4.03%	41.91%	53.41%	20.70%	24.30%
100000	24.05%	37.78%	25.38%	2.95%	38.67%	45.43%	13.06%	20.57%
500000	20.51%	24.12%	19.57%	0.93%	31.42%	31.54%	6.16%	12.76%
1000000	21.12%	19.73%	17.43%	0.61%	28.70%	25.93%	3.96%	10.39%

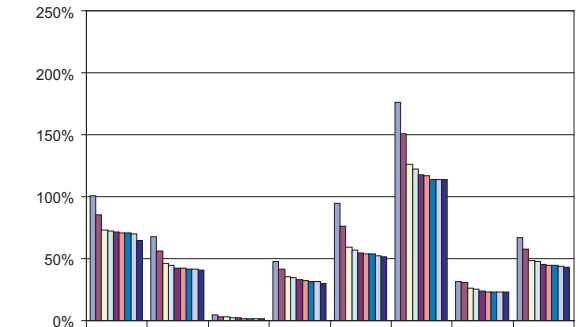
(a)



Sampling Granularity	compress	jess	db	javac	mpeg-audio	mtrt	jack	geo. mean
500+r	47.92%	76.49%	34.55%	39.56%	69.87%	81.86%	86.82%	59.00%
1000+r	46.21%	72.52%	32.23%	32.44%	67.38%	80.31%	76.92%	54.66%
5000+r	40.57%	62.15%	29.14%	16.20%	58.49%	71.01%	47.10%	42.06%
10000+r	35.29%	58.57%	28.62%	11.64%	56.37%	66.94%	38.13%	37.23%
50000+r	33.04%	46.47%	24.63%	4.67%	43.72%	52.13%	20.24%	26.05%
100000+r	24.11%	38.40%	24.09%	2.82%	38.88%	46.04%	13.45%	20.49%
500000+r	21.16%	24.95%	18.77%	0.98%	29.97%	31.51%	6.29%	12.84%
1000000+r	20.48%	19.50%	17.72%	0.60%	28.91%	26.05%	3.73%	10.25%

(b)

Figure 11. Relative profile size (percentage of exact profile size) for different sampling granularities. (a) Constant sampling granularity. (b) Randomized sampling granularity, r is a uniformly distributed random integer ($0 \leq r < 100$).



Sampling Granularity	com-press	jess	db	javac	mpeg-audio	mtrt	jack	geo. mean
500	100.5%	67.4%	4.4%	47.4%	94.3%	176.0%	31.6%	67.1%
1000	85.2%	56.4%	2.9%	41.9%	76.2%	151.1%	30.5%	57.9%
5000	73.2%	45.8%	2.7%	35.2%	59.0%	126.2%	25.9%	48.4%
10000	72.6%	44.7%	2.2%	34.8%	57.1%	122.1%	25.5%	47.4%
50000	71.7%	42.6%	2.0%	33.1%	54.5%	118.1%	23.8%	45.6%
100000	70.8%	42.3%	1.8%	32.3%	53.7%	116.7%	23.4%	45.0%
500000	70.5%	41.6%	1.8%	31.6%	53.5%	114.0%	23.4%	44.4%
1000000	69.6%	41.5%	1.7%	31.2%	52.1%	113.6%	23.4%	44.0%
No Prof.	64.6%	40.5%	1.7%	29.9%	51.8%	113.5%	23.1%	42.9%

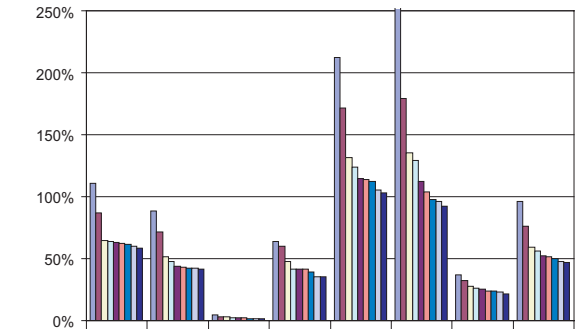
Figure 12. Profiling overhead for different sampling granularities on Sun JDK 1.5.0, Client VM.

Figures 12–14 show the profiling overhead for different sampling granularities on distinct JDKs. We always applied the optimizations of Section 5[‡‡]. For all measurements, *MaxPath* was set to 50. The first eight settings (sampling granularities 500–1 000 000) were measured with the optimized profiling agent presented in Figure 9 (constant sampling granularity). In the last setting 'No Prof.' we did not specify any profiling agent, resulting in the maximum sampling granularity of $2^{31} - 1$ (`Integer.MAX_VALUE`), as can be seen in the `initialize()`, `triggerSampling(MID[], int)`, and `triggerSamplingLeaf(MID[], int, MID)` methods of class `TC` (Figures 2 and 5). Hence, the setting 'No Prof.' does not produce any profiling statistics, but shows the lowest possible overhead that can be obtained with the present program transformation scheme.

