AGGREGATED ACCOUNTING OF MEMORY USAGE IN JAVA

Paul Bouché
Nokia Siemens Networks, An den Treptowers 1, Berlin, Germany

Martin von Löwis
Hasso-Plattner-Institute, Potsdam, Germany

Peter Tröger
Humbolt University, Berlin, Germany

Keywords: Software engineering, Algorithms and data structures, Software testing and maintenance.

Abstract: Profiling of application memory consumption typically includes a trade-off between overhead and accuracy. We present a new approach for memory usage accounting which has a comparatively low overhead and still provides meaningful results. Our approach considers the structure of modern applications by introducing the notion of memory accounts where application modules get "charged" for memory allocations. We have applied this approach to Java application servers and discuss important implementation aspects as well as experimental results of our prototype.

1 INTRODUCTION

Memory consumption is often a bottleneck of large object-oriented applications. System operators can often merely observe the total amount of memory consumed and have to cope with ever-increasing demand for more main memory. Various layers of abstractions, such as communication middleware, XML processing, and persistency layers, contribute to the system's intricacy from a memory management point of view: users of upper layers often have no knowledge what amount of memory is allocated by what specific operation. We have primarily experimented with Java application servers, which are a common source of these issues today. Thus, we will draw our examples from that domain, although we believe that the results are valid for any Java application, and can be applied to other object-oriented systems as well.

In order to deal with the complexity of the applications, analysis of resource consumption is an important issue. Performance indices must be related to specific parts of the applications in order to identify relevant points for optimization. The according tools are typically called *profilers*. A particular category of such tools are *memory profilers*.

Many memory profilers today have one major flaw, which is the correlation of a specific memory allocation and the responsible piece of source code. In cases where memory profilers are able to report such information, they usually cause a very high runtime and memory overhead due to the continuous storage of stack trace information for each object allocation.

Large software is usually organized into *modules*. A module encapsulates a set of tasks sharing a common goal such as SOAP message processing, servlet containment or business logic implementation. In Java, modules can be identified and structured with varying degrees of abstraction, e.g. on the class level, the package level, or the Java archive (*jar*) level. Attributing resource allocation costs to the correct module of an application is in all cases an important task, because the allocation of one object may cause subsequent allocations of other objects.

Furthermore, the calling of method `a()` in module *A* may cause the execution of another method `b()` in module *B*. Both methods might cause object allocation and therefore memory allocation to be accounted. It might be "unfair" to attribute the cost to the initial call since it originated in another module.

As an example, consider an application server

Bouché P., von Löwis M. and Tröger P. (2009).
AGGREGATED ACCOUNTING OF MEMORY USAGE IN JAVA.
In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 177-185
DOI: 10.5220/0002253701770185

hosting a Web service implementation. This application uses a third party library for XML processing. If all memory allocation costs that are caused by the XML processing are charged to the server implementation, the usage of a different XML processing library might wrongly indicate a better or worse performance of the server implementation. A misuse of the XML parser library implementation leading to a memory leak should not be accounted to the library itself, but to the originating source of the allocation request.

Therefore we defined the *principal-agent relationship* as one module "asking" another module directly or indirectly to perform a memory allocation, whereas traditional profilers capture the full stack trace for an allocation which contains information about principal-agent relationships. We introduce a novel scheme to perform the relevant accounting of memory allocation capturing only principal-agent relationships without unnecessary information and unnecessary effort spent.

The rest of this paper is structured as follows. In the next section we give a short explanation about the problem at hand. The following section 3 introduces the *principal-agent* relationship. In section 4 we present the concept of the *memory account* in detail and give reasons for its necessity and advantages. Section 5 covers the implementation strategy of our memory profiler. It is followed by section 6 where an experimental evaluation using standard benchmarks of the profiler is portrayed. The paper concludes with a discussion of related work, section 7, and closing remarks in the last section 8. Source code examples can be found in the appendix.

2 STATEMENT OF THE PROBLEM

To perform an analysis of memory consumption (i.e. memory profiling) in an object-oriented language, it is necessary to keep track of object creation and destruction, and possibly also to keep track of how object references pass through the system. Memory profiling can focus on various aspects, such as frequency of allocations, redundant allocations, etc.

