Efficient and Viable Handling of Large Object Traces

Philipp Lengauer[1] Verena Bitto[2] Hanspeter Mössenböck[1]

[1]Institute for System Software
Johannes Kepler University Linz, Austria
philipp.lengauer@jku.at

[2]Christian Doppler Laboratory MEVSS
Johannes Kepler University Linz, Austria
verena.bitto@jku.at

ABSTRACT

Understanding and tracking down memory-related performance problems is a tedious task, especially when it involves automatically managed memory, i.e., garbage collection. A multitude of monitoring tools show the substantial need of developers to deal with these problems efficiently. Unfortunately, state-of-the-art tools either generate an inscrutable amount of trace data or produce only a coarse-grained view of the application's memory behavior. While the first approach generates information that is very detailed albeit difficult to handle, the second approach is more efficient but may fail to provide vital information.

In this paper, we propose a method to combine the advantages of both approaches, i.e., a method to handle fine-grained tracing information efficiently. Specifically, we present an on-the-fly compression technique for tracing data with reasonable overhead. Furthermore, we show how to overwrite old parts of the trace to circumvent its unlimited growth, but almost without losing vital information.

We also provide a detailed evaluation of our approach, showing that the introduced run-time overhead is negligible compared to similar tracing tools as well as that the information quality recovers quickly after overwriting parts of the old tracing data.

Keywords

Tracing; Trace Reduction; Garbage Collection; Java

1. INTRODUCTION

The widespread use of programming languages with automatic memory management has stressed the need for memory profiling tools. Although managed memory relieves programmers from the error-prone task of freeing memory manually, it comes at the cost of performance problems that are hard to track down. When an allocation fails due to a full heap, the subsequent garbage collection (GC) pauses the application for a hard-to-predict period of time. Depending on the application's memory behavior, garbage collections may occur frequently and, thus, may have a negative impact on the application's throughput (long overall GC time) and availability (long GC pauses).

Existing profiling tools, such as GC Spy [12], operate on memory blocks instead of objects to reduce the overhead and to make visualization easier. Consequently, these tools are only able to provide coarse-grained statistics about memory blocks, such as how many objects of a specific type are contained in it. They are unable to provide information about the origin of an object, which is crucial, for example, to track down a memory leak.

Other tools, such as the one by Ricci et al. [14] collect more fine-grained information. However, they introduce an enormous overhead of up to 14000% [7].

In Lengauer et al. [7] we introduced AntTracks, a memory profiler based on the Java Hotspot™ VM which is able to record object allocations and object movements. Although information is recorded at object level, AntTracks produces a very low run-time overhead (4.86%) compared to other state-of-the-art tools. This has been mainly achieved through white-box compression, i.e., a compact, mostly precompilable event format, representing object allocation events and object move events with around 5 bytes on average. No event contains redundant information. Instead all information which can be reconstructed offline is omitted. Still, the generated trace files grow large quickly. The fastest-growing trace we have observed so far (SPECjvm compiler.compiler), produces up to 200 MB of trace data per second. Considering this amount of generated data, it is obvious that we need a method to keep the trace files small in order to allow efficient analysis. Interestingly, this is a problem which seems to have been disregarded in literature so far. Every continuously tracing monitoring tool either prolonging this problem with a low granularity or seems to ignore it at all.

In this paper, we will describe the effects of black-box compression on the trace, i.e., traditional compression that does not consider individual events but rather regards the trace as a large binary stream. Black-box compression is difficult during monitoring, because it has to be done on-the-fly, resulting in a trade-off between compression rate and run-time overhead. Furthermore, black-box compression does not solve but only postpones the problem of continuously growing trace files. Although the trace will grow more slowly, it will still grow unfeasibly large for a continuously running application.

Thus, in addition to black-box compression, we describe a new method to limit the absolute size of a trace file. When the trace file starts to exceed the given limit, old events are

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

DOI: http://dx.doi.org/10.1145/2851553.2851555

removed from it. The trace file thus always represents only the last part of the application's memory behavior. We call this *trace rotation*.

However, rotating the trace leads to some interesting challenges, especially when considering its performance impact and a minimalistic event format. For example, as events might omit reconstructible information, they depend on previous events as described in Bitto et al. [1]. Thus, if the front of the trace is cut off, the trace cannot be processed correctly because some information may be lost.

Our scientific contributions are (1) a method to integrate on-the-fly low-overhead compression of traces into a monitoring environment, (2) a novel technique to rotate object-level trace files incrementally, and (3) methods to restore information from before the cut-off point. Furthermore, we provide (4) a detailed evaluation of both, the overall overhead, as well as the information quality, i.e., the amount of reconstructible and non-reconstructible information.

This paper is structured as follows: Section 2 provides some necessary background information about the Hotspot™ VM as well as about AntTracks; Section 3 describes both black-box compression as well as trace rotation; Section 4 provides a detailed evaluation of our techniques; Section 5 describes related work; Section 6 presents future work and Section 7 concludes this paper.

2. BACKGROUND

This section briefly discusses the basics of the Java memory management as well as AntTracks' strategies to record and encode memory events.

2.1 Memory Management in Hotspot™

Java uses garbage collection to reclaim unused memory. In its default configuration the Parallel Collector splits the heap into two generations, i.e., a young generation and an old generation. The young generation is further split into the Eden space, the survivor-from space and the survivor-to space, while the old generation consists of the old space only. Usually, objects are allocated into the Eden space until it is exhausted. In this case, the JVM triggers a *minor collection* to free unused memory in the young generation using a stop-and-copy approach. Live objects from the Eden space are copied to the empty survivor-to space. The survivor-from space contains objects that have already survived at least one collection. Depending on how many collections they have already survived, they may be either copied into the survivor-to space or into the old space. After all live objects have been evacuated, the Eden space and the survivor-from space are declared empty and the two survivor spaces are swapped. The old generation is collected during *major collections* only. Major GCs are triggered rarely, since generational collection assumes that the majority of objects in the old space are alive for a long time. Instead of copying objects to another space, the entire heap is compacted towards its beginning.

To avoid extensive synchronization in multi-threaded applications, the JVM uses thread-local allocation buffers (TLABs) for fast allocations into the Eden space and promotion-local allocation buffers (PLABs) for fast copying during minor collections. These buffers are large chunks of memory which are claimed by a single thread while locking the heap. All following allocations or copying actions of a thread can then be done without locking. Major collections are done in parallel as well but without using PLABs. Instead the heap is divided into dedicated regions, which are collected by separate threads.

