# A Portable and Customizable Profiling Framework for Java Based on Bytecode Instruction Counting

Walter Binder

Ecole Polytechnique Fédérale de Lausanne (EPFL),
Artificial Intelligence Laboratory,
CH-1015 Lausanne, Switzerland
walter.binder@epfl.ch

**Abstract.** Prevailing profilers for Java, which rely on standard, native-code profiling interfaces, are not portable, give imprecise results due to serious measurement perturbation, and cause excessive overheads. In contrast, program transformations allow to generate reproducible profiles in a fully portable way with significantly less overhead. This paper presents a profiling framework that instruments Java programs at the bytecode level to build context-sensitive execution profiles at runtime. The profiling framework includes an exact profiler as well as a sampling profiler. User-defined profiling agents can be written in pure Java, too, in order to customize the runtime processing of profiling data.

**Keywords:** Profiling, program transformations, bytecode instrumentation, dynamic metrics, Java, JVM

## 1 Introduction

Most prevailing Java profilers rely on the Java Virtual Machine Profiling Interface (JVMPI) [14,15] or on the JVM Tool Interface (JVMTI) [16], which provide a set of hooks to the Java Virtual Machine (JVM) to signal interesting events, such as thread start and object allocations. Usually, such profilers 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 current stack trace.

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 platform-dependent native code, contradicting the Java motto 'write once and run anywhere'. Because exact profiling based on these APIs may cause an enormous overhead (in extreme cases we even experienced a slowdown of factor 4 000 and more), developers frequently resort to more efficient sampling-based profilers to analyze the performance of complex system, such as application servers. Many prevailing sampling profilers for Java use an external timer to trigger the sampling, resulting in 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. In short, most prevailing

K. Yi (Ed.): APLAS 2005, LNCS 3780, pp. 178–194, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Java profilers are either too slow or too imprecise to generate meaningful profiles for complex systems.

To solve these problems, we developed a novel profiling framework that relies neither on the JVMPI nor on the JVMTI, but directly instruments the bytecode of Java programs in order to create a profile at runtime. Our framework includes the exact profiler JP as well as the sampling profiler Komorium. Both profilers exploit the number of executed bytecodes[1] as platform-independent profiling metric, enabling reproducible profiles and a fully portable implementation of the profilers. Moreover, user-defined profiling agents written in pure Java may customize the profile generation.

As contributions, this paper introduces bytecode counting as profiling metric and defines an innovative profiling framework that is completely based on program transformations. Two profiling schemes are presented, one for exact profiling and one for sampling profiling. We implemented and evaluated both of them: The exact profiler JP causes 1–2 orders of magnitude less overhead than prevailing exact profilers. The sampling profiler Komorium causes less overhead than existing sampling profilers, and the resulting profiles are much more accurate.

The remainder of this paper is structured as follows: Section 2 argues for bytecode counting as profiling metric. Section 3 introduces the data structures on which our profiling framework is based. In Section 4 we explain the program transformation scheme underlying the exact profiler JP. Section 5 discusses the sampling profiler Komorium. In Section 6 we evaluate the performance of our profilers, as well as the accuracy of the profiles generated by the sampling profiler. Section 7 summarizes the limitations of our approach and outlines some ideas for future improvements. Finally, Section 8 discusses related work and Section 9 concludes this paper.

## 2 Bytecode Counting

Most existing profilers measure the CPU consumption of programs in seconds. Although the CPU second is the most common profiling metric, it has several drawbacks: It is platform-dependent (for the same program and input, the CPU time differs depending on hardware, operating system, and JVM), measuring it accurately may require platform-specific features (such as special operating system functions) limiting the portability of the profilers, and results may not be easily reproducible (the CPU time may depend on factors such as system load). Furthermore, measurement perturbation is often a serious problem: The measured CPU consumption of the profiled program may significantly differ from the effective CPU consumption when the program is executed without profiling. The last point is particularly true on JVMs where the use of the JVMPI disables just-in-time compilation.

For these reasons, we follow a different approach, using the number of executed bytecodes as profiling metric, which has the following benefits:

– **Platform-independ profiles:** The number of executed bytecodes is a platform-independent metric [10]. Although the CPU time of a deterministic program with a given input varies very much depending on the performance of the underlying

[1] In this paper ‘bytecode’ stands for ‘JVM bytecode instruction’.

hardware and virtual machine (e.g., interpretation versus just-in-time compilation), the number of bytecodes issued by the program remains the same, independent of hardware and virtual machine implementation (assuming the same Java class library is used).

- **Reproducible profiles:** For deterministic programs, the generated profiles are completely reproducible.
- **Comparable profiles:** Profiles collected in different environments are directly comparable, since they are based on the same platform-independent metric.
- **Accurate profiles:** The profile reflects the number of bytecodes that a program would execute without profiling, i.e., the profiling itself does not affect the generated profile (no measurement perturbation).
- **Portable and compatible profiling scheme:** Because counting the number of executed bytecodes does not require any hardware- or operating system-specific support, it can be implemented in a fully portable way. The profiling scheme is compatible also with JVMs that support neither the JVMPI nor the JVMTI, or that provide limited support for profiling in general.
- **Portable profiling agents:** Custom profiling agents can be written in pure Java and are better integrated with the environment. Hence, profiling agents are portable and can be used in all kinds of JVMs.
- **Flexible profiling agents**: Profiling agents can be programmed to preserve a trace of the full call stack, or to compact it at certain intervals, whereas existing profilers frequently only support a fixed maximal stack depth.
- **Fine-grained control of profiling agent activation:** Profiling agents are invoked in a deterministic way by each thread after the execution of a certain number of bytecodes, which we call the *profiling granularity*. Profiling agents can dynamically adjust the profiling granularity in a fine-grained way.
- **Reduced overhead:** The overhead is rather low compared to classical approaches, since it does not prevent the underlying JVM from putting all its optimization facilities to work during the profiling.

