Portable Profiling of Memory Allocation in Java

Walter Binder

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

walter.binder@epfl.ch

Abstract: This paper presents a portable framework to profile memory allocation in standard Java Virtual Machines. We extended our exact profiler JP, which generates a method call tree (MCT) for each thread in the system, in order to store information concerning object allocation in the MCT. Our primary design goals were to use platform-independent metrics for memory allocation and to minimize the extra overhead caused by memory profiling. For each method invocation context, the profiler preserves the number of allocated objects of each type. We exploit the fact that an object allocation is usually followed by a constructor invocation, in order to compute the number of object allocations from the MCT. A static analysis of constructor code allows to distinguish between the invocation of an alternate or superclass constructor and object allocation within the constructor. Arrays are treated specially, as we instrument array allocation instructions in order to preserve accumulated information on the type, number, and size of allocated arrays for each method invocation context. A performance evaluation shows that the extra overhead due to memory profiling is small.

Keywords: Java, memory profiling, profiling metrics, program transformations, bytecode instrumentation

1 Introduction

Memory profiling is an essential tool to analyze, characterize, and understand the memory behaviour of programs. Automatic memory management in Java [GJSB00] (by means of garbage collection) eases software development. Memory allocation is ubiquitous in Java programs, and therefore a big part of the program execution time may be spent in memory allocation and garbage collection. In order to help developers detect and analyze such performance issues, profiling tools are needed that compute statistics concerning the memory allocation behaviour of programs.

There are many profiling tools for the Java Virtual Machine (JVM) [LY99], most of them are based on the Java Virtual Machine Profiling Interface (JVMPI) [Suna] or on the JVM Tool Interface (JVMTI) [Sunb]. JVMPI is a set of hooks to the JVM which signals interesting events, such as object allocations. Its successor, the JVMTI, provides additional facilities for bytecode instrumentation.

However, most prevailing profilers based on the JVMPI or on the JVMTI cause excessive overhead when they are used to create exact profiles, i.e., tracking each method invocation, each object allocation, etc. E.g., for some benchmarks we measured a slowdown of more than factor 4 800 (!) when using the standard ‘hprof’ profiling agent (which is based on the JVMTI in Sun JDK 1.5.0) in its exact profiling mode. Moreover, profiling agents using the JVMPI or the JVMTI have to be written in native code, contradicting the Java motto ‘write once and run anywhere’.

We implemented an exact profiler for Java, called JP, which relies neither on the JVMPI nor on the JVMTI, but directly instruments the bytecode of Java programs in order to generate exact profiles [BH05]. JP creates a method call tree (MCT) for each thread in the system, which exposes the number of method invocations with the same call stack, as well as the number of bytecode instructions executed in each calling context. Using the number of executed bytecode instructions as platform-independent profiling metric has several advantages, such as making profiles reproducible (for deterministic programs) and directly comparable across different machines. JP is written in pure Java and fully portable, it has been successfully tested on several recent JVMs. Furthermore, JP supports user-defined profiling agents, which may be written in pure Java as well. The custom profiling agents are triggered periodically in a deterministic way (based on the number of executed bytecode instructions), in order to process the generated profiles (e.g., to integrate the MCT of each thread into a global MCT, to compute continuous metrics [DDHV03], etc.).

In this paper we describe an extension of JP to profile memory allocation. As platform-independence has been a major goal in the design of JP, the memory profiling must not compromise this feature. Hence, we do not directly profile the number of allocated bytes, but we keep track of the type and number of object allocations. In order to minimize the overhead (time and space) due to memory profiling, we compute this information directly from the MCT. Therefore, profiling object allocation does not require any additional instrumentation, and the MCT data structure need not be extended either. However, this approach works only to profile the allocation of objects that are not arrays. For array allocations, extra bytecode instrumentation is needed, in order to preserve the array element type, the number of allocated arrays, and the total size of the allocated arrays. A performance evaluation shows that our JP extension for memory profiling does not cause much extra overhead.