Java objects consist of two parts: the header, and the payload. The header contains meta information about the object, such as its type, its identity hash code, the lock state, and special information for the GC (such as the object's age and the mark bit). The payload contains all fields of the object.

The header is split into the mark word and the klass word, both being either 4 bytes or 8 bytes depending on the machine architecture. Figure 1 sketches the object header on a 64-bit architecture, using compressed pointers (enabled by default since Java SE 6u23). In this case, the mark word takes up 64 bits, while the klass word requires 32 bits only. The compressed klass word is made possible by storing pointers as offset from the heap base address. The mark

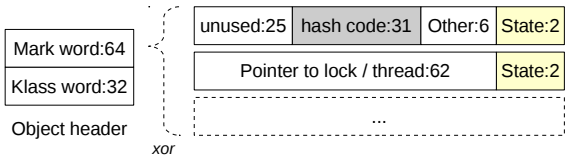

Figure 1: Object header in different object states

word is heavily over-allocated, meaning that it encodes different information depending on the object's state. The four different states are:

- neutral: The mark word contains the identity hash code and some flags regarding locking. The first 25 bits of the mark word remain unused.
- locked: The mark word holds a pointer to the native lock object.
- biased: The mark word holds a pointer to a thread towards which the object is biased. This thread may acquire the lock more efficiently.
- marked: The mark world holds a pointer to the location the object has been moved to by the GC.

Identity hash codes are generated lazily, i.e., they are initialized with zero and are generated when the identity hash is accessed for the first time. Zero is thus never a valid identity hash code.

2.2 AntTracks

AntTracks is a memory profiling tool based on the Hotspot™ VM. It records memory events, i.e., object allocations and object moves of a running Java application. The events are captured per thread as sketched in Figure 2. Each recording thread owns a private event buffer, which is fetched from a pool of empty buffers. All memory-related events of a single thread are then placed into that buffer. The size of the buffers is defined globally and is 16 kilobytes by default. The actual buffer size varies slightly to avoid multiple threads filling their buffers at the same time. Full buffers are enqueued into the flush queue which is drained by a dedicated worker thread. The worker thread undertakes the actual writing of the event buffers to the trace file. For every buffer a header is prepended, denoting the owning

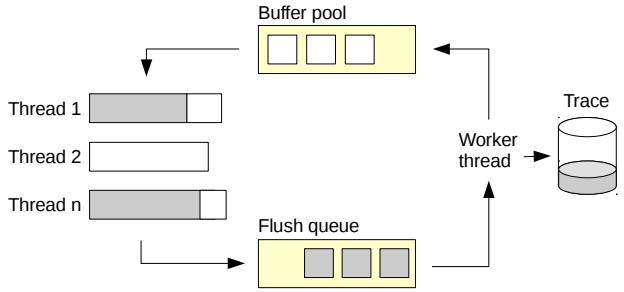

Figure 2: Event buffer management

thread and the buffer size, followed by the actual events. The processed event buffers are then returned to the buffer pool.

AntTracks has been specifically designed for low run-time overhead in order not to distort the actual memory behavior, while producing exact traces at the same time. Precisely for this reason, a lot of effort has been put into a binary, compact trace format. We omit any information from an event, which can be reconstructed offline, such that the majority of events are compile-time constants. As a result, dependencies between events arise which need to be resolved when processing the trace.

The characteristics of an object are stored in the allocation events only. For example, the *obj allocation generic* event stores the object's address, the size, the allocation site, and the allocation mode. The allocation site is defined as the location, i.e., the method and bytecode index, where the object has been allocated. The object's type can be inferred from the allocation site, and is therefore not recorded explicitly. Usually, only a single object type is allocated per allocation site. Hence, the object's size can be inferred from the allocation site as well. Only in those rare cases where this is not possible, the size is encoded into the allocation event as well. Whenever an object has been allocated into a TLAB, a *obj allocation optimized* event is fired instead, which omits the allocation address because it can be reconstructed offline by the analysis tool.

GC move events carry no explicit information about the moved objects. The *GC move generic* event (see Figure 3) contains the source address and the destination address only. The information, which object is actually moved can

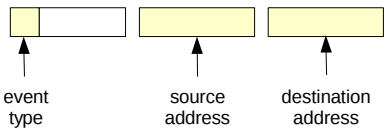

Figure 3: GC move generic event

be recovered from previous events, i.e., from the allocation address of the allocation event or from the destination address of the latest GC move event for this object. If an object is moved into a PLAB during minor collections, a *GC move optimized* event is fired, which omits the destination address. Adjacent objects, which are moved by the same offset during major collections are clustered into a single *GC move region* event. Like the *GC move generic* event, the *GC move region* event carries a source address and a destination address. However, in this case the addresses indicate only the location of the very first object, which is moved. Additionally, the event contains the number of objects which are affected by the move. The addresses of these adjacent objects can then be inferred. A complete description of all dependencies can be found in Bitto et al. [1]. The format of all fired events has been presented in Lengauer et. al [7].

3. APPROACH

The following sections describe our approach, i.e., black-box on-the-fly compression as well as how and when to rotate the trace.

3.1 Compressing the Trace on the Fly

A common way to reduce the size of trace files is to apply conventional compression algorithms. However, since AntTracks already encodes memory events in a highly compact, binary format (as described in Section 2.2), the potential compression rate is limited. Apart from that, compression is only worthwhile if not compromising overall tracing performance. Consequently, only such algorithms are applicable which are able to encode the information on the fly, i.e., in a single-pass while the application is traced. One well-known technique which fits our criteria is the Lempel-Ziv-Welch (LZW) algorithm [16]. The LZW builds up a dictionary of commonly occurring fragments in the input data. Compression is then achieved by substituting variable-length data fragments with the corresponding indexes of the dictionary.

