Building Custom, Efficient, and Accurate Memory Monitoring Tools for Java Applications

Verena Bitto
Christian Doppler Laboratory MEVSS
Johannes Kepler University Linz, Austria
verena.bitto@jku.at

Philipp Lengauer
Institute for System Software
Johannes Kepler University Linz, Austria
philipp.lengauer@jku.at

ABSTRACT

Traditional monitoring techniques can distort application behavior significantly. In this paper, we will provide an evaluation of state-of-the-art monitoring techniques and their impact on memory behavior. We will use AntTracks to show how VM-internal approaches can extract more diverse memory information at object level, vastly outperforming traditional techniques.

Keywords

Memory Monitoring; Garbage Collection; Java; Memory Monitoring Tools

1. INTRODUCTION

Higher-level programming languages like Java or C# relieve the programmer from freeing memory manually by applying automatic memory management, i.e., garbage collection (GC). Concomitantly, the actual memory behavior is hidden from the developer, making the detection of the actual source for memory anomalies a tedious task. Memory monitoring tools allow to keep track of memory internals, such as memory allocations and GC time. However, what kind of information can be tracked, at what granularity and at which costs depends mainly on the approach used for monitoring. Many existing tools impose an enormous memory footprint and run-time overhead on the monitored application, neglecting that such characteristics distort the actual memory behavior. This paper provides insights into all common monitoring strategies, i.e., *Sample-based Monitoring*, *Instrumentation-based Monitoring* and *VM-internal Monitoring*. We especially focus on VM-internal monitoring and our tool AntTracks, introduced at the ICPE'15 [3], to sketch which advantages as well as challenges custom monitoring tools implicate.

This paper is structured as follows: Section 2 describes state of the monitoring tools techniques, their benefits and drawbacks as well as their impact on the actual memory behavior; Section 3 compares all techniques by means of their information richness and information quality. Furthermore we show exemplary, which performance can be expected for different monitoring approaches; Section 4 concludes this paper.

ICPE'16, March 12-18, 2016, Delft, Netherlands

DOI: http://dx.doi.org/10.1145/2851553.2858664

2. STATE OF THE ART MONITORING TECHNIQUES

State-of-the-art approaches can be divided into 3 categories, i.e., *Sample-based Monitoring*, *Instrumentation-based Monitoring*, and *VM-internal Monitoring*. These approaches have fundamental differences in run-time overhead, information richness (such as the kind of data that can be captured) as well as quality (such as the accuracy of the generated results). We do not regard *Event-based Monitoring* as a distinct approach here, because one of the just mentioned approaches must be used to generate events in the first place.

The following sections will describe every approach in detail, including basic principles as well as a performance evaluation in the context of memory monitoring. We built tools according to the mentioned monitoring techniques with the aim to produce equivalent memory-related data in order to compare them accordingly. As basis for information comparison we used our tool AntTracks, since *VM-internal* techniques allow for the most diverse data to collect.

2.1 Sampling-based Monitoring

Sampling-based monitoring tools collect data periodically. In the best case, they reflect a statistically valid representation of the program under inspection. Commonly gathered information includes CPU usage information, e.g., the amount of time threads spend in specific methods, or memory information, e.g., the amount of memory data structures occupy. What kind of information can be extracted without modifying the virtual machine itself depends on the provided interface, e.g., the Java virtual machine tools interface (JVMTI) for Java or the ICorProfilerCallback interface for .NET.

A key requirement for sampling-based approaches is to sample at random time intervals. Otherwise the tool may record periodically recurring phases of a program over and over again, e.g., the stack traces of threads which are scheduled at certain intervals, while missing other phases in-between. However, randomizing sample intervals is difficult, since the program needs to be in a certain state, i.e., in safe points to pause application threads, for retrieving samples [1]. As a result, the accuracy of samples are limited by the underlying virtual machine. Consequently, different monitoring tools may deliver different results on specific metrics

like frequently called methods, as pointed out by Mytkowicz et al. [5].

2.2 Instrumentation-based Monitoring

Instrumentation-based monitoring tools allow to modify the original code of an application such that it records the desired information. Code can be instrumented either statically or dynamically, whereupon the former inserts code at compile-time, and the latter at run-time. In the latter case, the underlying virtual machine must support retrieving the original code and exchanging it with the manipulated one, e.g., by means of the JVMTI or the ICorProfilerCallback interface. However, these interfaces only provide means to instrument at bytecode level, not at machine code level.

Compared to sampling, instrumentation-based approaches allow to record more accurate information. However, due to the modified or injected code, instrumentation may impose a significant run-time overhead on the application (1) due to the additional code itself and (2) due to the just-in-time compiler not being able to apply the same optimizations.

