SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2008; **0**:1–33 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Platform-independent Profiling in a Virtual Execution Environment

Walter Binder,*[1] Jarle Hulaas,[2] Philippe Moret,[1] and Alex Villazón[1]

[1] *University of Lugano, Switzerland*
[2] *Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland*

SUMMARY

Virtual execution environments, such as the Java Virtual Machine, promote platform-independent software development. However, when it comes to analyzing algorithm complexity and performance bottlenecks, available tools focus on platform-specific metrics, such as e.g. the CPU time consumption on a particular system. Other drawbacks of many prevailing profiling tools are high overhead, significant measurement perturbation, as well as reduced portability of profiling tools, which are often implemented in platform-dependent native code. This article presents a novel profiling approach, which is entirely based on program transformation techniques, in order to build a profiling data structure that provides calling-context-sensitive program execution statistics. We explore the use of platform-independent profiling metrics, in order to make the instrumentation entirely portable and to generate reproducible profiles. We implemented these ideas within a Java-based profiling tool called JP. A significant novelty is that this tool achieves complete bytecode coverage, by statically instrumenting the core runtime libraries, and dynamically instrumenting the rest of the code. JP provides a small and flexible API to write customized profiling agents in pure Java, which are periodically activated to process the collected profiling information. Performance measurements point out that, despite the presence of dynamic instrumentation, JP causes significantly less overhead than a prevailing tool for the profiling of Java code.

KEY WORDS: Java; JVM; Profiling; Program Transformations; Bytecode Instrumentation; Dynamic metrics

1. INTRODUCTION

Virtual execution environments, such as the Java Virtual Machine (JVM) [35] or Microsoft's .NET framework [36], have become the basis for building complex, heterogeneous, component-based software systems, because they enable the development of fully portable software. The Java promoters'

*Correspondence to: W. Binder, Faculty of Informatics, University of Lugano, CH–6900 Lugano, Switzerland
E-mail: walter.binder@unisi.ch
Contract/grant sponsor: Part of this work was done while J. Hulaas was at Ericsson's Network Management Research Centre, Ireland, with a Marie Curie Fellowship grant. P. Moret benefits from a grant of the Swiss National Science Foundation.

Received 29 February 2008
Revised 13 May 2008

slogan ‘write once, run anywhere’ is meant to emphasize the platform-independence of compiled Java programs, which are represented as JVM bytecode [35].

Unfortunately, when it comes to performance analysis for Java programs, platform independence is lost. Prevailing profiling tools typically provide information concerning CPU time consumption on a particular execution platform, which is a highly system-specific dynamic metric, depending on hardware, operating system, and virtual machine implementation. Similarly, memory consumption is usually reported in bytes, a unit of measurement which takes into account platform-specific characteristics: object internal representation, alignment constraints, header size, etc.

In addition to the lack of support for platform-independent performance analysis, prevailing profiling tools often cause high overhead and therefore significantly perturbate the measurement. In other words, execution statistics reported by profiling tools do frequently not faithfully represent normal, non-profiled program executions, which may mislead software developers searching for performance bottlenecks. As another shortcoming, many profiling tools for virtual execution environments consist at least partly of platform-specific native code, limiting the portability of these tools.

Concretely, most profiling tools for Java are based on the Java Virtual Machine Profiling Interface (JVMPI) [44, 34] or its successor, the JVM Tool Interface (JVMTI) [45]. These interfaces provide a set of hooks into the JVM which signal events, such as method invocation, object allocation, etc. However, many profiling events prevent optimizations within the virtual machine; even just-in-time compilation may be disabled. Consequently, profiling can cause excessive overheads of more than factor 1 000 and completely perturbate the measurements. Moreover, these interfaces violate the ‘write once, run anywhere’ motto, because they require profiling agents to be implemented in native code.

One important use case for platform-independent profiling is in the area of *service-oriented architectures* (SOA). SOA aims at the construction of applications by integrating advanced service components [40], such as service repositories, matchmakers, service composition and orchestration engines, reputation mechanisms, etc. These components are typically deployed in heterogeneous environments, which means that the actual target platforms are often not known at development time. Therefore, it becomes necessary to have platform-independent profiling support that allows the developer to detect algorithmic inefficiencies, as illustrated in [19], where the authors present a profiling tool which enabled them to identify at the bytecode[*] level an inefficient sorting algorithm in a piece of non-trivial third-party software. Moreover, as components may involve complex algorithms, such as planners for automated service composition, the measurement overhead has to be low in order for such profiling tools to be applicable. Existing profilers are not appropriate for this setting because of their high overhead, their exclusive focus on highly platform-specific metrics, and frequent suffering from strong measurement perturbation.

In this article we present a novel approach for fully platform-independent profiling in virtual execution environments. We instrument programs in order to create a profiling data structure at runtime, which stores various calling-context-sensitive dynamic metrics. Each calling context includes a method invocation counter, a bytecode counter, as well as memory allocation counters. As opposed to sampling-based profiling, our approach aims at generating exact profiles, tracking each method invocation.

[*]In this article the term ‘bytecode’ is used as a synonym for ’virtual machine bytecode instruction’.

In this article we explain our approach using abstract datatypes. We assessed our approach with a Java implementation called JP. JP is implemented in pure Java and supports custom profiling agents that may be written in pure Java, too. Hence, JP is a fully portable profiling framework that can be used on any JVM, also on those that support neither the JVMPI nor the JVMTI or that provide limited support for profiling in general. A significant novelty is that JP also achieves complete bytecode coverage, by statically instrumenting the core runtime libraries, and dynamically instrumenting application code as well as any required additional libraries.

JP offers much flexibility with respect to the generation and processing of profiling data structures at runtime. JP provides a simple API to implement customized profiling agents. User-defined profiling agents can thus be programmed, e.g. to preserve a trace of the full call stack or, on the opposite, to compact it at certain intervals. This contrasts with existing profilers which frequently only support a fixed maximal stack depth.

In this article we show that the overhead caused by JP 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. The bigger part of the overhead is due to the generation of calling-context-sensitive profiling data structures, whereas the maintenance of bytecode and memory allocation counters causes relatively little overhead.

The contributions of this article are, first, novel techniques for platform-independent profiling in virtual execution environments based on program transformations, along with the introduction of portable and customizable profiling agents, and second, the presentation of a Java-based profiling tool that uses dynamic instrumentation to cover all executed bytecode, a thorough analysis of the profiling overhead in our implementation using the DaCapo benchmark suite [12], the SPEC JVM98 benchmark suite [48] and the SPEC JBB2005 [47] benchmark, as well as a comparison with the overhead caused by ‘hprof’, a standard profiler that comes with many distributions of the Java Development Kit (JDK).

This article is structured as follows: Section 2 introduces the platform-independent dynamic metrics used by our profiling approach. Section 3 presents our profiling data structures using abstract datatypes. Section 4 explains how call stacks are managed by the profiler, how applications are transformed at the bytecode level to generate the needed profiling information, and how the number of executed bytecodes is computed, which serves as profiling metric. Section 5 details our approach to profiling memory allocation. Section 6 outlines the implementation of JP. Section 7 presents our performance measurements, Section 8 discusses the benefits and limitations of our approach, whereas Section 9 compares our approach with related work. Finally, Section 10 concludes this article.

2. PLATFORM-INDEPENDENT DYNAMIC METRICS

In the following we introduce the platform-independent profiling metrics our approach focuses on: the number of method invocations (Section 2.1), the number of executed bytecodes (Section 2.2), and the number of object and array allocations (Section 2.3).

2.1. Method Invocations

Our profiling approach aims at computing calling-context-sensitive, platform-independent dynamic metrics. For each calling context, we store the number of method invocations (with the same stack of callers). Information on the number of method invocations is a common metric supported by many available profiling tools. However, while some profilers do not differentiate between different calling contexts or keep calling contexts only up to a pre-defined depth, our approach is able to maintain arbitrarily deep calling contexts. Nonetheless, user-defined profiling agents may discard or aggregate execution statistics for certain calling contexts.

In Section 5.1 we will show that method invocation counters enable the computation of object allocation statistics without causing extra runtime overhead in Java environments.

2.2. Dynamic Bytecode Metrics

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 virtual machine implementation), measuring it accurately may require platform-specific features (such as special operating system functions) limiting the portability of the profilers, and results are usually impossible to reproduce faithfully. 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 virtual machines where profiling disables just-in-time compilation (e.g., profiling based on the JVMPI [44, 34] prevents just-in-time compilation).

For these reasons, we follow a different approach, using the *number of executed bytecodes* as profiling metric [6], which has the following benefits:

- **Platform-independent profiles**: The number of executed bytecodes is a platform-independent metric [23]. Although the CPU time of a deterministic program with a given input varies very much depending on the performance of the underlying 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 virtual machine class library is used).
- **Reproducible profiles**: For deterministic programs, the generated profiles are fully reproducible. It must nevertheless be noted that few programs are actually fully deterministic. Timestamps and hashcodes are often hidden sources of non-determinism, since they depend on the time of the day, respectively the underlying memory management policy, when hashcodes are computed from memory addresses. Also, the activity of system threads of the JVM, such as the garbage collector and finalizer thread, will introduce non-determinism at the application level via the execution of finalizers or other callbacks.
- **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. However, for multi-threaded, non-deterministic programs, the profiling may affect the thread scheduling, resulting in some (usually minor) 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.
- **Fine-grained control of profiling agent activation**: User-defined 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. Upon invocation, profiling agents may process the execution statistics collected so far, which enables the generation of continuous metrics [23]. Continuous metrics represent profiling information at different stages of program execution.
- **Reduced overhead:** The overhead is rather low compared to classical approaches, since it does not prevent the underlying virtual machine from putting all its optimization facilities to work during the profiling.