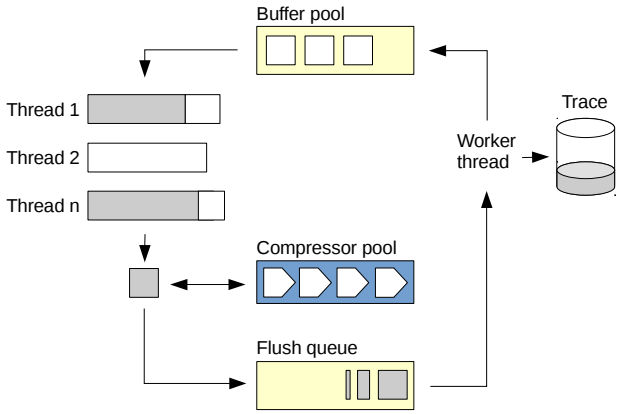

Figure 4: Event buffer management with conditional compression

We decided to introduce compression per tracing thread, i.e., a thread may compress its buffer before enqueuing it into the flush queue (see Figure 4). As a result, compression can take place in parallel without keeping the actual worker thread from writing the trace. We introduced a fixed number, i.e., the number of available cores on the machine, of compressors, each being able to compress a single buffer at a time. The compressors are retained in a dedicated compressor pool. A thread can request a compressor from that pool if he wants to compress its buffer before handing it over to the flush queue. If a compressor is available, the thread extracts it, compresses the buffer, and returns the compressor to the pool. The dictionary built by the compressor is constructed anew each time a thread retrieves a compressor from the pool, and cleared again when returning it to the

pool. Though this limits compression quality, because the compressor can use its compression dictionary for a single buffer only, assigning a compressor per thread would become memory-intensive quickly.

In order to investigate the trade-off between compression rate and run time, we implemented two different strategies: (1) All event buffers are compressed given that enough compressors are available; (2) Only some event buffers are compressed. In the latter case, an event buffer is only compressed if the flush queue is filled up to a certain threshold. Otherwise the uncompressed event buffer is enqueued to prevent the worker thread from becoming idle. Both strategies are evaluated in detail in Section 4.

3.2 Rotating the Trace

Trace rotation enables limiting the absolute size of the trace, which will then never be exceeded. When the trace reaches the given size limit, old events need to be overwritten with new events. To make overwriting easier, we write n small trace files instead of one big trace file. Every trace file starts with an index number, by means of which the analysis tool can reconstruct the correct order of the individual files. When the trace is rotated, we delete the data of the oldest trace file, assign it a new index number, and continue to write the trace into that file.

Consequently, the number of rotations depends on the maximum trace size, as well as on n, i.e., the number of trace files. The absolute trace size can be specified via a VM parameter, e.g., `MaxTraceSize=16G`. In addition to that, one can specify the maximum deviation from the maximum trace size, e.g., `MaxTraceSizeDeviation=0.1`. The actual number of trace files used (n) is then calculated based on the maximum deviation. For example, a maximum size of 16 GB and a deviation of 0.25 (25%) will result in 4 trace files, where every file will be at most 4 GB in size. Thus, there will always be at least three full trace files (12 GB of data) to analyze, even if the trace has just been rotated. Figure 5 shows this example after several iterations.

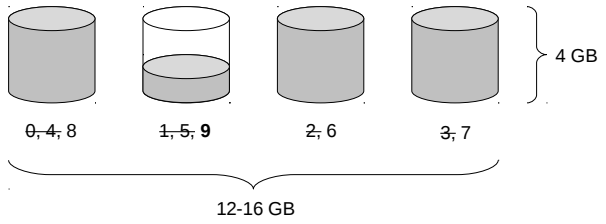

Figure 5: Trace rotation with a maximum trace size of 16 GB and a deviation of 25% after two complete iterations

3.3 Synchronization Points

As described in Lengauer et al. [7], the trace consists of events, i.e., object allocations and object movements. Initially the analysis tool starts with an empty data structure representing the heap. It then interprets every event as an incremental change to that data structure. This assumption is usually correct because every application starts with an empty heap. However, rotating the trace invalidates this assumption, because a part of the trace is missing, meaning the heap is not empty at the start of the trace. Consequently, as every event represents an incremental change to the previous state of the heap, the resulting state will not be correct. Even more so, events may not be parsed correctly because information (that was reconstructible before) is now missing as it was located in the overwritten part of the trace file.

Thus, since every trace file may be the starting point for processing, all trace files must start with a *synchronization point* that contains enough information to reconstruct the heap as it was at that point. A synchronization point is a snapshot of the heap, containing all information that is necessary to parse the subsequent events in the trace correctly. Specifically, the analysis tool needs to know the exact location and the type of every object in the heap at that point in time.

However, iterating through the heap and creating such a snapshot is a tedious task for various reasons: (1) to get a stable snapshot, the VM needs to be suspended; (2) iterating through the entire heap costs time and destroys CPU caches; (3) a large portion of the snapshot will be useless because it is about objects that will die at the next garbage collection anyway; and (4) such a snapshot will take up a considerable amount of space in the trace file.

To cope with these problems, we use a GC (minor or major) to rotate the trace on the fly. During such a GC, we can easily extend existing GC move events (see Section 2.2) by appending an object's type information. Thus, we do not have to write any additional events. Furthermore, the VM is already suspended during a GC and the move events are only emitted for live objects. Figure 6 shows multiple trace files, including GCs and synchronization points.

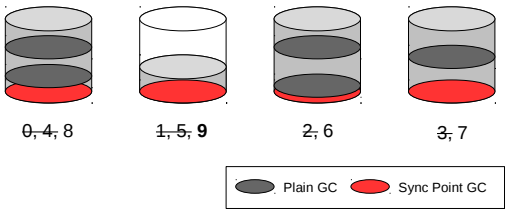

Figure 6: Trace rotation with a maximum trace size of 16 GB and a deviation of 25% after two complete iterations, including GCs and synchronization points

Major GC.

As described in Section 2.2, a move event is written for every live object during a major GC. When we are rotating the trace file at a major GC, all move events are replaced with *GC move sync* events. Such an event (as shown in Figure 7) carries all information of a regular move event plus an additional type identifier. This type identifier will be used to reconstruct the heap data structure, if the trace starts from a synchronization point in the oldest trace file. At synchronization points that are not in the oldest trace file, only the source address and the destination address are used and the type identifier is ignored.

If possible, AntTracks clusters several adjacent objects that are moved by the same distance into a single *move region* event (cf. Section 2.2). However, the analysis tool must know the types of all objects in the region for further analysis. Thus, this optimization is only used if the GC is not a synchronization point. If the GC is a synchronization

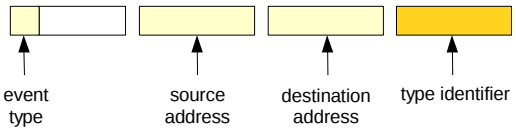

Figure 7: *GC move generic sync* event (plain *GC move generic* event with an additional field representing a type identifier)

point, *move region* events are split into individual *GC move sync* events to include type information.

Minor GC.