2.3 VM-internal Monitoring

Tools using the VM-internal monitoring approach require a modified VM to host the monitored application. The modified VM can expose internal data structures to the monitoring tool. For example, a VM may provide access to internal timers that keep track of the time of certain GC phases. Using this information, the monitoring tool may be able to explain long garbage collection times.

However, when already modifying the VM to expose additional data, one might consider to move as much as possible of the monitoring logic into the VM, because the information of interest is easier and faster to access.

In previous work (cf. Lengauer et al. [3] and Bitto et al. [2]), we have presented *AntTracks*, a custom VM able to trace object allocations and object deallocations imposing only a very small run-time overhead. Minimizing overhead is paramount to reduce the impact of the Observer Effect. Especially in the context of managed memory, a lot of adaptive heuristics and adaptive thresholds are in place to manage garbage collection. Even small changes in garbage collection time or in object allocations may lead to different choices by the GC, changing the overall garbage collection behavior and, in consequence, memory performance.

However, modifying a VM to support monitoring raises some interesting challenges if one of the primary goals is low run-time overhead and in-production use, such as (1) how to record the data, (2) how to maintain the data within the VM, (3) how to send the data to an external monitoring tool, (4) how to store the mass of data as well as (5) how to process the raw data and how to transform it to a meaningful memory representation.

Recording. The information of interest needs to be recorded, without distorting the application behavior. For example, AntTracks must fire an event for every new object by inserting code at every allocation. However, contrary to traditional instrumentation, it can insert the code during the lowering phase of the intermediate representation (i.e., in the abstract syntax tree) of the just-in-time compiler to machine code. If the allocation is optimized away, e.g., by escape analysis and scalar replacement, the allocation will not be lowered to machine code, and thus no instrumentation will be performed. Traditional bytecode instrumentation would impede escape analysis and scalar replacement, and thus change the application behavior by forcing the allocation of the object, no matter what.

Managing. The next step comprises the management of the recorded data, as we want to keep IO to a minimum and consequently do not want to process and send every event immediately. An obvious approach is to buffer data until enough has aggregated and send an entire chunk when the buffer is full, instead of firing every event one by one. However, buffering needs to be implemented carefully in order not to degrade performance. For example, when multiple threads record data in parallel, the buffer must be locked. Additionally, flushing the buffer (e.g., to a file) when it is full may block other threads for a significant amount of time. Thus, AntTracks uses thread-local buffers, i.e., every thread own its private buffer to write events to. Furthermore, when the buffer is full, the buffer is not flushed directly (which would stop the thread for too long), instead the buffer is submitted to a flush queue and a new buffer is fetched from a free list. A dedicated worker thread consumes buffers from this queue, flushes them, and submits it to the free list. Buffer sizes are randomized to avoid multiple threads with the same recording frequency to submit buffers to the flush queue at the same time.

Sending. Sending includes either storing the recorded data in a file for subsequent analysis or sending it, e.g., using a socket connection, to a tool directly. The performance of this step is heavily dependent on the underlying operating system and hardware performance.

Storing. Storing the recorded data for subsequent analysis can be challenging if the amount of data faces disk limitations. Most tools reduce their accuracy and send statistical aggregations only. However, aggregation must be done at run-time, whereupon it increases run-time overhead.

AntTracks uses a novel approach by cyclically overwriting old tracing information without losing most vital information (cf. Lengauer et al. [4]). It creates multiple trace files, clears them periodically, and starts every trace file with a synchronization point, which can be used for reattaching to the Java heap state.

Processing. Stored data is often highly compacted, keeping the storage performance and recording performance in mind. Thus, the raw data needs to be processed and transformed to meaningful memory representations. Depending on the amount of data collected, this can take up a significant amount of time.

AntTracks reconstructs thread-local heap states, and merges them at defined points in time. Thus the recorded data can be processed mostly in parallel.

3. COMPARISON

The following section compares monitoring techniques exemplary to give an estimate regarding information richness, information quality and performance. To compare the different approaches fairly, we have started from AntTracks and built tools trying to record as much of the same data as possible, once via sampling, i.e., dumping the heap periodically, and once via instrumentation at bytecode level.

Setup. All measurements were run on an Intel® Core TM i7-3770 CPU @ 3.4GHz x 4 (8 Threads) on 64-bit with 32 GB RAM and a Samsung SSD 840 PRO Series (DXM03B0Q), running Ubuntu Trusty Tahr 14.04 with the

Kernel Linux 3.11.0-23-generic. All unnecessary services were disabled in order not to distort the experiments.

3.1 Information Richness and Quality

Different monitoring techniques provide different levels of granularity of information. Figure 1 gives an overview about the different kind of traced information of the discussed monitoring strategies.