Consequently, dynamic bytecode metrics are key to the provision of a new class of portable, platform-independent profiling tools, with advantages for the tool users as well as for the tool implementors:

On the one hand, bytecode counting eases profiling, because thanks to the platform-independence of this metric [23], 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 reducing 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.

Other researchers have also found it valuable to have platform-independent profiling support, in order to allow the developer to detect algorithmic inefficiencies. In [19] the authors present a profiling tool which uses bytecode counting and enabled them to identify and replace an inefficient sorting algorithm in a piece of non-trivial third-party software.

In [29], a dynamic platform-independent analysis is made of a standard benchmark suite, which enables the authors to propose a number of improvements to existing Java compilers and execution environments.

2.3. Memory Allocation Metrics

Concerning dynamic memory allocation, most profilers use the number of allocated bytes as measurement unit. However, this metric depends on the particular virtual machine in use (object representation in memory, alignment, size of references, etc.).

In contrast, in our approach we track the number of allocated objects of each type, for each calling context [7]. This means that object allocations are described by triples of the form $\langle$calling context, object type, number of instances$\rangle$. This metric gives the developer a detailed, high-level view of object allocation in the profiled program. If desired, an estimation of the number of allocated bytes may be computed from this metric. As explained in Section 5.1, it is possible to compute this object allocation metric directly from method invocation counters without any additional instrumentation. Consequently, there is no extra overhead in the creation of profiling data structures (concerning execution time as well as the amount of memory required to store the profiling data structures).

Regarding array allocation, we preserve the element type, the number of allocated arrays, and the total number of array elements for each calling context. Concretely, array allocations are described by 4-tuples of the form $\langle$calling context, element type, number of arrays, number of array elements$\rangle$. In the case of Java, the element type may be one of the 8 basic types (`byte`, `short`, `int`, `long`, `char`, `boolean`, `float`, `double`), or a reference type. This metric may be used to compute other statistics, such as the average size of allocated arrays. Moreover, if the memory representation of arrays in a particular virtual machine is known, the metric may be used to compute the number of bytes consumed by allocated arrays.

For deterministic programs, these platform-independent memory allocation metrics yield reproducible profiles that are directly comparable across different machines. Measurement perturbation is not an issue, as we measure the number of objects that the unmodified program (without profiling) would allocate. In contrast, a metric such as the total amount of memory in use would be seriously perturbated by the measurement, since the profiling data structure itself consumes memory.

Our metrics for memory allocation do not expose the life-time of allocated objects. In general, the life-time of objects is hard to determine in virtual execution environments that rely entirely on automatic memory management (garbage collection), since there are no explicit de-allocation sites in the bytecode.

Custom profiling agents may put memory allocation metrics in relation to the number of executed bytecodes, in order to derive metrics such as the allocation density, i.e., the number of object allocations (or the approximate number of bytes allocated) per 1 000 executed bytecodes [23].

3. PROFILING DATA STRUCTURES

In this Section we define the data structures used by our profiling framework as abstract datatypes. Our profiler JP implements these abstract datatypes in Java.

3.1. Method Call Tree (MCT)

In our approach, we transform bytecode in order to create a Method Call Tree (MCT), where each node represents all invocations of a particular method with the same calling context (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, the number of bytecodes executed in the corresponding calling context (excluding the number of bytecodes executed by callee methods, since each callee has its own node in the MCT), as well as memory allocation counters.

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.[†]

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 [4], path profiling (i.e., preserving exact execution history) is feasible for a large portion of programs.

In the following we define two abstract datatypes to represent a MCT, the method identifier `MID` and the method invocation context `IC`. An instance of `IC` represents a node in the MCT. We assume the existence of the types `INT`, `STRING`, and `THREAD`, as well as the possibility to create aggregate types (`SET OF`).

3.1.1. Method Identifier.

- `createMID(STRING` $class$`, STRING` $name$`, STRING` sig`): MID`
  Creates a new method identifier, consisting of class name, method name, and method signature.

- `getClass(MID` mid`): STRING`
  Returns the class name of mid.
  `getClass(createMID(`c`,` x`,` y`))` = c.

- `getName(MID` mid`): STRING`
  Returns the method name of mid.
  `getName(createMID(`x`,` n`,` y`))` = n.

- `getSig(MID` mid`): STRING`
  Returns the method signature of mid.
  `getSig(createMID(`x`,` y`,` s`))` = s.

3.1.2. MCT Creation and Method Invocation Counter.

- `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` instance 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.

[†]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`.

`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`))` ≥ 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`)` ≥ 1.

3.1.3. *Bytecode Counter.*

Each `IC` instance stores a bytecode counter, which is initially zero. The bytecode counter is manipulated by the following operations:

- `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.

3.1.4. *Array Allocation Counter.*

Each `IC` instance stores array allocation counters, which are initially zero. The array allocation counters are manipulated by the following operations:

- `profileArrays(IC `c`, TYPE `t`, INT `$arrays$`, INT `$elements$`): IC`
  Registers array allocations in c. t is the element type of the arrays. $arrays$ represents the number of allocated arrays, while $elements$ is the total number of elements in all allocated arrays (i.e., $elements$ is the sum of the sizes of the allocated arrays).

`profileArrays(IC, TYPE, INT, INT)` returns c, after its array allocation statistics have been updated accordingly. This operation is not supported for the root of the MCT.

`TYPE` denotes the different data types supported by the virtual machine. In the case of the JVM, t may take one of the following values:

– `B`: Signed byte (`byte`).
– `C`: Unicode character (`char`).
– `D`: Double-precision floating point value (`double`).
– `F`: Single-precision floating point value (`float`).
– `I`: Integer (`int`).
– `J`: Long integer (`long`).
– `S`: Signed short (`short`).
– `Z`: True or false (`boolean`).
– `R`: Reference.

The first 8 values correspond to the encoding of basic types in the JVM [35]. The element type `R` indicates that the array stores references to objects, which may be instances of 'normal' classes or arrays.

- `getArrays(IC` c`, TYPE` t`): INT`
  Returns the number of arrays of element type t allocated in c.
  `getArrays(profileArrays(`x`,`t`,`$arrays$`,`y`),` t`)` $\geq arrays$.
  This operation is not supported for the root of the MCT.

- `getElements(IC` c`, TYPE` t`): INT`
  Returns the number of elements in arrays of element type t allocated in c.
  `getElements(profileArrays(`x`,`t`,`y`,`$elements$`),` t`)` $\geq elements$.
  This operation is not supported for the root of the MCT.

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` defined below represents an activation counter for each thread.

- `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.

4. PROFILING BYTECODE EXECUTION

In this Section we describe our program instrumentation scheme for platform-independent profiling, focusing on the MCT creation and the maintenance of bytecode counters. The profiling of dynamic memory allocation is the subject of Section 5.

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.

4.1. MCT Creation

We transform 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., we extend the signatures of all non-native methods with the additional argument). In the beginning of a method[‡] 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 instrumentation, we add 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 methods with the unmodified signatures.[§]

For each 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 method.

4.2. Bytecode Counting

For each method invocation context ic, we compute the number of executed bytecodes. We instrument the bytecode of methods in order to invoke `profileInstr(`ic`,` $bytecodes$`)` according to the number of executed bytecodes. For each (non-native) method, we perform a basic block analysis (BBA) to compute a control flow graph. In the beginning of each basic block, we insert a code sequence that implements this update of the bytecode counter.

[‡]Here we do not distinguish between methods and constructors, i.e., 'method' stands for 'method or constructor'.
[§]For native methods, which we cannot instrument, we add so-called 'reverse' wrappers which discard the extra `IC` argument before invoking the native method. The 'reverse' wrappers allow transformed code to invoke all methods with the additional argument, no matter whether the callee is native or not.

In the case of our Java profiler JP, the BBA algorithm is not hard-coded; 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, method return, exception throwing) end a basic block. Method 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 [11] and is related to the factored control flow graph [16].

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 though some of them may not be executed in case of an exception. The end result is that, 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 instrumented by JP.

4.3. Periodic Activation of Custom Profiling Agents

Our approach 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 provide an implementation of the abstract datatype `ProfilingAgent`, which includes two operations, `register(THREAD, IC)` and `processMCT(IC)`. In the case of JP, we mapped `ProfilingAgent` to a Java interface.

- `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 different threads into a global MCT, or to generate continuous metrics [23], 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.

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 in general a comparison against zero is more efficient than a comparison against an arbitrary, user-defined threshold. Concretely, the following polling 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 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 updated by an upper bound of the number of executed bytecodes.

The polling 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. Note that we do not implement full call/return polling [25], since we aim at minimizing the polling overhead and because variations in the number of executed bytecodes between invocations of the profiling agent are not a problem in our setting.