If a minor GC is a synchronization point, similarly to a major GC, every *GC move* event is replaced with a *GC move sync* event. As minor GCs use PLABs to move objects across spaces (cf. Section 2), there are also optimized events for moving objects into PLABs. Again, the original events are simply extended by a type identifier.

However, a minor GC does not collect the entire heap, but only the young generation. Consequently, after a minor GC synchronization point, the analysis tool cannot reconstruct the entire heap but only the spaces that have been collected, i.e., the Eden space, the survivor-from space, and the survivor-to space. To overcome this problem, we also iterate through the old space concurrently to the GC and create *GC sync* events for all objects there.

Using a minor GC is not as efficient as using a major GC because events for dead objects in the old generation may be created, and the run time no longer depends on the number of live objects, but on the size of the old space instead. However, it is still much more efficient than waiting for a major GC or, even worse, triggering one.

Deciding When to Rotate.

Usually, a minor GC is triggered only when the Eden space is full. Depending on the fill level of the survivor spaces and the old generation, a major GC will follow immediately or might even replace the minor GC entirely.

Consequently, there might not be the need for a GC when the trace must be rotated. In this case, we can either wait for a GC (and accept overshooting our trace size target), or rotate immediately and trigger a GC to create a synchronization point. Triggering a GC has the disadvantage that the GC behavior of the application is changed, and thus, the resulting trace is distorted.

Therefore, we use a hybrid approach by interpreting the allowed deviation not just as a lower deviation, but also as an upper deviation. For example, if the maximum trace size is 16 GB with 25% deviation (see Figure 8), this means that the trace might be somewhere between 12 GB and 20 GB. This allows us to wait for some time after reaching 16 GB in the hope that a GC will occur soon. If we reach the limit including the deviation, we trigger an emergency GC (minor) to create an artificial synchronization point.

When the trace size target has already been exceeded for a file, every allocation is adding another few bytes to an already over-full file. On the other hand, every allocation keeps filling the Eden space and thus increases the chance for triggering a GC. Thus, there is a linear relation between allocations and the probability of a GC. Therefore, if the absolute deviation is big enough, we will never have to introduce an artificial GC and thus never distort the trace file. We will show this to be correct in Section 4.

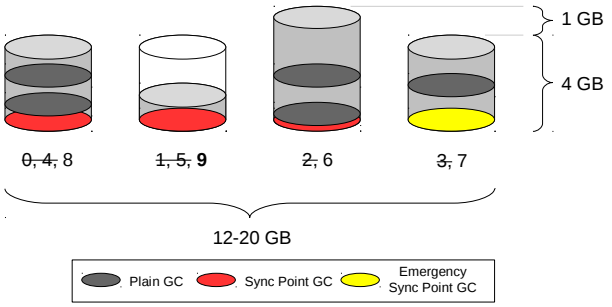

Figure 8: Trace rotation with a maximum trace size of 16 GB and a lower and upper deviation of 25% after two complete iterations, including GCs and synchronization points

3.4 Restoring Allocation Site Information

As described before, synchronization points contain extended events that allow the analysis tool to rebuild the heap data structure. However, these events contain only the bare minimum that is necessary to accomplish this task, i.e., the object's type. The type information is needed for inferring the size of an object. The size is crucial during trace processing in order to know how much space an object takes up in memory. Other information about objects, however, such as the allocating thread or the allocation site, is not included in *GC move sync* events.

The allocation site is the most interesting of the missing information, as it provides a clue about the origin and, thus, the purpose of an object. However, in contrast to the type, the allocation site cannot be inferred from an object. Information about the allocation site exists only during the allocation itself and is subsequently lost because it is not stored in the object.

Storing the Allocation Site.

Consequently, if we want to restore the allocation site information of an object, we need to save and store it during the allocation itself. However, finding a location to store the allocation site for every object is a challenging task.

A naïve approach would be to keep a map with object addresses as keys and allocation sites as values. Obviously, this approach is not feasible as this would require a significant amount of memory and is not trivial to maintain efficiently due to parallel allocations as well as due to the garbage collector moving objects.

The other possibility is to store the allocation site in the object itself. However, an object is already densely packed in order to fit as many objects as possible into the heap. As described in Section 2, there is neither space in the payload section nor in the header section of an object to store the allocation site. We could add an extra field in the object header to store the allocation site. However, this field would have to be at 8 bytes large to ensure proper object alignment. This would lead to an overall increase of the object size, and would thus lead to a GC time increase of at least the same magnitude.

Exploiting the Identity Hash Code.

Inspired by Odeira et al [10], we chose to exploit the identity hash code in the header. The identity hash code of an object does not have to be unique, but should only be as unique as possible to reduce hash collisions. Thus, we store the allocation site into the upper two bytes of the hash code, whereas the lower two bytes remain untouched.

Storing the allocation site into the identity hash code has two drawbacks: (1) An identity hash code must be generated eagerly for every object (usually, it is generated lazily on its first access). (2) The entropy of the identity hash code is reduced because it is now effectively only 2 bytes long.

In Section 4, we will show that the reduced entropy is negligible for the identity hash code. This is due to the fact, that most applications provide their own hash code implementation if they intend to hash objects of a specific class. Exploiting this observation, we were also able to reduce the eager generation of identity hash codes by optimistically not generating them for classes that provide a hash code implementation.

4. EVALUATION

This section provides a detailed evaluation of the black-box compression of traces as well as of their rotation in terms of overhead. Furthermore, it shows the worst-case performance as well as the performance of selected configurations.

Benchmarks.

To evaluate trace rotation, we need benchmarks that allocate a significant amount of objects. Only then, the trace is big enough so that it is rotated a number of times and the measurement can be declared to be significant.

We have examined the well-known DaCapo, DaCapo Scala, and SPECjvm (lagom) benchmark suites to evaluate whether they fit our needs. Not one of these suites had a benchmark that generates enough trace data in a single iteration (not even using the *huge* and *gargantuan* loads for DaCapo and DaCapo Scala). Thus, we have quadrupled the load of every SPECjvm benchmark. For the DaCapo and DaCapo Scala benchmarks, we chose the largest workload because only a fixed number of workloads is available. Furthermore we eliminated benchmarks, i.e., scimark.*, that did not create enough trace data.

In the following sections, we will mostly use only those benchmarks that generate enough trace data to show the impact of individual optimizations. Finally, we will show the overall overhead based on all benchmarks.

Before every measurement, we executed 20 warmups to stabilize JIT compilation. Every number shown in the following sections is the median of multiple runs. Unless otherwise noted, the standard deviation is negligible.

