JOHANNES KEPLER
UNIVERSITÄT LINZ

Submitted by
Dipl.-Ing. Philipp Lengauer

Submitted at
Institute for System Software

Supervisor and 1st Examiner
o.Univ.-Prof. Dipl.-Ing. Dr. Dr.h.c. Hanspeter Mössenböck

2nd Examiner
Associate Professor Doc. Ing. Petr Tuma, Dr.

February, 2017

Accurate and Efficient Memory Monitoring in a Java Virtual Machine

Doctoral Thesis
to obtain the academic degree of
Doktor der Technischen Wissenschaften
in the Doctoral Program
Engineering Sciences

Abstract

Modern applications are often written in an object-oriented style. They allocate large amounts of objects and rely on automatic garbage collection to reclaim them again. Memory-related problems such as out-of-memory exceptions, long garbage collection pauses or performance degradation due to excessive garbage collection activity are hard to find without proper tools. Common memory monitoring tools record the memory state of an application either by sampling (i.e., by periodically taking a snapshot of the memory state) or by instrumentation (i.e., by adding code snippets that record memory events such as object allocations or garbage collector activity). Sampling is often too coarse to track down intricate memory problems, while instrumentation often generates too much overhead.

This thesis presents the design and implementation of AntTracks, a novel Java virtual machine that is able to record memory events at object-level with negligible distortion of the application's behavior. The design includes a novel format for memory events designed to be compact and GC-independent, enabling a generic post-processing tool to analyze the memory behavior without any internal knowledge about the GC algorithm that generated the trace. Also, we describe how to generate these events efficiently in just-in-time compiled code and how to increase the trace accuracy with VM-internal information. The thesis also addresses the fact that capturing events at object-level leads to traces that are too big be stored in their full length. It proposes and evaluates several techniques for keeping traces small and manageable.

We also present novel techniques for parsing and analyzing such traces. To the best of our knowledge, our approach is the first that captures information at object-level and still only requires as much heap memory as the monitored application. The AntTracks tool is able to show an overview of the memory behavior throughout the application's lifetime. It can reproduce the heap for any point in time, enabling detailed analyses about the heap state. Furthermore, the tool can analyze the changes to the heap over time, enabling analyses about trends in the heap.

We provide both a qualitative evaluation, showing that the AntTracks virtual machine only imposes very little overhead compared to other state-of-the-art approaches, as well as a functional evaluation, showing how the recorded information can be used to track down memory-related performance problems.

Kurzfassung

Moderne Programme sind oft im objektorientierten Stil implementiert. Das heißt, sie legen große Mengen an Objekten an und verlassen sich auf die automatische Speicherbereinigung, um sie wieder freizugeben. Sollten sich nun Probleme im Speicher auftun, sind diese jedoch ohne geeignete Tools/Hilfsprogramme nur schwer zu finden. Aktuelle Werkzeuge beobachten das Speicherverhalten auf zwei Arten: entweder durch Abtasten (dh. durch periodisches Erfassen des gesamten Speicherzustandes) oder durch Instrumentierung (dh. durch Einfügen von Anweisungen um einzelne Allokationen aufzurechnen). Abtasten ist allerdings oft zu grob, um komplizierte Speicherprobleme aufzuspüren, während Instrumentierung große zusätzliche Laufzeitkosten verursacht.

Diese Arbeit entwirft und implementiert AntTracks, eine neuartige Java Virtuelle Maschine, die in der Lage ist, Speicherereignisse auf Objektebene aufzuzeichnen, dabei aber das Verhalten des beobachteten Programms nur minimal verzerrt. Ihr Design enthält ein neues Format für kompakte und implementierungsunabhängige Speicherereignisse, die von dem AntTracks Werkzeug verarbeitet und analysiert werden können, ohne, dass es Wissen über den benutzten Algorithmus zur automatischen Speicherbereinigung benötigt. Weiterhin wird beschrieben, wie diese Ereignisse effizient in synchron erstelltem Code aufgezeichnet werden können und wie man deren Genauigkeit auf Basis von internen Informationen der virtuellen Maschine verbessern kann. Außerdem beschäftigt sich die Arbeit auch mit dem Problem, dass die Erfassung auf Objektebene zu sehr vielen kleinen Ereignissen führt, die nicht alle auf unbestimmte Zeit gespeichert werden können. Abschließend werden neuartige Techniken zur Analyse von Speicherereignissen aufgezeigt.

Nach unserem Wissen, ist dies der erste Ansatz, der die Informationen auf Objektebene verarbeiten kann und dabei maximal genauso viel Speicher wie das überwachte Programm benötigt. Das AntTracks Werkzeug ist in der Lage eine Übersicht über das Speicherverhalten der gesamten Lebensdauer des Programms zu zeigen. Das bedeutet, es kann den Speicher für jeden Zeitpunkt rekonstruieren, was detaillierte Analysen über den Inhalt ermöglicht.

Diese Arbeit bietet sowohl eine qualitative Bewertung, als auch eine funktionale Bewertung der AntTracks. Dabei werden die Verzerrung des Programms und die zusätzlichen Laufzeitkosten während der Überwachung angezeigt, und darüber hinaus wird beschrieben, wie man mit den aufgezeichneten Ereignissen speicherbedingte Probleme finden kann.

Contents

Chapter 1

Introduction

"The story so far: In the beginning the Universe was created. This has made a lot of people very angry and been widely regarded as a bad move." - Douglas Adams

Automatically managed memory, i.e., garbage-collected memory, has gained wide-spread use in todays applications because it relieves programmers from the error-prone task of freeing memory manually. Moreover, many modern garbage collectors tend to compact the heap, leaving the heap in an unfragmented state, and thus enabling fast bump-pointer allocations.

Being able to allocate memory fast has promoted objects to first-class citizens in modern virtual machines (VMs). Consequently, developers heavily use object-oriented programming patterns, producing objects that differ significantly in terms of life time and in their relations to each other. For example, some patterns, such as *Strategies*, *Factories*, and *Singletons*, produce only few objects with a long life time. Other patterns, such as *Visitors*, *Iterators*, and *Commands*, create many objects with only a short life time but many interconnections. Additionally, patterns like *Caches* and *Pools* create objects with almost unpredictable life times, that may keep big data structures alive recursively.

Todays applications mix a wide variety of programming patterns, resulting in a complex overall memory behavior. This has led to the development of many different garbage collection algorithms, designed to handle different kinds of applications. Moreover, almost every garbage collector (GC) uses a combination of those algorithms and a multitude of configurable and adaptive heuristics to better cope with volatile behaviors and to enable the GC to adapt to new situations quickly.

While object allocations have a direct and easy-to-understand performance impact, garbage collection times are hard to predict. This is partially due to the fact, that a GC may collect the entire heap or only a subset of regions, depending on the chosen algorithm and the observed application behavior. Additionally, a GC may need to suspend the application for the duration of the collection. Consequently, the GC impacts the overall application performance (because the application cannot proceed when paused) as well as its availability and responsiveness.

Researchers as well as engineers have created a wide variety of tools to monitor memory behavior.

Some of these tools are simple visualizations based on log files whereas others work with sophisticated instrumentation algorithms to monitor a multitude of behavioral aspects. However, all tools so far suffer from the same information quality versus overhead tradeoff. This means, the information that is recorded and displayed is either very coarse, i.e., hardly detailed enough to diagnose the existence of a problem (let alone to track down the cause) but efficient, or it provides every bit of information one might need to track down the root cause but imposes so much overhead that the application's memory behavior is completely distorted, making the results of the analysis questionable at best.

1.1 Research Questions

This section discusses the three main research questions that will be addressed throughout this thesis.

1.1.1 Quantifying Application Behavior Distortion

The first research question addresses the observer effect in memory monitoring. To the best of our knowledge, there is no scientific work done on the observer effect in memory monitoring. Some monitoring tools provide a rough estimate about their overhead, some scientific work (cf. Section 9) is a little more specific in terms of run-time overhead and memory overhead. However, nobody has reported on the actual effects on GC time and application behavior when using state-of-the-art monitoring techniques such as memory dumps and instrumentation. **Research Question 1**: How big is the observer effect in memory monitoring?

1.1.2 Accurate and Efficient Monitoring

The second research question will deal with building a virtual machine that defeats the tradeoff of information granularity and overhead. All state-of-the-art tools either provide only very coarse-grained information, impose significant overhead that completely distorts application behavior, or build upon research VMs that are not usable for production deployment. Therefore, we need to retrofit a modern production VM that is able to provide fine-grained information without invalidating the analysis by imposing too much overhead. **Research Question 2**: How can we retrofit a modern production virtual machine to support accurate and efficient memory monitoring, enabling the reconstruction of the monitored application's heap at object level for any point in time?

1.1.3 Memory-related Performance Problems

The third research question deals with the detection and diagnosis of memory-related performance problems. It should explore how we can detect the presence of memory-related performance problems as well as whether the root cause can be determined. **Research Question 3**: What kinds of memory-related performance problems can be detected and tracked down to the root cause with information describing the full memory behavior at object-level?

1.2 Structure

This thesis is structured as follows: Section 1.3 discusses its contributions. Section 2 explains the state of the art in terms of modern allocation and garbage collection algorithms as well as in terms of monitoring tools with the capability to monitor memory behavior. Section 3 shows the observer effect when monitoring memory behavior with state-of-the-art methodology by example. Section 4 presents the design of a Virtual Machine that can monitor the memory behavior of any Java application at a fine-grained level with minimal overhead, and the format of the memory trace that is generated when recording. Section 5 shows how to efficiently reconstruct the heap for any point in time based on the trace presented in the previous section. Section 6 describes how to analyze the generated information to track down root causes for memory-related performance degradations. Section 7 shows several real-world memory-related performance problems and how they can be resolved with the techniques presented in this thesis. Section 8 presents a detailed evaluation in terms of overhead. Section 9 discusses related scientific work. Finally, Section 10 concludes this thesis.

1.3 Contributions

This thesis has four main contributions:

It presents (1) the design and implementation of a specialized Java VM that supports fine-grained memory monitoring at object level. Compared to state-of-the-art approaches, this VM only introduces minimal overhead and negligible distortion of the monitored application's behavior. This was achieved by (1.a) a compact format for memory event traces that exploits knowledge about operation frequencies to focus optimizations on them. Also, (1.b) events are generated efficiently by integrating the event firing mechanism into the JIT-compiled code and the Garbage Collector directly. The resulting event trace (1.c) exploits VM-internal knowledge about allocations sites, stack traces, and threads to improve the trace's accuracy. The specialized VM can also (1.d) capture pointers between objects efficiently by recording pointer values only at garbage collections and piggybacking pointer values onto GC events. It also (1.e) does not impede multi-threaded applications because it uses a thread-local buffering mechanism to record events in parallel without completely losing the temporal ordering of events. Finally, the tracing mechanism includes (1.f) techniques to rotate the trace to allow continuous monitoring for long-running applications.

This thesis also presents (2) an efficient technique to parse and analyze a memory trace. The presented approach includes (2.a) a technique to parse the trace in parallel without violating dependencies and (2.b) a sophisticated data structure to manipulate a virtual heap in parallel. Also, it enables to (2.c) reconstruct the heap for any point in time and to analyze that heap based on user-defined criteria. Additionally, the user can (2.d) analyze the changes to user-defined properties of the heap over an arbitrary interval. Furthermore, this approach (2.e) can analyze the heap with at most the same heap memory as the monitored application.

We also present (3) a detailed evaluation. We show the (3.a) extent of the observer effect in state-of-the-art memory monitoring. The evaluation also includes a (3.b) performance evaluation of the specialized VM, showing a minimal run-time overhead as well as that the introduced distortions are negligible. We also present (3.c) several case studies showing how three kinds of memory-related performance degradation can be tracked down using the approach presented in this thesis.

Finally, we provide an (4) extensive review of the capabilities of both academic as well as industrial tools in terms of memory monitoring.

Most of the work presented in this thesis has been published at international peer-reviewed conferences. In Lengauer et al. [26] we described the problem of memory monitoring, the deficits of current memory monitoring tools, and first experiments in terms of providing custom VMs for memory monitoring. In Lengauer et al. [27] we presented a custom virtual machine that is able to track memory usage per predefined feature, whereupon a feature is a user-assigned piece of source code. In Lengauer et al. [32] we showed how to automatically optimize GC parameters with an adaptive hill-climbing algorithm. We extended this work in Blaschek et al. [8] to reduce optimization time by identifying key parameters in terms of performance. In Lengauer et al. [29] we first presented our retrofitted virtual machine that is able to produce an object trace with minimal overhead as well as the employed trace format. In Bitto et al. [4] we showed how to reconstruct the heap based on that previously used trace format using less heap memory than the original application. We extended this work in Lengauer et al. [30] with methods to compress the trace as well as with a rotation mechanism for the trace to deal with long-running applications. With Bitto et al. [3] we presented a study on the observer effect in memory monitoring, which also included a tutorial at the conference. In Lengauer et al. [28] we extended our approach by also tracing pointers to reproduce object graphs. In Weninger et al. [48] we presented a novel approach for user-defined analyses and classification of heap states. In Lengauer et al. [31] we used AntTracks to study several state-of-the-art benchmark suites in terms of memory behavior.

Chapter 2

State of the Art

"If we knew what it was we were doing, it would not be called research, would it?"
- Albert Einstein

This section presents the state of the art in terms of object allocation techniques and garbage collection algorithms in managed languages. Although the algorithms are described in the context of Java, they are also applicable and used in many other managed languages, for example .NET-based languages. It also presents state-of-the-art memory monitoring techniques as well as popular memory monitoring tools.

2.1 Allocations

As objects are first-class citizens in modern managed languages, especially in languages like Java and .NET, allocating new objects is a critical operation in terms of performance. Thus, todays VMs employ a wide variety of strategies to enable fast object allocations.

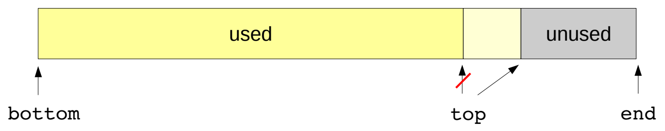
2.1.1 Free List Allocations

One of the earliest allocation algorithms employed are *free list allocations*. This allocation technique relies on all free memory areas being linked in a list. Objects are allocated by finding a free block in the list that is at least as big as the required object, and potentially returning the unused part of the block to the free list. As allocations always involve searching a suitable block in the list, this is the most expensive allocation strategy. Free list allocations are used in many unmanaged languages, i.e., C's `malloc`, and also in some modern GCs, e.g., Oracle's Concurrent Mark and Sweep.

To determine whether a free block is suitable for a new object, two different strategies can be used: (1) first fit, i.e., use the first block that is big enough to allocate the object into, and (2) best fit, i.e, select the smallest block the object fits into. While the first strategy can allocate objects faster, because it is very likely that a fitting block will be found quickly (especially for small objects), the latter minimizes heap fragmentation, thus making better use of the available memory. Knowlton et al. [25] first proposed to merge freed blocks to obtain fewer but bigger entries in the free list. As both strategies provide a unique advantage, they are often combined. Using a hybrid strategy, instead of a single list, a small group of free lists is maintained, whereupon every list only contains blocks of a predefined size range (e.g., the first list contains only blocks of 32 bytes or smaller, the second list only blocks of 32 to 64 bytes, and so on). The allocation algorithm then selects the appropriate list depending on the size of the object that must be allocated, and selects the first fitting block within that list. This has the advantage that we can quickly find a memory block while reducing fragmentation compared to the plain first fit strategy.

2.1.2 Bump Pointer Allocations

The bump pointer allocation strategy guarantees O(1) run-time complexity for every allocation. This strategy assumes that the heap can be divided into two parts, i.e., the used part and the unused part, i.e., it works only on unfragmented heaps. An unfragmented heap can be described with three pointers: (1) the bottom pointer, pointing to the start of the used part, (2) the top pointer, pointing to the end of the used part (exclusively) and the start of the unused region (inclusively), and (3) the end pointer, pointing to the end of the unused region. In this case, allocating an object is as easy as incrementing the top pointer by the size of the new object (i.e., bumping the top pointer), cf. Figure 1.



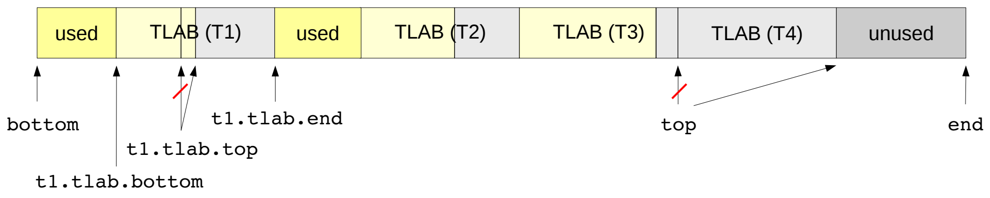
```
void* new_top = top + obj_size;
if(new_top >= end) allocation failure
void* new_obj = top;
top = new_top;
```

Figure 1: Allocating a new object with the bump pointer allocation strategy in an unfragmented heap

2.1.3 Thread-local Allocation Buffers

Although bump-pointer allocations already provide excellent run-time complexity in theory, their actual performance is rather poor because, in multi-threaded environments, many threads start racing for the top pointer. Therefore, modern VM's use bump pointer allocations in combination with thread-local allocation buffers. Using this strategy, the heap is still assumed to be unfragmented, but rather than allocating objects one by one, every thread allocates a large memory area, i.e., a thread-local allocation buffer (TLAB). Thus, every thread owns its private TLAB, and can allocate objects into that TLAB (using bump-pointer allocations again) without racing for the top pointer because every thread has its own bottom, top, and end pointer managing his TLAB. Threads still race for the global top pointer when they need to allocate a new TLAB, but not at every allocation.