4.4. Program Transformation Example

The example in Fig. 1 illustrates the aforementioned program transformations. To the left is the class `Foo` with the method `sum(int, int)` before transformation, to the right is the instrumented

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

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

   ...                               ...
}                                 }
```

Figure 1. Example program transformations for MCT creation, bytecode counting, and periodic invocation of a custom profiling agent.

version.[¶] `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. 1, 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 bytecode.

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

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 instrumented code, the static initializer allocates the method identifier `mid_sum` to represent invocations of `sum(int, int)` in the MCT. The instrumented method receives 2 extra arguments, the activation counter (type `AC`) and the caller's method invocation context (type `IC`). First, the instrumented 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 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 instrumented 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. PROFILING MEMORY ALLOCATION

In the following we present our approach for platform-independent profiling of dynamic memory allocation. While Section 5.1 deals with profiling allocations of objects that are not arrays, Section 5.2 addresses array allocations. The approach presented here is specific to Java, because we exploit the mechanism by which the JVM allocates and initializes objects resp. arrays.

5.1. Profiling Object Allocation

At the Java level, objects are allocated and initialized through class instance creation expressions (`new`), whereas at the bytecode level, object allocation and initialization are separated. Objects are allocated with the `new`<*class*> bytecode instruction, which leaves a reference to the created object instance (of type *class*) on the stack. Before the object can be used, a constructor has to be invoked in order to initialize the object. At the bytecode level, constructors are special methods with the name `<init>` that are invoked with the `invokespecial`<*method-spec*> bytecode instruction.[‖] `invokespecial`

[‖] `invokespecial` is also used for other purposes, such as calling private methods or methods in a superclass.

receives a reference to the previously allocated (and still uninitialized) object, as well as the method arguments on the stack. The method selection is based on the compile-time type given in *method-spec*, thus we can statically determine which constructor is invoked.

One way to profile object allocation would be to instrument each occurrence of the `new` bytecode instruction. However, as object allocation is rather frequent, this approach would produce a non-negligible overhead due to bytecode expansion and resulting extra execution time. Therefore, we chose a different approach, taking advantage of the MCT that is already created. In the MCT, each method invocation context maintains the set of non-native callee methods and their respective number of invocations. As constructors must not be native (see [35], Section 2.12.1: 'Constructor Modifiers'), all constructor invocations are present in the MCT. Because the JVM ensures that objects are initialized at most once and that uninitialized objects cannot be used[**] (see [35], Section 4.8: 'Constraints on Java Virtual Machine Code', and Section 4.9: 'Verification of Class Files'), we assume that constructor invocations correspond to object allocations. Hence, instead of sequentially counting object allocations as they take place, we use profiling data that is anyway collected – the number of constructor invocations – to infer the total number of allocated objects.

While this profiling scheme allows to compute the number of objects allocated by 'normal' methods, tracking the number of objects allocated by constructors requires some extra analysis, because every constructor, except the constructor of `java.lang.Object`, also has to invoke either an alternate or a superclass constructor. In other words, for all constructors but the constructor of `java.lang.Object`, we cannot assume that each constructor invocation corresponds to an object allocation. Even though the invocation of an alternate or superclass constructor usually happens in the beginning of the constructor code, it is not necessarily the first invocation of a constructor in the code, since the creation of the constructor arguments may involve object allocation and initialization.

For instance, consider the example in Fig. 2, which shows a class `A` with two constructors. To the right is the constructor bytecode generated by a standard Java compiler. The first constructor `A()` invokes the second constructor `A(java.lang.Object)` and passes a newly allocated and initialized object instance. In the bytecode of `A()`, the invocation of the constructor of `java.lang.Object` comes before the invocation of `A(java.lang.Object)`. In the MCT, `A()` has 2 callees, the constructor of `java.lang.Object` as well as the constructor `A(java.lang.Object)`, but only one of them corresponds to an object allocation.

In order to correctly profile the number of object allocations in constructors, we statically analyze the bytecode of each constructor during the instrumentation, in order to determine which alternate or superclass constructor is invoked. We use abstract interpretation in order to simulate the evolution of the stack and of local variables during execution of the constructor code. We only track the `this` reference, which is initially passed to the constructor in the local variable 0, until the first invocation on it (i.e., invocation of the alternate or superclass constructor). Our simulation is similar to the one performed by the JVM bytecode verifier [35], but it is simpler, because we are only interested in the first invocation on the `this` reference, whereas the JVM bytecode verifier has to ensure several properties.

JP produces a map $\mathcal{M}$ that associates each constructor (except the constructor of `java.lang.Object`) with the corresponding alternate or superclass constructor it invokes. In

[**]There are nevertheless abnormal cases where even recent JVMs do not prevent method invocations on uninitialized objects, such as inside finalizers. This issue is further discussed in Section 5.3.

```
public class A {
   A() {
      this(new Object());         aload_0
                                  new java/lang/Object
                                  dup
                                  invokespecial java/lang/Object/<init>()V
                                  invokespecial A/<init>(Ljava/lang/Object;)V
                                  return
   }

   A(Object o) {
      super();                    aload_0
                                  invokespecial java/lang/Object/<init>()V
                                  return
   }

}
```

Figure 2. Constructor example.

the map, the constructors are identified by their fully qualified name and signature. This map is loaded and accessed by the user-defined profiling agent in order to compute the correct number of object allocations from the MCT. In the following we consider $\mathcal{M}$: `MID` $\longrightarrow$ `MID` a (partial) function mapping a method identifier of a constructor to the method identifier of the associated alternate or superclass constructor. In the example in Fig. 2, $\mathcal{M}$(`createMID("A","<init>","()V")`) = `createMID("A","<init>","(java.lang.Object)V")`, and $\mathcal{M}$(`createMID("A", "<init>","(java.lang.Object)V")`) = `createMID("java.lang.Object", "<init>","()V")`.

Note that the assumption that each constructor (except the constructor of `java.lang.Object`) has exactly one associated alternate or superclass constructor may not hold for hand-crafted bytecode, as illustrated in Appendix A. However, this assumption is valid for compiled Java code, and the static analyzer is able to detect situations where the assumption is violated, producing a warning.

Fig. 3 explains how to compute the number of object allocations from the information stored in the MCT. Function `getAlloc(IC)` of Fig. 3 (a) returns the total number of objects allocated in a given method invocation context c. If c does not correspond to a constructor (or corresponds to the constructor of `java.lang.Object`), then `getAlloc(IC)` returns the total number of constructor invocations in the context of c. If c corresponds to a constructor (different from the constructor of `java.lang.Object`), then the sum has to be reduced by `getCalls(`c`)`, because each time the constructor corresponding to c is invoked, it will call its associated alternate or superclass constructor $\mathcal{M}$`(getMID(`c`))` once, without any object allocation taking place. Note that the computation of `getAlloc(IC)` does not require the map $\mathcal{M}$. Function `getAlloc(IC, STRING)` of Fig. 3 (b) returns the number of objects of a certain type $class$ allocated in a given method invocation context c. In contrast to `getAlloc(IC)`, function `getAlloc(IC, STRING)` differentiates between the invocations of constructors of different classes.

$$\texttt{getAlloc(IC } c\texttt{)=}\begin{cases}\left(\displaystyle\sum_{\substack{x\in\texttt{getCallees}(c),\\ mid_x=\texttt{getMID}(x),\\ \texttt{getName}(mid_x)=\texttt{<init>}}}\texttt{getCalls}(x)\right)-\texttt{getCalls}(c) & \text{if } mid_c\texttt{=getMID(}c\texttt{)}\wedge \texttt{getName(}mid_c\texttt{)=<init>}\wedge \texttt{getClass(}mid_c\texttt{)}\neq \texttt{java.lang.Object}\\ \displaystyle\sum_{\substack{x\in\texttt{getCallees}(c),\\ mid_x=\texttt{getMID}(x),\\ \texttt{getName}(mid_x)=\texttt{<init>}}}\texttt{getCalls}(x) & \text{otherwise}\end{cases}$$

(a) Total number of objects allocated by the method invocation context c.

$$\texttt{getAlloc(IC } c\texttt{, STRING } class\texttt{)=}\begin{cases}\left(\displaystyle\sum_{\substack{x\in\texttt{getCallees}(c),\\ mid_x=\texttt{getMID}(x),\\ \texttt{getName}(mid_x)=\texttt{<init>},\\ \texttt{getClass}(mid_x)=class}}\texttt{getCalls}(x)\right)-\texttt{getCalls}(c) & \text{if } mid_c\texttt{=getMID(}c\texttt{)}\wedge \texttt{getName(}mid_c\texttt{)=<init>}\wedge \texttt{getClass(}mid_c\texttt{)}\neq \texttt{java.lang.Object}\wedge \texttt{getClass(}\mathcal{M}(mid_c)\texttt{)} = class\\ \displaystyle\sum_{\substack{x\in\texttt{getCallees}(c),\\ mid_x=\texttt{getMID}(x),\\ \texttt{getName}(mid_x)=\texttt{<init>},\\ \texttt{getClass}(mid_x)=class}}\texttt{getCalls}(x) & \text{otherwise}\end{cases}$$

(b) Number of objects of type $class$ allocated by the method invocation context c.

Figure 3. Computing the number of allocated objects based on the number of constructor invocations.

5.2. Profiling Array Allocation

As array allocation does not involve any method/constructor invocation, the approach presented in Section 5.1 is not applicable to profile array allocations. Therefore, we instrument all occurrences of bytecode instructions that allocate arrays in order to preserve statistics of the type, number, and size of allocated arrays.

In the JVM, the bytecode instructions `newarray`<*type*>, `anewarray`<*type*>, and `multianewarray`<*type*><*allocDim*> are used to allocate arrays. While `newarray` allocates a 1-dimensional array of a basic type (`byte`, `short`, `int`, `long`, `boolean`, `char`, `float`, `double`), `anewarray` allocates a 1-dimensional array to hold references. Multi-dimensional arrays are represented as arrays of arrays. `anewarray` may be used to allocate one dimension of a multi-dimensional array. If several dimensions of a multi-dimensional array are to be allocated at once, it is more efficient to use `multianewarray`, which subsumes the functionality of `newarray` and of `anewarray`, and allows to allocate several array dimensions (the parameter $allocDim$) with a single bytecode instruction.

`newarray` and `anewarray` receive the size s of the array to allocate on the stack; s must be a non-negative integer value. In order to profile an array allocation `newarray`<*type*>, we insert a bytecode sequence directly before the `newarray` bytecode instruction, corresponding to `profileArrays(`c`, `t`, 1, `s`)`, where c represents the current method invocation context and t the corresponding element type of the array (`B`, `C`, `D`, `F`, `I`, `J`, `S`, or `Z`). For an array allocation `anewarray`<*type*>, we insert a bytecode sequence that corresponds to `profileArrays(`c`, R, 1, `s`)`.

`multianewarray`<*type*><*allocDim*> receives $allocDim$ non-negative integer values on the stack, which correspond to the sizes of the array dimensions to be allocated. If $allocDim = 1$, `multianewarray` could be replaced either by `newarray` or by `anewarray`. Hence, we can profile the array allocation as described for `newarray` resp. `anewarray`.

If $allocDim > 1$, the actual number of arrays and of array elements have to computed by multiplying the sizes of the dimensions [20]. The dimensionality of the array $arrayDim$ is encoded in the array type descriptor ($type$) [35]; $allocDim \leq arrayDim$. We distinguish two cases:

1. $allocDim < arrayDim$, or the base type of the array is an object type. In this case, only arrays that have references as elements (`R`) are allocated. For instance, the following array allocation examples fall into this category:

   - `multianewarray [[[I 2`
     Allocates the first two dimensions of a 3-dimensional integer array.
   - `multianewarray [[Ljava/lang/Object; 2`
     Allocates a 2-dimensional array of objects.

   In order to profile the array allocation, we insert a bytecode sequence that corresponds to one invocation of `profileArrays(IC, TYPE, INT, INT)`:[††]

$$\texttt{profileArrays(}c\texttt{,R,}\left[\sum_{i=0}^{allocDim-1}\prod_{j=1}^{i} dim(j)\right]\texttt{,}\left[\sum_{i=1}^{allocDim}\prod_{j=1}^{i} dim(j)\right]\texttt{)}$$

2. $allocDim = arrayDim$, and the base type of the array is a basic type t. In this case, two types of arrays are allocated: Arrays that have references as elements (`R`), as well as arrays that have a basic type as elements (`B`, `C`, `D`, `F`, `I`, `J`, `S`, or `Z`). For instance, the following array allocations fall into this category:

   - `multianewarray [[[I 3`
     Allocates a 3-dimensional integer array.

[††] $dim(j)$ refers to the j^{th} dimension of the array, $dim(j) \geq 0, 1 \leq j \leq allocDim$. $\prod_{j=1}^{0} x = 1$.