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.

4.1 Compression Overhead

In Section 3.1 we proposed two strategies for enabling on-the-fly compression of the event buffers: (1) All event buffers are compressed; (2) Only some event buffers are compressed. In the latter case, an event buffer is only compressed if the flush queue is filled up to a certain degree. In the following section we provide a detailed evaluation of black-box compression.

Trace size.

The trace size is the most obvious criterion to use for measuring the effectiveness of a compression algorithm. Figure 9 shows the size of the compressed trace relative to the original trace size.

Benchmark	Trace Size Full compression / partial compression	
compiler.compiler		32.1% / 86.0%
compiler.sunflow		29.7% / 88.4%
derby		15.6% / 94.8%
serial		21.0% / 87.0%
sunflow		10.8% / 90.3%
xml.transform		25.2% / 92.1%
xml.validation		25.6% / 89.3%
mean		21.6% / 89.7%

Figure 9: Size of fully compressed trace (blue, left) and size of partially compressed trace (cyan, middle) relative to original trace size (gray, right, 100%)

The second column shows the numbers when compressing all event buffers, while the third column shows the results for partial compression. The event buffer size has been fixed to 16 kilobytes in both cases. Increasing this parameter improves the compression rate and consequently reduces the trace size further. The larger an event buffer, the longer a thread can benefit from using the same dictionary for compression. The threshold of partial compression has been set to 0.2, meaning that buffers are only compressed when the flush queue is at least filled to 20%. Increasing this parameter would worsen the compression performance, since less buffers would be compressed.

Run time.

Though the trace size can be significantly decreased when enabling full compression, the application's run time suffers dramatically. Figure 11 shows the run-time overhead of full compression and partial compression, relative to the run time without compression. While on average the run-time overhead increases about 22% (geometric mean) with full compression, partial compression comes at the cost of only about 2% (geometric mean). Further evaluation showed that the increased run time of full compression can be attributed to the time required for compressing. The reduced IO time, caused by writing less data, cannot compensate for the compression time. Consequently, full compression is impossible without compromising AntTracks' overall monitoring performance. However, partial compression proves beneficial to further reduce the trace size without worsening the application's run time considerably.

Benchmark	Run-time overhead Partial compression / full compression
compiler.compiler	2.4% / 23.8%
compiler.sunflow	4.5% / 24.7%
derby	−0.5% / 25.8%
serial	3.7% / 24.3%
sunflow	2.0% / 15.5%
xml.transform	2.7% / 19.6%
xml.validation	1.4% / 20.1%
mean	2.3% / 21.9%

Figure 11: Run-time overhead of full compression (blue, right) and partial compression (cyan, middle) normalized relative to the run time with disabled compression (gray, left, 100%).

4.2 Rotation Overhead

Figure 10 shows the run-time overhead of tracing with rotation (8 GB maximum trace size with 10% deviation) relative to tracing without rotation on our selected benchmarks. Please note that these are already very allocation-intensive benchmarks, thus the average overhead is lower when using a more random set of benchmarks (see Lengauer et al. [7] and Paragraph Total Overhead Compared to Non-rotated Tracing).

We chose a relatively small maximum trace size (8 GB) intentionally to force benchmarks to rotate more often. Realistically, one would choose a much larger size, e.g, 32 GB or more. For example, doubling the maximum trace size would cut the number of synchronization points in half. Consequently, the performance can be improved by adjusting the maximum trace size. Thus, the following figures represent not the best setup but rather a minimalistic one.

The results show an average run-time overhead of 45% for rotation. This overhead can be reduced further by accepting a loss of data (cf. Paragraph Overhead without Allocation Site Information). The GC time (see right-hand side of Figure 10) is usually higher with rotation due to the additional work that has to be done for a synchronization GC.

Figure 12 reveals the ratio of synchronization GCs to plain GCs. It shows that the relatively small trace size of 8 GB is big enough for these benchmarks, as only 9.2% (geometric mean) of all GCs are synchronization points and none are emergency synchronization points. Figure 13 shows the sizes of trace data caused by garbage collections. It reveals that the required GC sync events are significantly larger than normal GC events. In the case of derby, one minor sync GC requires about 50 MB, while a normal minor GC takes up only 0.22 MB. For compiler.compiler a minor sync GC needs about 360 MB, which means that it occupies 45% of a single 800 MB trace file.

Benchmark	Sync GC / plain GC ratio
compiler.compiler	17.9%
compiler.sunflow	8.2%
derby	8.8%
serial	8.3%
sunflow	7.0%
xml.transform	5.0%
xml.validation	13.2%
mean	9.2%

Figure 12: Ratio of GCs used as synchronization points (green, left) in relation to plain GCs (gray, right), when trace rotation is enabled

Benchmark	Trace size [MB] Minor GC	Major GC	Sync events size overhead
compiler.compiler	25.16	6.03	1378.7%
compiler.sunflow	11.45	-	463.5%
derby	0.22	-	22387.5%
serial	0.10	-	749.8%
sunflow	1.14	-	252.4%
xml.transform	0.61	-	756.6%
xml.validation	12.17	1.43	2087.2%

Figure 13: Size of trace data produced by minor and major GCs, as well as sync events size overhead if a GC is used as a synchronization point

Overhead with Storing Allocation Site Information.

In Section 3 we described how to store allocation site information into identity hash codes, thus being able to restore them after a synchronization point. However, reducing the size of the identity hash code decreases its entropy. By removing 16 bits from the 31-bit identity hash code, we reduce the number of possible values by a factor of $2^{16} = 65536$. We thus reduce the entropy of the hash to $\frac{1}{65536} = 0.0015\%$ of its original entropy. In theory, this can considerably affect the performance of an application that uses hash-based data structures. In order to estimate the worst case over-

Benchmark	Run time	GC time
compiler.compiler	68.7%	41.6%
compiler.sunflow	36.5%	6.6%
derby	63.4%	166.9%
serial	42.8%	5.6%
sunflow	28.0%	4.9%
xml.transform	31.5%	11.1%
xml.validation	49.2%	66.9%
mean	45.0%	35.2%

Figure 10: Run time (lower is better) of tracing with rotation relative to the run time of tracing without rotation (gray, 100%) as well as GC time (lower is better) of tracing with rotation relative to the the GC time of tracing without rotation (gray, 100%), all without saving allocation site information

head of our approach, we designed a benchmark which puts $1 * 10^7$ plain Java objects into a `java.util.HashSet`, and subsequently queries $1 * 10^7$ times, where half of the queried objects are indeed in the HashSet. When saving allocation site information (and thus reducing the entropy of the identity hash code) the run-time overhead rises to 2291%. This overhead can be easily attributed to bad hash performance, which lets the hash map degenerate into a few long lists and thus, turns *put* and *get* into operations with linear complexity.