As expected, the overhead decreases with increasing sampling granularity, as the profiling agent is invoked less frequently. For the smallest sampling granularity we measured (500), on average the overhead is 67–96% (depending on the JVM). The worst case is the 'mtrt' benchmark with 176–279% overhead*.

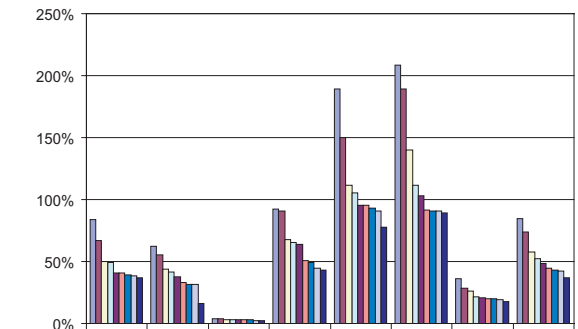
[‡‡]In particular, the leaf method optimization presented in Section 5.1 is essential to keep the profiling overhead low. Without this optimization, the overhead increases by 25% on average, and by up to 165% for the 'mtrt' benchmark. This additional overhead is due to the management of the reified stack in leaf methods and is independent of the sampling granularity.

*If the non-optimized profiling agent of Figure 7 was used with a sampling granularity of 500, the average overhead was 130–140% on Sun's JVMs and 230% on IBM's JVM, while the highest overhead was observed with the 'mpegaudio' benchmark (300–450% for Sun's JVMs and 880% for IBM's JVM). These results underline the importance of the optimization presented in Section 6.2 for low sampling granularities.



Sampling Granularity	com-press	jess	db	javac	mpeg-audio	mtrt	jack	geo. mean
500	111.0%	88.3%	4.8%	63.9%	212.0%	278.9%	36.8%	95.8%
1000	87.0%	71.5%	2.9%	59.8%	171.2%	179.4%	32.6%	76.3%
5000	64.9%	51.5%	2.8%	47.8%	131.2%	135.4%	27.4%	59.5%
10000	63.9%	48.0%	2.7%	41.6%	124.0%	129.0%	26.2%	56.3%
50000	62.8%	43.6%	2.4%	41.2%	114.5%	112.4%	25.1%	52.6%
100000	62.3%	43.1%	2.4%	41.2%	113.5%	104.1%	23.9%	51.3%
500000	61.8%	42.3%	1.7%	39.2%	112.2%	97.8%	23.6%	49.8%
1000000	60.2%	42.2%	1.6%	35.7%	105.7%	95.8%	22.8%	48.0%
No Prof.	58.3%	41.3%	1.5%	35.5%	103.3%	92.0%	21.3%	46.6%

Figure 13. Profiling overhead for different sampling granularities on Sun JDK 1.5.0, Server VM.



Sampling Granularity	com-press	jess	db	javac	mpeg-audio	mtrt	jack	geo. mean
500	83.9%	62.3%	3.9%	92.3%	189.3%	208.4%	36.0%	84.4%
1000	67.2%	55.8%	3.6%	91.0%	150.0%	189.6%	28.6%	73.8%
5000	50.0%	43.7%	3.4%	67.4%	111.6%	140.2%	26.2%	57.4%
10000	49.2%	41.2%	3.3%	65.3%	105.5%	111.9%	21.6%	52.4%
50000	41.1%	37.8%	3.3%	64.0%	95.5%	103.2%	21.1%	48.4%
100000	40.7%	33.0%	3.1%	50.5%	95.0%	91.6%	20.2%	44.3%
500000	39.5%	31.9%	2.9%	48.9%	92.9%	90.9%	20.1%	43.4%
1000000	38.4%	31.6%	2.6%	44.9%	90.7%	90.8%	19.2%	42.2%
No Prof.	36.6%	16.0%	2.1%	43.1%	78.0%	89.3%	17.9%	37.3%

Figure 14. Profiling overhead for different sampling granularities on IBM JDK 1.4.2.

Fortunately, the overhead quickly decreased with increasing sampling granularity. In Section 7.1 we have shown that with low (and constant) sampling granularities, the deterministic sampling mechanism is more vulnerable to spurious phenomena, such as systematically overlooking certain call stacks. A sampling granularity of 5000–10 000 gave accurate results both for constant and randomized sampling granularities. For a sampling granularity of 10 000, the average overhead is 47–56%. A lower bound for the average overhead caused by the present transformation scheme (setting 'No Prof.') is 37–47%.

For different sampling granularities, Figure 15 summarizes the average accuracy, the average relative profile size, as well as the average profiling overheads on Sun's and IBM's JVMs, i.e. Figure 15 compares the 'geo. mean' columns of Figures 10–14. The x-axis is the sampling granularity, the y-axis presents the average accuracy, the average relative profile size, respectively the average overhead. For the overhead, we also included the setting 'No Prof.', which does not produce any profile (hence, neither an average accuracy nor an average relative profile size can be shown for this setting), but shows the asymptotically lowest possible overhead with the present transformation scheme.