```
new Object[2][3][5] → profileArrays(c, R, 9, 38)
new Object[2][3][0] → profileArrays(c, R, 9, 8)
new Object[2][0][5] → profileArrays(c, R, 3, 2)
new Object[0][3][5] → profileArrays(c, R, 1, 0)

new int[2][3][5]    → profileArrays(c, R, 3, 8), profileArrays(c, I, 6, 30)
new int[2][3][0]    → profileArrays(c, R, 3, 8), profileArrays(c, I, 6, 0)
new int[2][0][5]    → profileArrays(c, R, 3, 2), profileArrays(c, I, 0, 0)
new int[0][3][5]    → profileArrays(c, R, 1, 0), profileArrays(c, I, 0, 0)
```

Figure 4. Examples: Profiling the allocation of multi-dimensional arrays.

- `multianewarray [[Z 2`
  Allocates a 2-dimensional boolean array.

In order to profile the array allocation, we insert a bytecode sequence that corresponds to two invocations of `profileArrays(IC, TYPE, INT, INT)`:

$$\texttt{profileArrays(}c\texttt{,R,}\left[\sum_{i=0}^{allocDim-2}\prod_{j=1}^{i} dim(j)\right]\texttt{,}\left[\sum_{i=1}^{allocDim-1}\prod_{j=1}^{i} dim(j)\right]\texttt{)}$$

$$\texttt{profileArrays(}c\texttt{,}t\texttt{,}\left[\prod_{j=1}^{allocDim-1} dim(j)\right]\texttt{,}\left[\prod_{j=1}^{allocDim} dim(j)\right]\texttt{)}$$

Fig. 4 illustrates the profiling of the allocation of multi-dimensional arrays with several examples.

At the implementation level, the inserted bytecode sequence to profile the allocation of a multi-dimensional array is generated according to the algorithm in Fig. 5. While the size of each array dimension is an `int`, the results of the arithmetic operations may exceed the range of an `int`. Hence, the variables $prod$, arr, and el are of the type `long` (i.e., each of them occupies two local variables).

5.3. Accuracy of Memory Profiling

In this Section we consider the accuracy of the profiling scheme presented in Section 5.1 and in Section 5.2. We discuss to which extent and under which conditions the generated memory allocation profiles are accurate.

An important limitation of our approach is that it cannot profile the execution of native code. Nonetheless, as constructors cannot be native, the MCT covers all constructor invocations. The consequence is that, in general, the information regarding the allocation of objects that are not arrays is present in the MCT, even though the profiled program may spend a considerable part of its execution time in native code.

Concerning the allocation of objects that are not arrays, the approach described in Section 5.1 tracks the allocation of all objects that are correctly initialized (i.e., the constructor returns normally). If an