In practice, however, the overhead is not as bad. This is mainly because most applications make little use of the identity hash code. Figure 14 shows the run-time overhead when allocation sites are stored in objects, relative to when they are not saved.

Benchmark	Run-time overhead
compiler.compiler	2.9%
compiler.sunflow	2.9%
derby	5.5%
serial	1.8%
sunflow	1.4%
xml.transform	6.4%
xml.validation	−0.1%
mean	2.9%

Figure 14: Run-time overhead when saving allocation sites relative to rotation without saving allocation sites

As expected, the run time slightly increases due to the decreased entropy of the hash (if the allocation site is stored), which distorts the performance of hash-based data structures, e.g., HashMaps and HashSets. Most of the benchmarks use their own implementation of hash code computation or simply do not use hash-based data structures extensively, so their run time increases only slightly. Xml.transform, however, rely on the identity hash code and thus it performance drops more significantly. This benchmark allocates a high amount of objects of class `...dom.SimpleResultTreeImpl.SimpleIterator`, which in return uses a HashMap based on identity hash codes.

Overhead with Eliminating Identity Hash Codes.

As discussed in Section 3, we can eliminate a lot of eager hash code generations by checking whether the allocated class provides its own hash code implementation. Figure 15 shows the number of eliminated hash code generations relative to the number of allocated objects. However, Figure 16 shows that there is no significant run-time reduction if the hash code generation is omitted. In case of serial, the run time even increases, because this benchmark relies on identity hash codes and the check for avoiding eager hash code generation costs some time.

Total Overhead Compared to Non-rotated Tracing.

Figure 17 shows the run-time overhead and the GC-time overhead when enabling trace rotation, storing allocation site information and eliminating hash codes, relative to our non-rotating tracing approach described in Lengauer et al. [7]. This time, not only selected benchmarks are evaluated, but we evaluate the full DaCapo, the DaCapo Scala and

Benchmark	Eliminated hashes
compiler.compiler	41.1%
compiler.sunflow	37.7%
derby	78.0%
serial	32.6%
sunflow	0.0%
xml.transform	36.9%
xml.validation	24.7%
mean	10.0%

Figure 15: Eliminated hash code generations relative to all object allocations

Benchmark	Run-time overhead
compiler.compiler	1.0%
compiler.sunflow	−0.4%
derby	−4.3%
serial	12.8%
sunflow	0.8%
xml.transform	−2.7%
xml.validation	0.1%
mean	0.9%

Figure 16: Run-time overhead when avoiding eager hash code generation

the SPECjvm benchmark suites. Only the SPECjvm scimark benchmarks have been excluded, since they produce traces of negligible size. Note, that especially in the DaCapo benchmark suite, some benchmarks (e.g., batik) do not fire a single GC. Nevertheless, they show a slight runtime overhead, caused from storing allocation sites, generating hash codes and eliminating hash codes. The same applies to benchmarks which are garbage collecting, but do not fire a single sync GC (cells in fourth column with 0% sync GCs ratio), because their data fit in a single trace file within one benchmark iteration. Especially DaCapo Scala kiama is sensitive to storing allocation site information into the hash code (run-time overhead of 103%, GC-time overhead of 75%), as it extensively allocates hash maps and hash entries. Comparing h2 and factorie reveals that the sync events themselves do not constitute the overhead, because although h2 generates sync event data of considerable size, this does not affect its run time negatively. On the other hand, factorie suffers most in terms of run time (about 323%), although the number of sync GCs are comparable to h2 and the size of sync event data is even smaller. However, experiments showed that if allocation site information is not stored, the overhead is reduced to 41%. This indicates that factorie makes use of identity hash codes and storing allocation sites as well as eliminating hash codes have a negative impact on performance. To achieve acceptable performance on those benchmarks, saving allocation site information and eliminating hash codes can be turned off. Benchmarks with a high amount of long-living objects, i.e,. objects which reside in the old generation, perform worse (c.f. Paragraph *Rotation Overhead*), since for every minor sync GC the old space needs to be traversed, causing the application to stall longer as needed. Consequently, how efficient trace rotation can be achieved depends mainly on the used data structures in the application and on the number of objects in the old

DaCapo	Run-time overhead	GC-time overhead	Sync GC / plain GC ratio	Sync events size [MB]
avrora	0.0%	0.0%	0.0%	/
batik	4.7%	/	/	/
eclipse	11.4%	/	/	/
fop	21.5%	/	/	/
h2	−2.8%	28.4%	6.4%	349.9
jython	11.8%	−14.0%	0.0%	/
luindex	0.0%	/	/	/
lusearch	3.3%	0.0%	0.0%	/
pmd	0.4%	5.5%	0.0%	/
sunflow	1.2%	6.6%	0.0%	/
tomcat	0.8%	0.0%	1.5%	3.3
tradebeans	0.4%	16.4%	6.2%	156.4
tradesoap	14.4%	1.7%	3.9%	85.6
xalan	3.9%	2.1%	0.0%	/
DaCapo Scala				
actors	−3.4%	−2.3%	1.6%	1.0
apparat	14.0%	9.3%	2.7%	26.5
factorie	323.0%	83.7%	8.8%	133.1
kiama	103.3%	75.0%	0.0%	/
scalac	3.2%	11.3%	0.0%	/
scaladoc	2.5%	−4.3%	0.0%	/
scalap	4.6%	/	/	/
scalariform	3.7%	−15.3%	0.0%	/
scalab	0.4%	−3.3%	0.0%	/
tmt	47.7%	27.7%	2.8%	3.5
SPECjvm				
compiler.compiler	75.4%	43.1%	11.7%	340.6
compiler.sunflow	39.8%	6.5%	7.4%	59.9
compress	−0.0%	0.0%	0.0%	/
crypto.aes	0.2%	0.0%	0.0%	/
crypto.rsa	1.2%	−5.0%	0.0%	/
crypto.signverify	2.9%	−1.4%	0.0%	/
derby	64.9%	204.9%	8.6%	68.4
mpegaudio	−0.1%	0.0%	0.0%	/
serial	64.1%	7.6%	7.6%	0.8
sunflow	31.0%	3.6%	6.6%	3.7
xml.transform	36.2%	7.4%	4.7%	5.7
xml.validation	49.0%	68.0%	10.7%	276.1
mean	6.5%	15.5%	5.2%	40.0

Figure 17: Run-time overhead and GC-time overhead when rotation is enabled

generation. In general, the additional overhead for rotation is low, with about 6.5% run-time overhead (geometric mean) and 15.5% GC-time overhead (geometric mean).