Figure 15 confirms that a sampling granularity of 10 000 yields a good tradeoff between high accuracy and reasonable overhead. For this setting, the differences (concerning the average accuracy and the average profile size) between a constant and a randomized sampling granularity are small. Therefore, using a constant sampling granularity, the profiling agent can be kept simple to reduce the overhead caused by its execution. In general, for low sampling granularities, the efficiency of the user-defined profiling agent is essential to prevent excessive overheads. For this reason, we implemented the optimization discussed in Section 6.2. Moreover, in the actual implementation of the profiling agent presented in Section 6, we replaced the standard `java.util.HashMap` class with an optimized map implementation.

The relative overhead due to the sampling profiling is the lowest on Sun's JDK 1.5.0 Client VM and the highest on Sun's JDK 1.5.0 Server VM. However, in absolute time, we found that for each granularity setting we measured, on IBM's JDK 1.4.2 the benchmarks ran about 50% faster than on Sun's JDK 1.5.0 Client VM and about 20% faster than on Sun's JDK 1.5.0 Server VM.

7.5. Comparison with 'hprof' profiling agent

In order to highlight the advantages of the Komorium profiler, we also evaluated the accuracy and the overhead for the 'hprof' profiling agent, which is distributed with many standard JDKs. For all measurements, the hardware and operating system configuration was the same as in Section 7.4. We could not show any results for IBM JDK 1.4.2, as its 'hprof' profiling agent crashed the JVM on our system (exception code: access violation). Therefore, we resorted to Sun JDK 1.4.2 Client VM and Sun JDK 1.4.2 Server VM.

On Sun JDK 1.5.0, we started the profiling agent 'hprof' with the '`-agentlib:hprof=cpu=samples,interval=1,depth=100`' option, which activates the sampling mode with a sampling interval of 1 ms, i.e. a sampling rate of 1000 samples per second (the highest possible sampling rate). We also specified a maximum stack depth of 100 (the default value is four), as some of the benchmarks create rather deep call stacks. Note that our profiling agent presented in Section 6 preserves arbitrarily deep call stacks. On Sun JDK 1.4.2, we used the '`-Xrunhprof:cpu=samples,depth=100`' option; the configuration of the sampling interval was not supported.

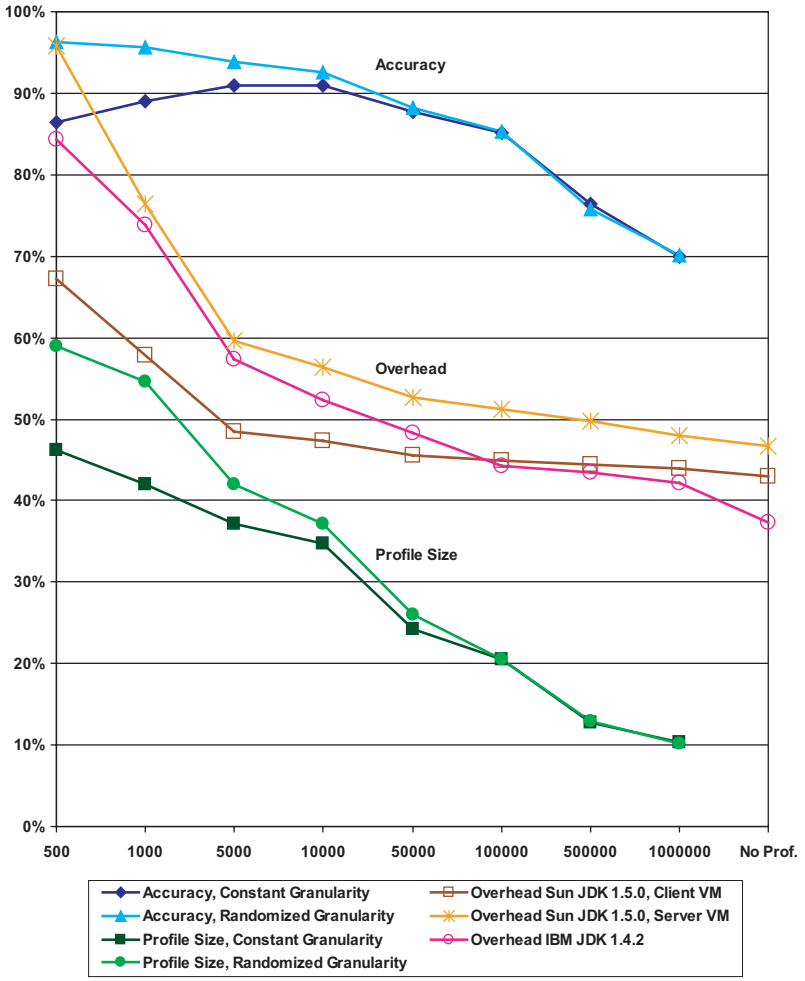


Figure 15. Sampling granularity versus average profile accuracy, average relative profile size, and average profiling overhead.