```
1. Save array dimensions (provided on the stack) in dedicated local variables. (As an optimization, the
   first array dimension can remain on the stack.)
2. Allocate further local variables to hold the current dimensional product $prod$, the number of arrays
   $arr$, and the number of array elements $el$.
   $prod := 1$.
   $arr := 0$.
   $el := 0$.
3. For each dimension $i$ ($1 \leq i \leq allocDim$):
   (a) If $i = allocDim = arrayDim$ and the base type of the array is a basic type:
       i. Invoke profileArrays($c$, R, $arr$, $el$).
       ii. $arr := 0$.
           $el := 0$.
   (b) $arr := arr + prod$.
   (c) Retrieve the size of the $i^{th}$ array dimension $dim(i)$ from the corresponding local variable. (The
       first array dimension may be directly duplicated on the stack.)
   (d) $prod := prod * dim(i)$.
   (e) $el := el + prod$.
4. Invoke profileArrays($c$, $t$, $arr$, $el$).
   If $allocDim = arrayDim$ and the base type of the array is a basic type, $t$ corresponds to that basic
   type; otherwise, $t$ = R.
5. Restore the array dimensions from local variables. (The first array dimension may be still on the
   stack.)
```

Figure 5. Algorithm to instrument allocations of multi-dimensional arrays.

exception occurs after object allocation but before the invocation of the constructor (e.g., an exception during the evaluation of the constructor arguments), the object allocation is not visible in the profile. A rare situation, in conjunction with finalizers, is when an uninitialized object is actually used, even though its constructor has not been invoked. Also in this case, the object allocation is not tracked. If we wanted to detect such abnormal cases, we might nevertheless add instrumentation to count all executions of the `new` bytecode instruction, and check if this produces the same result as our constructor-counting method.

If an exception is thrown in the constructor, the invocation of the constructor and hence the object allocation is visible in the profile. Nonetheless, if the exception occurs before the invocation of an alternate or superclass constructor, the computation of the number of object allocations within the constructor according to Fig. 3 may be incorrect. Note that the latter problem only concerns the computation of the number of object allocations in constructors, but not in other Java methods. Summing up, uninitialized objects may distort the computed object allocation profiles. Fortunately, this is rarely a problem in practice, because constructors that terminate abnormally throwing an exception are not frequent.

Regarding array allocation, we insert profiling code before the bytecode instruction that allocates the array. Thus, if the array allocation fails (e.g., the size of the array provided on the stack is negative or the JVM runs out of memory), the profile may be inconsistent. We did not consider the case of a negative array size, as this situation is usually a consequence of a programming error. To

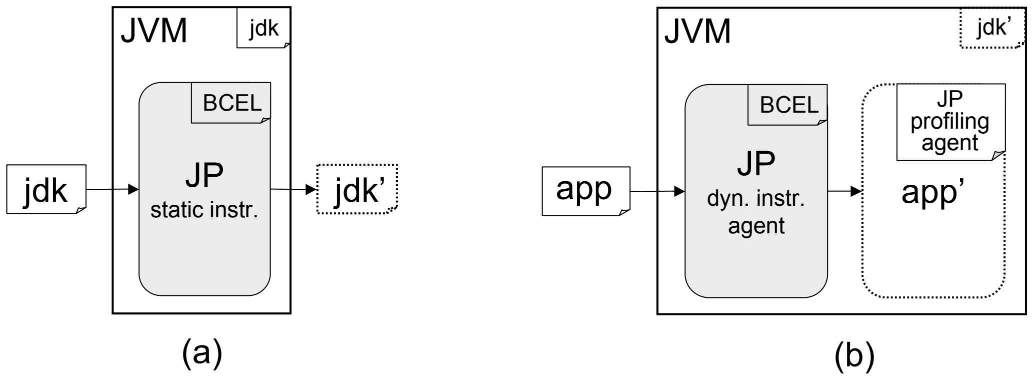

Figure 6. (a) Static instrumentation tool and (b) custom profiling agent and dynamic instrumentation agent of JP.

address this issue, we could insert conditionals in the profiling code in order to skip the invocation of `profileArrays(IC, TYPE, INT, INT)` if a negative array size was detected. As most applications are not designed to deal with occurrences of `OutOfMemoryError`, we did not consider this issue either. I.e., our profiler is intended to be used on well tested programs, which run successfully without causing such exceptions/errors.

It is also possible to defer the invocation(s) of `profileArrays(IC, TYPE, INT, INT)` after the actual array allocation bytecode, so that they get skipped in the case of an exception. This requires a slight modification of the instrumentation algorithm presented in Fig. 5, as the integer arguments of `profileArrays(IC, TYPE, INT, INT)` have to be stored either on the stack (below the slots with the array dimensions) or in local variables.

6. IMPLEMENTATION OF JP

In the following we outline our implementation principles. As illustrated in Figure 6, JP consists of (a) a static instrumentation tool for the preparation of the JDK, and (b) a runtime instrumentation agent which transforms application classes (and any remaining libraries) on the fly for collaboration with the custom profiling agent.

In order to implement the proposed profiling scheme, JP changes method (as well as constructor) signatures and bodies, and adds new methods and fields (with some restrictions, as described later). At the same time, JP has to (1) ensure that inserted profiling code will not be executed before the JVM has completed bootstrapping and (2) provide support for temporarily disabling the execution of profiling code for each thread.

The first constraint is important in order not to disrupt the startup of the JVM, since current standard JVMs may crash if certain JDK classes (e.g., `Object`, `String`, `Throwable`, `Thread`, etc.) are initialized in an unexpected order because of additional class dependencies. Another problem is that, as long as the JVM is not completely initialized, thread manipulation primitives may not behave according to their specified functionality, and therefore have to be considered as unavailable.

The second constraint allows dynamic instrumentation to be performed inside the same JVM process that runs the instrumented application, without creating artifacts in the collected profiling data.

6.1. Bootstrapping Support

Since the JVM offers no standardized support for bootstrapping with a customized JDK, we follow the safe and portable approach of considering the whole bootstrapping sequence as critical, and rely on the Java Language Specification [28], which mandates lazy class initialization, and hence guarantees that classes are not initialized before their first use. JP relies on a so-called "Java programming language agent" (for details, see the API documentation of the `java.lang.instrument` package) for dynamic instrumentation and for detecting the end of bootstrapping.

JP ensures that inserted profiling code is not executed while the JVM is bootstrapping. To this end, the following rules have to be enforced during bootstrapping:

- Method bodies that have been modified are not executed; instead, the original method bodies need to be executed. To this end, JP keeps a copy of the original method body together with the instrumented version and inserts a conditional in the beginning of the method that branches to the appropriate version depending on the state of the JVM.
- Added methods are not executed at all; they can only be invoked by profiling code after the end of bootstrapping. JP also patches Java's reflection API to ensure that added methods will not be accidentally used by existing code that relies on reflection.
- Classes used by profiling code are not initialized, unless they anyway would be initialized when bootstrapping an unmodified JDK.
- No fields are added to classes that are loaded during bootstrapping, unless these fields are simply initialized to their respective default value. This restriction is necessary to prevent the need for executing profiling code inside static initializers or constructors during bootstrapping. However, for all classes loaded during bootstrapping, JP will create "sister classes", the role of which is to hold any required additional static fields, such as the method identifiers (MIDs) introduced in Section 3. Because of lazy class initialization, and because the added static fields are accessed only by profiling code, the static initializers of the extra classes will not be executed during bootstrapping.

6.2. Dynamic Instrumentation Support

As illustrated in Figure 6(b), JP is designed to execute the dynamic instrumentation agent and the instrumented application inside the same JVM.

Dynamic instrumentation necessarily involves the invocation of methods of the JDK, which themselves are instrumented, since Java prohibits loading an instrumented and a non-instrumented version of a same JDK class in different classloader namespaces. Therefore, in order to prevent dynamic instrumentation from perturbing the collection of profiling data, JP allows switching on and off the execution of profiling code separately for each thread.

While other approaches, such as the NetBeans profiler [37], perform dynamic instrumentation in a separate JVM process, and use inter-process communication (IPC) to synchronize and transfer all class bytes, our in-process dynamic instrumentation has the advantage that it avoids the overhead of

IPC (which entails frequent process-level context switches) and the memory footprint of an additional JVM process.

Our tighter architecture makes it potentially easier for the developer to selectively disable the profiling of either a set of classes (by temporarily switching off dynamic instrumentation) or a number of threads (by choosing to bypass any inserted profiling code); this in turn helps the developer reduce the time spent profiling.

7. EVALUATION

In this Section we evaluate the overhead caused by our Java profiler JP in different settings and compare it with the 'hprof' profiling agent that is included in standard JDKs.

To evaluate the overhead caused by our profiling scheme, we used the DaCapo benchmark suite [12] (version 'dacapo-2006-10-MR2'), the SPEC JVM98 benchmark suite [48] (with problem size 100), as well as the SPEC JBB2005 benchmark [47] (warehouse sequence 1, 2, 3, 4). SPEC JVM98 consists of 7 benchmarks, whereas DaCapo, which is more recent, and is intended to provide more realistic workloads than SPEC JVM98 (a thorough comparison is given in [12]), consists of 11 benchmarks. Our test platform is a Linux Fedora Core 2 computer (Intel Pentium 4, 2.66 GHz, 1 GB RAM). The metric used for DaCapo and SPEC JVM98 is the execution time in seconds, whereas SPEC JBB2005 measures the throughput in operations/second. All benchmarks were run in single-user mode (no networking) and we removed background processes as much as possible in order to obtain reproducible results. For each setting and each benchmark, we took the median of 15 runs. For the DaCapo and SPEC JVM98 suites, we also computed the geometric mean of the respective benchmarks. Here we present the measurements made with the Sun JDK 1.7.0 platform ('early access', build b24) in its 'client' and 'server' modes.