Consequently, bytecode counting is key to the provision of a new class of portable, platform-independent profiling tools, which gives advantages to the tool users as well as to the tool implementors:

On the one hand, bytecode counting eases profiling, because thanks to the platform-independence of this metric [10], the concrete environment is not of importance. Thus, the developer may profile programs in the environment of his preference. Since 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.

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

- `createMID(STRING` *class*`, STRING` *name*`, STRING` *sig*`): MID`
  Creates a new method identifier, consisting of class name, method name, and method signature.
- `getName(MID` *mid*`): STRING`
  Returns the method name of *mid*.
  `getName(createMID(`*x*`,` *n*`,` *y*`))` = *n*.
- `getClass(MID` *mid*`): STRING`
  Returns the class name of *mid*.
  `getClass(createMID(`*c*`,` *x*`,` *y*`))` = *c*.
- `getSig(MID` *mid*`): STRING`
  Returns the method signature of *mid*.
  `getSig(createMID(`*x*`,` *y*`,` *s*`))` = *s*.

**Fig. 1.** Method identifier `MID`

- `getOrCreateRoot(THREAD` *t*`): IC`
  Returns the root node of a thread's MCT. If it does not exist, it is created.
- `profileCall(IC` *caller*`, MID` *callee*`): IC`
  Registers a method invocation in the MCT. The returned *IC* represents the callee method, identified by *callee*. It is a child node of *caller* in the MCT.
- `getCaller(IC` *callee*`): IC`
  Returns the caller `IC` of *callee*. It is the parent node of *callee* in the MCT.
  `getCaller(profileCall(`*c*`,` *x*`))` = *c*.
  This operation is not defined for the root of the MCT.
- `getCalls(IC` *c*`): INT`
  Returns the number of invocations of the method identified by `getMID(`*c*`)` with the caller `getCaller(`*c*`)`.
  `getCalls(profileCall(`*x*`,` *y*`))` $\geq$ 1.
  This operation is not defined for the root of the MCT.
- `getMID(IC` *c*`): MID`
  Returns the method identifier associated with *c*.
  `getMID(profileCall(`*x*`,` *callee*`))` = *callee*.
  This operation is not defined for the root of the MCT.
- `getCallees(IC` *c*`): SET OF IC`
  Returns the set of callee `IC`s of *c*.
  $\forall x \in$ `getCallees(`*c*`):getCaller(`*x*`)` = *c*.
  $\forall x \in$ `getCallees(`*c*`):getCalls(`*x*`)` $\geq$ 1.
- `profileInstr(IC` *ic*`, INT` *bytecodes*`): IC`
  Registers the execution of a certain number of bytecodes in *ic*. The bytecode counter in *ic* is incremented by *bytecodes*. Returns *ic*, after its bytecode counter has been updated. This operation is not defined for the root of the MCT.
- `getInstr(IC` *ic*`): INT`
  Returns the number of bytecodes executed in *ic*.
  `getInstr(profileInstr(`*x*`,` *b*`))` $\geq$ *b*.
  This operation is not defined for the root of the MCT.

**Fig. 2.** Method invocation context `IC`

## 3 Profiling Data Structures

In this section we define the data structures used by our profiling framework as abstract datatypes. A detailed presentation of the Java implementation of these data structures had to be omitted due to space limitations.

### 3.1 Method Call Tree (MCT)

For exact profiling, we rewrite JVM bytecode in order to create a Method Call Tree (MCT), where each node represents all invocations of a particular method with the same call stack. The parent node in the MCT corresponds to the caller, the children nodes correspond to the callees. The root of the MCT represents the caller of the main method. With the exception of the root node, each node in the MCT stores profiling information for all invocations of the corresponding method with the same call stack. Concretely, it stores the number of method invocations as well as the number of bytecodes executed in the corresponding calling context, excluding the number of bytecodes executed by callee methods (each callee has its own node in the MCT).

- `getOrCreateAC(THREAD` $t$`): AC`
  Returns the activation counter of a thread. If it does not exist, it is created.
- `setValue(AC` $ac$`, INT` $v$`): AC`
  Returns $ac$, after its value has been updated to $v$.
- `getValue(AC` $ac$`): INT`
  Returns the value of $ac$.
  `getValue(setValue(`$x$`,` $v$`)) =` $v$.