The main contributions of this paper are the presentation of the MCT using abstract datatypes, the introduction of platform-independent profiling metrics, a technique to compute the number of object allocations from the MCT, and an algorithm to instrument array allocations. Moreover, we examine the limitations of our approach in detail.

This paper is structured as follows: Section 2 presents the MCT data structure created by JP. In Section 3 we introduce our platform-independent metrics to profile memory allocation. Section 4 explains how to compute the number of object allocations from the information already present in the MCT. In Section 5 we detail the instrumentation of array allocations. Section 6 discusses issues regarding the accuracy of the generated profiles and limitations of our approach. In Section 7 we evaluate the overhead of our profiling scheme and show that the extra overhead due to memory profiling is small. Section 8 discusses related work, while Section 9 summarizes the benefits of our profiling approach. Finally, the appendix at the end of this paper shows how hand-crafted bytecode may circumvent certain restrictions usually enforced by the JVM or violate assumptions that hold for compiled Java code.

2 Method Call Tree (MCT)

JP rewrites JVM bytecode so that each thread in the system creates a method call tree (MCT), where each node represents all invocations of a particular method with the same call stack. The parent node in the MCT corresponds to the caller, the children nodes correspond to the callees. The root of the MCT represents the caller of the main method. With the exception of the root node, each node in the MCT stores profiling information for all invocations of the corresponding method with the same call stack. Concretely, JP stores the number of method invocations as well as the number of bytecode instructions executed by the corresponding method, excluding the number of bytecode instructions executed by callee methods (each callee has its own node in the MCT). In this paper we extend the MCT with memory allocation statistics.

The MCT is similar to the Calling Context Tree (CCT) [ABL97]. 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 [BL96], path profiling (i.e., preserving exact execution history) is feasible for a large portion of programs.

A detailed presentation of the MCT implementation is not in the scope of this paper. An abstract description of certain operations supported by the MCT is sufficient for the discussion in the following sections. For instance, we leave out all the details on bytecode instruction counting. We define two abstract datatypes to represent a MCT, the method identifier `MID` and the method invocation context `IC`. A method invocation context is a node in the MCT. In the following description, we assume the existence of the types `INTEGER`, `STRING`, and `THREAD`, as well as the possibility to create aggregate types (`SET OF`).

The method identifier `MID` offers the following operations:

- `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(`*class*`,` *x*`,` *y*`))` = *class*.

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

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

The method invocation context `IC` supports the following operations:

- `getOrCreateRoot(THREAD` t`): IC`
  Returns the root node of a thread's MCT. If it does not yet exist, it is created.

- `profileCall(IC` $caller$`, MID` $callee$`): IC`
  Registers a method invocation in the MCT. The returned IC represents the callee method, identified by $callee$. It is a child node of $caller$ in the MCT.

- `getCaller(IC` $callee$`): IC`
  Returns the caller `IC` of $callee$. It is the parent node of $callee$ in the MCT.
  `getCaller(profileCall(`$caller$`,` x`))` = $caller$.
  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.

- `getCalls(IC` c`): INTEGER`
  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.

- `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 Memory Allocation Metrics

The design of our profiler JP focuses on portability and platform-independence. JP does not rely on any platform-specific features in order to offer a completely portable profiling system that allows developers to profile their applications in their preferred environment, generating reproducible and directly comparable profiles. JP exploits the number of executed bytecode instructions as platform-independent profiling metric [DDHV03]. When we extended JP to profile memory allocation, we tried to use platform-independent metrics for memory allocation, too.

Most profilers use the number of allocated bytes as measurement unit. However, this metric depends on the particular JVM in use (size of references, alignment, object representation in memory, etc.).

In contrast, JP tracks the number of allocated objects of each type, for each method invocation context. I.e., object allocations are described by triples of the form $\langle$meth. invoc. 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 4, it is possible to compute this object allocation metric directly from the MCT without any additional instrumentation. Consequently, there is no extra overhead in the creation of the MCT (concerning execution time and the amount of memory required to store the MCT).