7.1. Time Overhead

Fig. 7 and 8 show the profiling overhead for different settings, with memory profiling enabled. For the DaCapo and SPEC JVM98 benchmarks (resp. the SPEC JBB2005 benchmark), the overhead is computed as a factor of $\frac{\text{execution time with profiling}}{\text{execution time without profiling}} \left(\text{resp. } \frac{\text{operations/second without profiling}}{\text{operations/second with profiling}}\right)$. 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 standard JDKs. We started the profiling agent 'hprof' with the '`-agentlib:hprof=cpu=times`' option, which activates JVMTI-based profiling. The argument '`cpu=times`' ensures that the profiling agent tracks every method invocation, as our own profiling scheme does.

For 'mtrt', the overhead due to 'hprof' is a factor close to 1 000 or more, depending on the execution mode ('client' or 'server'). The 'mtrt' benchmark produces by far the highest overhead with JP as well. This is because this benchmark is made of a very high number of method invocations (according to [23] it has the highest ratio of method invocations of the SPEC JVM98 suite); method entries are where most profiling activity takes place.

On the opposite side, 'db' produces the lowest overhead, both with JP and hprof. This may be explained by its unoptimized use of memory, since it spends most of the time sorting a small database

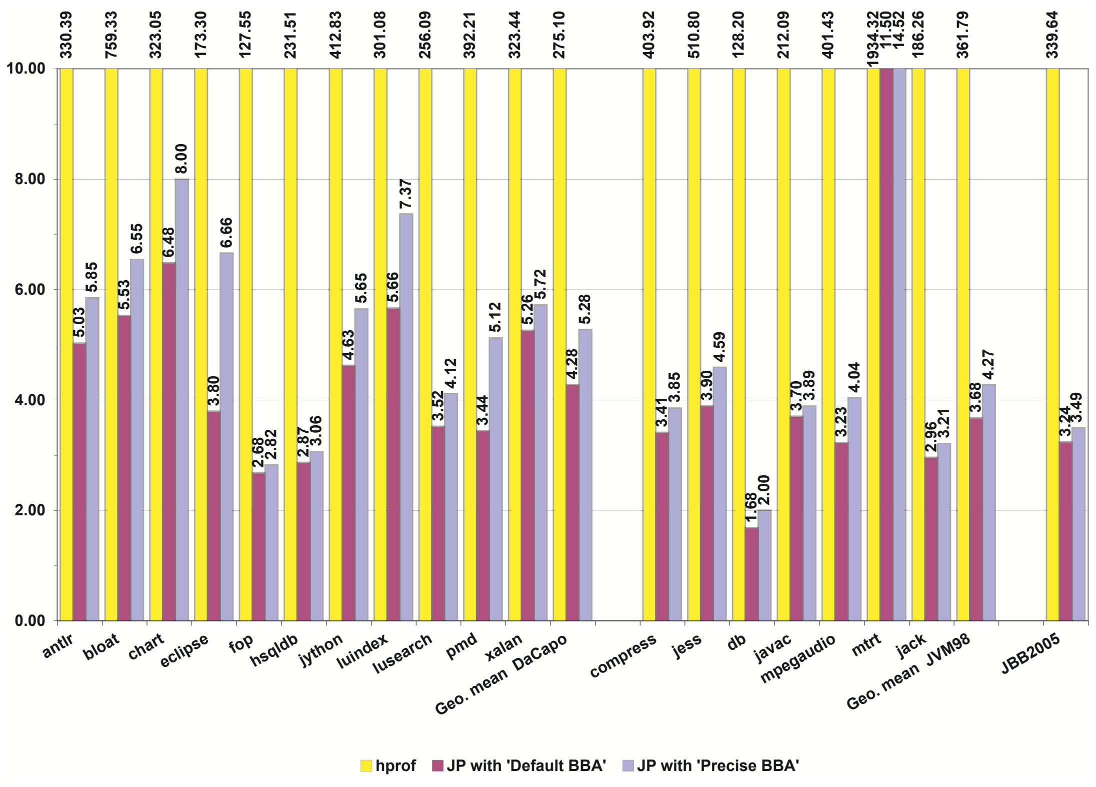

Figure 7. Profiling overhead (slowdown factor) for different profiler settings with Sun JDK 1.7.0 in client mode.

using a simple algorithm that ignores data locality, which results in serious thrashing of the underlying hardware memory management system [42]; therefore, this program spends comparatively more time in platform-level memory management operations than executing actual bytecodes. We have found (using a predecessor tool of JP [13]) that 'db' executes up to one order of magnitude fewer bytecode instructions per unit of time than other, more optimized benchmarks of SPEC JVM98.

For the DaCapo suite, on average, the slowdown due to the 'hprof' profiler is a factor 201–275, for the SPEC JVM98 suite, the slowdown is a factor 202–362, while for SPEC JBB2005, it is 340–436.

In all tests with JP, we used a simple profiling agent that is activated periodically (at the highest possible profiling granularity of $2^{31} - 1$), in order simply to integrate the MCT of each thread into a global MCT, and to reset some counters. This agent employs a JVM shutdown hook to generate the resulting profile in a file upon program termination. On average, the slowdown due to JP is a factor 4.2–5.3 for the DaCapo suite, 3.2–4.3 for the SPEC JVM98 suite, and 3.2–4.0 for SPEC JBB2005.

In Fig. 7 and 8 we evaluated JP for 2 different settings: with bytecode counting based on the 'Default BBA', and with bytecode counting using the 'Precise BBA'. In both settings, profiling of

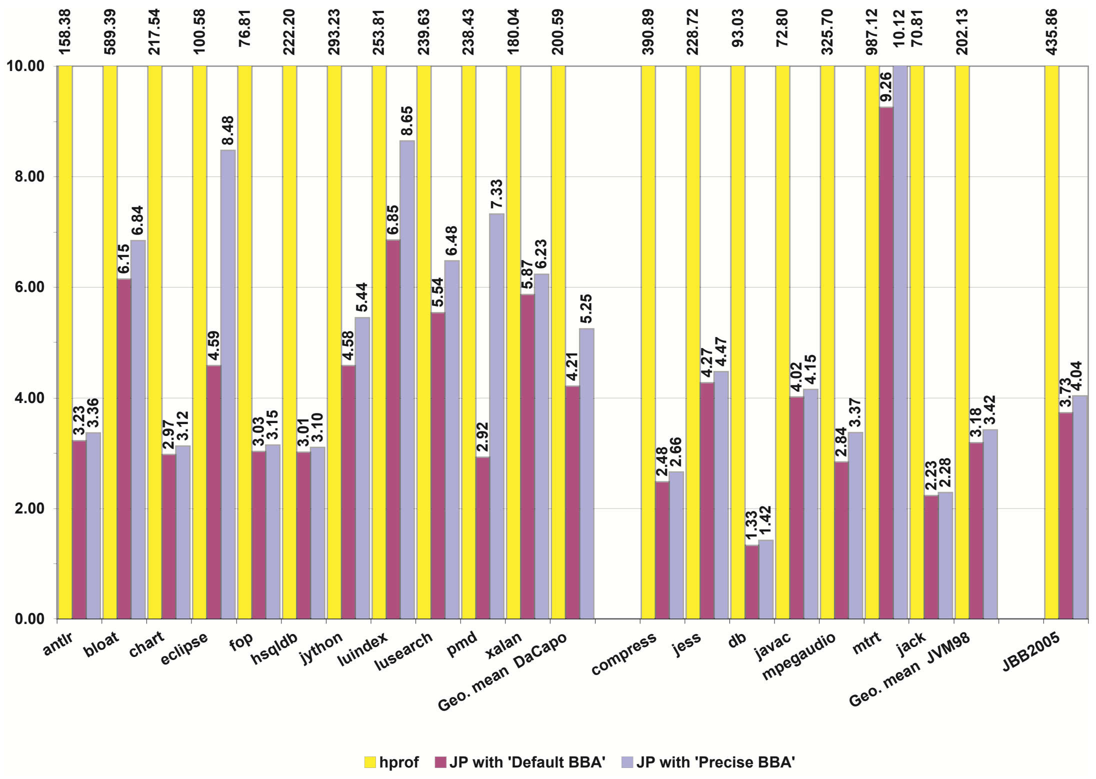

Figure 8. Profiling overhead (slowdown factor) for different profiler settings with Sun JDK 1.7.0 in server mode.

dynamic memory allocation was enabled and the profiling agent was activated exactly in the same way. We have experienced that the overhead caused by bytecode counting is relatively small compared to the overhead due to MCT creation.

We also tried to evaluate the extra time overhead due to the profiling of dynamic memory allocation. When memory profiling is enabled, the workload of the profiling agent increases slightly, since it has to compute the number of object allocations as explained in Section 5.1 and to process the data collected about array allocations. However, as we reduced the number of invocations of our profiling agent to a minimum in our evaluation (it only processed the profiling data upon program termination), this increased workload is negligible. Therefore, the main overhead in memory profiling comes from the instrumentation of array allocations. However, in the end, the extra overhead due only to memory profiling was not measurable in our setting.

Table I. Memory occupation by all IC instances.

	antlr	bloat	chart	fop	jython	luindex	lusearch	pmd	eclipse	xalan	hsqldb
#IC	555553	591237	24733	80007	1663347	21074	44268	769646	1870094	767972	127073
kB	21701	23095	966	3125	64974	823	1729	30064	73051	29999	4964

	compress	jess	db	javac	mpegaudio	mtrt	jack	JBB2005
#IC	2146	12063	5474	896256	3090	5803	49349	83295
kB	84	471	214	35010	121	227	1928	3254

7.2. Space Overhead

Besides the time overhead, we evaluated the extra memory occupied by the main profiling data structures. On our test platform, we found the IC instance to take up 40 bytes. This leads to Table I, which summarizes the number of IC instances created during one execution of the benchmarks, as well as the corresponding physical memory occupation, in kiloBytes. This amount represents the estimated size of the MCT created during one execution; it does not include any additional data structures introduced by customized profiling agents.

7.3. Precision of Basic Block Analysis

In order to measure the imprecision caused by the 'Default BBA', we compared profiles of the SPEC JVM98 benchmarks generated with the 'Default BBA' resp. with the 'Precise BBA' regarding the total number of bytecodes counted in all method invocation contexts ($b_{default}$ resp. $b_{precise}$). We measured the relative error δb as follows:

$$\delta b = \left(\frac{b_{default} - b_{precise}}{b_{precise}}\right) = \left(\frac{b_{default}}{b_{precise}} - 1\right)$$

For the SPEC JVM98 suite, we found the biggest relative error with 'jack', where δb is below 0.1%. The bigger relative error δb for 'jack' is not surprising, because 'jack' is known to be a particularly exception-intensive program [17, 39]. We conclude that in practice, the imprecision caused by the 'Default BBA' is minor.

7.4. Reproducibility of Profiles

For fully deterministic applications, profiles are reproducible as long as exactly the same environment – particularly the same Java class library – is used for profiling. However, in practice, many applications involve some non-determinism. For instance, when algorithms (e.g., hashtable operations) make use of the identity hashcodes of objects, they may follow different execution paths if the hashcodes vary between different runs of the program. Concurrency and unknown thread scheduling policies contribute to the observed non-determinism, too. Furthermore, on different platforms, distinct classes may be instantiated, depending e.g. on system properties; the use of different text line terminators in I/O is a well-known example.

We evaluated the reproducibility of SPEC JVM98 profiles collected on two different platforms. The first one is an Intel Pentium 4, 2.66 GHz, 1 GB RAM; Linux Fedora Core 2; Sun JDK 1.6.0_04, build b12, ‘server’ mode. The second platform is an Intel Core2 Duo 2.33Ghz, 2 GB RAM; Sun JDK 1.7.0, ‘early access’, build b25, ‘client’ mode.

We use an *overlap percentage* metric as in references [2, 26] to compare the profiles collected on the two platforms. Informally, the overlap is the amount of profiled information (weighted by execution frequency) that is present in both profiles. Two identical profiles have an overlap of 100%. For computing the overlap percentage, we use the bytecode counter in each calling context.

For ‘compress’, ‘jess’, ‘mpegaudio’, and ‘mtrt’, we measured an overlap percentage higher than 99.9%. However, for the other benchmarks, the overlap percentage was significantly lower: 69.3% for ‘db’, 95.3% for ‘javac’, respectively 91.5% for ‘jack’.

We analyzed the reasons for the observed low overlap percentages, and found that they were due to differences between the Java class libraries of JDK 1.6 and JDK 1.7. For ‘db’ and ‘jack’, the profile differences are caused by method `nextElement()` in the inner class `java.util.Vector$1`, which is a leaf method in JDK 1.6, but calls another method in JDK 1.7. Because this method is called extremely frequently when iterating through a vector, the difference between the two JDK versions causes a significant decrease in the overlap percentage. The ‘javac’ benchmark is also affected by this, in addition to another, identical JDK difference in method `digit(int, int)` of `java.lang.Character`.

8. DISCUSSION

In the following we discuss the strengths and limitations of our approach.

8.1. Benefits

Our goal has been to enable platform-independent and portable profiling in a virtual execution environment. The approach presented in this article, implemented by the Java profiler JP, is an important step into this direction. Instead of measuring CPU time, we compute the number of bytecodes that a program would execute without profiling. Hence, measurement perturbation is not an issue (at least for programs with deterministic thread scheduling) because the presence of measurements does not influence the measurement results, as it is the case with many existing profilers that may prevent optimizations in the virtual machine. Moreover, we also use platform-independent metrics to profile dynamic memory allocation. Further metrics can be derived from the dynamic bytecode metrics and the memory allocation metrics, such as e.g. the allocation density [23].

Another advantage of our approach is its portability. It can be implemented without resorting to any platform-specific features, as confirmed by our JP implementation: JP 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. JP has been successfully tested with several standard JVMs.

Our profiling framework can be customized by user-defined profiling agents, which themselves can be programmed in a completely platform-independent way. This is in contrast to standard profiling interfaces, such as the JVMPI [44] and the JVMTI [45], which require profiling agents to be implemented in native code.

We offer a simple but flexible API to implement a wide range of different profiling agents. A profiling agent can control the frequency of its periodic activation by adjusting the profiling granularity. The activation does not rely on the scheduling of the virtual machine (e.g., in the case of Java, scheduling is not well specified in the language and virtual machine specifications [27, 35]), because each thread in the system synchronously invokes the profiling agent after execution of a number of bytecodes that approximately corresponds to the current profiling granularity.