**Fig. 3.** Activation counter `AC`

In order to prevent race conditions, either access to the MCT has to be synchronized, or each thread has to maintain its own copy of the tree. To avoid expensive synchronization and to allow profiling agents to keep the profiling statistics of different threads separately, we chose to create a separate MCT for each thread in the system.[2]

The MCT is similar to the Calling Context Tree (CCT) [1]. However, in contrast to the CCT, the depth of the MCT is unbounded. Therefore, the MCT may consume a significant amount of memory in the case of very deep recursions. Nonetheless, for most programs this is not a problem: According to Ball and Larus [5], path profiling (i.e., preserving exact execution history) is feasible for a large portion of programs.

We define two abstract datatypes to represent a MCT, the method identifier `MID` (see Fig. 1) and the method invocation context `IC` (see Fig. 2). A method invocation context is a node in the MCT, encapsulating a method invocation counter and a bytecode counter. We assume the existence of the types `INT`, `STRING`, and `THREAD`, as well as the possibility to create aggregate types (`SET OF`).

### 3.2 Activation Counter

In order to schedule the regular activation of a user-defined profiling agent in a platform-independent way, our profilers maintain a counter of the (approximate) number of executed bytecodes for each thread. If this counter exceeds the current profiling granularity, the profiling agent is invoked in order to process the collected execution statistics. The abstract datatype `AC` (see Fig. 3) represents an activation counter for each thread.

## 4 Exact Profiling

In this section we describe a fully portable scheme for exact profiling. In order to validate our approach, we implemented the exact profiler JP, which is compatible with standard JVMs. JP relies neither on the JVMPI nor on the JVMTI, but directly instruments the bytecode of Java programs in order to obtain detailed execution statistics.

In Section 4.1 we explain how programs are transformed to create MCTs at runtime. While Section 4.2 discusses the necessary code instrumentation to maintain the bytecode counters within the MCTs, Section 4.3 explicates the periodic activation of a custom profiling agent. Finally, in Section 4.4 we illustrate the program transformations with an example.

[2] At the implementation level, a thread-local variable may be used to store a reference to the root of a thread's MCT. Each thread gets its own instance of the thread-local variable. In Java, thread-local variables are instances of `java.lang.ThreadLocal`.

### 4.1 MCT Creation

JP rewrites JVM bytecode in order to pass the method invocation context $ic_{caller}$ (type `IC`) of the caller as an extra argument to the callee method (i.e., JP extends the signatures of all non-native methods with the additional argument). In the beginning of a method[3] identified by $mid_{callee}$ (type `MID`), the callee executes a statement corresponding to

$ic_{callee}$ = `profileCall(`$ic_{caller}$`,` $mid_{callee}$`);`

in order to obtain its own (i.e., the callee's) method invocation context $ic_{callee}$.

Because native code is not changed by the rewriting, JP adds simple wrapper methods with the unmodified signatures which obtain the current thread's MCT root by calling `getOrCreateRoot(`$t$`)`, where $t$ represents the current thread. Therefore, native code is able to invoke Java methods with the unmodified signatures.[4]

For each Java method, we add a static field to hold the corresponding method identifier. In the static initializer we call `createMID(`$classname$`,` $methodname$`,` $signature$`)` in order to allocate a method identifier for each Java method.

### 4.2 Bytecode Counting

For each method invocation context $ic$, JP computes the number of executed bytecodes. JP instruments the bytecode of methods in order to invoke `profileInstr(`$ic$`,` $bytecodes$`)` according to the number of executed bytecodes. For each Java method, JP performs a basic block analysis (BBA) to compute a control flow graph. In the beginning of each basic block it inserts a code sequence that implements this update of the bytecode counter.

The BBA algorithm is not hard-coded in JP, via a system property the user can specify a custom analysis algorithm. JP itself offers two built-in BBA algorithms, which we call 'Default BBA' resp. 'Precise BBA'. In the 'Default BBA', 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 or JVM subroutine invocations do not end basic blocks of code, because we assume that the execution will return after the call. This definition of basic block corresponds to the one used in [7] and is related to the factored control flow graph [8].

The advantage of the 'Default BBA' is that it creates rather large basic blocks. Therefore, the number of locations is reduced where updates to the bytecode counter have to be inserted, resulting in a lower profiling overhead. As long as no exceptions are thrown, the resulting profiling information is precise. However, exceptions (e.g., an invoked method may terminate abnormally throwing an exception) may cause some imprecision in the accounting, as we always count all bytecodes in a basic block, even

[3] In this paper we do not distinguish between Java methods and constructors, i.e., 'method' stands for 'method or constructor'.

[4] For native methods, which we cannot rewrite, we add so-called 'reverse' wrappers which discard the extra `IC` argument before invoking the native method. The 'reverse' wrappers allow rewritten code to invoke all methods with the additional argument, no matter whether the callee is native or not.

- `register(THREAD` $t$`, IC` $root$`): INT`
  This operation is invoked whenever a new thread $t$ is created. It is called by `getOrCreateRoot(`$t$`)`, if a new MCT root node ($root$) has been allocated. For each thread, this operation is invoked only once, when it starts executing instrumented code (a wrapper method as discussed in Section 4.1). After `register(`$t$`,` $root$`)` has been called, the profiling agent must be prepared to handle subsequent invocations of `processMCT(IC)` by the thread $t$.
  `register(`$t$`,` $root$`)` returns the current profiling granularity for $t$, i.e., the approximate number of bytecodes to execute until $t$ will invoke `processMCT(IC)` for the first time.