4.3 Information Quality after Rotations

When a truncated trace is parsed, some properties of live objects that have been allocated before the synchronization point are unknown, e.g., the allocating thread and the allocation site (if it is not saved). We call the amount of information that is available about live objects the information quality, which is defined as a value between 0 and 1, 0 meaning that there is information missing about every object in the heap, and 1 meaning that we have complete information about all objects.

Figure 18 shows the quality for every benchmark right after a synchronization point (GC #0) as well as for the following 9 GCs (which have not been used for synchronization). The general high information quality can be explained by the large number of short-lived objects resulting in very few objects surviving a GC. Compiler.sunflow, derby, and xml.transform quickly gain a quality between 83% and 93% and do not show any significant improvement over the next GCs. The lack of improvement is caused by a pool of long-living objects that are kept alive for the entire application's life cycle and thus impede a better information quality once

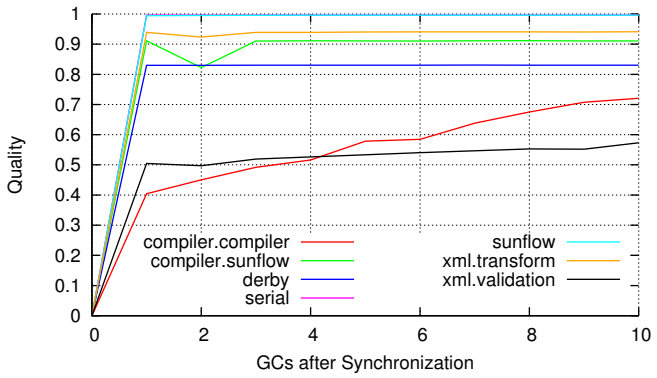

Figure 18: Information quality (percentage of objects with full information) after a synchronization point and after the following 9 GCs

their allocation is cut off. Please note the anomaly at the second GC of the compiler.sunflow benchmark in which the quality drops from 91% to 82% and rises back to 91%. This behavior is caused by a major GC collecting a large portion of objects and thus reducing the overall number of objects in the heap. In absolute numbers, the information quality is not dropping. Finally, compiler.compiler and xml.validation start off with a rather poor quality between 40% and 50%. However, as expected, the information quality increases with every GC as new objects replace old ones.

4.4 Rethinking Hash Strategies

VM implementers have spent a lot of time on thinking about good hash strategies for the identity hash code in order to increase the performance of hash-based data structures. There are multiple hash strategies of varying complexity and efficiency, such as the *Unguarded Global Park-Miller Random Number Generator (RNG)* by Park and Miller [11], the *Stable Stop-the-world (STW) with Address* hash, a *Constant Value* (for testing only), a *Global Counter* that is incremented for every object, the *Address* hash (using the lower bits of the address), and *Marsaglia's xor-shift Scheme* by Marsaglia [8]. Although the latter is the default implementation in the Hotspot™ VM, it can be overridden by setting the `-XX:hashCode` parameter.

However, when reducing the identity hash code from 31 to 15 bits, a different hash strategy might yield a better hash performance. Thus, we conducted an experiment in which we executed our synthetic worst-case benchmark (cf. Section 4.2), reduced the identity hash code to 15 bits, and tried all 6 hash code strategies mentioned above. Figure 19 shows the results of this experiment, revealing that almost all strategies perform as good as the default strategy.

The *Constant Value* strategy resulted in a timeout after taking more than 100 times as long as the default strategy, which is not surprising because it lets every hash map degenerate to a list. The *Global Counter* strategy however, resulted in a performance gain of 18.5%.

Hashing strategy	Run time
Park-Miller RNG	−0.1%
Stable STW w/ Address	−1.3%
Constant Value	
Global Counter	−18.5%
Address	2.5%
Marsaglia's xor-shift	base

Figure 19: Performance of our worst-case benchmark with different hash code strategies when reducing the identity hash code by 16 bits

5. RELATED WORK

There is substantial work on how to record the memory behavior of applications. Chilimbi et al. [4] propose a binary trace format for allocation and deallocation events. Hertz et al. [5, 6] capture object allocations, pointer updates as well as approximate object lifetimes. Ricci et al. [13, 14] additionally record method entry, method exit and exception events. While all of these approaches produce a large amount of data, none of them reveals the size of the generated trace. Neither Hertz et al. nor Ricci et al. present strategies for reducing the trace size. Only Chilimbi et al. discuss the performance of different encoding strategies to compress a trace. Extensive work on compressing data in general has been published. Amongst others, Brown et al. [2] and Burtscher et al. [3] present algorithms for especially encoding trace data efficiently. Although compression algorithms reduce the overall size of a trace, they cannot prevent its continuous growth.

Other approaches try to find strategies for reducing the trace data altogether. In the work of Printezis and Jones [12] the live heap of a running application can be monitored and traced if requested. Users can reduce the amount of collected data by filtering the information they are interested in. However, since we strive for reproducing the entire heap, we cannot simply omit specific object allocation events or object move events. Moreover, reducing the amount of data written just delays the problem of growing traces rather than solving it. Mohror and Karavanic [9] propose to achieve trace reduction by defining similarity metrics between sections of traces. However, the reduced traces are not accurate representations of the original trace. Instead, one has to deal with the introduced error. In AntTracks, an accurate event trace is indispensable for reconstructing precise snapshots of the heap. Consequently, errors in traces cannot be accepted. Wagner and Nagel [15] present strategies for keeping events in a single memory buffer. They intend to evolve fully in-memory event traces to overcome file system interactions. However, reducing the trace once the memory buffer is exhausted, e.g., by omitting information which is least important to the user or by simply stopping recording, is hardly applicable for AntTracks. While partial information of a heap may allow detecting local performance bottlenecks, e.g., of a single thread, it is insufficient for tracking down performance degradations over time, e.g., caused by garbage collection.

6. FUTURE WORK

The work presented in this paper is part of a larger project, which precisely records the memory behavior of Java applications. We plan to further investigate this field of research. Regarding trace reduction we plan to refine our strategies to improve information quality and reduce the introduced run-time overhead even more.

Compression.