In our approach, each method invocation (with the exception of native methods) is profiled. This is in contrast to frequently used sampling techniques, where the profiler is activated only periodically (e.g., every few milliseconds). While sampling causes less overhead, it is not always accurate: We evaluated the overlap percentage [2] of a sampling profile produced by the 'hprof' profiling agent included in standard JDK distributions (setting '`cpu=samples,interval=1`') with a profile generated by 'hprof' in its exact profiling mode (setting '`cpu=times`'). For the SPEC JVM98 benchmarks [48], the average overlap percentage was below 7%. Our profiling approach allows one to significantly reduce the overhead of exact profiling (see measurements in Section 7) which means that it also becomes applicable in settings where sampling techniques had to be used because of long program execution times.

In a system composed of multiple software components, it is possible to selectively instrument only certain components of interest, thus reducing the overall overhead. However, classes that may be used by different components, such as core classes of the virtual machine, should always be instrumented.

8.2. Limitations

Concerning limitations, the major hurdle of our approach is that bytecode instrumentation does not cover the execution of native code. This is an inherent problem, since we rely on the transformation of bytecode and focus on the computation of platform-independent metrics. For programs that heavily depend on native code, dynamic bytecode metrics may not be relevant. We have nevertheless shown in [10] that Java programs spend on average less than 5% of their execution time inside native code (measurements made using Sun JDK 1.6.0 as platform, and SPEC JVM98 and SPEC JBB2005 as benchmarks).

A related problem with native code is that the MCT does not preserve the full call stack for instrumented methods invoked by native code, which appear as children nodes of the root node in the MCT. However, in practice this is not a serious problem, because these callbacks from native code to instrumented code are not frequent.

The JVM specification [35] imposes restrictions on the size of different parts of a class file. For instance, the size of the constant pool is represented by a 16 bit unsigned integer. The size of methods is limited, as well. Hence, the instrumentation may fail if these limits are exceeded after insertion of static fields, methods, or bytecodes. However, this problem is very unlikely to occur with normal, hand-crafted Java code.

Another limitation is that the introduction of extra method arguments may break existing code that relies on reflection. In Java, the methods `getConstructors()`, `getDeclared-Constructors()`, `getMethods()`, and `getDeclaredMethods()` of `java.lang.Class` return arrays of reflection objects (i.e., instances of the `Constructor` resp. `Method` classes of package `java.lang.reflect`), representing wrapper methods (with the unmodified signatures) as well as methods with our extended signatures. 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 arguments, resulting in an `IllegalArgumentException`.

JP solves this issue by patching the aforementioned methods of `java.lang.Class` to filter out the reflection objects that represent methods with extended signatures. This modification is straightforward, because in standard JDKs these methods are implemented in Java (and not in native code).

The static fields inserted by JP are also accessible through reflection, and may in principle break existing code. However, we have not yet encountered such a problem in practice. Regarding serialization, while static fields are excluded by default, it is possible to customize the serialization mechanism such that it fails in the presence of added static fields.

Our approach introduces wrapper methods for compatibility with native code. However, the invocations of wrapper methods constitute extra stack frames, and may therefore break code relying on stack introspection and assuming a particular invocation sequence. We solved this issue similarly to reference [9], by cancelling ‘reverse’ wrappers for certain native methods and using code duplication instead of wrapping for the affected callers of these methods.

During bootstrapping, the execution of instrumented code is prevented; as a consequence, methods that started, but did not finish their execution before the end of bootstrapping will continue executing their original method body even after bootstrapping. This is because the decision of executing either the instrumented or the original method body is taken exclusively upon method entry; there is no other similar conditional inside the instrumented code. We believe that this limitation is not important in practice, because bootstrapping is already over before the application’s main method is invoked; application code is therefore normally not affected.

As we try to preserve the full call stack without any depth limitation, the MCT may consume a significant amount of memory in the case of very deep recursions. According to Ball and Larus, path profiling (i.e., preserving exact execution history) is feasible for a large portion of programs [4], an observation that is confirmed by our benchmarks. Nevertheless, our approach may easily be modified to compute a Calling Context Tree of bounded depth [1] instead of a complete MCT.

9. RELATED WORK

In the following we review some related work regarding *bytecode manipulation* and *profiling*.

9.1. Bytecode Manipulation

Altering Java semantics via bytecode transformations is a well-known technique [46] and has been used for many purposes that can be generally characterized as adding reflection or aspect-orientedness to programs. When working at the bytecode level, the program source code is not needed.

There are many tools for manipulating JVM bytecode. The bytecode engineering library BCEL [21] represents method bodies as graph structures. Individual bytecode instructions are mapped to Java objects. JP is based on BCEL, because the graph representation of method bodies eases instrumentation at the basic block level.

ASM [38] is a lightweight bytecode manipulation framework designed for dynamic load-time transformation of Java class files. While the instrumentation process using ASM may in many cases be more efficient than using BCEL, we found that BCEL gives finer control on the generated code, and that its representation of method bodies is better suited for instrumentation at the basic block level.

Javassist [14, 15], which is used by JBoss [31], enables structural reflection and provides convenient source-level abstractions.

Soot [49] is a framework for analyzing and transforming JVM bytecode that offers four intermediate code representations. For instance, *Jimple* is a typed, stack-less, three-address code intermediate represention. Soot is often used for bytecode optimization.

JOIE [18], JikesBT [30], and Serp [5] are further examples of bytecode manipulation libraries implemented in Java.

Some tools for aspect-oriented programming in Java, e.g. AspectJ [32], work at the bytecode level as well. However, usually such tools support only higher-level pointcuts, such as method invocations, whereas our collection of bytecode metrics requires transformations at the basic block level. Furthermore, the extension of method signatures, which we found essential to efficiently compute thread-local, calling context-sensitive profiling data, is usually not supported by aspect-oriented programming tools.

The use of AspectJ for profiling is explored in [41], and reported to yield mixed results. The AspectJ language itself lacks a number of joinpoints, such as for intercepting array allocations, that would be necessary for a complete coverage. From a performance perspective, it is difficult to compare their approach with ours, as they chose to adopt sampling-based profiling instead of exact profiling.

A salient disadvantage of using current aspect-oriented languages or tools is that they are, to the best of our knowledge, unable to process core libraries of the JDK, a decisive factor being that such tools introduce difficult-to-circumvent dependencies with their own runtime libraries, which prevent the JVM from bootstrapping.

9.2. Dynamic Metrics and Profiling

In [23] the authors present a variety of dynamic metrics, including bytecode metrics, for selected Java programs, such as the SPEC JVM98 benchmarks [48]. They introduce a tool called *J [24] for the metrics computation. *J relies on JVMPI [34, 44], which is known to cause very high measurement overhead (see Section 7) and requires profiling agents to be written in native code, contradicting the Java motto ‘write once, run anywhere’. Because of the high overhead, tools like *J may only be applied to programs with a short execution time. In contrast, our approach reduces the overhead and can be implemented in pure Java. Therefore, it is possible to instrument real applications in a way that is portable across different virtual execution environments.

There is a large body of related work in the area of profiling. Fine-grained instrumentation of binary code has been used for profiling by Ball and Larus [3, 33]. The ATOM framework [43] has been successfully used for many profiling tools that instrument binary code. However, as binary code instrumentation is inherently platform-dependent, this technique is not appropriate to build tools for the platform-independent performance analysis of software components.

The NetBeans Profiler (`http://profiler.netbeans.org/`) integrates Sun’s JFluid profiling technology [22] 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.

Whereas our approach is rather intended for use at development time, sampling-based profiling is often employed in already deployed systems as support for feedback-directed optimizations in dynamic compilers [2, 50]; indeed, sampling-based profilers may yield a sufficiently low overhead to become usable at production time. The framework presented in [2] 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 low overhead, as execution proceeds most of the time inside the lightly instrumented code portions.