To assess the accuracy of a profile produced by the 'hprof' sampling profiling agent, we computed again the overlap percentage with a perfect profile. As 'hprof' is timing-based, the perfect profile should reflect the actual CPU time spent in each calling context. We used 'hprof' in its exact profiling mode ('`-agentlib:hprof=cpu=times,depth=100`') to obtain the perfect profiles. Unfortunately, on our machine, the 'hprof' profiling agent of Sun JDK 1.4.2 crashed in its exact mode. As we were not able to generate perfect profiles for Sun JDK 1.4.2 and a comparison with

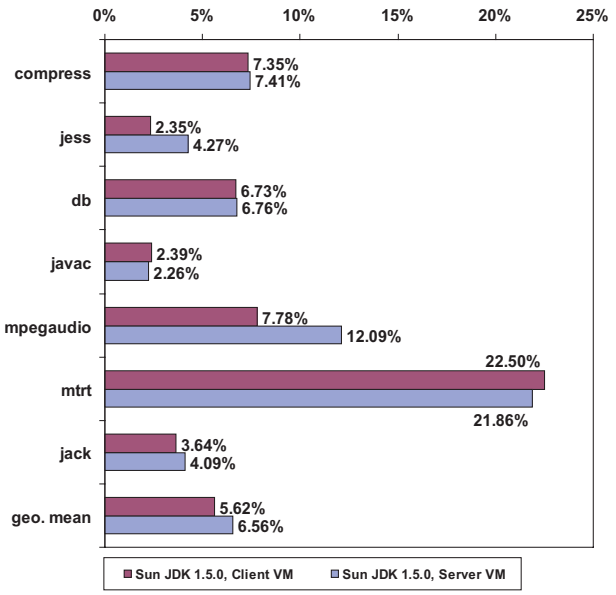


Figure 16. The 'hprof' profiling agent: accuracy of sampling profiles (overlap percentage).

the perfect profiles obtained on Sun JDK 1.5.0 would not be fair, we show the profile accuracy only for Sun JDK 1.5.0.

As illustrated in Figure 16, the achieved accuracy is very low, on average below 7%. This means that the sampling-based profiles generated by 'hprof' are not accurate at all, at least for rather short-running applications such as the SPEC JVM98 benchmarks.

Figure 17 presents the overhead caused by 'hprof'. As the overhead varies very much for the different benchmarks and JVMs, we used a logarithmic scale. On Sun JDK 1.5.0 Client VM, the average overhead (54%) is comparable with the overhead caused by our profiling agent with a sampling granularity of 10 000. For the other JVMs, the average overhead caused by 'hprof' is significantly higher: 89% for the Sun JDK 1.5.0 Server VM and 135–154% for Sun JDK 1.4.2. The average overhead on Sun JDK 1.5.0 Server VM is dominated by the enormous overhead for the 'jack' benchmark (975%). We confirmed this surprisingly high overhead with several extra runs of 'jack' in this setting.

The 'hprof' profiling agent of Sun's JDK 1.5.0 is based on the new JVMTI [5], which is supposed to cause less overhead than the previous JVMPI [4]. This is confirmed by Figure 17, which shows a significantly higher average overhead for the JVMPI-based profiling agent of Sun JDK 1.4.2.

We also computed the average sampling rate (samples per second) of our profiling framework. On our testing machine, a sampling granularity of 1 000 000 corresponds roughly to a sampling rate of 1000 samples per second, comparable to the sampling rate used by the 'hprof' profiling agent on Sun JDK 1.5.0. As we have shown in Section 7.1, a sampling granularity of 1 000 000 is too high to achieve good accuracy. A sampling granularity of 10 000 results in an average sampling rate of