Since we face a trade-off between compression rate and run time, full compression is not favorable in AntTracks. However, partial compression can be further improved, by applying encoding more thoughtfully. One can argue that data from event buffers of mutator threads can be compressed more tightly than event buffers of GC threads. This is due to the fact, that GC move events hold more distinct data than allocation events, i.e., GC move events contain unique addresses, while allocation events from the same allocation site all look similar. Consequently, it is worthwhile to apply compression more eagerly during mutator phases and more lazily during GC phases.

Similarly, when requesting a compressor, threads that produce large amounts of trace data can be favored over others that produce less trace data. In the current implementation the compression dictionary is constructed anew each time a thread retrieves a compressor from the pool. If the compressor is returned to the pool, the compression dictionary is cleared. As a result, a thread cannot reuse patterns of multiple mutator phases or GC phases. If a thread would own its private compressor, recurrent patterns of multiple phases could be exploited for encoding. We have not implemented this strategy so far, because it gets memory-intensive quickly and potentially distorts the actual memory behavior we want to monitor. However, threads producing lots of trace data would benefit from this strategy.

Rotation.

The performance of trace rotation depends mainly on two key factors: (1) The number of objects in the old generation, which need to be traversed in the case of a minor sync GC; (2) The number of live objects for which the allocation sites is stored. While the former will affect especially the performance of applications with long-living objects, the latter harms the entropy of hash-based data structure and thus, applications relying on them. Ideally, we could omit the traversal of the old generation altogether and reduce the number of stored allocation sites.

The former could be achieved by exceeding the maximum trace size, in order to wait for an upcoming major GC. This major GC could then be exploited as synchronization GC, and thus, reduce the overall GC-time overhead as well as run-time overhead of trace rotating. This approach may be especially feasible for allocation intensive real-world applications.

The latter can be tackled by avoiding to retain allocation sites for all objects. According to generational collection, the majority of young-generation objects tend to die anyway during the next collection. To estimate, whether an object is likely to die, we can inspect the object's age field, which records the number of times an object has already been evacuated. If an allocation site creates objects that turn out to die quickly, we can refrain from storing their allocation site. This strategy would allow us to reduce the generated data while at the same time not weakening the information quality. Reducing the number or the size of sync events is also beneficial because they currently require several hundred megabytes per GC (see Section 4.2). The more space is taken up by synchronization points, the less space is left for ordinary events and the sooner another trace file is needed that requires another synchronization point. However, omitting *GC move sync* events altogether is not feasible, since our analysis tool relies on at least the object type information to obtain the object's size.

7. CONCLUSIONS

We have presented novel techniques for reducing the size of event traces, as well as for limiting their maximum growth. On the one hand, partial on-the-fly black-box compression enables us to write an even more compact trace without compromising AntTracks' overall run-time performance. On the other hand, rotating trace files allows us to overcome disk limitations.

We showed that the quality of the trace data stabilizes quickly, with an average quality of 74% after the very first GC. In general, the additional run-time overhead of rotation (6.5%) is sufficiently small considering the DaCapo, the DaCapoScala and the SPECjvm benchmark suites. For allocation intensive benchmarks (selected from the SPECjvm benchmark suites), we achieved a run-time overhead of (45%) when limiting the maximum trace size to 8 GB. Certainly, raising the maximum trace size would further improve our results.

Tracking down performance bottlenecks in large and complex Java applications is crucial. While existing tools often record only coarse-grained data or ignore the practical problem of continuously growing traces, we proposed solutions for both problems. Our approach allows a precise representation of all object allocations and object moves. With the help of trace rotation we made a first step towards continuous and almost loss-free tracing.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

[1] V. Bitto, P. Lengauer, and H. Mössenböck. Efficient rebuilding of large java heaps from event traces. In *Proc. of the 12th ACM/SPEC Int'l. Conf. on Principles and Practice of Programming on the Java Platform: virtual machines, languages, and tools*, PPPJ '15, 2015.

[2] R. Brown, K. Driesen, D. Eng, L. Hendren, J. Jorgensen, C. Verbrugge, and Q. Wang. Step: A framework for the efficient encoding of general trace data. *SIGSOFT Softw. Eng. Notes*, 28(1):27–34, Nov. 2002.

[3] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam. The vpc trace-compression algorithms. *IEEE Trans. Comput.*, 54(11):1329–1344, Nov. 2005.

[4] T. Chilimbi, R. Jones, and B. Zorn. Designing a trace format for heap allocation events. In *Proc. of the 2nd Int'l. Symposium on Memory Management*, ISMM '00, pages 35–49, New York, NY, USA, 2000. ACM.

[5] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Error-free garbage collection traces: How to cheat and not get caught. In *Proc. of the 2002 ACM SIGMETRICS Int'l. Conf. on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, pages 140–151, New York, NY, USA, 2002. ACM.

[6] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Generating object lifetime traces with merlin. *ACM Trans. Program. Lang. Syst.*, 28(3):476–516, May 2006.

[7] 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.

[8] G. Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 7 2003.

[9] K. Mohror and K. L. Karavanic. Evaluating similarity-based trace reduction techniques for scalable performance analysis. In *Proc. of the Conf. on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 55:1–55:12, New York, NY, USA, 2009. ACM.

[10] R. Odaira, K. Ogata, K. Kawachiya, T. Onodera, and T. Nakatani. Efficient runtime tracking of allocation sites in java. In *Proc. of the 6th ACM SIGPLAN/SIGOPS Int'l. Conf. on Virtual Execution Environments*, VEE '10, pages 109–120, 2010.

[11] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Commun. ACM*, 31(10):1192–1201, Oct. 1988.

[12] T. Printezis and R. Jones. Gcspy: An adaptable heap visualisation framework. In *Proc. of the 17th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*, pages 343–358, 2002.

[13] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant tracks: Generating program traces with object death records. In *Proc. of the 9th Int'l. Conf. on Principles and Practice of Programming in Java*, PPPJ '11, pages 139–142, New York, NY, USA, 2011. ACM.

[14] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant tracks: Portable production of complete and precise gc traces. In *Proc. of the 2013 Int'l. Symposium on Memory Management*, ISMM '13, pages 109–118, New York, NY, USA, 2013. ACM.

[15] M. Wagner and W. E. Nagel. Strategies for real-time event reduction. In *Proc. of the 18th Int'l. Conf. on Parallel Processing Workshops*, Euro-Par'12, pages 429–438, Berlin, Heidelberg, 2013. Springer-Verlag.

[16] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984.