Concerning array allocation, we preserve the element type, the number of allocated arrays, and the total number of array elements for each method invocation context. I.e., array allocations are described by 4-tuples of the form $\langle$meth. invoc. context, element type, number of arrays, number of array elements$\rangle$. The element type may be one of the 8 basic types in Java (`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 JVM 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.[1] However, the instrumentation may change the thread scheduling of the JVM. Hence, the profiling may affect the progress of the different threads in a multi-threaded program. I.e., continuous metrics [DDHV03] may be distorted by the profiling.

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 Java, because of the automatic memory management (garbage collection). I.e., there are no explicit de-allocation sites in the bytecode.

JP supports the periodic activation of a user-defined profiling agent. The interval between consecutive activations of the profiling agent is customized by the agent itself. Hence, the agent may generate continuous statistics on the progress of memory allocation of a program during its execution. Moreover, the agent may combine the memory allocation metrics provided by JP with other metrics. E.g., the memory allocation metrics may be put in relation to the number of executed bytecode instructions 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 1000 executed bytecode instructions [DDHV03].

[1] In contrast, a metric such as the total amount of memory in use would be seriously perturbated by the measurement, since the MCT data structure itself may consume a large amount of memory.

4 Profiling Object Allocation

In this section we describe our approach to profile allocations of objects that are not arrays. Arrays are addressed in Section 5.

At the Java level, objects are allocated and initialized with 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.[2] `invokespecial` 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*. I.e., 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, the extra overhead due to such an instrumentation may be non-negligible. Therefore, we chose a different approach, taking advantage of the MCT that is already created by JP. 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 [LY99], 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[3] (see [LY99], 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. I.e., we do not directly profile object allocations, but we compute the number of allocated objects from the number of constructor invocations.

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`, invokes either an alternate or a superclass constructor in the beginning. I.e., 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 Figure 1, 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 invoca-