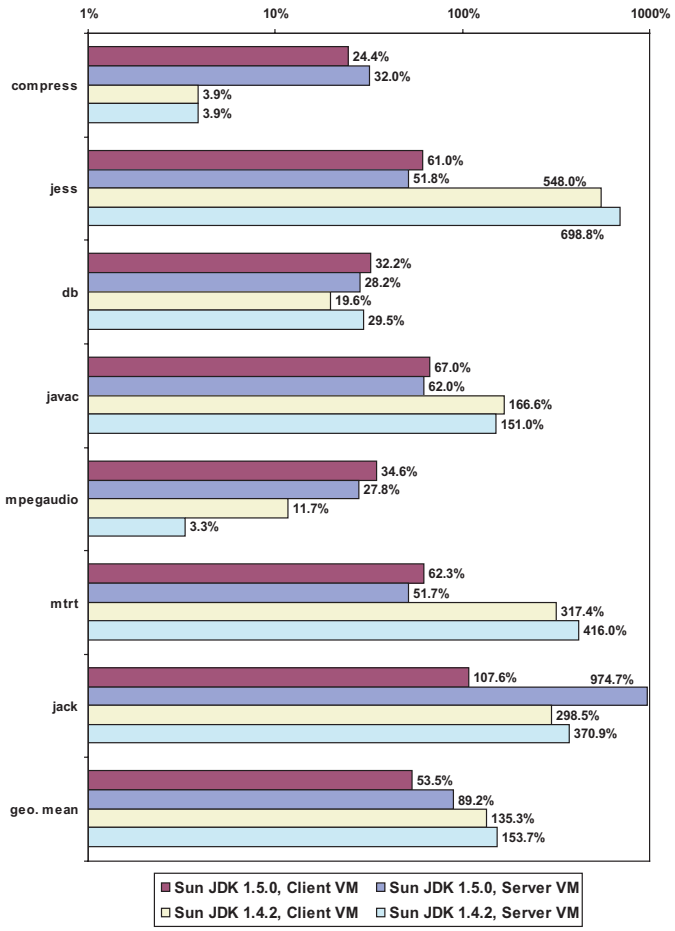


Figure 17. The 'hprof' profiling agent: overhead of sampling profiling.

approximately 80 000 samples per second and offers good accuracy. While 'hprof' with a sampling interval of 1 ms (on Sun JDK 1.5.0) and Komorium with a sampling granularity of 10 000 cause comparable overheads, Komorium achieves a 80 times higher sampling rate on our testing machine.

It can be expected that on a faster machine, 'hprof' with a sampling interval of 1 ms would give even less accurate results, whereas Komorium would produce the same profiles, as the sampling is based on the number of executed bytecodes. The sampling granularity allows the tradeoff between high accuracy and low overhead to be adjusted without considering the characteristics of the target machine. This is an advantage over timing-based sampling, where the sampling interval has to be adjusted to the processing power of the used hardware.

8. DISCUSSION

In this section we discuss the strengths and limitations of our profiling framework.

First and most importantly, our profiling scheme is fully portable. Komorium and all its runtime classes are implemented in pure Java and all program transformations follow a strict adherence to the specification of the Java language and virtual machine. Komorium has been successfully tested with several standard JVMs.

In contrast to profiling agents based on the JVMPI [4] and the JVMTI [5], which are implemented in native code, Komorium enables the implementation of portable profiling agents in pure Java, i.e. a single profiling agent may be deployed in all kinds of Java environments. Moreover, Komorium-based profiling is also applicable to JVMs that support neither the JVMPI nor the JVMTI. Komorium offers a simple but flexible API to implement a wide range of different profiling agents. The profiling agent can control the frequency of its periodic activation by adjusting the sampling granularity. The activation does not rely on the scheduling of the JVM (which is not well specified in the Java language and JVM specifications [1,2]), because each thread in the system synchronously invokes the profiling agent after executing a number of bytecodes corresponding to the current sampling granularity.

Komorium preserves the full method call stack, whereas most other Java profilers only preserve the call stack up to a limited depth. This allows a more detailed analysis of the program behaviour. The custom profiling agent may keep the full call stack of the samples, or process only part of it (e.g. to reduce the memory needed to store the samples or to reduce the processing time).

For deterministic programs, our profiling framework enables reproducible results. For multi-threaded programs without deterministic thread scheduling, the bytecode instrumentation may influence the thread scheduler decisions and consequently affect the execution of concurrent programs. If the task to be accomplished by each thread does not depend on the scheduling, profiles generated upon program termination can be reproducible, whereas continuous metrics [6] may be distorted by changes in the thread scheduling. In order to mitigate this issue, profiling agents could be coded carefully in order not to block threads in unexpected places. For instance, a profiling agent could keep a separate samples tree for each thread in order to avoid a synchronized method call (such as in the example agent presented in Section 6).

As explained in Section 7.1, a randomized sampling granularity helps to improve the accuracy of profiles. The profiling agent may still generate reproducible profiles by keeping a separate pseudo-random number generator for each thread and initializing it with a constant seed. However, exactly reproducible profiles may not always be a requirement, since sampling profiles are approximations anyway.

Compared with our exact profiler JP[†], Komorium significantly reduces the memory requirements: while JP transforms programs so that each thread in the system builds its whole method call tree on the heap, Komorium only reifies the current call stack; in general, the extra memory needed for each thread is constant. Moreover, the size of the generated sampling profiles is much smaller than the size of the corresponding perfect profiles, even for the lowest sampling granularity we measured (500).

[†] See Section 7.1 for more information on the exact profiler JP.

The size of the sampling profiles can be reduced by increasing the sampling granularity, which, however, may also reduce the accuracy of the profiles.

With a sampling granularity of 10 000, Komorium achieves a good tradeoff between high accuracy (an average overlap percentage with perfect profiles of more than 90%) and reasonable overhead of about 47–56% on average (depending on the JVM). In this setting, the worst-case overhead is 112–129% for the 'mtrt' benchmark. In contrast, JP causes an average overhead of 145–225% and a worst-case overhead of 900–1414% for the 'mtrt' benchmark (depending on the JVM). (Still, the overhead caused by JP is one or two orders of magnitude lower than the overheads caused by other exact profilers, such as by the 'hprof' profiling agent in its exact profiling mode.)