- `processMCT(IC` $ic$`): INT`
  This operation is periodically invoked by each thread in the system. Whenever `processMCT(`$ic$`)` is called, the profiling agent has to process the current thread's MCT. $ic$ is the method invocation context corresponding to the method that is currently being executed. The profiling agent may obtain the root of the current thread's MCT either from a map (to be updated upon invocations of `register(THREAD, IC))` or by successively applying `getCaller(IC).processMCT(IC)` allows the profiling agent to integrate the MCTs of the different threads into a global MCT, or to generate continuous metrics [10], which is particularly useful to display up-to-date profiling information of long running programs, such as application servers.
  `processMCT(IC)` returns the current profiling granularity for the calling thread, i.e., the approximate number of bytecodes to execute until the current thread will invoke `processMCT(IC)` again.

**Fig. 4.** `ExactProfiler`

though some of them may not be executed in case of an exception. I.e., using the 'Default BBA', we may count more bytecodes than are executed.

If the user wants to avoid this potential imprecision, he may select the 'Precise BBA', which ends a basic block after each bytecode that either may change the control flow non-sequentially (as before), or may throw an exception. As there are many bytecodes that may throw an exception (e.g., `NullPointerException` may be raised by most bytecodes that require an object reference), the resulting average basic block size is smaller. This inevitably results in a higher overhead for bytecode counting, because each basic block is instrumened by JP.

### 4.3 Periodic Activation of Custom Profiling Agents

JP supports user-defined profiling agents which are periodically invoked by each thread in order to aggregate and process the MCT collected by the thread. The custom profiling agent has to implement the abstract datatype `ExactProfiler` (see Fig. 4).

Each thread maintains an activation counter $ac$ (type `AC`) in order to schedule the regular activation of the custom profiling agent. The value of $ac$ is an upper bound of the number of executed bytecodes since the last invocation of `processMCT(IC)`. In order to make $ac$ directly accessible within each method, we pass it as an additional argument to all invocations of non-native methods. If the value of $ac$ exceeds the profiling granularity, the thread calls `processMCT(IC)` of the profiling agent. Note that the value of $ac$ is not part of the profiling statistics, it is only used at runtime to ensure the periodic activation of the profiling agent.

The value of $ac$ runs from the profiling granularity down to zero, because there are dedicated bytecodes for the comparison with zero. I.e., the following conditional is used to schedule the periodic activation of the profiling agent and to reset $ac$ ($ic$ refers to the current method invocation context):

```
if (getValue(ac) <= 0) setValue(ac, processMCT(ic));
```

The updates of $ac$ are correlated to the updates of the bytecode counters within the MCT (`profileInstr(IC, INT)`). However, in order to reduce the overhead, the value of $ac$ is not updated in every basic block of code, but only in the beginning of each method, exception handler, and JVM subroutine, as well as in the beginning of each loop. Each time it is decremented by the number of bytecodes on the longest execution path until the next update or until the method terminates. This ensures that the value of $ac$ is an upper bound of the number of executed bytecodes.

The conditional that checks whether `processMCT(IC)` has to be called is inserted in the beginning of each method and in each loop, in order to ensure its presence in recursions and iteration. As an optimization, we omit the conditional in the beginning of a method, if before invoking any method, each execution path either terminates or passes by an otherwise inserted conditional. For instance, this optimization allows to remove the check in the beginning of leaf methods.

### 4.4 Rewriting Example

The example in Fig. 5 illustrates the program transformations performed by JP: To the left is the class `Foo` with the method `sum(int, int)` before rewriting, to the right is the rewritten version.[5] `sum(int, int)` computes the following mathematical function: $sum(a, b) = \sum_{i=a}^{b} f(i)$. The method `int f(int)`, which is not shown in Fig. 5, is transformed in a similar way as `sum(int, int)`. In `sum(int, int)` we use an infinite `while()` 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.

For this example, we used the 'Default BBA' introduced in Section 4.2. `sum(int, int)` has 4 basic blocks of code: The first one (2 bytecodes) initializes the local variable `result` with zero, the second one (3 bytecodes) compares the values of the local variables `from` and `to` and branches, the third one (2 bytecodes) returns the value of the local variable `result`, and the fourth block (7 bytecodes) adds the return value of `f(from)` to the local variable `result`, increments the local variable `from`, and jumps to the begin of the loop.

In the rewritten code, the static initializer allocates the method identifier `mid_sum` to represent invocations of `sum(int, int)` in the MCT. The rewritten method receives 2 extra arguments, the activation counter (type `AC`) and the caller's method invocation context (type `IC`). First, the rewritten method updates the MCT and obtains its own (the callee's) method invocation context (`profileCall(IC, MID)`). The bytecode counter within the callee's method invocation context is incremented in the beginning of each basic block of code by the number of bytecodes in the block (`profileInstr(IC, INT)`).