We focus on "garbage" objects, i.e. objects that are not any longer used. In Java, many of these objects will be automatically released by the garbage collector. Unfortunately, the garbage collector cannot determine whether objects are unused, but only whether they are unreferenced. A common pitfall in Java and similar systems is that objects remain referenced even though the software developer believes that the last reference to the object should have been released. Even in cases where such references get released eventually, they may consume a significant amount of memory over some period of time.

Our objective is to detect such cases and to help developers and operators to adjust the system appropriately. For this analysis we have to determine three pieces of information:

- How many objects of what type are still allocated?
- Why had the objects been allocated originally?
- Why are they still referenced?

From this list we only support the first two aspects. We expect that users study the total number of objects per type, and the amount of memory that these objects consume, and then compare the numbers with their expectations. If they find that there are more objects of a certain type than they had expected, they will next need to find out where they came from. Once they have found out why the objects got allocated in the first place, they can then study why they had not been released.

It is important to note that the first two aspects in the above list can be represented in an aggregate manner. For the total number of objects and the total amount of memory the approach to aggregation is obvious. For the second question, we found a way of computing an aggregated number. For the last question, aggregated answers are typically not possible: to find out why a specific object is still referenced, one needs to find the specific container object (or objects) that still holds a reference. There are various debugging techniques available to find such "backwards references"; this issue is out of scope of our research.

3 PRINCIPALS AND AGENTS

To answer the second question, various profiling tools record the complete stack trace at the point of object allocation (Pauw et al., 1999; Dmitriev, 2003; Pearce et al., 2006), making it easier to investigate the conditions under which the allocation had originally occurred, even after the methods performing the allocation have already completed. Of course, recording the stack trace does not allow one to replay the full system state at the point of allocation, as access to various global and instance variables may have contributed to the parameters of the object allocation; these data might have changed at a later replay. The fact that tools often record the call stack indicates an important aspect of the problem: To understand an object's role, it is often sufficient to know the place in the code

where it was created – access to the fields of the object at the point of creation is often not necessary. At the same time, analysis of the existing tools [1] demonstrates that mere recording of the code line containing the `new`-expression is considered insufficient – developers need to inspect the call stack at allocation time to see which of the callers "actually" caused the allocation to happen.

Definition 1. *An **agent** module is a module that performs the allocation of an object on behalf of another object, the **principal** module.*

Within an application server the principal may be the server implementation and the agent may be a certain servlet implementation. The servlet in turn may be a principal for the XML processing library.

An agent which allocated some object for some principal might itself act as the principal with respect to another agent module, in the context of the allocation of another object, so the relationship between principals and agents is defined in the context of the allocation of a single object.

Notice that the principal-agent relationship is not necessarily an instance of an immediate caller-callee relationship. Instead, there might be several intermediary modules which delegate the object creation to another, until eventually agent code is invoked.

4 MEMORY ACCOUNTS

The principal-agent relationship, as described in the last section, denotes a situation where one module commissions another module to perform some task and the commissioned module allocates additional memory to perform this task. In order to align memory consumption to modules, we define:

Definition 2. *A module is **accounted** if it is marked for memory profiling during a program run.*

Definition 3. *A **memory account** represents the amount of all memory allocated within an accounted module. An object is allocated **within** a module if the module is accounted and no other intermediary module is accounted.*

Imagine the scenario of a Web service request processing within a Java EE application server. A SOAP request is received on a TCP network socket. The contained message payload is extracted and passed to the servlet container. The container determines the responsible servlet implementation and relays the SOAP request to an instance of this servlet. In order to provide a Java representation of the incoming package, it calls the currently registered XML processor for parsing the message's contents. Also the servlet container itself and the application server networking stack process the message's XML information, since relevant SOAP header entries might need to be considered. Typical examples are security or routing information.

In this particular example, several *principal-agent relationships* can be identified. In all cases, the XML parser allocated Java objects for the representation of XML data. The incoming package triggers memory allocation in the application server, which itself indirectly triggers memory allocation by the servlet implementation.