Comparing with the 'hprof' profiling agent in its sampling mode, Komorium causes comparable overhead, but generates much more accurate profiles. Moreover, in contrast to 'hprof', Komorium is able to distinguish between overloaded methods (i.e. methods in the same class with the same name, but with different signatures).

Summing up, Komorium is well suited for the profiling of long-running programs, where exact profiling would not be feasible because of the excessive overhead. Consequently, Komorium may become a valuable tool for Java software developers, an alternative to the common 'hprof' profiling agent.

Concerning limitations, the major hurdle of our approach is that it cannot directly account for the execution of native code. For programs that heavily depend on native code, the profiles obtained by Komorium may be incomplete. This is an inherent problem of our approach, since it relies on the transformation of Java code and on the counting of the number of executed bytecodes. A related problem with native code is that Komorium does not provide the full call stack for Java methods invoked by native code.

9. RELATED WORK

Altering Java semantics via bytecode transformations is a well-known technique [20] and has been used for many purposes that can be generally characterized as adding reflection or aspect-orientedness to programs. Working at the bytecode level, the program source code is not needed. There are several tools for manipulating JVM bytecode. For instance, Javassist [21] enables structural reflection and provides convenient source-level abstractions. Some tools for aspect-oriented programming in Java, such as AspectJ [22], also work at the bytecode level. However, such tools usually only support higher-level pointcuts, such as method invocations, whereas bytecode instruction counting requires transformations at the level of basic blocks of code. Therefore, we directly manipulate the JVM bytecode. For the low-level operations, our current implementation is based on the Bytecode Engineering Library BCEL [23].

Fine-grained instrumentation of binary code has been used for profiling in prior work [24,25]. In contrast, all profilers based on a fixed set of events such as that provided by the JVMPI [4] are restricted to traces at the granularity of the method call. This restriction also exists with the current version of Komorium and is justified by the fact that object-oriented Java programs tend to have shorter methods with simpler internal control flows than code implemented in traditional imperative languages.

Some Java profilers take advantage of bytecode instrumentation to be less obtrusive and to enable the JVM to function at full speed. The JVMPI, which has always been declared as an experimental interface

by Sun, but on which many commercial, e.g. JProbe (http://www.quest.com/jprobe/), and academic, e.g. JPMT [26], profilers are based, has been replaced by the JVMTI [5] in JDK 1.5.0. The JVMTI has built-in bytecode instrumentation facilities in order to let profiling agents go beyond the JVMTI API and implement customized, less-disruptive profiler events. Profiling agents based on the JVMTI still have to be written in native code.

JDK 1.5.0 also provides services that allow Java programming language agents to instrument programs running on the JVM. Java agents are specified with the '`-javaagent`' command line option and exploit the instrumentation API (package `java.lang.instrument`) to install bytecode transformations. Java agents are invoked after the JVM has been initialized, before the real application. They may even redefine the already loaded system classes. However, JDK 1.5.0 imposes several restrictions on the redefinition of previously loaded classes. For instance, fields or methods cannot be added, and the method signatures cannot be changed. The present transformation scheme is not compatible with these restrictions. As we want to transform all classes according to the same scheme, the current version of Komorium does not use the instrumentation API.

The NetBeans Profiler (http://profiler.netbeans.org/index.html) integrates Sun's JFluid profiling technology [27] into the NetBeans IDE. JFluid exploits dynamic bytecode instrumentation and code hotswapping in order to turn profiling on and off dynamically, for the whole application or just a subset of it. However, this tool needs a customized JVM and is therefore only available for a limited set of environments.

While Komorium is intended as a profiling tool for Java developers, sampling-based profiling is often used for feedback-directed optimizations in dynamic compilers [15,28], because in such systems the profiling overhead has to be reduced to a minimum in order to improve the overall performance. The framework presented in [15] uses code duplication combined with compiler-inserted, counter-based sampling. A second version of the code is introduced which contains all computationally expensive instrumentation. The original code is minimally instrumented to allow control to transfer in and out of the duplicated code in a fine-grained manner, based on instruction counting. This approach achieves high accuracy and low overhead, as most of the time the slightly instrumented code is executed. Implemented directly within the JVM, the instruction counting causes only minimal overhead. In our case, code duplication would not help, because we implement all transformations at the bytecode level for portability reasons. The bytecode instruction counting itself contributes significantly to the overhead.

Hardware performance counters that record events, such as instructions executed, cycles executed, pipeline stalls, cache misses, etc., are often exploited for profiling. In [14] hardware performance metrics are associated with execution paths. Recently, the Jikes RVM [29], an open-source research virtual machine that offers a flexible testbed for prototyping virtual machine technology, has been enhanced to generate traces of hardware performance monitor values for each thread in the system [30]. In [31] the authors introduce 'vertical profiling', which combines hardware and software performance monitors in order to improve the understanding of system behaviour by correlating profile information from different levels. All these approaches aim at generating precise profiling information for a particular environment, with a focus on improving virtual machine implementations. In contrast, Komorium is a developer tool that helps in program analysis. Komorium does not rely on any platform-specific features in order to offer a completely portable profiling system that allows developers to profile their applications in their preferred environment, generating reproducible and directly comparable profiles (for deterministic programs).