Figure 2 shows several threads allocating in parallel without racing for the same top pointer. In that example, thread 1 is currently allocating a new object in its own private TLAB. The thread needs to check whether there is still enough space in its TLAB, and can then allocate the object using the bump pointer allocation technique, without needing to care about synchronizing with other threads. Thread 4 is currently in the process of allocating a new TLAB. In this case, the top pointer must be locked, but allocating a new TLAB is a rare operation compared to object allocations.



```
if(t1.tlab.top + sizeof(new_obj) >= t1.tlab.end)    t4.tlab.bottom = top;
  t1.tlab = allocate_new_tlab();                    t4.tlab.top = top;
void* new_obj = t1.tlab.top;                        top = top + tlab_size;
t1.tlab.top = t1.tlab.top + sizeof(new_obj);        t4.tlab.end = top;
```

Figure 2: Allocating a new object with the TLAB allocation strategy in an unfragmented heap

The example shows that the heap is now split into more than 2 parts. There is still the used part at the bottom and an unused part at the end, but in between the heap is heavily fragmented. However, bump pointer allocations (and also some GC algorithms) rely on the fact that the heap remains unfragmented. To guarantee that, instead of just discarding a TLAB when a new object does not fit anymore, a TLAB is retired. Retiring a TLAB involves filling the rest of the TLAB with a filler object, i.e., an `int[]`. This filler object is not referenced by anyone, and contains no references itself. Thus, the object can be deallocated at the next GC, but it ensures a continuous stream of objects in the heap. Also, when a GC starts, all application threads are forced to retire their TLABs in preparation for the collection.

2.2 Garbage Collection

Garbage collection has become very popular in todays languages because it relieves programmers from the error-prone task of freeing memory manually. A garbage collector reclaims all objects that are unreachable from the root pointers (e.g., static fields, local variables, and so on). It is automatically triggered when a thread fails at allocating a new object and is often considered the primary benefit of a managed language. However, building a garbage collector that is able to collect large heaps efficiently, with only small, ideally predictable, pause times, and that is also able to quickly adapt to new situations and new memory behavior is a difficult task. This section aims at providing an overview about garbage collection techniques in general and their respective properties. It also describes some state-of-the-art collectors in Java.

2.2.1 Stop-the-world Collectors with Safepoints

Many collectors are stop-the-world collectors, meaning that when a collection is triggered (usually by a thread being unable to allocate a new object), all application threads are suspended. This enables the GC to safely collect the heap without the application threads interfering by trying to allocate new objects or by mutating pointer fields.

However, suspending all application threads in a state that enables to GC to safely inspect their method stacks as well as the value stack and the register values (all might contain root pointers) is a difficult task. All registers and all values on the method stacks as well as on the value stack may contain primitive values, pointers, or even pointers to fields within objects (or to elements within an array). The GC must safely determine whether a value is a pointer to an object start (after all, it could just be a numeric value that randomly matches the address of an object).

To overcome the problem of decoding thread-local stacks and registers correctly, VMs use *safepoints*. Safepoints are special instructions inserted into the code, forcing the thread to check whether it must stop. As a collection may only be executed when all threads have reached such a safepoint, the JIT compiler remembers all necessary information for those positions in the code to be able to correctly decode pointers from the current thread state (e.g., what registers and stack positions contain pointers). To be able to decode every stack frame, every call must be a safepoint. Furthermore, every operation that allocates a new object, and thus may trigger a collection if the heap is full, must also be a safepoint. Return instructions as well as loop back edges are also often used as safepoints.

As safepoint checks, i.e., checking whether the current thread must be stopped, are expensive, safepoints are removed and hoisted out of loops whenever possible. Therefore, a hot loop containing no operations that force a mandatory safepoint might not contain any safepoints. If a collection must be executed, the GC must wait until every thread has reached its next safepoint, even if a thread is executing a long running loop without any safepoints.

2.2.2 Generational Collectors and Card Tables

Lieberman and Hewitt [33] first noticed that GCs can exploit the fact that applications allocate many short-living objects and few long-living objects and thus introduced generational garbage collection. Generational collectors split the heap into several (often two or three) generations with ascending ids, which can be collected independently from each other. New objects are always allocated in the generation with the smallest id. If an object survives a predefined (possibly adaptive) number of garbage collections, it is promoted to the next generation, until it finally reaches the last generation. Splitting objects into generations has the advantage that these generations can be collected independently and with different collection algorithms. For example, on the one hand, assuming that most objects die young, many collectors use a very run-time-efficient collection algorithm in the younger generations at the cost of wasting a lot of space. On the other hand, older generations can be collected with a space-efficient algorithm, assuming it is unlikely for an object to die when it has reached an older generation. This produces many run-time-efficient collections in the young generation, and rare run-time-inefficient collections in the old generation.

Many algorithms split the heap into only two generations, i.e., the young generation, and the old generation. However, as we want to collect them independently from each other, e.g., just the young generation, we need to consider all pointers from one generation to another as root pointers. The positions of cross-generational pointers are remembered in the generation's remembered set.

To keep the remembered set up-to-date, the GC must track all pointer updates. As tracking every pointer assignment and checking whether the new pointer value references an object of another generation would be far too costly, the GC employs a card table (cf. Wilson and Mohr [49]) to track pointer modifications. A card is a fix-sized heap region and usually contains several objects. The card table is a mark list for all cards of the heap. Whenever an application thread mutates a pointer field, it marks the appropriate card dirty. When a collection starts, the GC only needs to search objects in dirty cards for new cross-generational pointers.

2.2.3 Stop and Copy Collector

The *stop and copy* collection algorithm achieves superior run-time performance at the cost of memory performance. This technique splits the heap into two spaces, a *from* and a *to* space. New objects are allocated into the *from* space and the *to* space is always empty. A garbage collection is triggered when no new object can be allocated into the *from* space. The GC stops all application threads and starts walking through the object graph, starting with the root pointers. Every object the GC finds in the *from* space is immediately copied into the *to* space and a forwarding pointer to the new object is installed in the header of the old object. Subsequently, the GC repeats the copying process with all children (i.e., pointer fields) of the copied objects. While copying the children of an object, all pointer fields in the parent are adjusted accordingly. If the GC arrives at an object that has already been copied, the forward pointer provides the address to the new object and enables the GC to update the pointer field correctly. As soon as all objects have been copied (and the *from* space therefore contains only garbage), the spaces are swapped, i.e., the previous *from* space becomes the new *to* space and the previous *to* space becomes the new *from* space.

As this algorithm only deals with live objects (unreachable objects are never visited), its run-time complexity depends on the amount of live memory only. However, one space is always empty, effectively using only half of the available memory.

2.2.4 Mark and Compact Collector

The *mark and compact* collection algorithm focuses on memory performance (as it does not waste memory compared to the *stop and copy* algorithm) at the cost of run-time performance. This algorithm is divided into three phases, i.e., the *mark* phase, the *summarize* phase, and the *compact* phase. In the *mark* phase, the GC traverses the object graph (starting with the root pointers) and marks every object it reaches as alive by setting the mark bit in its header. Thus, after the entire object graph has been traversed, all reachable objects are marked. The *summarize* phase walks through the heap (from left to right, including unmarked objects) and calculates the new position for every object (dead objects will be overwritten). Finally, the *compact phase* copies all objects to their new positions and adjusts all pointers accordingly. Thus, after a collection, the heap is unfragmented as all objects have been copied towards the beginning of the heap, separating the heap into a used region (full with objects) and a completely unused region.

This algorithm focuses on memory performance, as it compacts all objects as efficiently as possible, making maximum use of the available memory in the process. However, in terms of run-time complexity, the *mark and compact* algorithm depends on the size of the heap. Consequently, even if only a few objects are alive, the GC still needs to walk the entire heap in the *summarize* phase.

The *mark and compact* algorithm is based on the *mark and sweep* algorithm, which replaces the *summarize* and the *compact* phase with a single *sweep* phase. In this phase, the GC walks the heap and inserts all free memory blocks into a free list. As the *mark and sweep* algorithm does not compact the heap after marking, it produces a fragmented heap.

2.2.5 Oracle's ParallelOld Collector

Oracle's *ParallelOld Collector* is the current default GC in Java 8 as well as in previous versions. As proposed by Ungar [46], it employs the *parallel generational scavenging* technique. This technique splits the heap into two generations, i.e., a young generation and an old generation. The young generation is split again into the *Eden* space, the *survivor from* and the *survivor to* space. New objects are allocated into the *Eden* space. A garbage collection is triggered when the *Eden* space is full.

The *ParallelOld Collector* can either perform a *minor* or a *major* collection. A *minor* collection will only collect the young generation, whereas a *major* collection collects the entire heap. The *minor* collection uses the *stop and copy* approach to evacuate the *Eden* space and the *survivor from* space into the *survivor to* space. If an object has reached a certain age (i.e., it has been copied for a certain number of times), it is promoted into the old generation instead. The *major* collection uses the *mark and copy* approach to collect the entire heap (i.e., the young generation as well as the old generation). Both collection types are stop-the-world collections, i.e., the application is fully suspended while the heap is collected. Figure 3 shows the heap layout of the *ParallelOld Collector*.

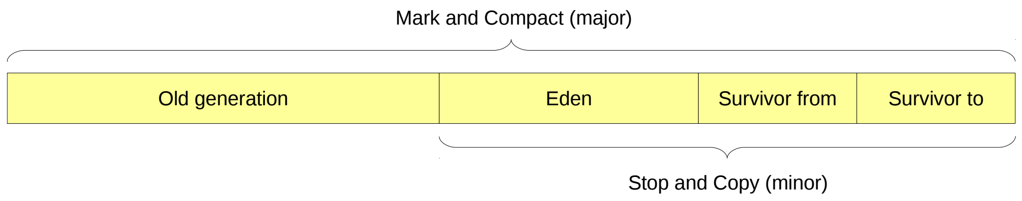


Figure 3: Heap layout and collection algorithms of Oracle's *ParallelOld Collector*

As the *minor* GC evacuates two spaces into the *survivor to* space instead of just one (like the plain algorithm would), a *minor* collection may fail if not all objects fit into the *survivor to* space. In this case, the collection is aborted, and a *major* collection is triggered. However, as the collection cannot be rolled back, some objects have already been copied, while others have not. To overcome this problem, a failed *minor* collection is immediately followed by a *major* collection. Although the first collection has been aborted, and there might be two copies of some objects, the GC still guarantees that all pointers either point to the new or to the old version, making sure that only one of the copies is collected.

As the *major* GC compacts towards the beginning of the heap and disregards all space boundaries, all objects are automatically promoted to the old generation. However, if the heap is almost full, not all objects might fit into the old generation, leaving some in the *Eden* space, maybe even in the *survivor* spaces. Therefore, it is possible that, in extreme cases, objects are moved back into the *Eden* space.

Both the *minor* and the *major* GC algorithms are implemented to make use of multi-core processors by using multiple threads to collect the heap. The *minor* collection algorithm walks the object graph in parallel. If two threads arrive at the same object at the same time, both will try to copy it. However, only one thread will succeed in atomically installing the forwarding pointer into the old object's header. The other thread will then undo its copy operation. The *major* collection algorithm also marks and compacts in parallel. The *marking* phase can easily be executed in parallel because marking an object twice does not do any harm. The *compact* phase splits the heap into several regions, which are processes by multiple GC threads in parallel. Please note that those regions do not necessarily start and end at object boundaries. Thus different parts of an object can be copied by different GC threads in any order. Only the *summarize* phase is not executed in parallel.

To summarize, the *ParallelOld Collector* uses the run-time efficient *stop and copy* approach for the young generation (assuming most objects die young) and the more memory-efficient *mark and compact* algorithm when the entire heap must be collected. Also, its heap layout enables the collector to resize spaces if necessary.

2.2.6 Oracle's Garbage First Collector

Oracle's *Garbage First (G1) Collector*, as proposed by Detlefs et al. [14, 15], is a garbage collector designed to handle big heaps with relatively low pause times. Similarly to the *ParallelOld Collector*, it also divides the heap into a young and an old generation, whereupon the old generation is split again into an *Eden* space, and the *Survivor* space. Also, this collector uses the *stop and copy* approach for *minor* collections and the *mark and compact* algorithm for *major* collections. However, in contrast to the *ParallelOld Collector*, the individual heap spaces are not continuous. Rather, the *G1 Collector* splits the heap into many fixed-size regions, whereupon every region is logically assigned to a space. Thus, the *minor* GC can select any subset of the regions to collect. Figure 4 shows the heap layout of the *G1 Collector*.

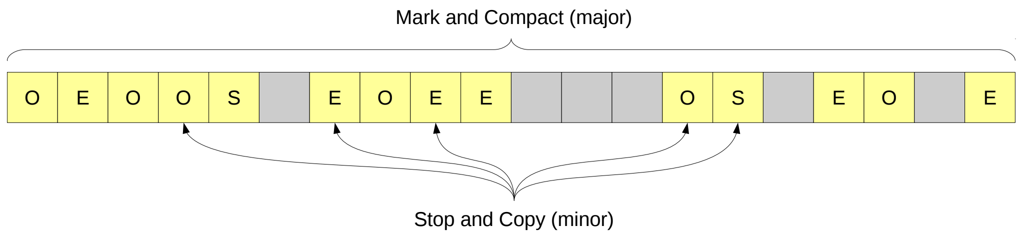


Figure 4: Heap layout and collection algorithms of Oracle's *G1 Collector* (O: old generation, S: survivor space, E: Eden space)

A *minor* collection starts by suspending all application threads and collecting all root pointer values. However, it then resumes the application threads and marks live objects concurrently. It uses the *snapshot at the beginning* (SATB) approach to mark objects that are mutated while marking. When all reachable objects have been marked, the application threads are suspended again for evacuation.

A *minor* collection can select any combination of regions to collect. However, every region can only be evacuated to a region of a specific type, i.e., all *Eden* space regions must be evacuated to a *survivor* space region, all *Survivor* space regions to either other *survivor* space regions or to an old generation region, and the old generation regions only to other old generation regions. In fact, whenever a region is evacuated, one of the empty regions is selected and assigned the proper space. Afterwards, the evacuated region is not only empty, but completely cleared, having no assigned space. Which regions are evacuated in a *minor* collection are determined by the amount of garbage that they contain. Even during a minor GC, the entire heap must be marked. Then, the GC can compute what spaces contain the most garbage and can collect those, freeing as much memory as possible with the least effort.

When the regions have been selected, the GC evacuates all objects to other regions of the appropriate space. Finally, all application threads are resumed.

Only if a *minor* GC has failed (e.g., because the heap is so full that there are not enough regions available as target for evacuation), a *major* GC will be performed. This collection will suspend all application threads, *mark and compact* the entire heap and resume again.

Nevertheless, a *major* GC is rather rare and will only occur when the application runs out of memory. This is a significant advantage compared to other GCs (like the *ParallelOld GC* that will eventually run into *major* collections).

2.2.7 Oracle's Concurrent Mark and Sweep Collector

Oracle's *Concurrent Mark and Sweep Collector* is very similar to the *ParallelOld Collector*. It uses the same generational layout, i.e., an *Eden* space, two *survivor* spaces, and an old generation. Also, the *minor* collection fpr the young generation uses exactly the same algorithm as the *ParallelOld Collector*, i.e., it suspends all applications threads and evacuates all objects from the *Eden* space and the *survivor from* space to the *survivor to* space or the old generation.

However, the Concurrent Mark and Sweep GC also supports a *minor* collection that collects only the old generation and uses the *mark and sweep* algorithm instead of the *mark and compact* algorithm. Also, this collection is executed concurrently, i.e., the application threads continue to run for most of the collection time. They are suspended shortly the start of a collection, and at the end to insert all unmarked areas into the free list. This concurrent *major* collection enables the collector to work on large heaps but still keep pause times low.

A *minor* collection in the old generation is triggered based on different heuristics and the fill levels of the individual spaces. However, when a *minor* collection in the old generation is currently in progress and mutator threads fill up the heap concurrently, a *minor* collection in the young generation is triggered. In this case, the collection in the old generation is suspended and continued after the collection in the young generation has finished.

2.2.8 Red Hat's Shenandoah Collector

Red Hat's *Shenandoah Collector* is a relatively new collector, presented in Flood et al. [16], that uses a *mark and copy* technique. It divides the heap into regions, similar to the *G1 Collector*, and copies all live objects of collected regions into other regions. However, in contrast to the very similar *stop and copy* algorithm, this technique does not stop while copying, but rather copies objects concurrently. After a concurrent marking phase, the collector copies all objects concurrently. Small pauses are only introduced at the beginning and the end of the marking phase. There is no pause when copying.

In order to avoid the concurrently running mutator threads to modify different versions of the same object, mutator threads must not write to objects that are scheduled for being copied but have not been copied so far. If a thread wants to modify such an object, the object has to be copied before. The collector uses write barriers to enforce this behavior on all mutator threads.

2.2.9 Azul's C4 Collector

Azul's *C4 (Continuously Concurrent Compacting Collector) Collector*, presented by Tene et al. [45], is a fully concurrent collector, introducing no pauses at all. This collector first marks all reachable objects. In the second phase, it evacuates objects from all regions that have been selected for collection. The third and final phase corrects all references to the new locations.

To keep application threads from mutating old objects or even mutating objects while they are copied, the memory pages from which objects are moved are protected. This mechanism is called a *Load Value Barrier*. Accessing a value on such a page forces the application thread to execute a signal handler, which takes care of copying the object and correcting the reference to point to the correct location.

Compared to the other collectors, Azul's *C4 Collector* is the only production GC that is fully pauseless.

2.3 Memory Monitoring Techniques

State-of-the-art memory monitoring tools can be classified into four categories based on the techniques used to record information. These techniques differ significantly in terms of overhead and how they distort the memory behavior of the monitored application. This section describes these techniques in detail and concludes with a comparison of functionality.

2.3.1 API-based Monitoring

API-based monitoring uses a given API from the underlying system (i.e., from the virtual machine) to access the desired information. Although the Java VM provides the Java Virtual Machine Tool Interface

(JVMTI)[1] for profilers and debuggers, in terms of memory monitoring no information is provided out of the box. JVMTI just provides an API to iterate through the heap and to collect data manually. For individual objects, JVMTI allows accessing field values, the object's type, and the object's size. The JVM does not provide any physical addresses for security reasons (if one had access to the physical address of an object, one might tamper with it in an illegal way). To summarize, JVMTI provides a API to implement other memory monitoring techniques, but hardly provides anything out of the box.

2.3.2 Sampling-based Monitoring

Sampling-based monitoring queries the state of the monitored application periodically and extracts information based on individual samples or their respective change over time. In the case of memory monitoring, a sample can be as simple as a single value describing the current heap consumption or as complex as a complete heap dump containing all objects including their respective contents. Taking a heap snapshot is an expensive operation as it involves iterating through the entire heap object by object and recording all the required information. This information can then be written to disk for later analysis or processed immediately.