Different levels of abstraction of memory accounts can be chosen for this example. The network core and the servlet container can be mapped to one memory account. Thus, allocations of the whole application server implementation and a servlet implementation can be cleanly separated. This is especially useful if one wants to detect possible memory leaks in a servlet implementation, regardless of the XML processing library or the application server. Each time the execution enters the XML processing, all allocations should be attributed to its corresponding memory account. When the execution returns allocations need to be charged to the previously active memory account. The execution context determines the previously active memory account.

In the Java virtual machine, all calls are synchronous unless an exception occurs. In the ideal case, all memory account states are maintained on a stack in sync with the Java execution stack. When a call leaves one module, the memory account needs to be saved and when the call returns it needs to be restored. Exceptions need to be handled appropriately as their processing may cause object allocation, e.g. the exception object itself.

When objects are deallocated by the garbage collector of the Java runtime, the corresponding memory account needs to be refunded. Therefore, every Java object must be mappable to the memory account its allocation cost was charged to.

The next section describes how the basic idea of memory accounts can be implemented in a Java runtime environment.

5 IMPLEMENTATION

We have named our memory profiler *ASGMemProf* as it was created in the context of the Adaptive Service Grid (ASG) project.

[1] See section 7 for details

ASGMemProf is based on the notion of Java packages and classes. These are the primary mechanisms for modularization of Java code; code that is from a single author or which fulfills a single function is often concentrated into a single class or package. As a consequence, by attributing memory allocation to classes and packages, we can typically identify the "culprit" for a memory allocation: the software function that caused the object to be allocated.

Possibly contrary to intuition, it is not necessary to record the exact line of code within the principal module that caused the memory allocation. When the developer finds that a certain package has caused the allocation of a number of objects of class X, the developer will often know what part of the package (class/method/line) caused the allocation. Only when a class has many instances that are allocated in many places (e.g. String objects), might the developer want to know where exactly the object has been allocated. However, such objects often form a part of a larger structure, so that the developer would want to track the *root* object of the structure instead.

Consequentially, a memory account is identified by a package name or by a package pattern. We implemented a pattern matching which allows the usage of a wildcard (*) allowing definition of different memory accounts for different sub packages. A single package name without a wildcard, e.g. `a.b`, will create a memory account for `a.b` and will attribute all costs to classes within `a.b`. The pattern `a.b.*` will match all classes in this package (direct match) and all subpackages (wildcard match) Therefore, definition 3 means that an allocation is accounted within the memory account whose package pattern matches the package of the allocating method the closest in the order in which they appear in the allocation's stack trace. Before profiling, memory accounts need to be defined by appropriate package names or package name patterns.

As a general implementation strategy we have employed on-the-fly code-rewriting (also known as dynamic bytecode instrumentation) based on the Java Virtual Machine Tool Interface (JVM TI)[2] and the Java Instrumentation API[3]. As a Java class is loaded, where and whether modifications of its bytecode need to take place in order to employ our memory accounting is determined automatically.

As we have previously mentioned the currently active memory account needs to be maintained in sync with the execution stack. For our scheme only those methods are of interest which reside in (sub-) packages for which a memory account has been defined.

The current memory account is stored in a *thread local variable*. Upon method entry it is saved into an added local variable, updated with the method's memory account and restored upon exit. The memory account a method "belongs to" is determined at load time, identified via an integer ID and added as a constant to the method's class. Methods outside of the defined memory accounts do not alter the current memory account. Therefore, allocations will be charged correctly to the memory account closest to the top of the execution stack.

We have implemented our own thread local storage which is based on the thread ID (added in Java 1.5.0) as an index into an array of `ThreadInfo` objects. The size of that array is static and currently 1,000,000. This should be sufficient even though in Sun's JVM the thread ID is a consecutive number.

This implementation strategy avoids the expensive capture of a stack trace upon each allocation event. The combination of a thread local variable and local variables implicitly constitutes a stack where the thread local variable always shows the top element of the stack and the elements below are entailed within the normal execution stack frames of the VM.

Concerning the allocation and deallocation events the following aspects need to be considered: measuring object size, accounting object allocation, accounting object deallocation and association with the correct memory account.