Information	Smpl. Ref.	Instr. Ref.	ET	VM AT
Object allocations	Some	All	All	All
– Size	All	All	All	All
– Type	All	All	All	All
– Pointers	All	None*	All	All
– Allocation site	None	All	All	All
– Address	None	None	None	All
– Allocating subsystem	None	None	None	All
Object deaths	Some	All	All	All
– Time unreachable	None	None	All	None
– Time collected	Some	All	None*	All
– Liveness from GC	None	None	None	All
Object lifecycle	None	None	None	All
Object movements	None	None	None	All
Heap structure	None	None	None	All
Method entries/exits	None*	None*	All	None
Temporal ordering	All	Some	All	Some
Explicit GCs triggered	None	None	All	None
Arbitrary VMs	All	Some	Some	None

Figure 1: Comparison of the capabilities of sampling-based (our reference implementation), instrumentation-based (our reference implementation as well as ElephantTracks (ET), the most similar tool to AntTracks) and VM-internal (AntTracks - AT) monitoring. (*Recording of information possible, but not implemented)

Object allocations can be tracked with every monitoring technique, although the amount of monitored information differs. Sampling stands out as the only method that may miss allocations altogether, e.g., objects may be allocated and freed in the time between two samples. Although all techniques provide basic object information like its size, its type and its pointers, only VM-based monitoring allows to collect internal information like the object's address or its allocating subsystem, i.e., the interpreter, C1 compiler or C2 compiler. Object deaths can be monitored with all techniques as well. Sampling-based approaches suffer again from incompleteness due to non-sampled time intervals. Other approaches differ in their definition of object death. ET considers the time an object becomes unreachable, while our instrumentation-based reference implementation and AT track the time an object is actually freed by the GC. The instrumentation approach uses `PhantomReferences` (a more scalable variant of `WeakReferences`) to detect deallocations. Since garbage collection data is only available via VM-internal tracking, ET has to run its own reachability analysis to determine liveness of objects, by explicitly triggering additional GCs. Movements of objects, e.g., during garbage collection, are only monitored by the VM-internal approach. As a result, VM-internal techniques can reproduce the entire lifecycle of objects. This includes changes to the object's pointers as well as its location in the heap.

Likewise, the heap structure can only be reconstructed by the VM-internal approach. This covers both object related information, e.g., the location of objects in the heap, as well as memory related information, e.g., address ranges of a specific heap region.

Method-related data cannot be monitored via sampling since heap dumps lack this kind of information. In case of AT and our sample-based reference implementation, it currently is not monitored by choice, since the hotness of methods is derived differently.

Temporal ordering concerns the chronological order of monitored information. Since AT and our instrumentation-based reference implementation record events in a thread-local manner contrary to ET, event ordering among different thread is not ensured. Sample-based approaches are by definition in order, due to the periodically drawing of samples.

Although, on the one hand, every JVM supports heapdumps, instrumentation on the other hand requires an interface like JVMTI which is supported from JDK 5.0 onwards. VM-internal tracking requires a modified VM and therefore lacks portability.

3.2 Performance

All three approaches have major differences in terms of overhead as well as behavior distortion they introduce. Figure 2 and 3 show the run-time overhead and the GC-time overhead respectively, normalized compared to the application without any active monitoring.

To make the sampling approach comparable to the instrumentation approach and the VM-internal approach, we adjusted the sampling rate for every benchmark individually, so that the sampling approach generates as many dumps as the other approaches can reproduce a full and consistent heap state.

In terms of *run-time*, the VM-internal approach, i.e., AntTracks, outperforms all other tools. In some cases, i.e., mpegaudio, avrora, and scalaxb, the sampling strategy performs almost as good because these benchmarks only have few allocations and even fewer live objects, consequently resulting in small and fast dumps. For allocation intensive applications, however, sampling produces an overhead of at least 100%. The instrumentation approach produces the most overhead, due to impeding the scalar replacement of objects and its need for additional object allocations (phantom references).

In terms of *GC-time*, the sampling strategy performs best with only one exception (factorie) because it only introduces pauses while the application is running. Consequently, sampling shows only slight deviations regarding GC-time, due to the changed application-to-GC ratio. The instrumentation-based approach actually introduces so much GC overhead, that three benchmarks (compiler.compiler, xml.validation, factorie) crash altogether, because the VM spends more than 98% of its time on garbage collection. This overhead is caused by effectively doubling the amount of objects, i.e., one phantom reference for every object allocated, and the additional handling of those special references by the GC. In those benchmarks that do not crash, the overhead is with a minimum (with the least-allocating benchmark mpegaudio) of 2922% and a maximum of 48408% tremendous. The VM-internal approach changes the GC behavior because of