The outputs of sampling-based memory monitoring tools are usually memory consumption metrics over time or heap states for a specific point in time. A heap state contains a list of all objects, usually aggregated by type or a similar characteristic. Unfortunately, the sampling-based memory technique cannot recreate information about object allocations, e.g., the allocating thread or the allocation site, as this information is not stored within the objects and is therefore lost after allocation. This information cannot be reproduced in any way. Like in the API-based approach, physical addresses are not accessible.

Also, based on heap samples only, no object lifecycle can be computed. If two subsequent samples both contain an object of a specific type, one cannot tell whether (1) it is the same object, whether (2) the first object has been replaced with a new one, or (3) whether any number of objects of that type have been allocated and deallocated in the meantime. Consequently, it is impossible to determine the lifetime of an object or the rate in which objects with specific properties are allocated and deallocated.

In theory, this information can be approximated by forcing a complete collection after every memory-write (i.e., after pointer assignments and allocations). There is some academic work employing this strategy (cf. Section 2.4). However, as this imposes a enormous overhead (as shown by that work), we do not consider this strategy feasible for production environments.

2.3.3 Instrumentation-based Monitoring

Instrumentation-based monitoring intercepts class loading, analyzes the code that is about the be loaded, and inserts small code snippets to record information. In terms of memory behavior, monitoring tools search for any instruction allocating objects, and insert code to record the allocation. As the allocation itself is observed, this approach can record the allocation site, the allocating thread and other context information about the allocation.

However, one has to be very careful when instrumenting all allocating instructions because the added code might impede escape analysis. Escape analysis and scalar replacement are optimizations that try to determine whether a newly allocated object escapes the compilation unit (i.e., the method) by returning its reference to the caller or to another method, or by assigning its reference to a field. If the compiler can guarantee that an object does not escape, it puts the fields of that object onto the method stack as

[1]JVMTI: https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html

local variables (i.e., stack allocation, scalar replacement) and replaces all field accesses with accesses to the newly introduced local variables. Adding instrumentation code that uses the allocated object often makes it impossible to determine whether the object escapes and, thus, forces it to be allocated onto the heap.

A pure instrumentation-based approach is unable to record any deallocations because there are no specific deallocation instructions that can be instrumented. There are three virtual-machine features that current instrumentation-based memory tools employ to record deallocations: (1) finalizers, (2) weak references, and (3) object tags. Like in the sampling-based approach, one can use manually forced GCs to approximate the earliest time an object becomes unreachable.

Finalizers are methods that are called by the GC before an object is deallocated. However, as many GCs suspend all application threads to prohibit any interfering pointer mutation , a finalizer cannot be called right away. Instead the GC puts all collectible objects into a queue and calls the finalizers for them when the application continues. These objects are then collected in the next GC run (if no new reference to the object has been created). This means that finalizers have a significant impact on the GC performance, especially if all objects use them, because the GC needs at least two runs to collect an object. Also, as these objects have to be handled in a special way and are considered a rarely used feature by VM implementers, allocating such objects is not optimized. In terms of recording deallocations, weak references work similarly to finalizers.

Weak references are wrapped by `WeakReference` objects and are handled differently by the GC. The GC considers an object to be dead if it is only references by weak references. If this is the case, a language-level event is fired, indicating that the reference will soon be cleared by the GC. Java supports a few variants of weak references, most importantly `PhantomReference`s that are used by memory monitoring tools. These references work exactly like weak references, except that the event that is fired does not provide access to the object. This enables the GC to clear the reference immediately and to deallocate the object instead of keeping it alive until the event handling has been finished.

Finally, object tags are marked objects for which a GC will generate special events (natively via JVMTI). However, the VM assumes that these tags will be used sparsely only for some objects and is, thus, very inefficient when used on all objects.

To summarize, the instrumentation-based approach can easily record data about any object allocation. However, recording deallocations is difficult and will involve significant performance degradations.

2.3.4 Event-based Monitoring

Event-based monitoring uses events generated by the underlying virtual machine to record memory behavior. However, as one of the other approaches must be used to generate these events, we do not consider this technique any further in terms of performance.

2.3.5 Comparison

In Figure 5 we compare API-based, sampling-based, and instrumentation-based monitoring. We look at information about object allocations and the objects' properties (i.e., type, size, allocation mode (e.g., interpreted, compiled)), information about object deaths (i.e., the time they become unreachable as well as the time they are finally collected), the object lifecycle (i.e., allocation and death) and its movements through the heap, as well as the heap layout (i.e., object positions in the heap) and the heap content (i.e., field values).

The API-based monitoring approach provides the least amount of information. Using JVMTI, an agent can query the size and the type of an object but no information about the allocation is provided. Some object deaths can be observed by tagging objects, but this is not feasible for many objects.

Information	API (Java)	Sampling	Instrumentation		
				+ Phantom refs.	+ Forced GCs
Object allocations	None	Some	All	All	All
– Size	All	All	All	All	All
– Type	All	All	All	All	All
– Pointers	None	All	None	None	All
– Address	None	None	None	None	None
– Allocation site	None	None	All	All	All
– Allocating thread	None	None	All	All	All
– Allocating mode	None	None	None	None	None
Object deaths	Some	Some	None	All	All
– Time unreachable	None	None	None	None	All
– Time collected	Some	Some	None	All	None
Object lifecycle	Some	Some	None	None	None
Object movements	None	None	None	None	None
Heap layout	None	None	None	None	None
Heap content	None	Some	All	All	All

Figure 5: Comparison of functionality for API-based, sampling-based, and instrumentation-based memory monitoring approaches (None (red): no information can be obtained, Some: (orange) the information can be obtained for some, but not for all objects, All (green): the information can be obtained for all objects without exception

The sampling-based approach provides more information about the field values. Also, some allocations and deaths can be detected if subsequent samples show an increase or decrease in the amount of objects with fixed properties. However, no information can be inferred about the heap layout or when an object becomes unreachable.

The instrumentation-based approach provides the most information. It is the only technique that can provide the allocation site and the allocating thread. If we also use phantom references or forced GCs, we can also detect when an object is collected or becomes unreachable respectively.

Finally, we can see that none of the above techniques can provide the heap layout, the physical addresses of objects, or the complete lifecycle of all objects, including their movement through the heap.

2.4 Monitoring Tools

We have examined the capabilities and memory monitoring overheads of several profiling tools. This section will discuss every one of those tools and conclude with a comparison.

Java Hotspot^{TM} VM. The Java Hotspot^{TM} VM already comes with some built-in profiling mechanisms. As every VM supports the much richer Java Virtual Machine Tool Interface (JVMTI) that external tools can access, these mechanisms are only designed to give a quick hint to obvious problems.

Using the `-Xprof` flag, the VM will output some coarse per-thread profile of frequently called (i.e., hot) methods. These statistics are gathered by sampling, i.e., by periodically suspending the VM and examining all call stacks. An example for the output is shown in Figure 6. The profile provides a detailed listing of hot methods, as well as the portion that the thread was blocked (i.e., waiting for a lock or an IO operation), and the type of code the thread was in (i.e., in compiled code, in native code, or in runtime stub code). An adept programmer might get some insight about the allocation behavior based on the hot methods, but realistically, this flag is useless in terms of memory profiling.

```
Flat profile of 3,18 secs (302 total ticks): Thread-158

     Compiled + native   Method
  2,1%     0 +      6    org.apache.xalan.templates.TemplateList.getHead
  1,0%     0 +      3    org.apache.xalan.templates.TemplateList.getTemplateFast
  1,0%     1 +      2    org.....ElemApplyTemplates.transformSelectedNodes
  0,7%     0 +      2    sun.misc.URLClassPath.getLoader
  0,3%     0 +      1    org.apache.xpath.axes.LocPathIterator.executeCharsToContentHandler
  0,3%     0 +      1    java.util.zip.ZipFile.getInputStream
  5,6%     1 +     15    Total compiled

         Stub + native   Method
 68,9%     0 +    197    java.io.FileOutputStream.writeBytes
 10,1%     0 +     29    java.io.FileInputStream.readBytes
  4,5%     0 +     13    java.lang.Throwable.fillInStackTrace
  4,2%     7 +      5    java.security.AccessController.doPrivileged
  2,4%     0 +      7    java.io.FileInputStream.read0
  1,0%     0 +      3    java.io.UnixFileSystem.getBooleanAttributes0
  1,0%     0 +      3    java.util.zip.Inflater.inflateBytes
  0,7%     0 +      2    java.lang.ClassLoader.findLoadedClass0
  0,3%     0 +      1    java.lang.String.intern
  0,3%     1 +      0    java.security.AccessController.doPrivileged
  0,3%     0 +      1    java.util.zip.ZipFile.getEntry
  0,3%     0 +      1    java.util.zip.ZipFile.read
 94,4%     8 +    262    Total stub

  Thread-local ticks:
  5,3%    16             Blocked (of total)
```

Figure 6: Output of the Hotspot^{TM} VM when setting the `-Xprof` flag

Much more suited for at least determining the existence of a memory-related performance problem is the `-verbose:gc` flag. Using this flag (and some similar flags for more details), the VM can produce output shown in Figure 7. This log contains a single line for every garbage collection that is performed, including the reason for the collection, the changes (in bytes) of all generations that are collected, and the time it took. By examining the GC times alone, e.g., if they are constantly growing, one can easily determine if there is a problem. Also, we get some limited insight into the nature of the problem, e.g., a constantly full young generation indicates there are a lot of temporary objects, or a constantly full old generation indicates that a big data structure cannot be freed. Although this log might give the user some hint, the actual cause of the performance problem remains unknown.

The parameter `-XX:+PrintClassHistogramBeforeFullGC`, the output of which is shown in Figure 8, prints instance counters for every class at every GC. By examining several sequential outputs like this, the user can search for trends, e.g., when the number of instances of a class grows over time. However, the output shows only plain counts, there is no information about object identity. For example, in Figure 8, there are currently 28505 instances of the class `[B` (`byte[]`) in the heap. If the next GC reports the same number again, we do not know whether these are the same 28505 objects, of whether they have all been

```
[GC (Allocation Failure) [PSYoungGen: 2693792K->10608K(2693632K)]
    2700234K->17082K(3033600K), 0,0106794 secs] [Times: user=0,02 sys=0,00, real=0,01 secs]
[GC (System.gc()) [PSYoungGen: 1386378K->4320K(2699264K)] 1392852K->10801K(3039232K),
    0,0071578 secs] [Times: user=0,00 sys=0,01, real=0,01 secs]
[Full GC (System.gc()) [PSYoungGen: 4320K->0K(2699264K)] [ParOldGen: 6481K->6126K(339968K)]
    10801K->6126K(3039232K), [Metaspace: 9222K->9222K(1058816K)], 0,0171786 secs] [Times:
    user=0,07 sys=0,00, real=0,01 secs]
```

Figure 7: Output of the Hotspot^{TM} VM when setting the `-verbose:gc -XX:+PrintGCDetails` flags

freed and allocated again. Even if the next output reports only 10 objects more, we do not know whether just 10 objects have been allocated, or all were deallocated and 28505 + 10 were allocated anew.

```
 num     #instances         #bytes  class name
----------------------------------------------
   1:         28505        6024712  [B
   2:         23829         953160  java.lang.ref.Finalizer
   3:         12908         863328  [C
   4:          9485         531160  java.util.zip.ZipFile$ZipFileInflaterInputStream
   5:          9485         531160  java.util.zip.ZipFile$ZipFileInputStream
   6:          1048         307080  [I
   7:         11997         287928  java.lang.String
   8:          7255         174120  java.util.LinkedList$Node

 518:             1             16  sun.util.locale.provider.SPILocaleProviderAdapter
 519:             1             16  sun.util.resources.LocaleData
 520:             1             16  sun.util.resources.LocaleData$ResourceBundleControl
Total        138586       11224568
```

Figure 8: Output of the Hotspot^{TM} VM when setting the `-XX:+PrintClassHistogramBeforeFullGC` flag

To summarize, the Hotspot^{TM} VM already provides some crude profiling mechanisms, some of which can even be used to detect memory-related performance problems. However, to properly analyze the data and to track down the root cause of the problem, we need better tooling support. Analyzing the output of the VM directly is a boring and tedious task and, after all, we have computers for that.

JConsole. *JConsole*[2] is a simple tool by Oracle to read data from exported *Java Management Beans*. However, this tool is limited to the information exposed by these beans, which is in terms of memory just the current consumption per memory pool (i.e., Eden, survivor, old). This data provides far too little information to determine whether a problem exists, much less the root cause of the problem,

jhat. The *Java Heap Analysis Tool* (jhat)[3] is a dump-based tool that can only display simple accumulated statistics such as instance counts per class.

[2]JConsole: http://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html
[3]jhat: http://docs.oracle.com/javase/6/docs/technotes/tools/share/jhat.html

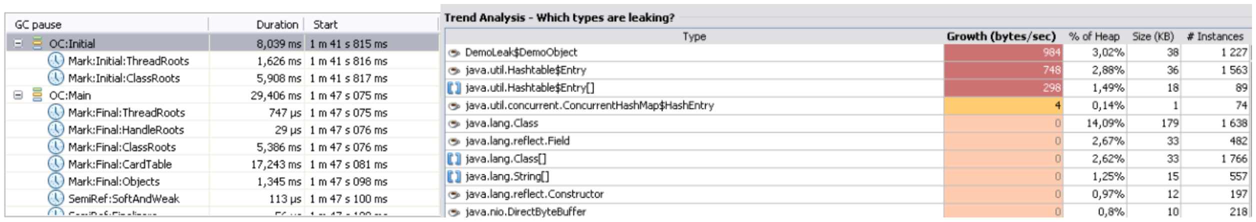
JRockit Mission Control. The *JRockit Mission Control* is a profiling tool published by Oracle. The VM and the according tools are designed to *manage, profile, and eliminate memory leaks in your Java application without introducing the performance overhead normally associated with tools of this type.*[4]. The JRockit Mission Control can detect hot methods as well as latency problems (involving network sockets and other IO operations). In terms of memory analysis, this tool can do a generic memory analysis and it can also perform a memory leak analysis.

The generic memory analysis can show several aspects of the current memory usage:

- GC times, GC pause time ratio, and global heap usage over time.
- GC phase times in detail, indicating what kind of task took the longest, shown in Figure 9 (left hand side).
- Allocation rate per thread (also some statistics like biggest object or average allocated object size).
- Heap contents, i.e., the number of instances and the aggregated size of all instances per class.

However, the interpretation of this data is left to the user. Thus, this functionality is only little more than a human-readable visualization of what the Hotspot^{TM} VM can log.

The memory leak detection, however, is more sophisticated. This analysis starts by detecting growing instance numbers per class, shown in Figure 9 (right hand side). That behavior could also be detected by manually examining consecutive Hotspot^{TM} VM histogram logs. However, in JRockit, the user can then drill down and look at the connection of individual objects and follow the pointers to the root cause. The interpretation of individual types is left to the user, i.e., the user must know about the semantics of all individual types in order to determine the root cause of memory leak.



GC pause	Duration	Start
OC:Initial	8,039 ms	1 m 41 s 815 ms
Mark:Initial:ThreadRoots	1,626 ms	1 m 41 s 816 ms
Mark:Initial:ClassRoots	5,908 ms	1 m 41 s 817 ms
OC:Main	29,406 ms	1 m 47 s 075 ms
Mark:Final:ThreadRoots	747 µs	1 m 47 s 075 ms
Mark:Final:HandleRoots	29 µs	1 m 47 s 076 ms
Mark:Final:ClassRoots	5,386 ms	1 m 47 s 076 ms
Mark:Final:CardTable	17,243 ms	1 m 47 s 081 ms
Mark:Final:Objects	1,345 ms	1 m 47 s 098 ms
SemiRef:SoftAndWeak	113 µs	1 m 47 s 100 ms

Trend Analysis - Which types are leaking?

Type	Growth (bytes/sec)	% of Heap	Size (KB)	# Instances
DemoLeak$DemoObject	984	3,02%	38	1 227
java.util.Hashtable$Entry	748	2,88%	36	1 563
java.util.Hashtable$Entry[]	298	1,49%	18	89
java.util.concurrent.ConcurrentHashMap$HashEntry	4	0,14%	1	74
java.lang.Class	0	14,09%	179	1 638
java.lang.reflect.Field	0	2,67%	33	482
java.lang.Class[]	0	2,62%	33	1 766
java.lang.String[]	0	1,25%	15	557
java.lang.reflect.Constructor	0	0,97%	12	197
java.nio.DirectByteBuffer	0	0,8%	10	218

Figure 9: GC phase times and LeakDetector of JRockit (Source: http://www.oracle.com/technetwork/middleware/jrockit/overview/missioncontrol-whitepaper-june08-1-130357.pdf)

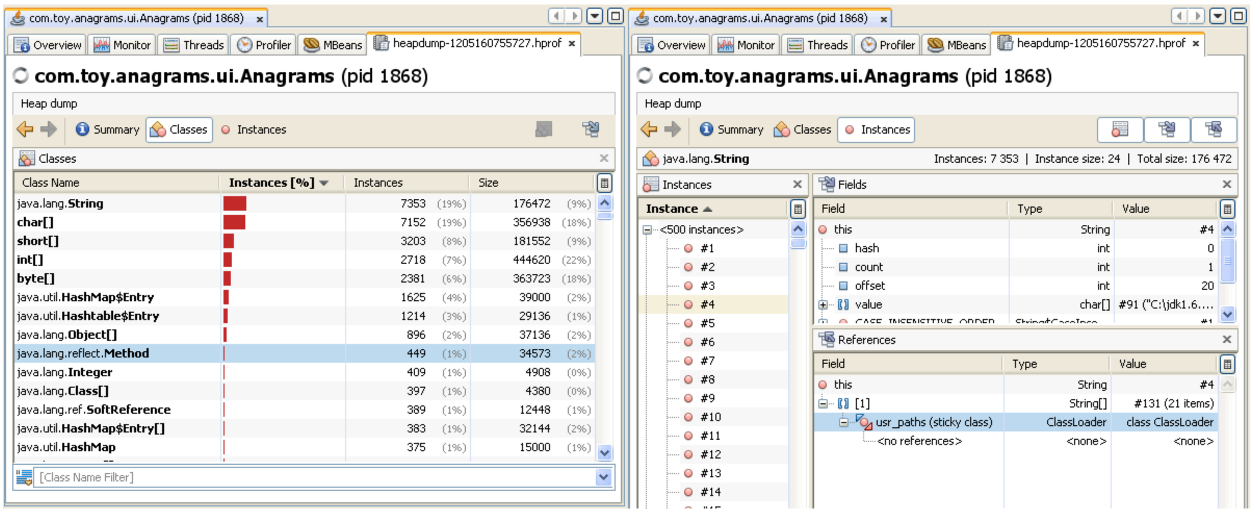
Another difference is that JRockit piggybacks the memory leak analysis onto the mark phase of the GC. Oracle thus claims only minimal overhead while examining the trends of objects per class.