Measuring Object Size. We define an object's allocation costs as the objects size in contrast to the object graph it may refer to, i.e. the object graph is only implicitly considered as the sum of all monitored objects. The size of an object can be measured based on its class definition. The Java Instrumentation API offers a method which is implemented this way, but it needs an object as parameter. It would be desirable to know the size of an object based on its class before an instance is ever created. The implementation of `getObjectSize()` iterates over all fields and adds up the size based on the field's type. Since a call to that method upon every object allocation is very costly, the result is cached in a weak hash map.

Accounting Object Allocation. Object allocations are tracked by instrumenting the constructor of `java.lang.Object` with a global guard condition which when true will relay the control to a static method in the profiler along with the `this` reference. This method records object allocations (`trackAllocs(Object)`). This idea came from the documentation of the JVM TI reference. The object constructor will be called for all created Java objects

[2] `http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html`

[3] `http://java.sun.com/javase/6/docs/api/java/lang/instrument/package-summary.html`

including objects created via reflection and native code excluding array objects and `java.lang.Class` objects. Therefore, each `newarray` bytecode of all loaded classes will be instrumented with a call to a corresponding profiler method passing along the array object reference, the array's length and the array's content type.

Accounting Object Deallocation. There are several options for accounting object deallocation: (1) finalizers, (2) usage of reference objects from `java.lang.ref` and (3) JVM TI object tagging. Using finalizers for object deallocation accounting means adding or instrumenting the `finalize()` method for *every* class. This approach was tested and degraded the performance of the JVM drastically. Since our implementation is in pure Java we decided for reference objects. There are two types of references that can be used for accounting object deallocation, i.e. *weak* and *phantom* references. Reference objects can be registered with a reference queue. They need to be *cleared*., i.e. their referent needs to be set to null. When the garbage collector reclaimes a garbage object the referring weak and soft references, if any, will be cleared automatically and added to a reference queue, if any. Phantom references are not cleared automatically and are only enqueued when the object's finalizers have been run. In our opinion, weak references are least intrusive into the garbage collector, and do not prevent the garbage collection of an object, although in the case of reviving finalizers (which are rare), these are less accurate than phantom references. Nevertheless, we decided to use weak references.

Association with the correct Memory Account. The accounting methods will use the currently active memory account stored in the thread local variable of the currently executing thread as the account to which the costs are to be attributed. Allocation cost data for each class, thread and memory account need to be maintained. We have implemented this mainly by using two or three layers of weak hashmaps. They are weak in the sense that the value reference is a weak reference. This is important for example in the case of threads. If a thread terminates the profiler must not prevent the thread object from being reclaimed.

ASGMemProf can be set to periodically dump the collected data to disk (snapshot). The snapshot is in parsable text form so as to allow easier post processing. We employed a non-blocking scheme to create a snapshot. A shutdown hook ensures a final data dump.

For a more detailed explanation of the profiler implementation please refer to (Bouché, 2007).

5.1 Detecting Memory Leaks

Before each snapshot a full GC ensures the clearance of garbage objects and the recharging to the corresponding memory accounts. Although we focused our work on Java EE server side components such as Servlets or EJBs ASGMemProf can be used to profile Java SE applications as well.

Servlets or EJBs do not have a classic main method, but several entry points. Adding to that is the more complex life cycle of these components. Therefore, it is harder to define a point in time when all used memory for a given task should have been released. If it has not been, this is a strong hint at a memory leak. On the other hand Java SE applications have one main method and it is reasonable to say that when that method has finished all used memory should be freed or at least be eligible for reclamation. Yet, static variables complicate this as well.

In any case we detect memory leaks by analyzing the difference between two or more profiling snapshots. Currently this is done manually. For this to work memory accounts for different subsystems or areas of interest have to be carefully defined. Each account will list the live and total number of created objects for each class optionally grouped by the method(s) causing the allocation. Usually a memory leak is indicated by a rising number of live objects over time. A time correlation of events in the application and the time stamp of the snapshot is necessary. For example, the execution of a certain Servlet may cause a memory leak by misuse of another subsystem or third party library leaving the memory account of the Servlet engine clean, but causing a sustained raised number in another memory account.