The activation counter `ac` is updated in the beginning of the method and in the loop. It is reduced by the number of bytecodes on the longest execution path until the next

[5] For the sake of better readability, in this paper we show all transformations on Java-based pseudo-code, whereas our profiler implementations work at the JVM bytecode level. The operations on the abstract datatypes `MID`, `IC`, `AC`, and `ExactProfiler` are directly inlined in order to simplify the presentation.

```
class Foo {

   static int sum(int from,
                  int to) {

      int result = 0;
      while (true) {

         if (from > to) {

            return result;
         }

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

}
```

```
class Foo {
   private static final MID mid_sum;
   static {
      String cl = Class.forName("Foo").getName();
      mid_sum   = createMID(cl, "sum", "(II)I");
   }

   static int sum(int from,
                  int to, AC ac, IC ic) {
      ic = profileCall(ic, mid_sum);
      profileInstr(ic, 2);
      setValue(ac, getValue(ac) - 2);
      int result = 0;
      while (true) {
         profileInstr(ic, 3);
         setValue(ac, getValue(ac) - 10);
         if (getValue(ac) <= 0)
            setValue(ac, processMCT(ic));
         if (from > to) {
            profileInstr(ic, 2);
            return result;
         }
         profileInstr(ic, 7);
         result += f(from, ac, ic);
         ++from;
      }
   }

   static int sum(int from, int to) {
      Thread t = Thread.currentThread();
      return sum(from, to, getOrCreateAC(t),
                 getOrCreateRoot(t));
   }
}
```

**Fig. 5.** Rewriting example: Program transformations for exact profiling

update or method termination. For instance, in the loop it is incremented by 10 $(3 + 7)$, as this is the length of the execution path if the loop is repeated. The other path, which returns, executes only 5 bytecodes $(3 + 2)$. The conditional is present in the loop, but not in the beginning of the method, since the only possible execution path passes by the conditional in the loop before invoking any method.

A wrapper method with the unmodified signature is added to allow native code, which is not aware of the additional arguments, to invoke the rewritten method. The wrapper method obtains the current thread's activation counter as well as the root of its MCT before invoking the instrumented method with the extra arguments.

## 5 Sampling-Based Profiling

Even though exact profiling based on the program transformation scheme presented in Section 4 causes considerably less overhead than prevailing exact profilers, the overhead may still be too high for complex applications. Moreover, for applications with many concurrent threads, maintaining a separate MCT for each thread may consume a large amount of memory. For these reasons, we developed the sampling profiler Komorium, which is also based on program instrumentation. Komorium relies on the periodic activation of a user-defined profiling agent to process samples of the call stack.

Sampling profiles are different from exact ones, since they expose neither the absolute number of method invocations nor the absolute number of executed bytecodes. However, sampling profiles can be used to estimate the relative distribution of processing effort, guiding the developer in which parts program optimizations may pay off. 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., $\frac{\text{number of occurrences of } S}{\text{total number of samples}}$) approximates the proportion of processing spent in the corresponding calling context.

### 5.1 Call Stack Reification

In order to make the call stack available at execution time, Komorium reifies the current call stack as a pair $\langle mids, sp \rangle$, where $mids$ is an array of method identifiers (type `MID[]`) and $sp$ is the stack pointer (type `INT`), which denotes the next free element on the reified stack. $mids[i]$ are the identifiers of the activated methods ($0 \leq i < sp$). $mids[0]$ is the bottom of the reified stack and $mids[sp-1]$ is its top, corresponding to the currently executing method.

Komorium transforms programs in order to pass the reified call stack of the caller as 2 extra arguments to the callee method (i.e., Komorium extends the signatures of all non-native methods with the additional arguments). In the beginning of a method identified by $mid$, the callee executes a statement corresponding to

$mids$`[`$sp$`++] = `$mid$`;`

in order to push its method identifier onto the reified stack. The integer $sp$ 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 $mids$ 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()`.

Compatibility with native code is achieved in a similar way as explained in Section 4.1: Because native code is not changed by the rewriting, Komorium adds simple wrapper methods with the unmodified signatures which allocate an array to represent the reified stack. The initial value of the stack pointer is zero.

### 5.2 Periodic Sampling

The custom profiling agent has to implement the abstract datatype `SamplingProfiler` (see Fig. 6). In order to schedule the regular activation of the custom profiling agent, each thread maintains an activation counter $ac$, in a similar way as described in Section 4.3. $ac$ is updated in the beginning of each basic block of code; the basic blocks are computed by the 'Default BBA' introduced in Section 4.2. The conditional that checks whether `processSample(MID[], INT)` has to be invoked is inserted in each basic block after the update of $ac$. In contrast to the exact profiler, we do not reduce overhead by computing an upper bound of the number

- `register(THREAD` $t$`): INT`
  This operation is invoked whenever a new thread $t$ is created. After `register(`$t$`)` has been called, the profiling agent must be prepared to handle subsequent invocations of `processSample(MID[], INT)` by $t$.
  `register(`$t$`)` returns the current profiling granularity for $t$.

- `processSample(MID[]` $mids$`, INT` $sp$`): INT`
  This operation is periodically invoked by each thread in the system in order to process a sample of the reified call stack, which is represented by the pair $\langle mids, sp \rangle$.
  `processSample(MID[], INT)` returns the current profiling granularity for the calling thread.