Also, the memory leak detector can trace allocations, i.e., what methods allocate the most. However, they are not able to drill down to individual allocation sites within a method, and the allocation traces can only be enabled for one type at a time. Hirt and Lagergren [5] even state that *enabling allocation traces for types with a high allocation pressure can introduce significant overhead. For example, it is, in general, a very bad idea to enable allocation traces for java.lang.Strings.*.

[4] JRockit Mission Control: http://www.oracle.com/technetwork/middleware/jrockit/mission-control/index.html

[5] *Oracle JRockit* by Hirt and Lagergreen, ISBN 978-1-847198-06-8, p399

Visual VM. *Visual VM* is a popular and easy-to-use *visual tool integrating several commandline JDK tools and lightweight profiling capabilities*[6]. Besides hot method-based profiling, it also supports crude memory profiling via memory dumps. A user can attach the tool to any running JVM instance and request a memory dump. This request suspends the VM while taking and transferring a complete memory dump, including all objects and their field values, to the tool for analysis. The tool does not run any automatic analyses, but the user can easily browse both accumulated heap statistics, as well as individual objects (shown in Figure 10). Depending on the heap size, this can introduce a significant overhead to the monitored application.


com.toy.anagrams.ui.Anagrams (pid 1868)
Overview
Monitor
Threads
Profiler
MBeans
heapdump-1205160755727.hprof
com.toy.anagrams.ui.Anagrams (pid 1868)
Heap dump
Summary
Classes
Instances
Classes
[Class Name Filter]

Class Name	Instances [%]	Instances	Size
java.lang.String		7353 (19%)	176472 (9%)
char[]		7152 (19%)	356938 (18%)
short[]		3203 (8%)	181552 (9%)
int[]		2718 (7%)	444620 (22%)
byte[]		2381 (6%)	363723 (18%)
java.util.HashMap$Entry		1625 (4%)	39000 (2%)
java.util.Hashtable$Entry		1214 (3%)	29136 (1%)
java.lang.Object[]		896 (2%)	37136 (2%)
java.lang.reflect.Method		449 (1%)	34573 (2%)
java.lang.Integer		409 (1%)	4908 (0%)
java.lang.Class[]		397 (1%)	4380 (0%)
java.lang.ref.SoftReference		389 (1%)	12448 (1%)
java.util.HashMap$Entry[]		383 (1%)	32144 (2%)
java.util.HashMap		375 (1%)	15000 (1%)

java.lang.String Instances: 7 353 | Instance size: 24 | Total size: 176 472

Instances

Instance
<500 instances>
#1
#2
#3
#4
#5
#6
#7
#8
#9
#10
#11
#12
#13
#14

Fields

Field	Type	Value
this	String	#4
hash	int	0
count	int	1
offset	int	20
value	char[]	#91 ("C:\jdk1.6....

References

Field	Type	Value
this	String	#4
[1]	String[]	#131 (21 items)
usr_paths (sticky class)	ClassLoader	class ClassLoader
<no references>	<none>	<none>

Figure 10: Analyzing heap dumps with Visual VM (Source: http://visualvm.java.net/heapdump.html)

Elephant Tracks. *Elephant Tracks*, by Ricci et al. [42, 43], can produce an event trace for arbitrary Java applications, containing method entry and method exit events, object allocation and object death events, and pointer update events[7]. They instrument classes at loaded-time to track object allocations and to periodically timestamp objects when used.

According to Ricci et al. [43], *Elephant Tracks* introduces an overhead of 24,580% (geometric mean, max 324,580%). These numbers also include method entries and exists, which make up a majority of events according to their analysis. Based on the overhead reports and the event distribution analysis, we can calculate a rough estimate about the overhead when tracking object allocations and deaths only. We do this under the assumption that every event produces roughly the same overhead, and break down the total overhead using the event distributions reported. The results in Figure 11 show a enormous overhead of up to 7237.86%. The DaCapo benchmarks tradebeans and tradesoap have been excluded because they fail due to internal timeouts of the underlying communication layer. Furthermore, the authors admit that they cannot monitor any application running on Java 7 or higher, and even on Java 6 only IBM J9 is supported.

[6]Visual VM: http://visualvm.java.net
[7]Elephant Tracks: http://www.cs.tufts.edu/research/redline/elephantTracks/

Benchmark	Overhead	Memory events	Overhead estimation
avrora	43610.00%	0.24%	104.66%
batik	8410.00%	1.77%	148.86%
eclipse	160360.00%	3.23%	5179.63%
fop	13020.00%	4.63%	602.83%
h2	358310.00%	2.02%	7237.86%
jython	77490.00%	2.79%	2161.97%
luindex	7160.00%	0.28%	20.05%
lusearch	32790.00%	2.72%	891.89%
pmd	23020.00%	4.27%	982.95%
sunflow	158330.00%	3%	4749.90%
tomcat	17540.00%	6.33%	1110.28%
xalan	71540.00%	1.52%	1087.41%

Figure 11: ElephantTracks overhead, including a rough estimate of using memory events only based on data provided by Ricci et al. [43]

Dynatrace. *Dynatrace*[8] is a state-of-the-art monitoring tool that excels at tracking transactions from the start (e.g., a click in a web browser) over several tiers using different technologies (e.g., C#, Java, native) down into a database. This tool is the leader in application performance management according to *Gartner Magic Quadrant for Application Performance Monitoring*[9]. Nevertheless, the capabilities of Dynatrace in terms of memory monitoring are comparable to other tools. Besides the common memory overview information, i.e., memory usage and garbage collection ratio over time, they also offer *trending memory snapshots* to detect growing instance numbers and a complete *memory dump* for manual browsing.

Trending memory snapshots are a sequence of automatically taken memory dumps that are compared to each other. Using these snapshots, the tool can detect changes in instance counts, i.e., a rise or drop of the instance count per class. These trending memory snapshots can be used to diagnose memory leaks by identifying rising object counts for specific classes in correlation with rising GC times. However, these snapshots have no concept of object identity. This means assuming a constant instance count for a specific class, *Dynatrace* cannot tell whether no new objects have been allocated, or whether all objects have been reclaimed by the GC and reallocated afterwards.

To actually determine the root cause of a memory leak, a complete memory dump can be taken. This dump contains not only instance counts per class, but also all field values. The user can browse through objects by following pointers down from the referrer to the referee, but also from the referee to all referrers. By examining the referers, the user can identify who keeps a potentially big data structure alive. However, due to the increased data in a complete memory dump, they produce inherently more overhead than a trending dump. To reduce the run-time overhead of taking the snapshot as well as the memory necessary to process it, the complete dump can be configured to omit `java.lang.String`s and all primitive values.

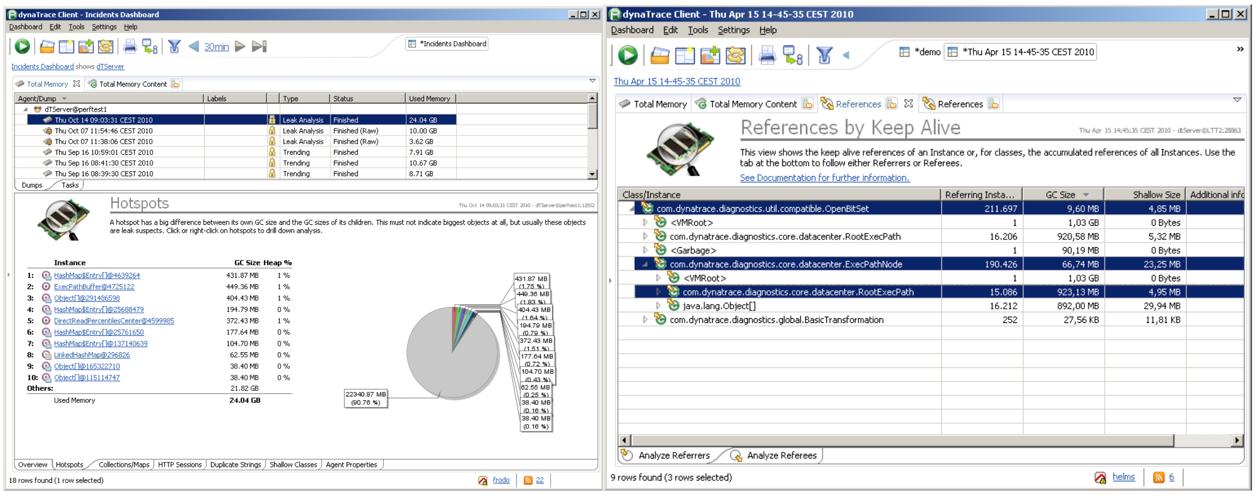
A detailed description of Dynatrace's capabilities in terms of memory monitoring can be found at one of their community pages[10]. Figure 12 shows an accumulated heap snapshot as well as how-to browse individual objects in the heap.

There are no official records about the performance overhead when taking trending snapshots or when

[8]Dynatrace: https://www.dynatrace.com/platform/offerings/application-monitoring/

[9]Dynatrace in the Gartner Magic Quadrant: https://www.dynatrace.com/en/gartner-magic-quadrant-application-performance-monitoring-2015.html

[10]Dynatrace Memory Monitoring: https://community.dynatrace.com/community/display/DOCDT63/Memory+Diagnostics


dynaTrace Client - Incidents Dashboard
Dashboard Edit Tools Settings Help
30min
*Incidents Dashboard
Incidents Dashboard shows dTServer
Total Memory
Total Memory Content

Agent/Dump	Labels	Type	Status	Used Memory
dTServer@perftest1				
Thu Oct 14 09:03:31 CEST 2010		Leak Analysis	Finished	24.04 GB
Thu Oct 07 11:54:46 CEST 2010		Leak Analysis	Finished (Raw)	10.00 GB
Thu Oct 07 11:38:06 CEST 2010		Leak Analysis	Finished (Raw)	3.62 GB
Thu Sep 16 10:59:01 CEST 2010		Trending	Finished	7.91 GB
Thu Sep 16 08:41:30 CEST 2010		Trending	Finished	10.67 GB
Thu Sep 16 08:39:30 CEST 2010		Trending	Finished	8.71 GB

Hotspots

Thu Oct 14 09:03:31 CEST 2010 - dTServer@perftest1:12052

A hotspot has a big difference between its own GC size and the GC sizes of its children. This must not indicate biggest objects at all, but usually these objects are leak suspects. Click or right-click on hotspots to drill down analysis.

	Instance	GC Size	Heap %
1:	HashMap$Entry[]@4639264	431.87 MB	1 %
2:	ExecPathBuffer@4725122	449.36 MB	1 %
3:	Object[]@291486598	404.43 MB	1 %
4:	HashMap$Entry[]@25688479	194.79 MB	0 %
5:	DirectReadPercentilesCenter@4599985	372.43 MB	1 %
6:	HashMap$Entry[]@25761650	177.64 MB	0 %
7:	HashMap$Entry[]@137140639	104.70 MB	0 %
8:	LinkedHashMap@296826	62.55 MB	0 %
9:	Object[]@165322710	38.40 MB	0 %
10:	Object[]@115114747	38.40 MB	0 %
Others:		21.82 GB	
	Used Memory	24.04 GB	

18 rows found (1 row selected)

frodo
22
dynaTrace Client - Thu Apr 15 14-45-35 CEST 2010
Dashboard Edit Tools Settings Help
*demo
*Thu Apr 15 14-45-35 CEST 2010
Thu Apr 15 14-45-35 CEST 2010
Total Memory
Total Memory Content
References
References

References by Keep Alive

Thu Apr 15 14:45:35 CEST 2010 - dtServer@LTT2:28863

This view shows the keep alive references of an Instance or, for classes, the accumulated references of all Instances. Use the tab at the bottom to follow either Referrers or Referees.

See Documentation for further information.

Class/Instance	Referring Insta...	GC Size	Shallow Size	Additional info
com.dynatrace.diagnostics.util.compatible.OpenBitSet	211.697	9,60 MB	4,85 MB	
<VMRoot>	1	1,03 GB	0 Bytes	
com.dynatrace.diagnostics.core.datacenter.RootExecPath	16.206	920,58 MB	5,32 MB	
<Garbage>	1	90,19 MB	0 Bytes	
com.dynatrace.diagnostics.core.datacenter.ExecPathNode	190.426	66,74 MB	23,25 MB	
<VMRoot>	1	1,03 GB	0 Bytes	
com.dynatrace.diagnostics.core.datacenter.RootExecPath	15.086	923,13 MB	4,95 MB	
java.lang.Object[]	16.212	892,00 MB	29,94 MB	
com.dynatrace.diagnostics.global.BasicTransformation	252	27,56 KB	11,81 KB	

9 rows found (3 rows selected)

Figure 12: Analyzing heap dumps with Dynatrace
(Source: https://community.dynatrace.com/community/display/DOCDT63/Memory+Diagnostics)

taking a complete memory dump. However, according to unofficial reports by developers, analyzing a complete memory dump requires more than twice the size of the analyzed heap. In terms of run-time overhead, no data exists, but developers confirmed that Dynatrace suspends the VM while taking a snapshot via JVMTI.

App Dynamics. The *App Dynamics*[11] tool can track business transactions and user experience. It can also drill down to problem root causes at code level by identifying method hotspots or memory leaks.

In terms of memory monitoring, this tool can count object instances by instrumenting allocating instructions. Furthermore, it instruments collection classes to detect growing data structures, including callers to `put` and `add` methods as well as stale elements that are never retrieved. This way, the user does not have to dig through big heap dumps to make sense of complex object graphs, but is instead presented with information about a collection, unused elements, and modifying call stacks.

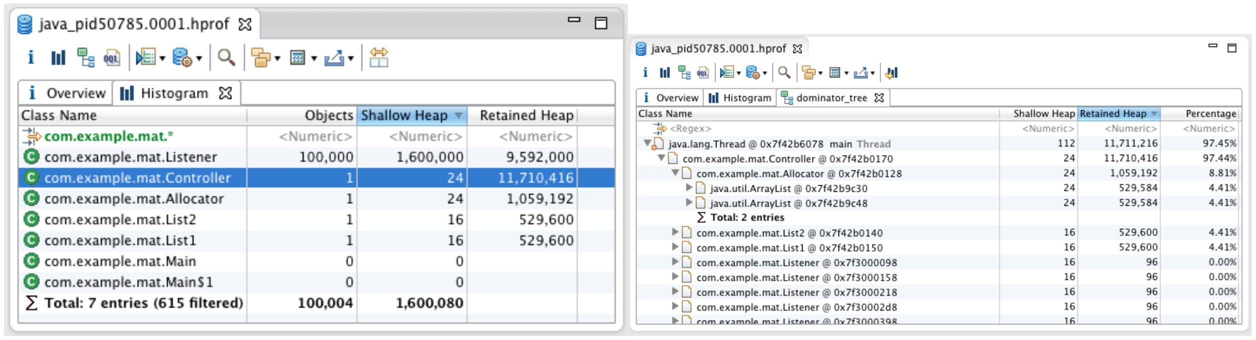
New Relic APM. The *New Relic APM*[12] tool can also track business transactions end-to-end, starting from the browser down into the database. In terms of memory monitoring, however, it is limited to what JMX Beans already expose. The tool can show memory usage (per memory pool) and GC time, but cannot drill down into the heap. It also does not provide simple accumulated statistics about the memory usage, such as instance counts per class.

Netbeans Profiler. Like many other IDE profilers, the *Netbeans Profiler* can detect method hotspots by sampling stack traces. However, it can also be configured to track object allocations, as well as the liveness of objects[13]. Allocations are tracked by instrumenting all allocating instructions. The liveness is tracked by

[11] Memory Monitoring in AppDynamics: https://www.appdynamics.com/info/jvm-monitor-memory
[12] New Relic APM: https://newrelic.com
[13] Memory Monitoring in the Netbeans Profiler: https://profiler.netbeans.org/docs/help/5.5/profile_memory_short.html

creating `java.lang.WeakReference`s for some of the allocated objects and by recording when the reference is set to `null` by the garbage collector. However, the developers advise to not do this for all objects to keep the overhead acceptable: *it is advised that only a small proportion of objects, say 1 out of 10 or 1 out of 20, is tracked in order to keep both temporal and spatial overhead under control.*

Eclipse Memory Profiler. The *Eclipse Memory Profiler*[14] enables the user to analyze a heap dump in detail. Besides the usual accumulated statistics such as instance counts and used memory per class, this profiler can also show the *retained heap*, i.e., the size of all that is kept alive by a specific group of objects (cf. Figure 13). Also, it provides some automatic analysis on the fill level of collections, and displays accesses to every collection. Finally, the user can also use the Object Query Language (OQL) to perform custom analyses.