In [10] the authors show that profiles based on bytecode instruction counting are valuable to detect algorithmic inefficiencies and help the developer to focus on those parts of a program that suffer from high algorithmic complexity. However, in [10], a simple, exact profiler is used, which causes excessive overhead. Moreover, aspects concerning multi-threading and native code are not addressed.

Much of the know-how worked into Komorium comes from our previous experience gained with the Java Resource Accounting Framework, Second Edition (J-RAF2) [8,9], which also uses bytecode instrumentation in order to gather dynamic information about a running application. J-RAF2 maintains a bytecode instruction counter for each thread, comparable to the `instrCounter` in class `TC` (see Figure 2). When the number of executed bytecodes exceeds a given threshold, a resource manager is invoked which implements a user-defined accounting or control policy. We also experimented with a sampling profiler based on J-RAF2 (which used the `Throwable` API to obtain stack traces), but the resulting accuracy was disappointing, because the rules for bytecode instruction counting and resource manager activation follow a different objective: J-RAF2 computes an upper bound of the number of bytecodes executed by the transformed program and inserts granularity checks as sparingly as possible in order to reduce the accounting overhead. Moreover, the `Throwable` API has several limitations as described in Section 4.1 and its use caused high overhead for low sampling granularities.

10. FUTURE WORK

Bytecode instruction counting and CPU time are distinct metrics for different purposes. Profiles based on bytecode instruction counting are platform-independent, reproducible, directly comparable across different environments, and valuable to gain insight into algorithm complexity. Nonetheless, the value of our profiling tools would further increase if we could use profiles based on bytecode instruction counting to accurately estimate CPU time on a particular target system. This would enable a new way of cross-profiling. The developer could profile an application on his preferred platform P_{develop}, providing the profiler some configuration information concerning the intended target platform P_{target}. The profile obtained on P_{develop} would allow the developer to approximate a CPU time-based profile on P_{target}.

For this purpose, individual (sequences of) bytecode instructions may receive different weights according to their complexity. This weighting is specific to a particular execution environment (hardware and JVM) and can be generated by a calibration mechanism. However, the presence of garbage collection and dynamic compilation may limit the achievable accuracy. Therefore, we focus first on simple JVM implementations (interpreters), such as JVMs for embedded systems, which do not involve complex optimization and re-compilation phases. Next, we will consider JVMs with a deterministic compilation scheme, such as CACAO (http://www.complang.tuwien.ac.at/cacaojvm/) [32], and estimate the number of (native) machine instructions the just-in-time compiler would generate for individual basic blocks. Related work on performance prediction for Java programs shows that such an approach is feasible [33].

We are also exploring the extent to which our profiling approach is applicable to other virtual machines, the most obvious challenger being the *.NET* platform.

Moreover, we are working on more sophisticated profiling agents to profile long-running programs, such as application servers, continuously providing the user with up-to-date execution statistics (continuous metrics [6]).

11. CONCLUSION

In this article we presented a novel sampling-based profiling framework for Java which makes extensive use of program transformations in order to overcome many limitations of prevailing Java profilers. Our framework does not rely on native code, it may be used with any JVM. It offers a flexible API to implement custom profiling agents in pure Java.

The periodic sampling is based on bytecode instruction counting, i.e. the periodic invocation of the profiling agent depends on the number of executed bytecodes and not on an external timer. Therefore, for deterministic programs, the resulting profiles are reproducible and do not depend on parameters such as processor speed and system load. In this article we have shown that sampling based on bytecode instruction counting offers a good tradeoff between high accuracy of profiles and reasonable overhead.

ACKNOWLEDGEMENT

Thanks to Jarle Hulaas who evaluated the overhead caused by the exact profiler JP.