In reference [8] we introduced a portable sampling profiler for the JVM based on bytecode counting. A profiling agent is periodically invoked in a deterministic way after the execution of a certain number of bytecodes (sampling interval). The profiling data structure generated by the sampling profiler can be regarded as a partial MCT, covering only a subset of the calling contexts. While this data structure allows us to accurately estimate the relative distribution of executed bytecodes in different calling contexts, it completely lacks method invocation counters as well as memory allocation statistics. The advantage of the sampling profiler is its lower overhead.

ProfBuilder [19] is a toolkit for building Java profilers. However, ProfBuilder does not address issues regarding multi-threading and native code, and the generated profiling tools described in [19] cause high overhead. The authors of ProfBuilder show with a case study that profiles based on dynamic bytecode metrics are valuable to detect algorithmic inefficiencies and help the developer focus on those parts of a program that suffer from high algorithmic complexity.

10. CONCLUSION

In this article we presented a novel approach for platform-independent, portable, and customizable profiling in a virtual execution environment. We rely on program transformations at the bytecode level in order to compute a calling-context-sensitive profiling data structure that collects platform-independent dynamic metrics, such as the number of method invocations, the number of executed bytecodes, as well as statistics on memory allocation. In addition to platform independence, our approach ensures largely reproducible results, minimized measurement perturbation, and largely reduced overhead.

We implemented our approach as the Java profiler JP. JP, as well as user-defined profiling agents, are programmed in pure Java. Thanks to its ability to perform dynamic instrumentation, JP is, to the best of our knowledge, the first Java profiling tool to ensure full coverage of executed bytecodes. Our evaluation confirms that JP causes significantly less overhead than hprof, a prevailing profiler based on a standard JVM profiling interface.

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. 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.
3. T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

4. T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.
5. BEA. Serp. Web pages at `http://serp.sourceforge.net/`.
6. W. Binder. A portable and customizable profiling framework for Java based on bytecode instruction counting. In *Third Asian Symposium on Programming Languages and Systems (APLAS 2005)*, volume 3780 of *Lecture Notes in Computer Science*, pages 178–194, Tsukuba, Japan, Nov. 2005. Springer Verlag.
7. W. Binder. Portable profiling of memory allocation in Java. In *Net.ObjectDays 2005 (NODe 2005)*, volume P-69 of *Lecture Notes in Informatics*, pages 110–128, Erfurt, Germany, Sept. 2005.
8. W. Binder. Portable and accurate sampling profiling for Java. *Software: Practice and Experience*, 36(6):615–650, 2006.
9. W. Binder and J. Hulaas. Extending standard Java runtime systems for resource management. In *Software Engineering and Middleware (SEM 2004)*, volume 3437 of *LNCS (Lecture Notes in Computer Science)*, pages 154–169, Linz, Austria, Sept. 2004.
10. W. Binder, J. Hulaas, and P. Moret. A quantitative evaluation of the contribution of native code to Java workloads. In *2006 IEEE International Symposium on Workload Characterization (IISWC-2006)*, San Jose, CA, USA, Oct. 2006.
11. 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).
12. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, Oct. 2006. ACM Press.
13. A. Camesi, J. Hulaas, and W. Binder. Continuous Bytecode Instruction Counting for CPU Consumption Estimation. In *QEST 2006 (3rd International Conference on the Quantitative Evaluation of SysTems), 11-14 September 2006*, Riverside, California, USA, 2006. IEEE Computer Society Press.
14. S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer Verlag, Cannes, France, June 2000.
15. S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. *Lecture Notes in Computer Science*, 2830:364–376, 2003.
16. 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.
17. M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. *ACM SIGPLAN Notices*, 35(5):13–26, May 2000.
18. G. Cohen, J. Chase, and D. Kaminsky. Automatic program transformation with JOIE. In *1998 USENIX Annual Technical Symposium*, pages 167–178, 1998.
19. B. F. Cooper, H. B. Lee, and B. G. Zorn. ProfBuilder: A package for rapidly building Java execution profilers. Technical Report CU-CS-853-98, University of Colorado at Boulder, Department of Computer Science, Apr. 1998.
20. G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33, 10 of *ACM SIGPLAN Notices*, New York, USA, Oct. 1998.
21. M. Dahm. Byte code engineering. In *Java-Information-Tage 1999 (JIT'99)*, Sept. 1999. `http://jakarta.apache.org/bcel/`.
22. 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.
23. B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, Nov. 2003.
24. B. Dufour, L. Hendren, and C. Verbrugge. *J: A tool for dynamic analysis of Java programs. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 306–307, New York, NY, USA, 2003. ACM Press.
25. M. Feeley. Polling efficiently on stock hardware. In *the 1993 ACM SIGPLAN Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 179–187, June 1993.
26. P. Feller. Value profiling for instructions and memory locations. Master Thesis CS1998-581, University of California, Sa Diego, Apr. 1998.
27. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java language specification*. Java series. Addison-Wesley, Reading, MA, USA, second edition, 2000.
28. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, 2005.

29. D. Gregg, J. Power, and J. Waldron. Platform independent dynamic java virtual machine analysis: the java grande forum benchmark suite. *Concurrency and Computation Practice and Experience*, 15(3–5):459–484, 2003.
30. IBM. Jikes Bytecode Toolkit. Web pages at `http://www.alphaworks.ibm.com/tech/jikesbt`.
31. JBoss. Open source middleware software. Web pages at `http://www.jboss.com/`.
32. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP-2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001.
33. J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software–Practice and Experience*, 24(2):197–218, Feb. 1994.
34. 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.
35. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.
36. Microsoft. Microsoft .NET Framework Developer Center. Web pages at `http://msdn.microsoft.com/netframework/`.
37. NetBeans. The NetBeans Profiler Project. Web pages at `http://profiler.netbeans.org/`.
38. ObjectWeb. ASM. Web pages at `http://asm.objectweb.org/`.
39. T. Ogasawara, H. Komatsu, and T. Nakatani. A study of exception handling and its dynamic optimization in Java. *ACM SIGPLAN Notices*, 36(11):83–95, Nov. 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).
40. M. P. Papazoglou and D. Georgakopoulos. Introduction: Service-oriented computing. *Communications of the ACM*, 46(10):24–28, Oct. 2003.
41. D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, June 2007.
42. Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of java workloads: a structured view and opportunities for optimizations. In *SIGMETRICS 2001/Performance 2001*, pages 194–205, Cambridge, MA, June 2001.
43. A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, 2004.
44. Sun Microsystems, Inc. Java Virtual Machine Profiler Interface (JVMPI). Web pages at `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/`.
45. Sun Microsystems, Inc. JVM Tool Interface (JVMTI), Version 1.0. Web pages at `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/`.
46. E. Tanter, M. Ségura-Devillechaise, J. Noyé, and J. Piquer. Altering Java semantics via bytecode manipulation. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002), USA*, volume 2487 of *LNCS*, pages 283–298, Oct. 2002.
47. The Standard Performance Evaluation Corporation. SPEC JBB2005 (Java Business Benchmark). Web pages at `http://www.spec.org/osg/jbb2005/`.
48. The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Web pages at `http://www.spec.org/osg/jvm98/`.
49. R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.
50. 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.

Appendix A – INITIALIZATION WITH DIFFERENT CONSTRUCTORS

The computation of the number of object allocations in a constructor (Section 5.1) is based on the assumption that each constructor (except the constructor of `java.lang.Object`) has exactly one associated alternate or superclass constructor. This assumption is backed by the Java Language Specification [27]. However, at the bytecode level, a constructor may invoke a different alternate or superclass constructor depending on its arguments without causing any verification error [35]. The example in Fig. 9 illustrates this. The main method allocates two objects of the same type and initializes them with the same constructor (but passing different constructor arguments). However, the alternate constructor, which takes no arguments, is invoked only once. The output of the program is '`count = 1`'.

Fortunately, this kind of situation only occurs with hand-crafted bytecode. If a standard Java compiler is used to generate bytecode from Java code, such bytecode is not created. Nonetheless, the static analyzer that examines constructor code (see Section 5.1) is able to detect this situation and produces a warning.

```
public class DifferentConstructors {
   static int count = 0;

   public static void main(String[] args) {
      new DifferentConstructors(false);
      new DifferentConstructors(true);
      System.out.print("count = " + count);
   }

   DifferentConstructors(boolean x) {
      // The following is manually crafted bytecode.
      // Depending on the argument, the alternate or
      // the superclass constructor is called:
      aload_0
      iload_1
      ifeq superclassConstructor
alternateConstructor:
      invokespecial DifferentConstructors/<init>()V
      return
superclassConstructor:
      invokespecial java/lang/Object/<init>()V
      return
   }

   DifferentConstructors() {
      super();
      ++count;
   }
}
```

Figure 9. Depending on the constructor argument, the alternate or the superclass constructor is invoked.