As a practical application we needed a feasible memory profiler for the ASG execution platform which is implemented in Java EE. Under certain conditions there was an out of memory error, i.e. we had a memory leak. Employing available memory profilers to find the leak was practically impossible however, because either the system slowdown was making it unresponsive or the amount of data collected numbered several gigabytes. Certainly the great size of the ASG platform contributed to this.

We profiled the ASG execution platform with ASGMemProf and though there was a significant slowdown it was bearable and the amount of profiling data was greatly reduced through the employed aggregation. After analyzing the profiling snapshots we were finally able to identify the memory leak. It was caused by a server management subsystem using the API of the data abstraction library Hibernate [4] wrongly.

[4] `http://www.hibernate.org/`

6 DISCUSSION AND EVALUATION

It is important to quantify the memory and runtime overhead of a profiler experimentally in order to provide a basis for a prediction of the overhead it incurs on average. This is also a measure of quality for a profiler implementation. We present the overhead measurements of our implementation in this section.

6.1 Runtime Overhead Measurements

To measure the additional runtime an application requires when instrumented we used two benchmark suites. The DaCapo (Blackburn et al., 2006) benchmark suite v2006-10-MR2 was used to measure Java SE performance and SpecJBB 2005 v1.07[5] was used to measure server side performance.

All tests ran on Sun's Java HotSpot VM version 1.6.0-b105 in mixed, client mode. The computer hardware was a 2 GHz AMD Athlon processor with 1 GB of RAM running Microsoft Windows XP with Service Pack 2. The test results were computed from an average of three consecutive runs for each benchmark. The DaCapo benchmarks were executed with input size small and SpecJBB 2005 was carrying out Warehouses one through four each measuring 240 seconds. The results can be found in table 1.

The table shows how long each benchmark took without the profiler (plain) and with the profiler applied. Additionally, we measured the performance of the JFluid profiler (Dmitriev, 2003), whose technology has become part of the NetBeans[6] Java IDE which we used in version 6.5., as a comparison. This profiler works with similar technology to our profiler though it takes a stack sample upon each allocation event. In order to be fair we disabled this feature. For the SpecJBB2005 benchmark ASGMemProf was set to create a memory account for the package `spec.*`. For the DaCapo benchmarks, allocations were charged to an overall memory account.

For each of the profilers the table shows two columns: the time the benchmark took to complete and the incurred overhead given as a factor of the original execution time. For SpecJBB 2005 the overhead was computed by dividing the original throughput by the profiled throughput.

[5]Standard Performance Evaluation Corporation, SPECjbb2005 (Java Server Benchmark), `http://www.spec.org/jbb2005/`

[6]Sun Microsystems Inc. and NetBeans contributors, `http://www.netbeans.org/kb/index.html`

6.1.1 Discussion

Generally speaking, the overhead incurred with our current implementation makes it only feasible for development uses, but not applicable for production use. An acceptable overhead for production use is cited in the literature to be approximately 0.3 (Pauw et al., 1999) or less which our profiler clearly does not deliver. But in comparison with a competitive profiler implementation the results are more than encouraging.

The advantage of our implementation is clearly visible. In all cases ASGMemProf incurs overhead less than or equal to JFluid. In most cases ASGMemProf is twice as fast as JFluid.

The benchmark `antlr` incurs the least overhead, 1.72 which is getting into the region of acceptable overhead (1.3). This rather low overhead is due to the fact that `antlr` does not create many objects and those which are created live until termination. Therefore, much less time is spent in the profiling methods. Especially the rather expensive slow down of the GC via weak references is reduced due to its inactivity.

We have done preliminary tests of a worst case scenario in an application which continually creates a lot of objects with a very short life time. Creating a weak reference object for each allocated object already degrades the performance significantly - not to speak of the time needed to account object deallocation. The GC must treat `WeakReference` objects specially and obviously the implementation in Sun's JVM for this is only of average quality. All our attempts to further reduce the profiling overhead failed because most overhead is incurred when creating and tracking `WeakReferences` which is VM implementation dependent.