REFERENCES

1. Gosling J, Joy B, Steele GL, Bracha G. *The Java Language Specification* (*Java Series*) (2nd edn). Addison-Wesley: Reading, MA, 2000.
2. Lindholm T, Yellin F. *The Java Virtual Machine Specification* (2nd edn). Addison-Wesley: Reading, MA, 1999.
3. Liang S, Viswanathan D. Comprehensive profiling support in the Java virtual machine. *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS-99)*, 3–7 May 1999. USENIX Association: Berkeley, CA, 1999; 229–240.
4. Sun Microsystems, Inc. Java Virtual Machine Profiler Interface (JVMPI). http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/ [9 August 2005].
5. Sun Microsystems, Inc. JVM Tool Interface (JVMTI). http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/ [9 August 2005].
6. Dufour B, Driesen K, Hendren L, Verbrugge C. Dynamic metrics for Java. *ACM SIGPLAN Notices* 2003; **38**(11):149–168.
7. Binder W, Hulaas JG, Villazón A. Portable resource control in Java. *Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01)*, November 2001. *ACM SIGPLAN Notices* 2001; **36**(11):139–155.
8. Hulaas J, Binder W. Program transformations for portable CPU accounting and control in Java. *Proceedings of PEPM'04 (2004 ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation)*, Verona, Italy, 24–25 August 2004. ACM Press: New York, 2004; 169–177.
9. Binder W, Hulaas J. A portable CPU-management framework for Java. *IEEE Internet Computing* 2004; **8**(5):74–83.
10. Cooper BF, Lee HB, Zorn BG. ProfBuilder: A package for rapidly building Java execution profilers. *Technical Report CU-CS-853-98*, University of Colorado at Boulder, Department of Computer Science, April 1998.
11. Choi J-D, Grove D, Hind M, Sarkar V. Efficient and precise modeling of exceptions for the analysis of Java programs. *Proceedings of the ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 1999. ACM Press: New York, 1999; 21–31.
12. The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/ [9 August 2005].
13. Binder W, Hulaas J. Exact and portable profiling for Java using bytecode instruction counting. *Technical Report EPFL-IC-2005011*, Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, March 2005.
14. Ammons G, Ball T, Larus JR. Exploiting hardware performance counters with flow and context sensitive profiling. *PLDI'97: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*. ACM Press: New York, 1997; 85–96.
15. Arnold M, Ryder BG. A framework for reducing the cost of instrumented code. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. *ACM SIGPLAN Notices* 2001; **36**(5):168–179.

16. Feller P. Value profiling for instructions and memory locations. *Master Thesis CS1998-581*, University of California, San Diego, CA, April 1998.
17. Anderson J *et al.* Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems* 1997; **15**(4):357–390.
18. Cierniak M, Lueh G-Y, Stichnoth JM. Practicing JUDO: Java under dynamic optimizations. *ACM SIGPLAN Notices* 2000; **35**(5):13–26.
19. Ogasawara T, Komatsu H, Nakatani T. A study of exception handling and its dynamic optimization in Java. *Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01)*, November 2001. *ACM SIGPLAN Notices* 2001; **36**(11):83–95.
20. Tanter E, Ségura-Devillechaise M, Noyé J, Piquer J. Altering Java semantics via bytecode manipulation. *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, October 2002 (*Lecture Notes in Computer Science*, vol. 2487). Springer: Berlin, 2002.
21. Chiba S. Load-time structural reflection in Java. *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, Cannes, France, June 2000 (*Lecture Notes in Computer Science*, vol. 1850). Springer: Berlin, 2000; 313–336.
22. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An overview of AspectJ. *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP-2001)* (*Lecture Notes in Computer Science*, vol. 2072), Knudsen JL (ed.). Springer: Berlin, 2001; 327–353.
23. Dahm M. Byte code engineering. *Java-Information-Tage 1999 (JIT'99)*, September 1999. Available at: http://jakarta.apache.org/bcel/ [9 August 2005].
24. Ball T, Larus JR. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems* 1994; **16**(4):1319–1360.
25. Larus JR, Ball T. Rewriting executable files to measure program behavior. *Software—Practice and Experience* 1994; **24**(2):197–218.
26. Harkema M, Quartel D, Gijsen BMM, van der Mei R. Performance monitoring of Java applications. *Proceedings of the 3rd International Workshop on Software and Performance (WOSP-02)*, New York, July 2002. ACM Press: New York, 2002; 114–127.
27. Dmitriev M. Profiling Java applications using code hotswapping and dynamic call graph revelation. *WOSP'04: Proceedings of the 4th International Workshop on Software and Performance*. ACM Press: New York, 2004; 139–150.
28. Whaley J. A portable sampling-based profiler for Java Virtual Machines. *Proceedings of the ACM 2000 Conference on Java Grande*, June 2000. ACM Press: New York, 2000; 78–87.
29. Alpern B *et al*. The Jalapeño virtual machine. *IBM Systems Journal* 2000; **39**(1):211–238.
30. Sweeney PF, Hauswirth M, Cahoon B, Cheng P, Diwan A, Grove D, Hind M. Using hardware performance monitors to understand the behavior of Java applications. *Virtual Machine Research and Technology Symposium*. USENIX Association: Berkeley, CA, 2004; 57–72.
31. Hauswirth M, Sweeney PF, Diwan A, Hind M. Vertical profiling: Understanding the behavior of object-oriented applications. *OOPSLA'04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM Press: New York, 2004; 251–269.
32. Krall A. Efficient JavaVM just-in-time compilation. *International Conference on Parallel Architectures and Compilation Techniques*, Paris, October 1998, Gaudiot J-L (ed.). IFIP/ACM/IEEE: North-Holland, 1998; 205–212.
33. Turner JD. A dynamic prediction and monitoring framework for distributed applications. *PhD Thesis*, Department of Computer Science, University of Warwick, May 2003.