**Fig. 6.** `SamplingProfiler`

```
class Foo {
   static int sum(int from,
                  int to) {

      int result = 0;
      while (true) {

         if (from > to) {

             return result;
         }

         result += f(from);
         ++from;
      }
   }
}
```

```
class Foo {
   private static final MID mid_sum;
   static {
      String cl = Class.forName("Foo").getName();
      mid_sum   = createMID(cl, "sum", "(II)I");
   }

   static int sum(int from,
                  int to, AC ac, MID[] mids, int sp) {
      mids[sp++] = mid_sum;
      setValue(ac, getValue(ac) - 2);
      if (getValue(ac) <= 0)
         setValue(ac, processSample(mids, sp));
      int result = 0;
      while (true) {
         setValue(ac, getValue(ac) - 3);
         if (getValue(ac) <= 0)
            setValue(ac, processSample(mids, sp));
         if (from > to) {
            setValue(ac, getValue(ac) - 2);
            if (getValue(ac) <= 0)
               setValue(ac, processSample(mids, sp));
            return result;
         }
         setValue(ac, getValue(ac) - 7);
         if (getValue(ac) <= 0)
            setValue(ac, processSample(mids, sp));
         result += f(from, ac, mids, sp);
         ++from;
      }
   }

   static int sum(int from, int to) {
      AC ac = getOrCreateAC(Thread.currentThread());
      return sum(from, to, ac, new MID[STACKSIZE], 0);
   }
}
```

**Fig. 7.** Rewriting example: Program transformations for sampling-based profiling

of executed bytecodes and by minimizing the insertion of conditionals, because such optimizations would reduce the accuracy of the generated sampling profiles.

### 5.3 Rewriting Example

Fig. 7 illustrates the program transformations performed by Komorium with the same example as in Section 4.4. `STACKSIZE` is a constant that defines the maximum depth of a reified stack; typical values are between 1 000 and 10 000.

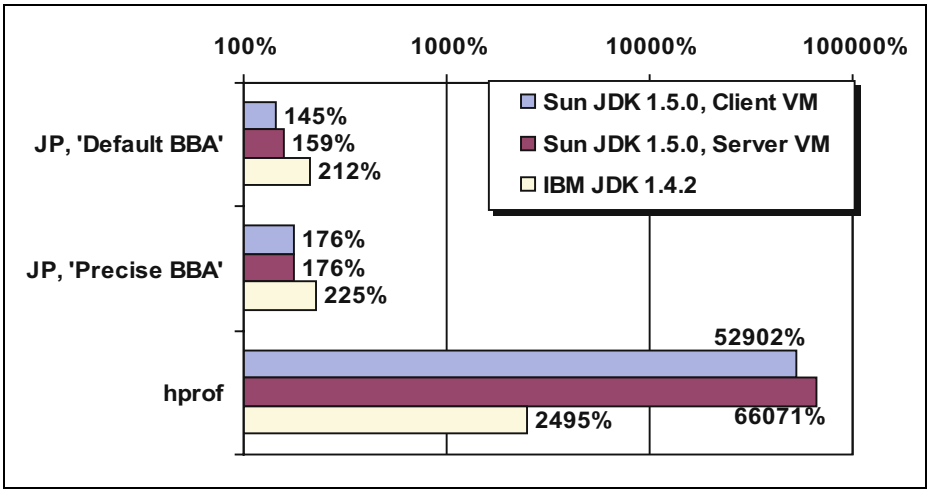


**Fig. 8.** JP: Profiling overhead for different profiler settings and JDKs

## 6 Evaluation

In the following we present our measurement results for JP and Komorium. While Section 6.1 provides performance measurements for JP, Section 6.2 discusses the accuracy of sampling profiles as well as the overhead caused by Komorium.

### 6.1 Exact Profiling (JP)

To evaluate the overhead caused by our exact profiler JP, we ran the SPEC JVM98 benchmark suite [17]. We removed background processes as much as possible in order to obtain reproducible results. For all settings, the entire JVM98 benchmark suite (consisting of several sub-tests) was run 10 times, and the final results were obtained by calculating the geometric mean of the median of each sub-test. Here we present the measurements made with Sun JDK 1.5.0 Client VM, Sun JDK 1.5.0 Server VM, as well as with IBM JDK 1.4.2.

Fig. 8 shows the profiling overhead for two different settings of JP, using the 'Default BBA' resp. the 'Precise BBA'. For both settings of JP, we used a simple profiling agent with the highest possible profiling granularity (the profiling agent was invoked by each thread after the execution of approximately $2^{31} - 1$ bytecodes). Upon program termination, the agent integrated the MCTs of all threads and wrote the resulting profile into a file using a JVM shutdown hook. Depending on the JVM, the average overhead for JP using the 'Default BBA' is 145–212%. For the 'Precise BBA' the average overhead is slightly higher (176–225%). We experienced the highest overhead of 900–1 414% with the 'mtrt' benchmark.