Class Name	Objects	Shallow Heap	Retained Heap
com.example.mat.*	<Numeric>	<Numeric>	<Numeric>
com.example.mat.Listener	100,000	1,600,000	9,592,000
com.example.mat.Controller	1	24	11,710,416
com.example.mat.Allocator	1	24	1,059,192
com.example.mat.List2	1	16	529,600
com.example.mat.List1	1	16	529,600
com.example.mat.Main	0	0	
com.example.mat.Main$1	0	0	
Σ Total: 7 entries (615 filtered)	100,004	1,600,080	

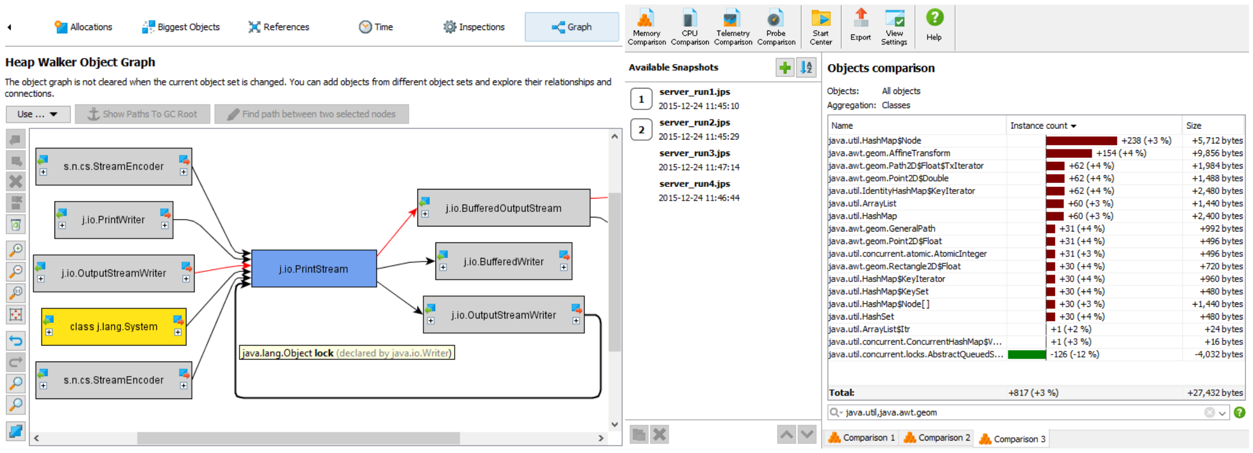
Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.lang.Thread @ 0x7f42b6078 main Thread	112	11,711,216	97.45%
com.example.mat.Controller @ 0x7f42b0170	24	11,710,416	97.44%
com.example.mat.Allocator @ 0x7f42b0128	24	1,059,192	8.81%
java.util.ArrayList @ 0x7f42b9c30	24	529,584	4.41%
java.util.ArrayList @ 0x7f42b9c48	24	529,584	4.41%
Σ Total: 2 entries			
com.example.mat.List2 @ 0x7f42b0140	16	529,600	4.41%
com.example.mat.List1 @ 0x7f42b0150	16	529,600	4.41%
com.example.mat.Listener @ 0x7f3000098	16	96	0.00%
com.example.mat.Listener @ 0x7f3000158	16	96	0.00%
com.example.mat.Listener @ 0x7f3000218	16	96	0.00%
com.example.mat.Listener @ 0x7f30002d8	16	96	0.00%
com.example.mat.Listener @ 0x7f3000398	16	96	0.00%

Figure 13: Analyzing a heap dump with the Eclipse Memory Profiler (Source: http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/)

JProfiler. *JProfiler*[15] can detect unresponsive or slow database queries or service requests. The tool can show hot methods as well as hot queries per transaction to determine the root cause of a performance problem. In terms of memory monitoring, the tool can visualize object graphs so the user can manually determine the root cause for a memory leak (cf. Figure 14). Furthermore, the tool also supports a simple comparison of accumulated heap statistics over time, enabling fast diagnosis of a change in memory behavior.

GC Viewer. The *GC Viewer*[16] is a small tool that can only visualize the GC behavior, e.g., in terms of pause times, heap usage, or freed memory, over time, as shown in Figure 15. Although this tool only offers limited functionality, it can be quite useful because it visualizes the information based on the GC log files that many VMs can already create. It thus simplifies the analysis of a GC log significantly.

[14]Tutorial for the Eclipse Memory Profiler: http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/
[15]JProfiler: https://www.ej-technologies.com/products/jprofiler/overview.html
[16]GC Viewer: http://www.tagtraum.com/gcviewer.html



Heap Walker Object Graph

The object graph is not cleared when the current object set is changed. You can add objects from different object sets and explore their relationships and connections.

Available Snapshots

1 **server_run1.jps** 2015-12-24 11:45:10
2 **server_run2.jps** 2015-12-24 11:45:29
server_run3.jps 2015-12-24 11:47:14
server_run4.jps 2015-12-24 11:46:44

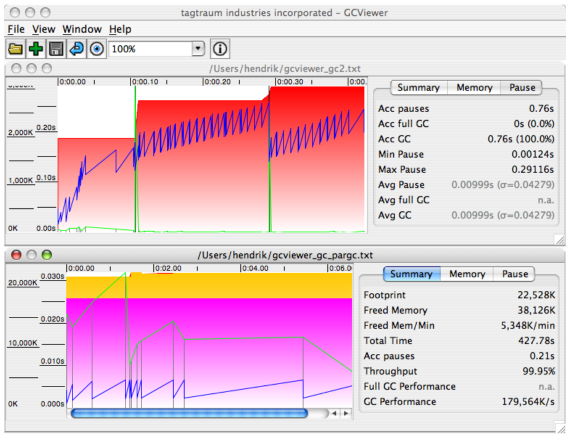
Objects comparison

Objects: All objects
Aggregation: Classes

Name	Instance count	Size
java.util.HashMap$Node	+238 (+3 %)	+5,712 bytes
java.awt.geom.AffineTransform	+154 (+4 %)	+9,856 bytes
java.awt.geom.Path2D$Float$TxIterator	+62 (+4 %)	+1,984 bytes
java.awt.geom.Point2D$Double	+62 (+4 %)	+1,488 bytes
java.util.IdentityHashMap$KeyIterator	+62 (+4 %)	+2,480 bytes
java.util.ArrayList	+60 (+3 %)	+1,440 bytes
java.util.HashMap	+60 (+3 %)	+2,400 bytes
java.awt.geom.GeneralPath	+31 (+4 %)	+992 bytes
java.awt.geom.Point2D$Float	+31 (+4 %)	+496 bytes
java.util.concurrent.atomic.AtomicInteger	+31 (+3 %)	+496 bytes
java.awt.geom.Rectangle2D$Float	+30 (+4 %)	+720 bytes
java.util.HashMap$KeyIterator	+30 (+4 %)	+960 bytes
java.util.HashMap$KeySet	+30 (+4 %)	+480 bytes
java.util.HashMap$Node[ ]	+30 (+3 %)	+1,440 bytes
java.util.HashSet	+30 (+4 %)	+480 bytes
java.util.ArrayList$Itr	+1 (+2 %)	+24 bytes
java.util.concurrent.ConcurrentHashMap$V...	+1 (+3 %)	+16 bytes
java.util.concurrent.locks.AbstractQueuedS...	-126 (-12 %)	-4,032 bytes
Total:	+817 (+3 %)	+27,432 bytes

java.util,java.awt.geom

Figure 14: Analyzing a heap dump with JProfiler (Source: https://www.ej-technologies.com/products/jprofiler/overview.html)


tagtraum industries incorporated – GCViewer
File View Window Help
100%
/Users/hendrik/gcviewer_gc2.txt
/Users/hendrik/gcviewer_gc_pargc.txt
Summary
Memory
Pause

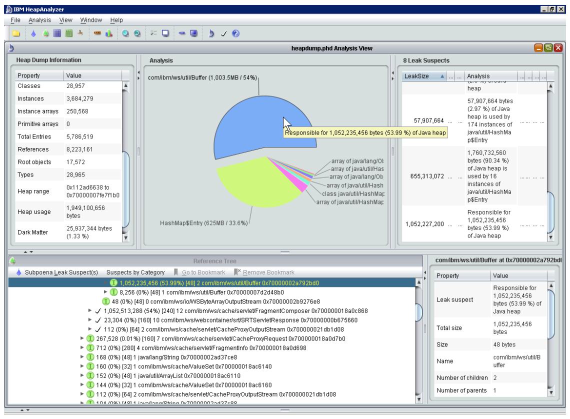
Acc pauses	0.76s
Acc full GC	0s (0.0%)
Acc GC	0.76s (100.0%)
Min Pause	0.00124s
Max Pause	0.29116s
Avg Pause	0.00999s (σ=0.04279)
Avg full GC	n.a.
Avg GC	0.00999s (σ=0.04279)

Footprint	22,528K
Freed Memory	38,126K
Freed Mem/Min	5,348K/min
Total Time	427.78s
Acc pauses	0.21s
Throughput	99.95%
Full GC Performance	n.a.
GC Performance	179,564K/s

Figure 15: Analyzing a GC log with the GC Viewer (Source: http://www.tagtraum.com/images/gcviewer-screenshot.png)

IBM Heap Analyzer. The *IBM Heap Analyzer*[17] is a tool specifically designed to analyze heap dumps. As shown in Figure 16, it can show accumulated statistics such as instance counts and memory usage per class. It can also categorize objects by size or by the accumulated size of all children. Using the latter metric, the tool can heuristically detect memory leaks by identifying large data structures. In terms of performance, neither run-time overhead nor memory requirements are explicitly given. However, the website of this tool states that the hardware requirements include *memory larger than the size of Java heaps*.

[17]IBM Heap Analyzer: https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=4544bafe-c7a2-455f-9d43-eb866ea60091



Property	Value
Classes	28,957
Instances	3,684,279
Instance arrays	250,568
Primitive arrays	0
Total Entries	5,786,519
References	8,223,161
Root objects	17,572
Types	28,965
Heap range	0x112ad6638 to 0x70000007fe7f1b0
Heap usage	1,949,100,656 bytes
Dark Matter	25,937,344 bytes (1.33 %)

LeakSize	...	...	Analysis	...	...	...	...
			heap				
57,907,664	...	...	57,907,664 bytes (2.97 %) of Java heap is used by 174 instances of java/util/HashMap$Entry	...	...	...	...
655,313,072	...	...	1,760,732,560 bytes (90.34 %) of Java heap is used by 16 instances of java/util/HashMap$Entry	...	...	...	...
1,052,227,200	...	...	Responsible for 1,052,235,456 bytes (53.99 %) of Java heap	...	...	...	...

```
        1,052,235,456 (53.99%) [48] 2 com/ibm/ws/util/Buffer 0x70000002a792bd0
          8,256 (0%) [48] 1 com/ibm/ws/util/Buffer 0x70000007d2d48b0
        48 (0%) [48] 0 com/ibm/ws/io/WSByteArrayOutputStream 0x70000002b9276e8
      ✓ 1,052,513,288 (54%) [240] 12 com/ibm/ws/cache/servlet/FragmentComposer 0x700000018a0c868
      ✓ 23,304 (0%) [160] 10 com/ibm/ws/webcontainer/srt/SRTServletResponse 0x70000000b675660
      ✓ 112 (0%) [64] 2 com/ibm/ws/cache/servlet/CacheProxyOutputStream 0x700000021db1d08
    267,528 (0.01%) [160] 7 com/ibm/ws/cache/servlet/CacheProxyRequest 0x700000018a0d7b0
    712 (0%) [280] 4 com/ibm/ws/cache/servlet/FragmentInfo 0x700000018a0d698
    168 (0%) [48] 1 java/lang/String 0x70000002ad37ce8
    160 (0%) [32] 1 com/ibm/ws/cache/ValueSet 0x700000018ac6140
    152 (0%) [48] 1 java/util/ArrayList 0x700000018ac6110
    144 (0%) [32] 1 com/ibm/ws/cache/ValueSet 0x700000018ac6160
    112 (0%) [64] 2 com/ibm/ws/cache/servlet/CacheProxyOutputStream 0x700000021db1d08
```

Property	Value
Leak suspect	Responsible for 1,052,235,456 bytes (53.99 %) of Java heap
Total size	1,052,235,456 bytes
Size	48 bytes
Name	com/ibm/ws/util/Buffer
Number of children	2
Number of parents	1

Figure 16: Analyzing a heap dump with the IBM Heap Analyzer (Source: https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=4544bafe-c7a2-455f-9d43-eb866ea60091)

Comparison. Figure 17 shows a comparison of the capabilities of the tools discussed above. Most of the tools support accumulated statistics for the entire heap. Object-based statistics and metrics over time are only supported by some tools.

In terms of overhead, only *ElephantTracks* reports it in a scientific way. A few other tools, provide some vague advice about their configuration in terms of overhead, e.g., *JRockit* states that tracing allocations should only be enabled for one type at a time and for `java.lang.String`s not at all. The majority of tools do not report any run-time overhead when monitoring memory.

Tool	Global/time	Heap-based	Object-based	Implementation
Hotspot^{TM}VM	yes	yes	no	dump-based
JConsole	no	yes	no	API-based
jhat	no	yes	no	dump-based
JRockit Mission Control	yes	yes	yes	instrumentation-based
Visual VM	yes	yes	yes	dump-based
Elephant Tracks	no	yes	yes	instrumentation-based
Dynatrace	yes	yes	yes	dump-based
App Dynamics	yes	yes	yes	dump-based
New Relic APM	yes	no	no	dump-based
Netbeans Profiler	no	yes	yes	instrumentation-based
Eclipse Memory Profiler	no	yes	yes	dump-based
JProfiler	yes	yes	yes	dump-based
GC Viewer	yes	no	no	log-based
IBM Heap Analyzer	no	yes	yes	dump-based

Figure 17: Comparison of functionality (i.e., global metrics over time such as GC pauses, accumulated heap-based metrics such as instance counts per class for a specific point in time, and object-based metrics such as pointers) and the used technique for state-of-the-art memory monitoring tools

Chapter 3

Observer Effect

"Every measurement of a system changes the system." - Unknown

When monitoring an application's memory behavior, the observed behavior can be significantly distorted. The magnitude of the distortion can be so vast that analysis results become useless or even misleading. Also, the distortion may not be limited to an increased overall run-time, but also to an increased GC frequency, an increased GC time or even an increased memory usage. To make matters worse, the same metrics might even show significant decreases due to the VM's heuristics reacting differently to the changed environment.

The following sections explore the impact of a sampling approach, i.e., regular heap dumps that are taken throughout the application's execution, a plain instrumentation approach, i.e., just instrumenting allocating instructions to record allocations, and an instrumentation approach that also monitors object deaths, i.e., in additional to recording all allocating instructions, a `java.lang.reference.PhantomReference` object is used to detect object deallocations. We have created prototype implementations for these three approaches to build a foundation for a scientifically valid comparison.

A `java.lang.reference.PhantomReference` is a special form of a *weak reference*. An object that is only referenced by weak references can be collected and the GC has to set all weak references to that object to null. Additionally, the GC triggers an event when such reference is cleared, enabling monitoring tool to record the death of the object. There are alternative approaches to record the death of an object in Java, e.g., using the `finalize` method that is called on every object before it is collected or manually forcing a GC in regular intervals (or even at every GC point as *ElephantTracks* does) to check what objects are still alive. However, both alternative approaches distort the result even more. A finalizer postpones the collection of an object because the GC first has to detect that the object is unreachable, then continue the application, and finally collect the object when the finalizer has finished and no new reference to the object has been created in the meantime. A manually forced GC at every GC point obviously generates quite different heap behavior because the used heap tends to stay very small and different GC heuristics kick in as the GC time is by far outweighing the mutator's execution time. We prefer a `PhantomReference` over a `SoftReference` or a plain `WeakReference`. A `SoftReference` can be cleared by the GC as well, however, the GC tries to keep the object alive as long as possible. This kind of reference is used for cache implementations, which distorts the GC behavior because objects are kept alive as long as possible. The difference between a `WeakReference` and a `PhantomReference` is that the GC can immediately collect an object that is just referenced by a `PhantomReference`, making it ideal for recording object deaths. Using a `WeakReference`, the event that is triggered still provides a hard reference (a `PhantomReference` triggers the event but does not provide the reference because the object has already been collected), creating the same problem that arises when using the `finalize` methods.

As the instrumentation approaches generate death events only after a GC (because the references are cleared there), we adjusted the sampling frequency for the sampling approach so that the amount of samples is approximately the same as the amount of garbage collections. Otherwise, when choosing an arbitrary sampling frequency, the results are hardly comparable to the instrumentation approaches.

We have conducted the following experiment to examine the extent of the observer effect follow the same methodology and use the same setup as described in Section 8. However, for simplicity we have reduced the set of benchmarks to only 7 representing a variety of different behavior. The *SPECjvm compiler.compiler* allocated a significant amount of fairly small objects with long lifetimes, thus creating big heaps in the process. Similarly, the *SPECjvm xml.validation* also allocates many objects, but with only very small lifetimes. Most of the objects do not survive the first collection. The *SPECjvm mpegaudio* creates only few, but very large objects (mainly arrays). *DaCapo avrora* is a benchmark with only a few allocations but it employs many threads and makes heavy use of locking, which makes this benchmark interesting in terms of how long it takes to reach the next safepoint. *DaCapo jython* generates a lot of dynamic code allocating short-living objects. The *DaCapoScala factorie* benchmark creates a big heap with only one thread, resulting in an intense load for the GC. *DaCapoScala scalaxb* is a small benchmark, implemented in Scala, that allocates only a few objects.

3.1 Run Time

Figure 18 shows the run-time impact of memory monitoring when using sampling, plain instrumentation, or instrumentation with recording object deaths. Using the sampling approach, we can see that the overhead is fairly small for all benchmarks where we expected small heaps. The plain instrumentation overhead, however, depends on the number of allocated objects, not on the heap size. When also monitoring the object deaths, the instrumentation approach crashes three benchmarks with `OutOfMemoryErrors` and internal timeouts because the imposed overhead is too big. This behavior is consistent with state-of-the-art tools using instrumentation, such as *ElephantTracks*.

Overall, it is easy to see that every approach imposes unacceptable overhead for production use in some cases.

3.2 GC Count

Figure 19 shows the GC count when using sampling, plain instrumentation, or instrumentation with recording object deaths. For most benchmarks, the GC count remains almost unchanged in the sampling approach. Since no additional objects are allocated by that approach, the GC has effectively the same work to do.

The *DaCapoScala factorie* benchmark, however, shows a 52% increase in the number of garbage collections. Even although everything remains effectively the same for the GC, more GCs are performed. This is because the *factorie* benchmark shows a 277% overhead in the overall run time. The overhead produces

Benchmark	Sampling	Instrumentation (plain)	Instrumentation (+death)
SPECjvm compiler.compiler	179.20%	52.62%	crashed
SPECjvm mpegaudio	2.44%	0.82%	4.25%
SPECjvm xml.validation	278.26%	49.34%	crashed
DaCapo avrora	2.56%	0.76%	13.90%
DaCapo jython	113.74%	98.88%	415.03%
DaCapoScala factorie	277.40%	349.79%	crashed
DaCapoScala scalaxb	35.17%	40.08%	155.88%

Figure 18: Observer effect on the overall run time when using sampling, plain instrumentation, or instrumentation with recording object deaths (gray is the base run time, red the introduced overhead)

Benchmark	Sampling	Instrumentation (plain)	Instrumentation (+death)
SPECjvm compiler.compiler	0.00%	0.00%	crashed
SPECjvm mpegaudio	6.38%	84.04%	−80.85%
SPECjvm xml.validation	0.00%	0.00%	crashed
DaCapo avrora	0.00%	22.22%	622.22%
DaCapo jython	1.74%	13.37%	15.12%
DaCapoScala factorie	52.22%	172.70%	crashed
DaCapoScala scalaxb	6.30%	30.71%	−36.22%

Figure 19: Observer effect on the GC count when using sampling, plain instrumentation, or instrumentation with recording object deaths (gray is the base GC count, red the introduced overhead)

a significant change in the ratio of execution versus GC time, resulting in different GC heuristics being applied. This behavior is a perfect example of how, in theory, the sampling approach should not distort the application's behavior, and yet, it unexpectedly does.