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

[3] In Appendix A at the end of this paper we show that there are cases where recent JVMs do not prevent method invocations on uninitialized objects.

```
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 1: Constructor example.

tion 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 rewriting, 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 [LY99], 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.

Our rewriting tool 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 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` → `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 Figure 1, $\mathcal{M}$(`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 B. 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.

Figure 2 explains how to compute the number of object allocations from the information

$$\texttt{getAlloc(IC } c\texttt{)=} \begin{cases} \left(\displaystyle\sum_{\substack{x\in\texttt{getCallees}(c),\\ mid_x=\texttt{getMID}(x),\\ \texttt{getName}(mid_x)=\texttt{<init>}}} \texttt{getCalls}(x)\right) - \texttt{getCalls}(c) & \begin{array}{l}\text{if } mid_c\texttt{=getMID(}c\texttt{)} \land \\ \quad \texttt{getName(}mid_c\texttt{)=<init>} \land \\ \quad \texttt{getClass(}mid_c\texttt{)} \neq \\ \qquad\qquad \texttt{java.lang.Object}\end{array} \\[2em] \displaystyle\sum_{\substack{x\in\texttt{getCallees}(c),\\ mid_x=\texttt{getMID}(x),\\ \texttt{getName}(mid_x)=\texttt{<init>}}} \texttt{getCalls}(x) & \text{otherwise} \end{cases}$$

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

$$\begin{array}{l}\texttt{getAlloc(IC } c\texttt{,}\\ \quad\texttt{STRING } class\texttt{)=}\end{array} \begin{cases} \left(\displaystyle\sum_{\substack{x\in\texttt{getCallees}(c),\\ mid_x=\texttt{getMID}(x),\\ \texttt{getName}(mid_x)=\texttt{<init>},\\ \texttt{getClass}(mid_x)=class}} \texttt{getCalls}(x)\right) - \texttt{getCalls}(c) & \begin{array}{l}\text{if } mid_c\texttt{=getMID(}c\texttt{)} \land \\ \quad \texttt{getName(}mid_c\texttt{)=<init>} \land \\ \quad \texttt{getClass(}mid_c\texttt{)} \neq \\ \qquad\qquad \texttt{java.lang.Object} \land \\ \quad \texttt{getClass(}\mathcal{M}(mid_c)\texttt{)} = class\end{array} \\[2em] \displaystyle\sum_{\substack{x\in\texttt{getCallees}(c),\\ mid_x=\texttt{getMID}(x),\\ \texttt{getName}(mid_x)=\texttt{<init>},\\ \texttt{getClass}(mid_x)=class}} \texttt{getCalls}(x) & \text{otherwise} \end{cases}$$

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

Figure 2: Computing the number of allocated objects based on the number of constructor invocations.

stored in the MCT. `getAlloc(IC)` (Figure 2 (a)) returns the total number of objects allocated by a given method invocation context c. If c does not correspond to a constructor (or corresponds to the constructor of `java.lang.Object`), `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`), the sum has to be reduced by `getCalls(`c`)`, because each time the constructor corresponding to c is invoked, it calls its associated alternate or superclass constructor $\mathcal{M}$`(getMID(`c`))` once without allocating an object. Note that the computation of `getAlloc(IC)` does not require the map $\mathcal{M}$. `getAlloc(IC, STRING)` (Figure 2 (b)) returns the number of objects of a certain type $class$ allocated by a given method invocation context c. In contrast to `getAlloc(IC)`, `getAlloc(IC, STRING)` differentiates between the invocations of constructors of different classes.

5 Profiling Array Allocation

As array allocation does not involve any method/constructor invocation, the approach presented in Section 4 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. This also requires an extension of the MCT. We add the following operations to the method invocation context `IC`:

- `profileArrays(IC` c`, TYPE` t`,`
  `INTEGER` $arrays$`, INTEGER` $elements$`): IC`
  Registers array allocations in c. t is the element type of the arrays, it 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 [LY99]. The element type `R` indicates that the array stores references to objects, which may be instances of 'normal' classes or 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, INTEGER, INTEGER)` returns c, after its array allocation statistics have been updated accordingly. This operation is not supported for the root of the MCT.

- `getArrays(IC` c`, TYPE` t`): INTEGER`
  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`): INTEGER`
  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.

The bytecode instructions newarray<*type*>, anewarray<*type*>, and multianewarray<*type*><*allocDim*> are used 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. In Java and in the JVM, 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. The dimensionality of the array $arrayDim$ is encoded in the array type descriptor (*type*) [LY99]; $allocDim \leq arrayDim$. 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, INTEGER, INTEGER):[4]

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

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

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.
   - `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, INTEGER, INTEGER)`:

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

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

Figure 3 illustrates the profiling of the allocation of multi-dimensional arrays with several examples.

```
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 3: Examples: Profiling the allocation of multi-dimensional arrays.