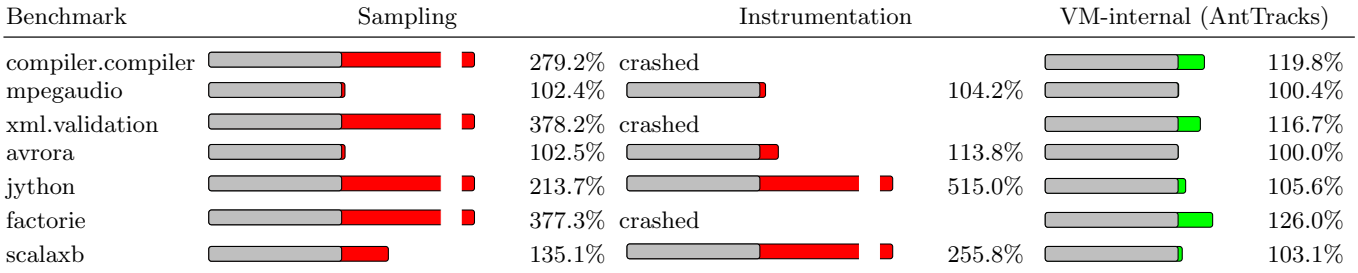

Figure 2: Run time of all three approaches, relative to the respective application's run-time without any monitoring.

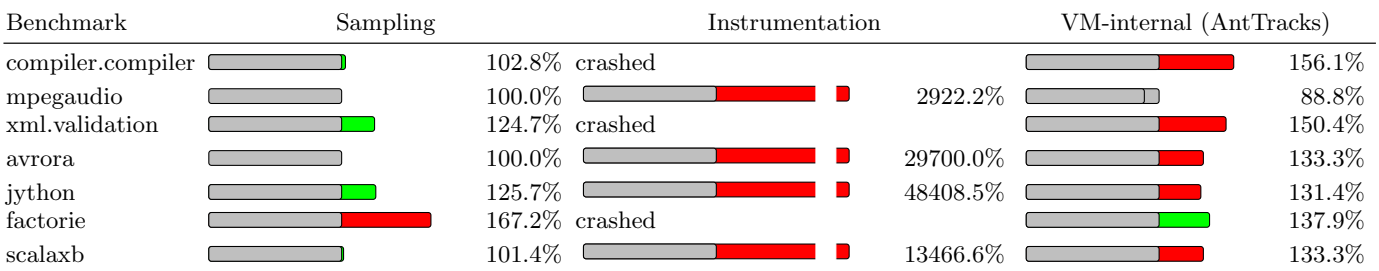

Figure 3: GC time of all three approaches, relative to the respective application's GC time without any monitoring.

its internal instrumentation. Consequently, the overall GC-time is higher, however, the overhead is pretty constant and predictable compared to the other approaches.

The *data size* (bytes generated) is variable depending on the approach. In general, the instrumentation approach produces more data than the VM-internal approach, because the first has to send an event for every deallocated object, whereas the latter has to send only events for live objects. Considering that only a small portion of objects survive their first GC, the former has to generate more data. The data size of the sampling approach is highly unpredictable, depending on when a dump is taken, i.e., right before vs. right after a garbage collection.

4. CONCLUSION

In this paper, we evaluated different monitoring techniques with respect to three metrics, i.e., information quality, information richness and overhead in terms of run time, GC time, and heap memory. Although a VM-internal approach, such as AntTracks, poses more complex and VM-specific challenges to overcome, we showed that it outperforms the traditional techniques in all three metrics. Our approach incorporates new techniques for efficient monitoring, that can be easily adapted for other high-performance and almost distortion-free tools.

5. ACKNOWLEDGMENTS

This work was supported by the Christian Doppler Forschungsgesellschaft, and by Dynatrace Austria GmbH.

6. REFERENCES

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño virtual machine. *IBM Syst. J.*, 39(1):211–238, Jan. 2000.

[2] V. Bitto, P. Lengauer, and H. Mössenböck. Efficient rebuilding of large java heaps from event traces. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, pages 76–89, New York, NY, USA, 2015. ACM.

[3] P. Lengauer, V. Bitto, and H. Mössenböck. Accurate and efficient object tracing for java applications. In *Proc. of the 6th ACM/SPEC Int'l. Conf. on Performance Engineering*, ICPE '15, pages 51–62, 2015.

[4] P. Lengauer, V. Bitto, and H. Mössenböck. Efficient and viable handling of large object traces. In *Proc. of the 7th ACM/SPEC Int'l. Conf. on Performance Engineering*, ICPE '16, pages 51–62, 2016.

[5] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of java profilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 187–197, New York, NY, USA, 2010.