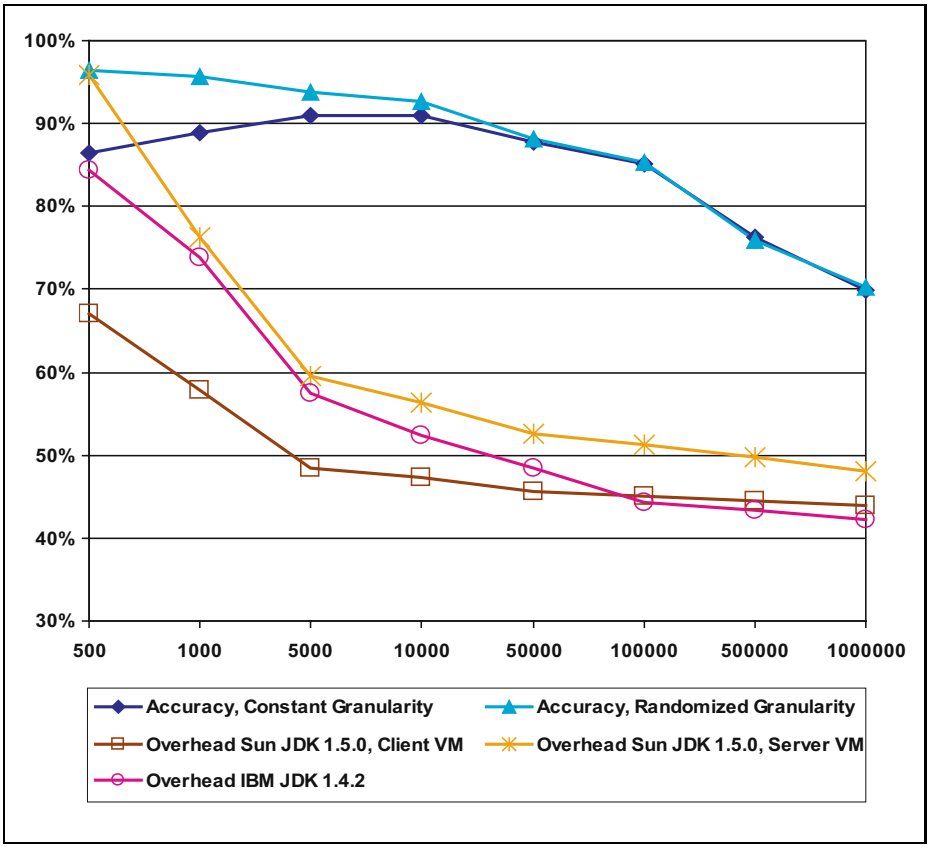


**Fig. 9.** Komorium: Average profile accuracy (overlap percentage) and average profiling overhead for different profiling granularities (X axis) and JDKs

To compare our profiler with a standard profiler based on the JVMPI/JVMTI, we also evaluated the overhead caused by the 'hprof' profiling agent shipped with the standard JDKs. On Sun's JVMs we started the profiling agent 'hprof' with the '`-agentlib:hprof=cpu=times`' option, which activates JVMTI-based profiling (available since JDK 1.5.0), whereas on IBM's JVM we used the '`-Xrunhprof:cpu=times`' option for JVMPI-based profiling. The argument '`cpu=times`' ensures that the profiling agent tracks every method invocation, as our profiling scheme does.

Because the overhead caused by the 'hprof' profiling agent is 1–2 orders of magnitude higher than the overhead caused by JP, Fig. 8 uses a logarithmic scale. On average, the slowdown due to the 'hprof' profiler is 52 902–66 071% on Sun's JVMs and 2 495% on IBM's JVM. For 'mtrt', the overhead due to 'hprof' exceeds 300 000% on both Sun JVMs.

### 6.2 Sampling Profiling (Komorium)

Even though the overhead caused by JP is 1–2 orders of magnitude lower than the overhead due to prevailing exact profilers, it may still be too high for certain complex applications. Komorium aims at computing accurate sampling profiles with lower overhead than JP. Fig. 9 shows the average profile accuracy and profiling overhead of Komorium for different profiler settings and JVMs.

We use an *overlap percentage* metric as defined in [3,11] to compare profiles. 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%. In Fig. 9 we show the average accuracy of sampling profiles (i.e., the overlap percentage of sampling profiles created by Komorium with the corresponding perfect profiles generated by JP using the 'Precise BBA') obtained with different profiling granularities for the SPEC JVM98 benchmark suite.

With a constant profiling granularity, the achievable accuracy is 91%. The best results are obtained with a profiling granularity of 5 000–10 000. As one could expect, the accuracy decreases with increasing profiling granularity (i.e., fewer samples). Surprisingly, also for lower granularities (500–1 000), the accuracy decreases. This is because program behaviour may correlate with our deterministic sampling mechanism, reducing the accuracy of profiles [3].

The accuracy can be improved by adding a small random value $r$ to the profiling granularity [2]. For our measurements, $r$ was equally distributed with $0 \leq r < 100$. This randomization increases the average overlap percentage for lower profiling granularities up to 96%. In order to enable reproducible results despite of the randomization, the profiling agent may use a separate pseudo-random number generator for each thread and initialize it with a constant seed.

For all measured profiling granularities, Komorium causes less overhead than JP. With a profiling 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).

We also evaluated the standard 'hprof' profiler in its sampling mode. On JDK 1.4.2 (JVMPI-based profiling), the average overhead is about 150%, while on JDK 1.5.0 (JVMTI-based profiling) the average overhead is about 90%. These overheads are higher than the overheads caused by Komorium with a profling granularity of 10 000. Moreover, the accuracy of the sampling profiles generated by 'hprof' is very low: On average, the overlap percentage of the sampling profiles with exact ones (generated by 'hprof' in its exact profiling mode) is below 7%. The primary reason for this inferior accuracy is the low sampling rate used by 'hprof' (max. 1 000 samples/second).