At the implementation level, the inserted bytecode sequence to profile the allocation of a multi-dimensional array is generated according to the algorithm in Figure 4. 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).

```
1. Save array dimensions (provided on the stack) in unused local variables. (As an optimization, the first array dimension can remain on the stack.)
2. Allocate unused 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 4: Algorithm to instrument allocations of multi-dimensional arrays.

6 Accuracy of Profiles

In this section we discuss the accuracy of the profiling scheme presented in Section 4 and in Section 5. We discuss to which extend and under which conditions the generated memory allocation profiles are accurate.

The most severe limitation of our approach is that it cannot profile the execution of native code. This is an inherent problem of our profiling scheme, since it relies on the instrumentation of Java code. Therefore, the MCT does not cover any method invocation context that would correspond to a native method. Consequently, for a program that heavily depends on native code, the generated MCT is incomplete and may be misleading.

Nonetheless, as constructors cannot be native, the MCT covers all constructor invocations. I.e., 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. A minor limitation of our current implementation is that the call stack is not preserved when a native method invokes Java code. All Java methods/constructors invoked by native code appear as children nodes of the root node in the MCT, i.e., as siblings of the main method. However, in practice this is not a big problem, because these callbacks from native code to instrumented Java code are not frequent.

Concerning the allocation of objects that are not arrays, the approach described in Section 4 tracks the allocation of all objects that are correctly initialized (i.e., the constructor returns normally). If an 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 is illustrated in Appendix A at the end of this paper: 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 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 Figure 2 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 exceptions in constructors 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 address this issue, we could insert conditionals in the profiling code in order to skip the invocation of `profileArrays(IC, TYPE, INTEGER, INTEGER)` 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, INTEGER, INTEGER)` after the array allocation so that they are skipped in the case of an exception. This requires a slight modification of the rewriting algorithm presented in Figure 4, as the integer arguments of `profileArrays(IC, TYPE, INTEGER, INTEGER)` have to be stored either on the stack (below the sizes of the array dimensions) or in local variables.

7 Performance Evaluation

To evaluate the overhead caused by our memory profiling scheme, we ran the SPEC JVM98 benchmark suite [The] on a Windows XP computer (Intel Pentium 4, 2.4 GHz, 512 MB RAM).[5] In order to obtain reproducible results, all benchmarks were run under the same conditions on a very lightly loaded system. For all settings, the entire JVM98

[5] Note that the results presented in this paper are not directly comparable with previous results in our technical report [BH05], as they were collected on a Linux system.

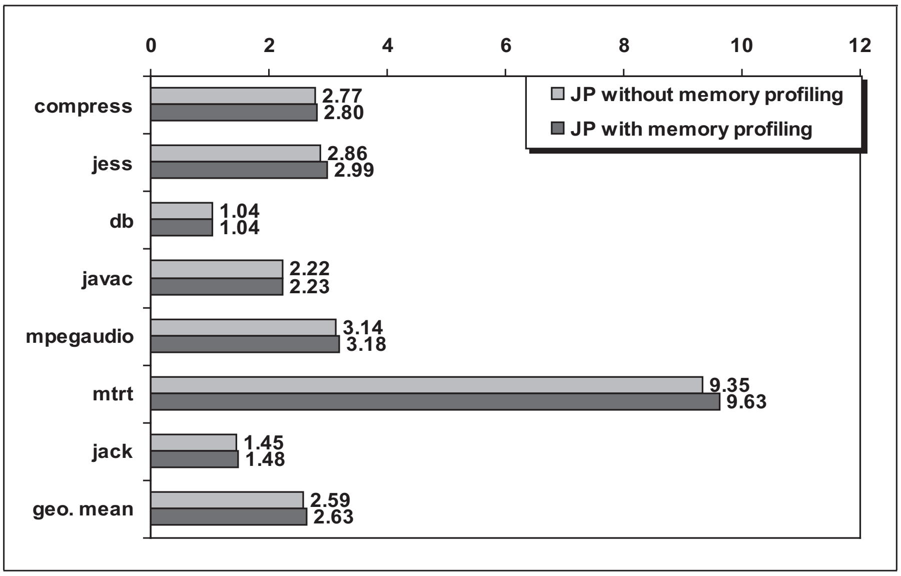

Figure 5: Profiling overhead (slowdown factor) for 2 different profiler settings.

benchmark suite (consisting of several sub-tests) was run 10 times, and the final results were obtained by calculating the geometric mean of the median of each sub-test. Here we present the measurements made with the Sun JDK 1.5.0_01 platform in its 'server' mode.

Figure 5 shows the profiling overhead (as a slowdown factor of $\frac{\text{execution time with profiling}}{\text{execution time without profiling}}$) for two settings: JP without memory profiling versus JP with memory profiling. The setting without memory profiling corresponds to the previous version of JP, which computes a MCT for each thread in the system, including the number of bytecode instructions executed in each method invocation context. In the setting with memory profiling, JP also preserves statistics concerning array allocations in the MCT.

For both measurements we used a simple profiling agent which periodically integrates the MCT of each thread into a global MCT, which is written into a file upon program termination (using a shutdown hook). If memory profiling is enabled, the workload of the profiling agent increases, since it has to compute the number of object allocations as explained in Section 4 and to process the collected data concerning array allocations. However, as we reduced the number of invocations of the custom profiling agent to a minimum in our evaluation, the extra overhead due to the increased workload of the profiling agent is negligible. The extra overhead because of memory profiling, as shown in Figure 5, is due to the instrumentation of array allocations. As we expected, the extra overhead is small for all benchmarks (4% on average), compared with the overhead caused by the creation of the MCT and tracking the number of executed bytecode instructions.