The same effect is even more visible in the instrumentation approaches. The plain instrumentation approach also increases the GC count (although, again, the effective work for the GC remains the same) because slight alterations in the timings result in different GC behavior. When also object deaths are recorded, the effect is so big that three benchmarks crash because the amount of objects is doubled (for every object, a `PhantomReference` object must be allocated). The other benchmarks show completely unpredictable behavior, some show significantly increased GC activity (e.g., *DaCapo avrora*), whereas others show significant drops (e.g, -80% for *SPECjvm mpegaudio*).

3.3 GC Time

Figure 20 shows the overall GC times when using sampling, plain instrumentation, or instrumentation with recording object deaths. Although the GC count stays the same for almost all benchmarks using the sampling approach, the GC time shows unexpected but significant changes. The same holds for the plain instrumentation approach. When also object deaths are recorded, the GC time is increased by up to 48154% although for some benchmarks the GC count even dropped. Although we might have expected something like this, the extent is surprising as it shows no correlation with the heap usage or the amount of allocated objects.

Benchmark	Sampling	Instrumentation (plain)	Instrumentation (+death)
SPECjvm compiler.compiler	2.48%	6.66%	crashed
SPECjvm mpegaudio	3.80%	79.75%	2893.67%
SPECjvm xml.validation	24.76%	6.02%	crashed
DaCapo avrora	16.00%	8.00%	31960.00%
DaCapo jython	24.05%	102.85%	48154.43%
DaCapoScala factorie	67.15%	85.34%	crashed
DaCapoScala scalaxb	2.10%	45.56%	13410.18%

Figure 20: Observer effect on the overall GC time when using sampling, plain instrumentation, or instrumentation with recording object deaths (gray is the base GC time, red the introduced overhead)

3.4 Memory Usage

Figure 20 shows the memory usage, i.e., the maximum used heap size, when using sampling, plain instrumentation, or instrumentation with recording object deaths. Interestingly, the memory usage stayed almost the same or dropped for all benchmarks when using the sampling approach and the plain instrumentation approach. When recording object deaths, however, the amount of additional memory used exceeds the expected values by far (we would have expected at most twice the memory usage as we double the amount of objects). Due to different timings, GCs are at different points in time and different heuristics kick in, making the behavior unpredictable.

Benchmark	Sampling	Instrumentation (plain)	Instrumentation (+death)
SPECjvm compiler.compiler	−0.31%	1.22%	crashed
SPECjvm mpegaudio	−3.43%	−33.22%	864.43%
SPECjvm xml.validation	0.80%	0.16%	crashed
DaCapo avrora	1.99%	−2.37%	60.82%
DaCapo jython	−0.42%	−2.80%	508.06%
DaCapoScala factorie	−27.48%	−39.84%	crashed
DaCapoScala scalaxb	−6.00%	−3.72%	1166.19%

Figure 21: Observer effect on the overall memory usage when using sampling, plain instrumentation, or instrumentation with recording object deaths (grey is the base memory usage, red the introduced overhead)

Chapter 4

Low-overhead Object and GC Tracing

"Never memorize something that you can look up." - *Albert Einstein*

The tracing of memory operations, such as object allocations, object deallocations, and object movements is a difficult task if the traced application is to remain undistorted (cf. Section 3). There is ample work on monitoring tools working at object-level granularity, but all fail at that very task (cf. Section 2). This chapter presents concepts and solutions for extending existing virtual machines to support the recording of all events necessary to rebuild the heap state offline for any point in time, imposing a significantly lower overhead than current state of the art approaches (cf. Section 8 for a detailed evaluation of that claim). These concepts have been implemented in the *AntTracks Virtual Machine*, a prototypical VM based on the Java Hotspot^{TM} VM. Therefore, the term *AntTracks* will be used synonymously for the general concept and the prototypical VM. AntTracks has first been published in Lengauer et al. [29].

The goal of AntTracks is to reconstruct a *heap state* for any point in time after the application has terminated. We define the heap state conceptually as a map from addresses to object information. The object information contains the objects content as well as meta information about the object, which includes:

1. Type (e.g., `java.lang.String` or `char[]`)
2. Size (e.g., 16 bytes)
3. Length (if it is an array) (e.g., 10 elements)
4. Allocation site (e.g., `java.lang.String::substring(II)` including as many callers as possible)
5. Allocating thread (e.g., `Java Thread` with the name `Worker Thread 42`)
6. Allocating code (e.g., compiled code or interpreted code)
7. Allocating mode (e.g., fastly into a TLAB or slowly outside a TLAB)

Consequently, all this information must be tracked when an object is allocated.

Figure 22 shows the basic architecture of AntTracks and how it is embedded into an existing JVM. AntTracks comprises two main components, i.e., the *Events Processor* and the *Symbols Store*. The Events Processor receives all events, takes care of synchronization and buffering, and writes them to a trace file. The Symbols Store maps symbols needed for the events, e.g., type names, type sizes, and method locations, to IDs. These IDs are used int the events passed to the Events Processor in order to keep them compact.

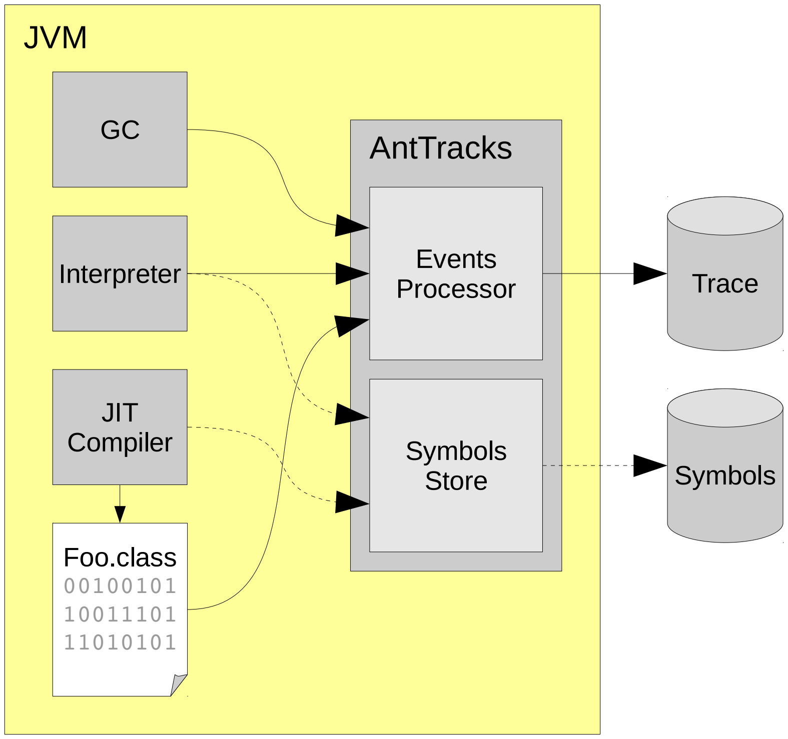


Figure 22: Overview of the AntTracks components within a virtual machine

Any event-based architecture has two performance-critical operations when firing an event: (1) building the event and (2) transporting the event to its destination. Both operations must be done as fast as possible. However, not every event is performance-critical as some events occur rarely or represent an operation that is slow itself. For example, an allocation in compiled and highly-optimized code is a very frequent and fast operation in modern virtual machines. Consequently, firing an event representing this allocation is performance-critical. An allocation done by interpreted code, on the other hand, is slow anyway. Therefore, firing an event for that allocation is not performance-critical. AntTracks optimizes a lot of events based on an distinction between the *common case* and other *rare cases*.

4.1 Common Case Optimizations

Tracking memory operations is a tedious task because they are highly optimized and diverse in todays virtual machines. Even an operation as simple as a plain object allocation can be carried out in a number of ways depending on the method state (e.g., is the method compiled? How aggressively was it compiled?), on the current VM state (e.g., what is the current VM configuration? What is the state of adaptive heuristics? How many threads are actively allocating?), on the current heap state (i.e., what is the current fragmentation level? Is the heap limit about to be reached?), and on the GC (i.e., is a GC pending, what kind of algorithm will be used to collect).

Consequently, we have analyzed which operation variants are common and which are rare. We can then optimize for the frequent variants and thus increase the overall performance. Please note, that we do not neglect rare operations, but rather focus optimization efforts on the frequent ones.

The following experiments to determine optimization opportunities follow the same methodology and use the same setup as described in Section 8.

4.1.1 Compiled Code Allocations

A core concept of Java Virtual Machines is hot code compilation in combination with feedback-guided optimizations. Therefore, we claim that most objects are allocated by compiled code and that interpreted allocations will almost vanish after the application has warmed up as well as that compiled allocations have good optimization potential.

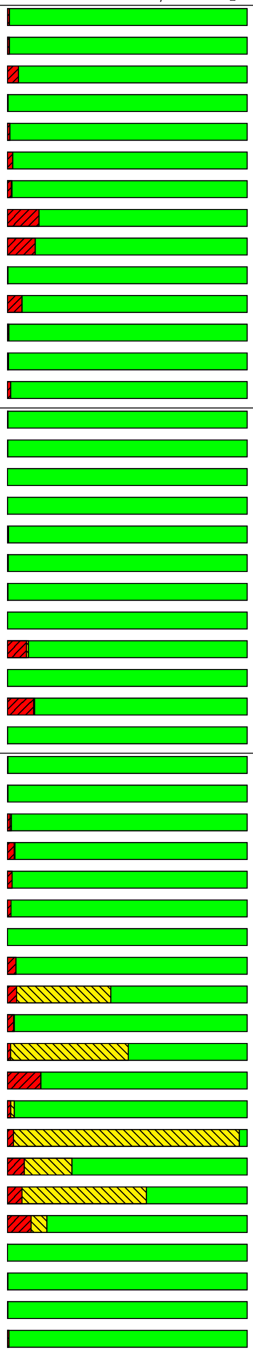
Figure 23 shows the allocations per benchmark (number of objects as well as memory size) and the distribution of allocations executed by the VM, by the interpreter, and by compiled code respectively. As expected, compiled-code allocations are dominant over interpreted allocations. Only some scimark benchmarks still execute a lot of interpreted allocations. This is due to the fact, that those benchmarks do a lot of number crunching and rarely allocate, thus allocating methods are executed rarely and take longer to warm up. Thus it is difficult to completely warm up those methods.

Also, some benchmarks show a significant portion of VM-internal allocations, e.g., lusearch and pmd. VM-internal allocations are allocations of helper objects that are executed by VM-internal native code with no corresponding Java code. A closer investigation revealed that those benchmarks use exceptions for control flow, i.e., they throw a `java.io.IOException` to indicate the end of a stream rather than returning a dedicated `EOF` token. However, allocating an exception object calls the `fillInStackTrace` method, which stores the current stacks in VM-internally allocated arrays.

4.1.2 Thread-local Allocations and Thread-local Moves

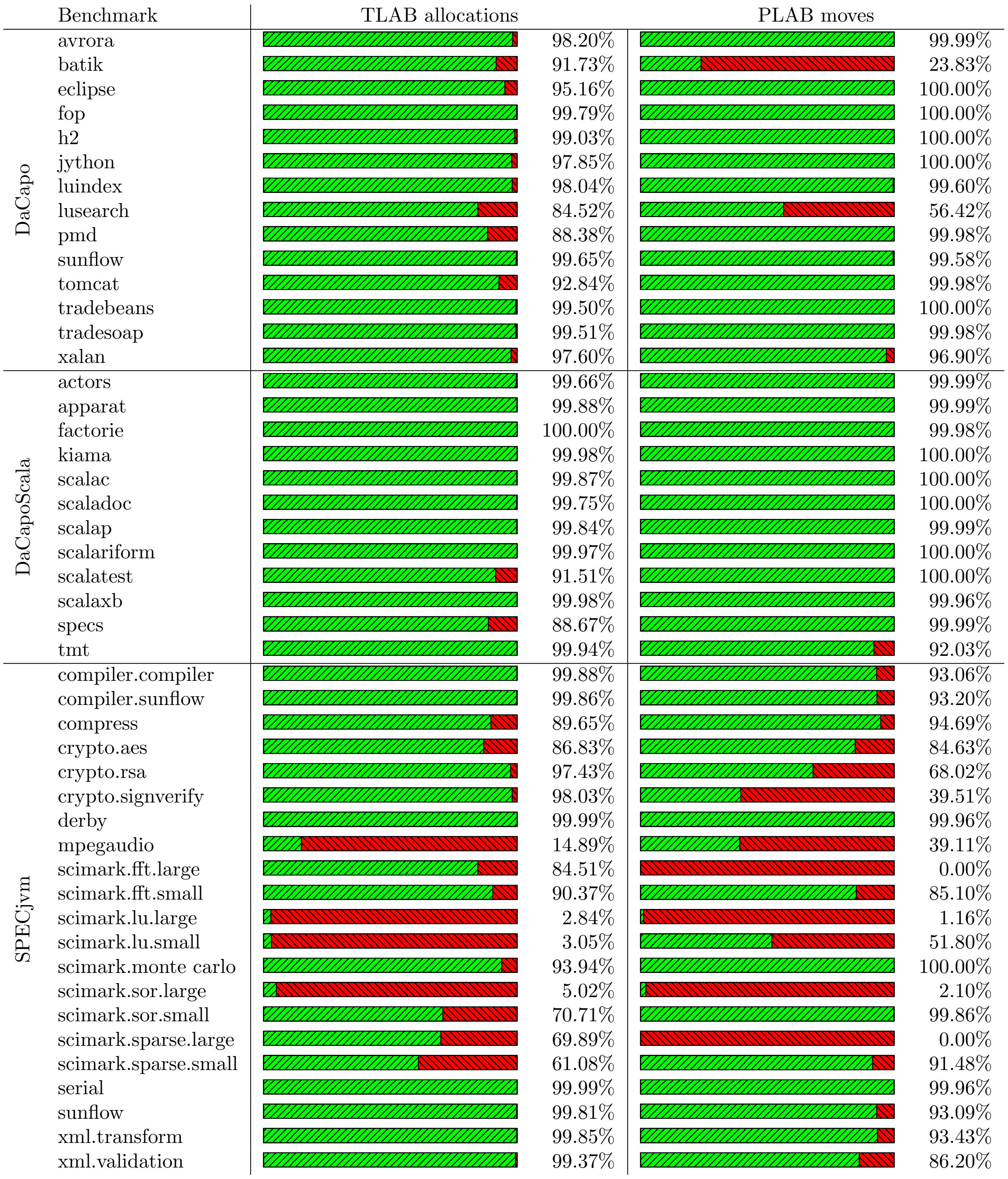
Todays applications may use hundreds of threads that allocate in parallel. Moreover, modern GC algorithms make use of multi-core machines and collect the heap with multiple threads at once, i.e., by walking the object graph in parallel and evacuating live objects to a survivor space, or by splitting the heap into several regions that are collected by different GC threads. As such operations would require locking the entire heap (to prevent two threads allocating an new object at the same location), these operations are done in a thread-local context. This means that every mutator thread maintains a thread-local allocation buffer (TLAB) and every GC thread maintains a promotion-local allocation buffer (PLAB). Therefore, the heap must only be locked rarely for allocating a large junk of memory at once (TLAB or PLAB), which is then owned by the allocating thread and can be used for allocations without locking.

We claim that (1) the allocation of TLABs and PLABs is rather rare, and that (2) allocations and moves into TLABs and PLABs respectively are frequent compared to allocations and moves outside this thread-local context. Figure 24 shows the ratio of TLAB allocations and the ratio of PLAB moves relative to the overall allocations and the overall moves respectively.