## 7 Limitations and Future Work

While in Section 2 we stressed the benefits of our profiling framework, this section discusses its limitations and outlines some ideas for future improvements.

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 generated profiles

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.

Concerning our exact profiler JP, which does not limit the depth of the generated MCTs, memory consumption may be an issue in the case of deep recursions. However, the developer may resort to our sampling profiler Komorium, for which the extra memory needed per thread is constant. Moreover, the size of the generated sampling profiles is factor 2–10 smaller than the size of the corresponding exact profiles, even for the lowest profiling granularity we measured (500).

Finally, we note that bytecode counting and CPU time are distinct metrics for different purposes. While profiles based on bytecode counting are platform-independent, reproducible, directly comparable across different environments, and valuable to gain insight into algorithm complexity, more research is needed in order to assess to which extend and under which conditions these profiles also allow an accurate prediction of actual CPU time for a concrete system.

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 native code, garbage collection, and dynamic compilation may limit the achievable accuracy. Therefore, we will focus first on simple JVM implementations (interpreters), such as JVMs for embedded systems, which do not involve complex optimization and re-compilation phases.

## 8 Related Work

Fine-grained instrumentation of binary code has been used for profiling in prior work [4,13]. In contrast, all profilers based on a fixed set of events such as the one provided by the JVMPI [15] are restricted to traces at the granularity of the method call. This restriction also exists with the current version of our profilers 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.

The NetBeans Profiler[6] integrates Sun's JFluid profiling technology [9] 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.

[6] `http://profiler.netbeans.org/index.html`

While Komorium is intended as a profiling tool for Java developers, sampling-based profiling is often used for feedback-directed optimizations in dynamic compilers [3,18], 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 [3] 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.

Much of the know-how worked into JP and Komorium comes from our previous experience gained with the Java Resource Accounting Framework, Second Edition (J-RAF2) [6,12], which also uses bytecode instrumentation in order to gather dynamic information about a running application. J-RAF2 only maintains a single bytecode counter for each thread, comparable to the activation counter used by our profilers (type `AC`). When the number of executed bytecodes exceeds a given threshold, a resource manager is invoked which implements a user-defined accounting or control policy. However, J-RAF2 is not suited for profiling: J-RAF2 cannot compute MCTs, and experiments with a sampling profiler based on J-RAF2 (using the `Throwable` API to obtain stack traces in the user-defined resource manager) were disappointing (low accuracy and high overhead).

## 9 Conclusion

In this paper we presented a novel profiling framework for Java, consisting of an exact and a sampling profiler. Our profilers exploit the number of executed bytecodes as platform-independent profiling metric. This metric is key to generate reproducible and directly comparable profiles, easing the use of the profiling tools. Moreover, this profiling metric allowed us to implement the profiling framework in pure Java, achieving compatibility with standard JVMs. Our profiling framework supports user-defined profiling agents in order to customize the creation of profiles. In contrast to many existing profiling interfaces which require profiling agents to be written in native code, we support portable profiling agents implemented in pure Java. The activation of the profiling agents follows a deterministic scheme, where the agents themselves control the activation rate in a fine-grained manner.

Performance evaluations revealed that our exact profiler causes 1–2 orders of magnitude less overhead than prevailing exact profilers. Nonetheless, for complex systems the execution time and memory overheads due to our exact profiler may still be too high. For these cases, our sampling profiler is able to generate highly accurate profiles with an overhead lower than standard sampling-based profilers, which often produce inferior profiles.

## Acknowledgements

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

## References

1. G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96. ACM Press, 1997.
2. J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, R. Sites, V. Vandervoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4), Nov. 1997.
3. M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.
4. T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.
5. T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.
6. W. Binder and J. Hulaas. A portable CPU-management framework for Java. *IEEE Internet Computing*, 8(5):74–83, Sep./Oct. 2004.
7. W. Binder, J. G. Hulaas, and A. Villazón. Portable resource control in Java. *ACM SIGPLAN Notices*, 36(11):139–155, Nov. 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).
8. J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Proceedings of the ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 21–31. ACM Press, 1999.
9. M. Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance*, pages 139–150. ACM Press, 2004.
10. B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, Nov. 2003.
11. P. Feller. Value profiling for instructions and memory locations. Master Thesis CS1998-581, University of California, Sa Diego, Apr. 1998.
12. J. Hulaas and W. Binder. Program transformations for portable CPU accounting and control in Java. In *Proceedings of PEPM'04 (2004 ACM SIGPLAN Symposium on Partial Evaluation & Program Manipulation)*, pages 169–177, Verona, Italy, August 24–25 2004.
13. J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software–Practice and Experience*, 24(2):197–218, Feb. 1994.
14. S. Liang and D. Viswanathan. Comprehensive profiling support in the Java virtual machine. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS-99)*, pages 229–240, Berkeley, CA, May 3–7 1999. USENIX Association.
15. Sun Microsystems, Inc. Java Virtual Machine Profiler Interface (JVMPI). Web pages at `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/`.
16. Sun Microsystems, Inc. JVM Tool Interface (JVMTI). Web pages at `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/`.
17. The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Web pages at `http://www.spec.org/osg/jvm98/`.
18. J. Whaley. A portable sampling-based profiler for Java Virtual Machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 78–87. ACM Press, June 2000.