This can clearly be seen in benchmark `jython` where a lot of short living objects are created over a sustained amount of time. Hence, the expensive operations of weak reference creation, maintenance and notification upon referent reclamation occur often. Preliminary test with the `mtrt` benchmark of the SpecJVM98 suite showed a slowdown of 25, respectively 60 for JFluid. This is obviously not feasible even for development circumstances.

SpecJBB 2005 models typical server side Java behavior by emulating users accessing a rather big in-memory database inserting, updating and deleting records. The database is implemented as binary object graphs. Here the negative GC behavioral influence of weak references comes into play as well. They still perform a lot better than the native implementation of JFluid via JVM TI object tagging, but are very unacceptable for the production use of the profiler. Yet, the advancement of our implementation

Table 1: Runtime Overhead of ASGMemProf vs. JFluid/NetBeans.

DaCapo	plain (ms)	ASGMemProf (ms)	(factor)	JFluid/NetBeans (ms)	(factor)
antlr	813	1406	1.72	2750	3.38
bloat	2609	25813	9.89	49703	19.05
chart	2812	27062	9.63	60844	21.63
eclipse	10531	40171	3.81	67938	6.45
fop	1218	7688	6.31	26484	21.74
hsqldb	3406	10531	3.09	13641	4.00
jython	532	8860	16.65	9375	17.62
luindex	781	5047	6.46	5422	6.94
lusearch	2938	12625	4.29	27985	9.52
pmd	500	1172	2.34	8703	17.40
xalan	3343	17078	5.10	26594	7.95
SpecJBB 2005	(bops)	(bops)	(factor)	(bops)	(factor)
warehouse 1	4895	608	8.05	294	16.64
warehouse 2	4854	615	7.89	336	14.46
warehouse 3	4833	545	8.86	237	20.39
warehouse 4	4754	520	9.14	274	17.35

strategy is clearly visible in SpecJBB 2005 as well.

Weak reference or phantom reference objects still seem to degrade the performance drastically under heavy load. Yet, they remain the only viable option to do exact memory profiling apart from a native JVM TI agent implementation. Maybe another mechanism to track object deallocation in a standard manner for the Java platform needs to be found.

One optimization remains. Short run, often called methods cause a lot of unnecessary runtime overhead, especially if the calling code is from within the same memory account. This could be optimized by using static code analysis and only updating the account when necessary. Additionally, only those methods that (could) cause object allocation are to be instrumented. It is not insignificant to determine this at load time.

6.2 Memory Overhead Measurements

In order to measure the space overhead our profiler incurs we recorded the peak value of the *Private Bytes* performance indicator of the VM process during a benchmark suite run on the same hardware as previously mentioned. The reported result is an average of three runs for each benchmark. The results are listed in table 2.

Table 2 shows two columns for each ASGMemProf and JFLuid/NetBeans and one column for the benchmark without the profiler. The peak private byte size is given in kilo bytes. The overhead factor is determined as a factor of the plain value.

6.2.1 Discussion

The memory overhead does not vary as much as the runtime overhead.

In SpecJBB2005 both profilers incur practically the same amount of overhead. The main factor contributing to the space overhead is each reference object that has to be allocated for each newly allocated object in order to track its deallocation. These reference objects are allocated on the Java heap and in the current implementation weigh 40 bytes. If an application creates a lot of small objects, the overhead will be very high.

For the DaCapo benchmarks JFluid gives a better performance than our profiler, though the difference is relatively low. NetBeans produces slightly lower space overhead than ASGMemProf. This is probably due to the fact, that the NetBeans profiler implementation is in native code and instead of weak reference objects for deallocation accounting it uses the tag mechanism of the JVM TI which must require less space than a weak reference object.