While for some programs (e.g., 'mtrt') the slowdown due to exact profiling may be as high as factor 10, the overhead caused by JP is still 1–2 orders of magnitude lower than the overhead caused by the standard 'hprof' profiling agent in its exact profiling mode (setting '`cpu=times`'), which exceeds factor 4 800 for the 'mtrt' benchmark.

8 Related Work

Fine-grained instrumentation of binary code has been used for profiling in prior work [BL94, LB94]. In contrast, all profilers based on a fixed set of events like the one provided by JVMPI [Suna] are restricted to traces at the granularity of the method call. This restriction also exists with JP and is justified by the fact that object-oriented Java programs tend to have shorter methods, with simpler internal control flows than code implemented in traditional imperative languages.

Some Java profilers take advantage of bytecode instrumentation to be less obtrusive and to enable the JVM to function at full speed. The JVMPI, on which many commercial (e.g., JProbe[6]) and academic (e.g., JPMT [HQGvdM02]) profilers are based, has been replaced by the JVMTI [Sunb] in JDK 1.5.0. The JVMTI has built-in bytecode instrumentation facilities in order to let profiling agents implement customized, less disruptive profiler events. Profiling agents based on the JVMTI still have to be written in native code.

JDK 1.5.0 also provides services that allow Java programming language agents to instrument programs running on the JVM. Java agents are specified with the '`-javaagent`' command line option and exploit the instrumentation API (package `java.lang.instrument`) to install bytecode transformations. Java agents are invoked after the JVM has been initialized, before the real application. They may even redefine the already loaded system classes. However, JDK 1.5.0 imposes several restrictions on the redefinition of previously loaded classes.

The NetBeans Profiler[7] integrates Sun's JFluid profiling technology [Dmi04] 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.

JRes [CvE98] is a resource accounting and control system for Java, which takes CPU, memory, and network resource consumption into account. For its implementation, JRes does not need any modification of the JVM, but relies on a combination of bytecode rewriting and native code libraries. For memory accounting, JRes uses bytecode instrumentation, but still needs the support of a native method to account for memory occupied by array objects. JRes uses the number of allocated bytes as metric.

J-SEAL2 [Bin01, BHV01], an extended and improved version of JavaSeal [VBB98] that is also compatible with the Seal Calculus [VC99], adds resource management features to Java, too. In contrast to JRes, J-SEAL2 is written in pure Java, it does not rely on native code. J-SEAL2 tries to keep track of the active memory hold by each component in the

[6] `http://www.quest.com/jprobe/`

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

system. It uses an estimation of the number of allocated bytes as metric. J-SEAL2 keeps a weak reference to each allocated object in order to notice when the object is reclaimed by the garbage collector. Neither JRes nor J-SEAL2 computes a MCT. They maintain only a single counter of the number of allocated bytes for a component or a group of threads. Hence, neither of them is suited for call-path-sensitive memory profiling.

9 Conclusion

JP is a novel, exact profiling framework for Java that is completely based on program transformations at the bytecode level. It is portable and fully compatible with any standard JVM and allows custom profiling agents to be written in pure Java. JP computes a MCT for each thread to store call-path-sensitive profiling information, such as the number of method invocations and the number of executed bytecode instructions for each calling context. Because JP uses platform-independent metrics, profiles are reproducible (for deterministic programs). In all our measurements, JP causes 1–2 orders of magnitude less overhead than prevailing exact profilers for Java.

In this paper we presented an extension of JP to track memory allocation using platform-independent metrics: For objects that are not arrays, we preserve the type and the number of allocated instances. For arrays, we store the element type, the number of arrays, and the total size of the arrays. From the profiling information generated by JP various other metrics can be derived, such as the average size of allocated objects, the allocation density, as well as continuous metrics.