	Benchmark	Allocations [#, MB]		VM / Interpreted / Compiled [%]		
DaCapo	avrora	7481780	214.34	0.77%	0.04%	99.19%
	batik	459405	31.64	0.69%	0.15%	99.16%
	eclipse	682930	40.21	4.55%	0.02%	95.42%
	fop	2520097	101.96	0.18%	0.06%	99.75%
	h2	394466207	14271.58	0.97%	0.00%	99.03%
	jython	180035686	7454.61	2.14%	0.00%	97.86%
	luindex	217429	10.06	1.59%	0.30%	98.11%
	lusearch	21899743	2108.43	13.11%	0.00%	86.89%
	pmd	15710105	540.53	11.53%	0.02%	88.45%
	sunflow	138438818	5960.33	0.17%	0.00%	99.82%
	tomcat	6972352	381.81	5.99%	0.11%	93.90%
	tradebeans	921049446	36893.99	0.49%	0.00%	99.51%
	tradesoap	944964802	43070.77	0.39%	0.00%	99.61%
	xalan	102428226	4739.75	1.29%	0.00%	98.71%
DaCapoScala	actors	236220524	5556.11	0.15%	0.00%	99.85%
	apparat	393032748	12089.15	0.05%	0.00%	99.94%
	factorie	5793168425	132698.34	0.00%	0.00%	100.00%
	kiama	11478471	381.57	0.01%	0.01%	99.98%
	scalac	41416801	1258.83	0.13%	0.29%	99.59%
	scaladoc	42707749	1751.17	0.24%	0.01%	99.74%
	scalap	3479182	83.73	0.10%	0.04%	99.86%
	scalariform	50525439	1197.28	0.02%	0.00%	99.98%
	scalatest	3824928	160.46	7.90%	0.81%	91.29%
	scalaxb	98635027	2331.27	0.01%	0.01%	99.98%
	specs	6426379	290.60	10.85%	0.47%	88.68%
	tmt	2662914172	62365.64	0.02%	0.00%	99.98%
SPECjvm	compiler.compiler	460537491	14663.88	0.10%	0.00%	99.90%
	compiler.sunflow	1205127548	40451.25	0.12%	0.00%	99.88%
	compress	36726	62.00	1.11%	0.50%	98.39%
	crypto.aes	73072	350.36	2.81%	0.31%	96.88%
	crypto.rsa	32741578	1864.99	1.90%	0.00%	98.10%
	crypto.signverify	3979253	786.52	1.46%	0.00%	98.54%
	derby	2001049501	74367.55	0.00%	0.00%	100.00%
	mpegaudio	551750	265.31	3.50%	0.03%	96.47%
	scimark.fft.large	1162	82.98	3.79%	39.33%	56.88%
	scimark.fft.small	84312	333.74	2.56%	0.26%	97.18%
	scimark.lu.large	34352	65.44	1.29%	49.12%	49.60%
	scimark.lu.small	3723904	821.89	13.87%	0.01%	86.12%
	scimark.monte carlo	13506	1.36	1.33%	1.49%	97.18%
	scimark.sor.large	17864	33.22	2.47%	94.27%	3.25%
	scimark.sor.small	11146	1.51	7.00%	19.88%	73.12%
	scimark.sparse.large	1375	59.81	6.04%	51.93%	42.04%
	scimark.sparse.small	5580	35.28	9.86%	6.56%	83.58%
	serial	3345308405	127549.31	0.00%	0.00%	100.00%
	sunflow	2046338355	88704.80	0.09%	0.00%	99.91%
	xml.transform	261451398	9471.40	0.08%	0.00%	99.92%
	xml.validation	809767272	26745.53	0.57%	0.00%	99.43%

Figure 23: Distribution of allocations executed by the VM, by interpreted code, and compiled-code



	Benchmark	TLAB allocations	PLAB moves
DaCapo	avrora	98.20%	99.99%
	batik	91.73%	23.83%
	eclipse	95.16%	100.00%
	fop	99.79%	100.00%
	h2	99.03%	100.00%
	jython	97.85%	100.00%
	luindex	98.04%	99.60%
	lusearch	84.52%	56.42%
	pmd	88.38%	99.98%
	sunflow	99.65%	99.58%
	tomcat	92.84%	99.98%
	tradebeans	99.50%	100.00%
	tradesoap	99.51%	99.98%
	xalan	97.60%	96.90%
DaCapoScala	actors	99.66%	99.99%
	apparat	99.88%	99.99%
	factorie	100.00%	99.98%
	kiama	99.98%	100.00%
	scalac	99.87%	100.00%
	scaladoc	99.75%	100.00%
	scalap	99.84%	99.99%
	scalariform	99.97%	100.00%
	scalatest	91.51%	100.00%
	scalaxb	99.98%	99.96%
	specs	88.67%	99.99%
	tmt	99.94%	92.03%
SPECjvm	compiler.compiler	99.88%	93.06%
	compiler.sunflow	99.86%	93.20%
	compress	89.65%	94.69%
	crypto.aes	86.83%	84.63%
	crypto.rsa	97.43%	68.02%
	crypto.signverify	98.03%	39.51%
	derby	99.99%	99.96%
	mpegaudio	14.89%	39.11%
	scimark.fft.large	84.51%	0.00%
	scimark.fft.small	90.37%	85.10%
	scimark.lu.large	2.84%	1.16%
	scimark.lu.small	3.05%	51.80%
	scimark.monte carlo	93.94%	100.00%
	scimark.sor.large	5.02%	2.10%
	scimark.sor.small	70.71%	99.86%
	scimark.sparse.large	69.89%	0.00%
	scimark.sparse.small	61.08%	91.48%
	serial	99.99%	99.96%
	sunflow	99.81%	93.09%
	xml.transform	99.85%	93.43%
	xml.validation	99.37%	86.20%

Figure 24: Ratio of TLAB allocations (green) relative to non-TLAB allocations (red) and ratio of PLAB moves (green) relative to non-PLAB moves (red)

With the exception of the scimark.lu.large, scimark.lu.small and scimark.sor.large benchmarks, all benchmarks have TLAB allocations as their dominant method of allocation. The same holds for PLAB moves. Consequently, we want to optimize for both TLAB allocations and PLAB moves (we will also optimize non-PLAB moves in Section 4.1.5). Allocations outside a LAB can be neglected in terms of overall run-time performance.

4.1.3 Small Array Allocations

A common assumption is that most objects are either instances or small arrays and that large arrays are the exception. As the array length will be part of the resulting events, optimizing for instances and arrays of which the length can be stored in a single byte (instead of in four bytes) would be an obvious optimization.

To quantify this claim, we looked at the distribution of instances and arrays as well as at the average array length and at the amount of arrays with a length smaller than 255 (0xFF). Figure 25 shows the distribution of array lengths as well as the percentage of instances, small arrays and big arrays.

We can easily see that in all benchmarks except for mpegaudio, and the scimark benchmarks, the median array length is 33 or smaller. The benchmarks with a bigger median are number crunching benchmarks using large arrays as buffers or matrix data structures.

The distribution of instances, small arrays, and big arrays shows that only a small portion of objects are big arrays. Thus, we want to optimize for instances as well as for small arrays.

4.1.4 Survivor Ratio

As the run-time complexity of many modern GC algorithms depends on either the number of live objects or the total heap size, a GC will try to collect when the number of survivors is expected to be the smallest. Because dead objects can only get more and not less, this means that a GC will collect as late as possible. We therefore claim, that the number of dead objects will usually outweigh the number of surviving objects.

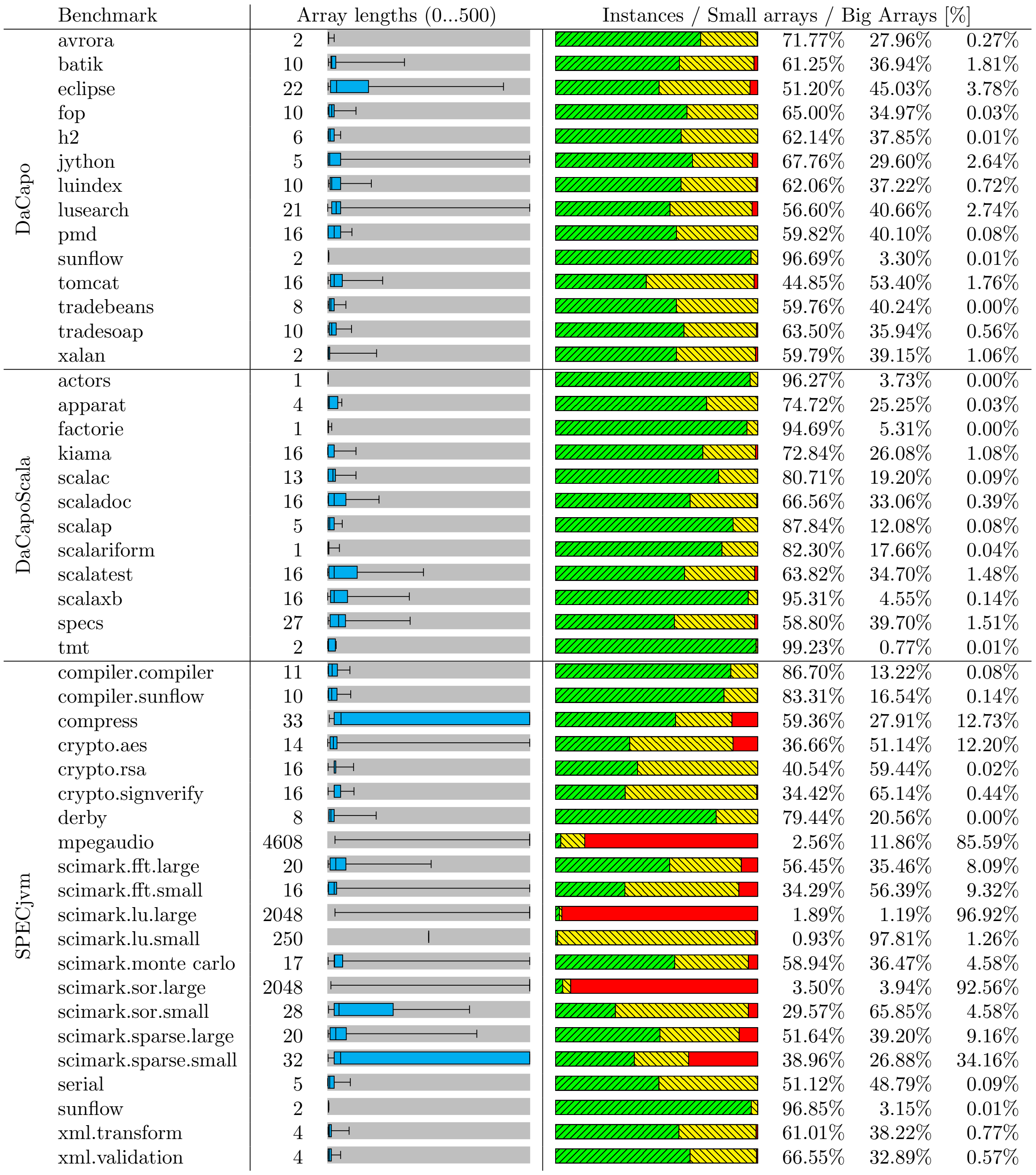
Figure 26 shows the distribution of survivor ratios for minor and major GCs, i.e., every data point is the survivor ratio of a single GC within that benchmark run. This figure also shows the ratio of minor GCs relative to major GCs. Manually triggered GCs (`System.gc()` and `JVMTI ForceGC`) have been excluded.

As expected, the results show that minor GCs are much more frequent over major GCs. Furthermore, in minor GCs, most objects do not survive. On the other hand, the survivor ratio of major GCs is relatively high, meaning that old objects are likely to survive again. During a minor collection, we therefore want to optimize for live objects rather than dead objects. For major collections, however, we will not optimize for dead objects, but rather use the optimization described in Section 4.1.5.

4.1.5 Clustered Moves

Objects that reference each other are obviously often used in combination together. Consequently, it provides good performance if they are located next to each other because then they are more likely to reside in the same cache line. Additionally, objects that are allocated one after another are also likely to reference each other because they have been allocated in the same context. Garbage collectors try to keep referencing objects near to each other for those reasons.

Therefore, we claim that objects are likely to survive or die in clusters. Figure 27 shows the average cluster size of surviving objects during major collections. Consequently, optimizing for clusters will improve overall performance.



	Benchmark	Array lengths (0...500)	Instances / Small arrays / Big Arrays [%]		
DaCapo	avrora	2	71.77%	27.96%	0.27%
	batik	10	61.25%	36.94%	1.81%
	eclipse	22	51.20%	45.03%	3.78%
	fop	10	65.00%	34.97%	0.03%
	h2	6	62.14%	37.85%	0.01%
	jython	5	67.76%	29.60%	2.64%
	luindex	10	62.06%	37.22%	0.72%
	lusearch	21	56.60%	40.66%	2.74%
	pmd	16	59.82%	40.10%	0.08%
	sunflow	2	96.69%	3.30%	0.01%
	tomcat	16	44.85%	53.40%	1.76%
	tradebeans	8	59.76%	40.24%	0.00%
	tradesoap	10	63.50%	35.94%	0.56%
	xalan	2	59.79%	39.15%	1.06%
DaCapoScala	actors	1	96.27%	3.73%	0.00%
	apparat	4	74.72%	25.25%	0.03%
	factorie	1	94.69%	5.31%	0.00%
	kiama	16	72.84%	26.08%	1.08%
	scalac	13	80.71%	19.20%	0.09%
	scaladoc	16	66.56%	33.06%	0.39%
	scalap	5	87.84%	12.08%	0.08%
	scalariform	1	82.30%	17.66%	0.04%
	scalatest	16	63.82%	34.70%	1.48%
	scalaxb	16	95.31%	4.55%	0.14%
	specs	27	58.80%	39.70%	1.51%
	tmt	2	99.23%	0.77%	0.01%
SPECjvm	compiler.compiler	11	86.70%	13.22%	0.08%
	compiler.sunflow	10	83.31%	16.54%	0.14%
	compress	33	59.36%	27.91%	12.73%
	crypto.aes	14	36.66%	51.14%	12.20%
	crypto.rsa	16	40.54%	59.44%	0.02%
	crypto.signverify	16	34.42%	65.14%	0.44%
	derby	8	79.44%	20.56%	0.00%
	mpegaudio	4608	2.56%	11.86%	85.59%
	scimark.fft.large	20	56.45%	35.46%	8.09%
	scimark.fft.small	16	34.29%	56.39%	9.32%
	scimark.lu.large	2048	1.89%	1.19%	96.92%
	scimark.lu.small	250	0.93%	97.81%	1.26%
	scimark.monte carlo	17	58.94%	36.47%	4.58%
	scimark.sor.large	2048	3.50%	3.94%	92.56%
	scimark.sor.small	28	29.57%	65.85%	4.58%
	scimark.sparse.large	20	51.64%	39.20%	9.16%
	scimark.sparse.small	32	38.96%	26.88%	34.16%
	serial	5	51.12%	48.79%	0.09%
	sunflow	2	96.85%	3.15%	0.01%
	xml.transform	4	61.01%	38.22%	0.77%
	xml.validation	4	66.55%	32.89%	0.57%

Figure 25: Median array lengths (as well as the distribution of array lengths in the range of 0 and 500) and distribution of instances (green), small arrays (with a length of 255 or less) (yellow), and big arrays (with a length of 256 or higher) (red)

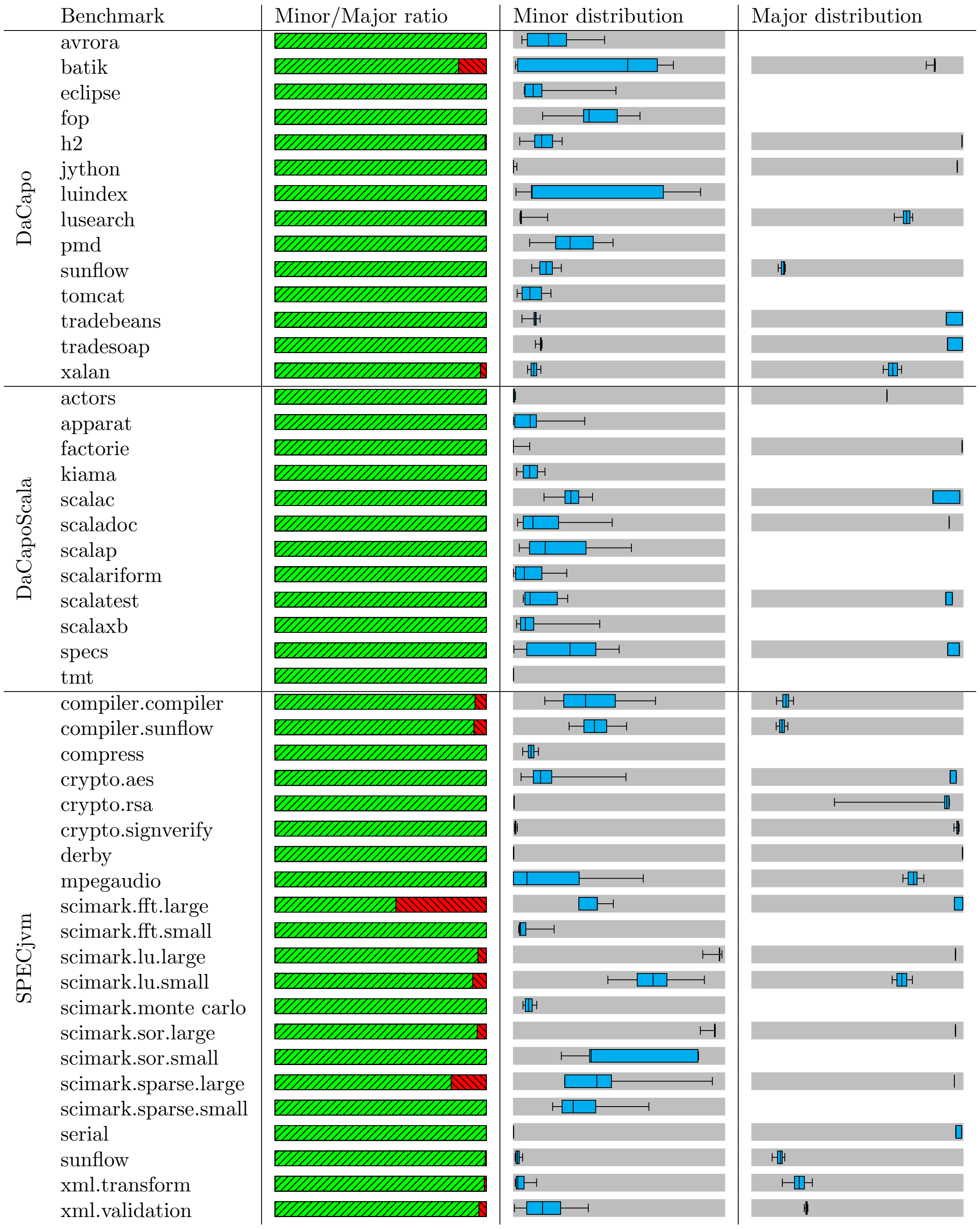


Figure 26: Ratio of minor GCs (green) relative to major GCs (red) and distribution of the survivor ratios of minor GCs and major GCs respectively

Benchmark	Avg. cluster size
batik	807.30
lusearch	107.60
sunflow	214.54
xalan	216.26
compiler.compiler	231.35
compiler.sunflow	346.75
crypto.rsa	226.86
crypto.signverify	207.96
mpegaudio	232.72
scimark.lu.small	105.82
sunflow	97.00
xml.transform	170.79
xml.validation	207.32

Figure 27: Arithmetic mean of survivor cluster size (number of objects) during major GCs

4.2 Events

In order to be able to analyze the memory behavior of an application, all memory-related operations are recorded and written to a trace file by the virtual machine. This trace file contains a sequence of events, where every event represents a single operation.

Figure 28 shows all events, including a short description. The events are split into 4 categories, i.e., *Phase Events*, *Bookkeeping Events*, *Allocation Events*, and *Move Events*. The trace is a sequence of these events. Every event starts with the ID (1 byte) followed by the event data. A compact format for the event data is paramount to keep the trace small, and, consequently, to reduce IO time. A detailed evaluation of the event distribution can be found in Section 8.2.1.