7 RELATED WORK

Profiling of Java applications is an ongoing research topic. Additionally, there are several commercial and open source profilers available. Several publications address the usage of CPU and time measurement of single methods. Among them are ProfBuilder (Cooper et al., 1998), JaViz (Kazi et al., 2000), JInsight (Sevitsky et al., 2001), JFluid (Dmitriev, 2003), J-Seal2 (Binder et al., 2001), JSpy/JPaX (Goldberg

Table 2: Space Overhead of ASGMemProf vs. JFluid/NetBeans.

benchmark	plain (KB)	ASGMemProf (KB)	(factor)	JFluid/NetBeans (KB)	(factor)
SpecJBB2005	242,420	549,212	2.26	599,696	2.47
DaCapo	183,456	356.416	1.94	225.792	1.23

and Havelund, 2003), JBOLT (Brear et al., 2003), JPMT (Harkema et al., 2003), Twilight/Aksum (Seragiotto and Fahringer, 2005), JP Tool (Binder and Hulaas, 2006) and eDragon/JIS (Carrera et al., 2003).

These works are concerned with researching, discussing and evaluating different concepts and implementations for measuring the CPU time usage of methods, threads and whole modules. Some employ bytecode instrumentation (JPMT, Twilight/Aksum, ProfBuilder J-Seal2, JP Tool) and others use the provided profiling interface functions.

Much less attention has been given to memory profiling. J-Seal2 which is concerned with accounting and enforcing restriction on mobile code execution environments has a memory profiling subsystem. Their implementation also uses memory accounts and employs bytecode instrumentation. Techniques for associating context information with an allocation are similar. Yet, a memory account is not based on modules, but on predefined execution environment restrictions which are valid for a whole application. We employ a more refined model and J-Seal2 does not capture principal-agent relationships. A more recent publication in conjunction with that research is JP Tool where an expensive sampling of the stack is likewise avoided by extending a method's signature with a reference to the memory accounting object. Binder notes that this technique cannot be applied to Java core classes, whereas our rewriting scheme allows profiling of core classes as well.

The JFluid profiler which has been integrated into the NetBeans development environment employs memory accounting techniques similar to those we have used. The implementation uses weak references for object deallocation notification as well. Results are aggregated into a calling context tree (CCT) (Ammons et al., 1997). This aggregation technique is very common among profilers. While information for memory accounts can be extracted from a CCT, unnecessary stack trace data has been collected and effort expended. The NetBeans profiler contains a very interesting technique for detecting memory leaks. The object generation metric for a class which is the number of different ages for all objects. An object age is the number of garbage collections it has survived.

DJProf (Pearce et al., 2006) which is a profiler based on aspect oriented programming (AOP) using AspectJ (Kiczales et al., 2001) is employed to perform the bytecode instrumentation (the defined aspects are *woven* into the code). It uses phantom references to capture object deallocation. The goal was to test the suitability of AOP for profiler implementations. There is a short discussion on where to attribute allocation costs to using the example of a constructor allocating other objects. It is decided to take the same direction as we do, but not generalized into the notion of a memory account.

There are several commercial tools which allow memory profiling such as YourKit, JProbe, JProfiler, Borland OptimizeIt, Intel VTune, IBM Rational Quantify and Wily Introscope.

Furthermore, attempts are being made to reduce the overhead of exact profiling with sampling techniques for the cost of accuracy. There are several works that discuss this issue (Arnold and Ryder, 2001; Factor et al., 2004; Ammons et al., 1997; Dmitriev, 2003). Arnold points out a scheme for reducing the overhead cost of instrumented code and presents data showing that recording only every 10th event will still yield an accuracy of 98%. This is something we can investigate for ASGMemProf in the future. Yet, a sampling technique is not feasible for finding memory leaks.

8 CONCLUSIONS AND FUTURE WORK

We have presented a novel model for memory profiling: the principal-agent relationship, and the concept of a memory account. These concepts attempt to reduce the overhead of exact memory profiling by performing a sensible aggregation of data. In particular, even though each individual object is accounted for, we can avoid computing and preserving the stack trace that lead to the allocation of a specific object.

Our approach inherently avoids taking a stack sample for each allocation and therefore delivers less information than traditional memory profilers. As we have explained this is in effect not a loss but a gain. Although this constitutes less accuracy in terms of amount of information. Optionally the profiler can be set to record the top stack frame for the allocation at no additional cost.