In order to reduce the extra overhead due to memory profiling to a minimum, we compute the number of object allocations directly from the number of constructor invocations in the MCT. A static analysis of constructor code allows us to correctly calculate the number of objects allocated by constructors. As this approach works only for objects that are not arrays, additional bytecode instrumentation is needed to preserve statistics concerning array allocation in the MCT. A performance evaluation shows that the extra overhead due to memory profiling is negligible.

References

[ABL97] Glenn Ammons, Thomas Ball, and James 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.

[BH05] Walter Binder and Jarle Hulaas. Exact and Portable Profiling for Java Using Bytecode Instruction Counting. Technical Report EPFL-IC-2005011, Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, March 2005.

[BHV01] Walter Binder, Jarle G. Hulaas, and Alex Villazón. Portable resource control in Java.

ACM SIGPLAN Notices, 36(11):139–155, November 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[Bin01] Walter Binder. Design and Implementation of the J-SEAL2 Mobile Agent Kernel. In *The 2001 Symposium on Applications and the Internet (SAINT-2001)*, pages 35–42, San Diego, CA, USA, January 2001.

[BL94] Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[BL96] Thomas Ball and James R. Larus. Efficient Path Profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[CvE98] Grzegorz Czajkowski and Thorsten 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, October 1998.

[DDHV03] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, November 2003.

[Dmi04] Mikhail 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.

[GJSB00] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java language specification*. Java series. Addison-Wesley, Reading, MA, USA, second edition, 2000.

[HQGvdM02] M. Harkema, D. Quartel, B. M. M. Gijsen, and Rob van der Mei. Performance monitoring of Java applications. In *Proceedings of the 3rd International Workshop on Software and Performance (WOSP-02)*, pages 114–127, New York, July 2002. ACM Press.

[LB94] James R. Larus and Thomas Ball. Rewriting Executable Files to Measure Program Behavior. *Software–Practice and Experience*, 24(2):197–218, February 1994.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

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

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

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

[VBB98] Jan Vitek, Ciarán Bryce, and Walter Binder. Designing JavaSeal or How to Make Java Safe for Agents. Technical report, University of Geneva, July 1998. `http://cui.unige.ch/OSG/publications/OO-articles/TechnicalReports/98/javaSeal.pdf`.

[VC99] Jan Vitek and Giuseppe Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, 1999.

Appendix A:
Method Invocations on Uninitialized Objects

In Section 4 we argued that the JVM specification [LY99] prevents the use of uninitialized objects. Trying to invoke a method on an uninitialized object causes a verification error. However, there is a way to work around this restriction: Even on recent JVMs, finalizers are executed on uninitialized objects. The code example in Figure 6 allocates an object but does not invoke its constructor. Nonetheless, the finalizer (the method `finalize()`) is executed on the uninitialized object. We tested this code example on Windows XP with Sun JDK 1.5.0_01 and with IBM JDK 1.4.2, and in both cases the resulting output was '`value = 0`'.

This illustrates that there are 'grey areas' in the JVM specification, and that statements, such as 'the JVM guarantees that uninitialized objects are not used', may not hold in certain cases. However, this example has been manually constructed, and we can assume that such extreme situations are not frequent in practice.

```
public class UninitializedObject {

   public static void main(String[] args) {
      allocUninitializedObject();
      System.gc();
      System.runFinalization();
   }

   static void allocUninitializedObject() {
      // The following is manually crafted bytecode.
      // An object is allocated, but not initialized:
      new UninitializedObject
      pop
      return
   }


   final int value;

   UninitializedObject() { value = 1; }

   void print() { System.out.print("value = " + value); }

   public void finalize() { print(); }

}
```

Figure 6: The finalizer is invoked on an uninitialized object.

Appendix B:
Initialization with Different Constructors

The computation of the number of object allocations in a constructor (Section 4) 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 [GJSB00]. However, at the bytecode level, a constructor may invoke a different alternate or superclass constructor depending on its arguments without causing any verification error [LY99]. The example in Figure 7 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 4) 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 7: Depending on the constructor argument, the alternate or the superclass constructor is invoked.