While initial results obtained from the approach

are promising, end-user experience from a variety of applications still needs to be obtained and studied, to find out whether the presented approach is practical for analyzing memory consumption. Analyzing the memory and run-time overhead is feasible, as we have demonstrated above. Analyzing the value of our tools to developers is more difficult; based on past publications in this field, we expect that any report on utility and viability of this approach will remain anecdotal.

We envision two application areas for this approach: development and operations. Our applications of the tool had primarily been in the field of development – helping the developer to find out memory leaks in the application, so that the code can be improved.

In operations, the application of the approach would be different. For example, the operator might apply memory accounting to different services running in the same service container, and then take service management decisions based on the amount of memory used by each service (e.g. to migrate a service with high memory consumption to a different machine). As another example, the approach could be used for the self-policing of application containers: the container could enforce an upper limit on memory consumption, and let allocations from a principal fail if the principal's memory account is overdrawn.

Further implementation details can be found in (Bouché, 2007). The profiler is available at `sourceforge.net`.

REFERENCES

Ammons, G., Ball, T., and Larus, J. R. (1997). 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, New York, NY, USA. ACM Press.

Arnold, M. and Ryder, B. G. (2001). A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179.

Binder, W. and Hulaas, J. (2006). Exact and portable profiling for the JVM using bytecode instruction counting. volume 164, pages 45–64.

Binder, W., Hulaas, J. G., and Villazon, A. (2001). Portable resource control in java. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 139–155. ACM Press.

Blackburn, S. M., Garner, R., and Hoffmann, C. (2006). The DaCapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06*, pages 169–190, New York, NY, USA. ACM.

Bouché, P. (2007). A comparative study of J2EE profiling approaches for usage within asg. Master's thesis, Hasso-Plattner-Institute for IT-Systems Engineering of the University of Potsdam.

Brear, D. J., Weise, T., Wiffen, T., Yeung, K. C., Bennett, S. A. M., and Kelly, P. H. J. (2003). Search strategies for java bottleneck location by dynamic instrumentation. In *Software, IEE Proceedings*, volume 150, Issue: 4, pages 235– 241.

Carrera, D., Guitart, J., Torres, J., Ayguade, E., and Labarta, J. (2003). Complete instrumentation requirements for performance analysis of web based technologies. In *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*, pages 166– 175.

Cooper, B., Lee, H., and Zorn, B. (1998). Profbuilder: A package for rapidly building java execution profilers.

Dmitriev, M. (2003). Design of JFluid: A profiling technology and tool based on dynamic bytecode instrumentation. Technical report, Sun Microsystems Inc.

Factor, M., Schuster, A., and Shagin, K. (2004). Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 288–300, New York, NY, USA. ACM Press.

Goldberg, A. and Havelund, K. (2003). Instrumentation of java bytecode for runtime analysis.

Harkema, M., Quartel, D., van der Mei, R., and Gijsen, B. (2003). JPMT: A java performance monitoring tool. Technical Report TR-CTIT-03-25 Centre for Telematics and Information Technology, University of Twente, Enschede.

Kazi, I. H., Jose, D. P., Ben-Hamida, B., Hescott, C. J., Kwok, C., Konstan, J., Lilja, D. J., and Yew, P.-C. (2000). JaViz: A client/server java profiling tool.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK. Springer-Verlag.

Pauw, W. D., Jensen, E., and Konuru, R. (1999). Jinsight, a visual tool for optimizing and understanding java programs. ibm corporation, research division. `http://www.alphaWorks.ibm.com/tech/jinsight/`. Url last visited: 23. May 2009.

Pearce, D. J., Webster, M., Berry, R., and Kelly, P. H. J. (2006). Profiling with aspectj. In *Software: Practice and Experience*. John Wiley & Sons, Ltd.

Seragiotto, C. and Fahringer, T. (2005). Analysis of distributed java applications using dynamic instrumentation. In *IEEE International Conference on Cluster Computing (Cluster 2005)*.

Sevitsky, G., de Pauw, W., and Konuru, R. (2001). An information exploration tool for performance analysis of java programs. In *TOOLS '01: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 85, Washington, DC, USA. IEEE Computer Society.