4.2.1 Phase Events

The trace can be split into two alternating phases, i.e., the mutator phase and the GC phase. The mutator phase contains events triggered by the application, i.e., mostly allocation events, whereas the GC phase contains events triggered by the GC, i.e., mostly move events. The *GC start* and *GC end* events indicate a change from the mutator phase to the GC phase and from the GC phase back to the mutator phase respectively. These two events carry the time (relative to the application start) as well as some information about the kind of GC, e.g., whether it is a major GC (collecting the entire heap) or a minor GC (collecting only some spaces of the heap).

Obviously, phase events must not occur out of order with other events. For example, a GC start event must be written to the trace after all events of the preceding mutator phase, and before all events of the succeeding GC phase. Similarly, the GC end event must be written after the preceding GC phase and before the succeeding mutator phase.

Additionally, the *GC info* event and the *GC failed* event can follow a GC start event, or precede a GC end event respectively, to modify the semantics. The GC info event specifies what spaces are collected in case of a minor GC, whereas the GC failed event indicates that a GC in a specific space has failed. Consequently, if the GC start event specifies a major GC, all spaces are collected. If the GC start event specifies a minor GC, it must be followed by a GC info event specifying the space that is collected. If a GC failed for a space,

	ID	Name	Semantics
	00	Nop	nothing
	01	Mark	marks a point in the trace
Phase	02	GC Start	a GC started
	03	GC End	the GC ended
	04	GC Info	meta information about the GC
	05	GC Failed	the GC failed
Bookkeeping	06	Space Create	a new space has been created
	07	Space Allocate	a space has been assigned a specific purpose
	08	Space Release	a space has been released and is not in use anymore
	09	Space Redefine	a space has been redefined in terms of start and end
	0A	Space Destroy	a space has been destroyed
	0B	Thread Alive	a thread has been detected as alive
	0C	Thread Death	a thread has just died
	0D	TLAB Allocation	a thread has just allocated a new TLAB
	0E	PLAB Allocation	a thread has just allocated a new PLAB
Allocations	0F	Allocation Slow	an object has been allocated by the VM
	10	IR Allocation Slow	an object has been allocated by interpreted code
	11	IR Allocation Slow with Type	an object has been allocated by interpreted code
	12	IR Allocation Normal	an object has been allocated by interpreted code
	13	IR Allocation Fast	an object has been allocated by interpreted code
	14	C1 Allocation Slow	an object has been allocated by c1-compiled code
	15	C1 Allocation Slow with Type	an object has been allocated by c1-compiled code
	16	C1 Allocation Normal	an object has been allocated by c1-compiled code
	17	C1 Allocation Fast	an object has been allocated by c1-compiled code
	18	C1 Allocation Fast Special	an object has been allocated by c1-compiled code
	19	C2 Allocation Slow	an object has been allocated by c2-compiled code
	1A	C2 Allocation Slow with Type	an object has been allocated by c2-compiled code
	1B	C2 Allocation Normal	an object has been allocated by c2-compiled code
	1C	C2 Allocation Fast	an object has been allocated by c2-compiled code
	1D	C2 Allocation Fast Special	an object has been allocated by c2-compiled code
Moves	1E	GC Move Slow	an object has been moved
	1F	GC Move Fast Wide	an object has been moved
	20	GC Move Fast	an object has been moved
	21	GC Move Fast Narrow	an object has been moved
	22	GC Move Region	several objects have been moved by the same offset
	23	GC Keep Alive	an object has been kept alive without moving
	24	GC Move Sync Wide	an object has been moved
	25	GC Move Sync	an object has been moved
	26	GC Deallocation	an object has been deallocated

Figure 28: All events, including their IDs and a short description of their semantics, events with equal semantics have alternating formats for different use cases.

the GC is aborted for that space, redefining all movements from that space as copy operations instead, effectively duplicating the object.

All phase events carry a GC ID. This ID is used to associate events with one another if multiple GCs run concurrently. However, a single space can be collected by only one GC at a time.

Figure 29 shows the format for all phase events The GC start and GC end event carry the type (minor vs major) in the second byte, the GC cause in the third byte (GC causes can be defined via the symbols), and some additional flags in the fourth byte (e.g., whether the GC has failed). The next 8 bytes carry the time of the event relatively to the application start in milliseconds. Finally, the last 8 bytes carry the new relative address base for all subsequent events. Since many events carry addresses, addresses make up a significant amount of the trace. Therefore, we define a base address here which is most likely not far from addresses we need to write to the trace in the next phase. Some events will try to encode their addresses relatively to this base address and will therefore be able to use addresses that fit into 3-4 bytes. For example, if the event is a GC start event, the base address will be set to the start of the space where new objects will be put. If the event is a GC end event, the base address will be set to the space where objects will be moved to or moved from.

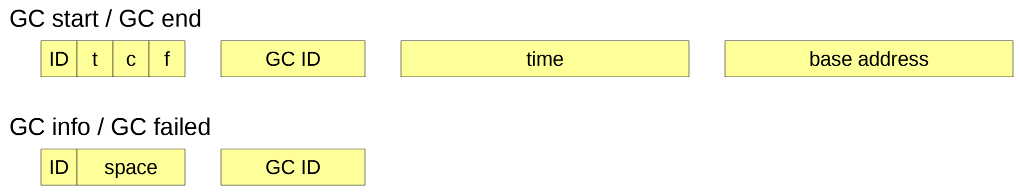


Figure 29: Event formats for phase events

4.2.2 Bookkeeping Events

There are several kinds of bookkeeping events in the trace, i.e., events that do not describe allocation or GC behavior, but that are nevertheless necessary to interpret other events correctly. These events describe the space layout of the heap, active threads, as well as their LABs. Bookkeeping events are not performance-critical as they occur rarely compared to allocation and move events.

The first category are space events, describing how the entire heap is split into individual spaces and what they are used for. Every space event carries a space id to uniquely identify a space. A *space create* event indicates that a new space has just been created with the given start and end address. Depending on the GC algorithm, the VM uses this event to statically define spaces at startup (e.g., in case of the ParallelOld GC) or to adaptively create new spaces (e.g., in case of the G1 GC). The *space allocate* event denotes that the specified space is in use from now on, and also, how this space is intended to be used. For example, the VM specifies a mode and a type: the mode is either `normal`, `humongous start`, or `humongous continued`, and the type is either `Eden`, `survivor`, or `old`. Normal denotes that the space is filled with objects that do not overlap space boundaries, humongous start indicates that the last object of the space extends into the next adjacent space, and humongous continued indicates that the start of the first object is in the previous adjacent space. The *space release* event indicates that the specified space is not in use anymore (counterpart to space allocate). If any objects remain in that space, they can be regarded as dead. However, the space can be re-allocated with the space allocation event later. The *space redefine* event redefines the start and the end address of the specified space. This event is used to describe the adaptive changes the GC can make to space sizes. Finally, the *space destroyed* event constitutes that the specified space has been fully destroyed (counterpart to space create), i.e., the ID of that space will not be used in any future events, unless the space create event is repeated. Figure 30 shows the format of space events.

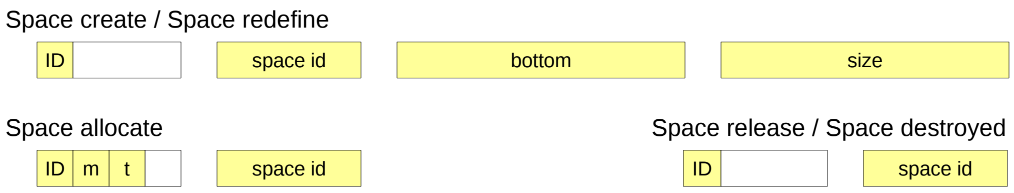

Space create / Space redefine
ID
space id
bottom
size
Space allocate
ID
m
t
space id
Space release / Space destroyed
ID
space id

Figure 30: Event formats for space events

Please note, that before any space event (except for space create), it is valid for the space to contain objects. Depending on the event, the objects either remain unchanged (e.g., in case of a space being reallocated to another use or redefined) or they can be considered dead (e.g., in case of the space being released or destroyed.

The *thread alive* and *thread dead* events are used by the VM to indicate that a thread is alive or has just died. The alive event can be sent multiple times, for example, to indicate a change in the threads name. The *TLAB allocation* and *PLAB allocation* events describe the current LABs the specified thread is holding. Every thread can hold one TLAB or one PLAB per space mode. If, for example, two TLAB allocation events are sent for the same thread, that means that the second TLAB will replace the first TLAB as the active one. However, in the case of PLABs, a second PLAB allocation event does not automatically replace the old PLAB, but rather only if they fall into spaces with matching modes. Figure 31 shows the format of space events. Please note, that the TLAB allocation and PLAB allocation event do not carry any thread ID, but this information will be reconstructed from the context of the events.

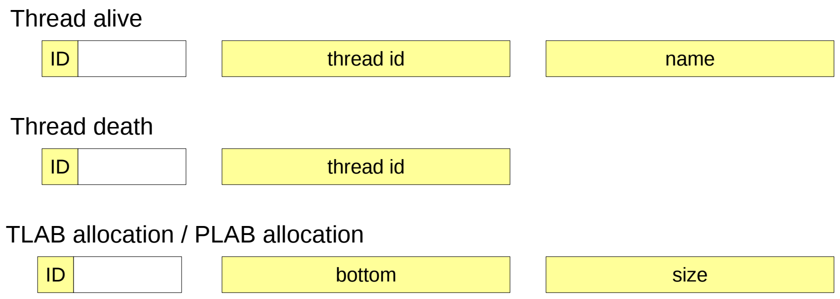


Figure 31: Event formats for thread events

4.2.3 Allocation Events

For every object allocation, exactly one allocation event is recorded. Allocation events are performance-critical because an object allocation is a frequent operation and they will make up a large portion of the trace (cf. Section 8).

Reconstructing Omitted Information

To keep the events as small as possible, all information is omitted that can be reconstructed offline. Candidates for omission are the the object's size, the object's type, and the object's address. However, reconstructing all the missing information is a difficult task, as presented in this section.

Reconstructing Object Types. The object type can be inferred from the allocation site. The symbol information associates every allocation site ID with a method, which a bytecode index (BCI) in that method, with the method's declaring class, as well as with the type of the objects usually allocated at that site. Thus, the post-processing tool can infer the type of the object by looking at the information associated with the allocation site ID. Figure 32 shows a method, in which the type of all allocated objects can be inferred from their allocation site. For example, if we see an allocation at BCI 4, 22 and 39, a `java.lang.StringIndexOutOfBoundsException` is allocated, whereas at BCI 65 a `java.lang.String` is allocated.

Reconstructing Object Sizes. Similar to the object type, we can also infer the object size from the allocation site, because the size of every type is stored in the symbols file. In case of array objects, the symbols file contains the header size as well as the element size. Consequently, the size of an array object can be computed by multiplying the element size with the array length stored in the event (and adding the header size).

Reconstructing Object Addresses. The VM uses TLABs to allocate objects without having to synchronize on the entire heap. After allocating a TLAB, the owning thread will put its first object at the beginning of its TLAB. The next object will be put right after the first object. Consequently, assuming that we know the address of the TLAB, and all allocation events are in correct temporal order (at least within the context of one thread), the address can be computed by adding the address of the TLAB to the sum of the sizes of all objects previously allocated into the same TLAB. Figure 33 shows an example on how to incrementally reconstruct object addresses within one TLAB, whereas Figure 34 shows the mathematical definition.

Unreconstructable Information. There are some corner cases in which the omitted information can not be reconstructed correctly. In these cases, a slow event format is used, that carries an explicit type identifier as well as the object size. Whether this information is included in the event must be indicated by the allocation site ID. By looking at the allocation site ID and by fetching the respective symbols information, the post-processing tool must be able to decide whether the object type and the object size are included.

There are two cases in which the object type and one case in which the object size cannot be reconstructed: (1) Calls to `java.lang.Object::clone()` are implemented via intrinsics in the JIT compiler. This means, that the call is replaced with an allocation. However, the generated code does not (and cannot) know what kind of object it will be allocating. Rather it resolves the object size at run-time and copies the entire content of the object (including the type information in the header); (2) Allocations of arrays with multiple dimensions can use a single bytecode to allocate all levels of the array. Therefore, a statement like `new int[2][3][4]` results in a single bytecode (and consequently in a single allocation site) allocating a total of 1 `int[][][]`, 2 `int[][]`, and 6 `int[]`. In this case, the VM must trigger multiple array allocation events in which it must specify the types of the arrays explicitly. The VM cannot trigger just a single event, because the arrays may be allocated into different areas in the heap, and the allocations might even be interrupted by

```
 0: iload_1
 1: ifge            13
 4: new             #6   // class java/lang/StringIndexOutOfBoundsException
 7: dup
 8: iload_1
 9: invokespecial   #7   // Method java/lang/StringIndexOutOfBoundsException."<init>":(I)V
12: athrow
13: iload_2
14: aload_0
15: getfield        #3   // Field value:[C
18: arraylength
19: if_icmple       31
22: new             #6   // class java/lang/StringIndexOutOfBoundsException
25: dup
26: iload_2
27: invokespecial   #7   // Method java/lang/StringIndexOutOfBoundsException."<init>":(I)V
30: athrow
31: iload_2
32: iload_1
33: isub
34: istore_3
35: iload_3
36: ifge            48
39: new             #6   // class java/lang/StringIndexOutOfBoundsException
42: dup
43: iload_3
44: invokespecial   #7   // Method java/lang/StringIndexOutOfBoundsException."<init>":(I)V
47: athrow
48: iload_1
49: ifne            65
52: iload_2
53: aload_0
54: getfield        #3   // Field value:[C
57: arraylength
58: if_icmpne       65
61: aload_0
62: goto            78
65: new             #43  // class java/lang/String
68: dup
69: aload_0
70: getfield        #3   // Field value:[C
73: iload_1
74: iload_3
75: invokespecial   #72  // Method "<init>":([CII)V
78: areturn
```

Figure 32: Bytecode of the `java.lang.String::substring(II)` method in which, for all allocations, the allocated type can be inferred from the respective allocation site

a garbage collection. (3) Finally, a `java.lang.Class` object contains all static fields of the class it describes as non-static fields. Thus, the size of objects of this class can differ, so the size must be specified explicitly.

Fortunately, the VM can easily determine whether one of these cases applies and can add the missing information to the event. After reading the first event block, the post-processing tool must determine whether

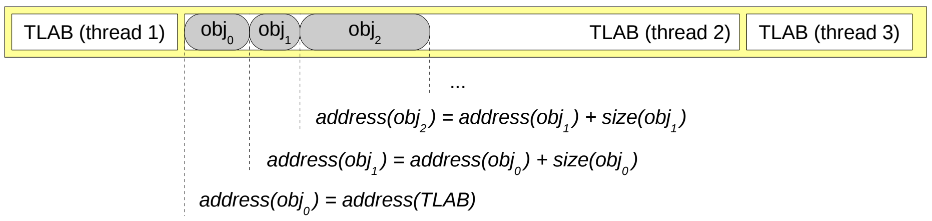

TLAB (thread 1)
obj0
obj1
obj2
TLAB (thread 2)
TLAB (thread 3)
...
address(obj2) = address(obj1) + size(obj1)
address(obj1) = address(obj0) + size(obj0)
address(obj0) = address(TLAB)

Figure 33: Example for incrementally reconstructing an objects address based on all previously allocated objects

$$address(obj_n) = \begin{cases} address(TLAB) & \text{if } n = 0 \\ address(obj_{n-1}) + size(obj_{n-1}) & \text{else} \end{cases}$$

Figure 34: Recursive definition for reconstructing object addresses

one of these cases applies by fetching the respective symbols information, and parsing the rest of the event appropriately.

There is also one case in which the object address cannot be reconstructed. If objects are too large for a TLAB or there is no more space for TLAB but just for some individual objects, the VM may refrain from allocating into a TLAB. In this case, the object address cannot be reconstructed because the object may be allocated at any free location. This is different from the above mentioned cases, as whether this has happened cannot be determined by examining the allocation site. Consequently, a dedicated event type is used for these cases (normal events and slow events) to indicate that the address is stored explicitly in the event

Formats. There are a total of 10 different allocation event formats, as shown in Figure 35. Allocation event formats can be categorized using two dimensions: (1) slow, normal, or fast, and (2) IR, C1, or C2. The first dimension tells how the object was allocated, i.e., slowly by the VM using an VM-internal allocation method (i.e., slow path), normal (directly into the heap, not using a TLAB), or fast (into a TLAB). The second dimension indicates what kind of code allocated the object, i.e., the VM itself (none), interpreted code (IR), code compiled with the client compiler (C1), or code compiled by the server compiler (C2).

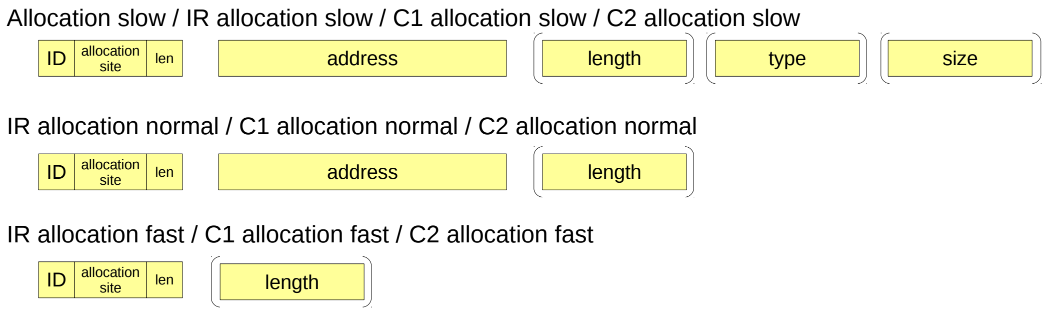

Allocation slow / IR allocation slow / C1 allocation slow / C2 allocation slow
ID
allocation site
len
address
length
type
size
IR allocation normal / C1 allocation normal / C2 allocation normal
ID
allocation site
len
address
length
IR allocation fast / C1 allocation fast / C2 allocation fast
ID
allocation site
len
length

Figure 35: Event formats for allocation events

All allocation event formats carry the allocation site and the length, if the allocation was an array. If the length is smaller than 255, it will be put into the `len` field in the first event block. If the array length is equal or bigger than 255, the `len` field will be set to 255, and a 32-bit block containing the length will be appended. This is because we have observed that most arrays are actually very small, allowing for this valuable optimization (cf. Section 4.1).

Slow and Normal Allocation Event. Slow and normal allocation events use a very similar format. Both carry the allocation site and the length (if possible) in their first event block, the absolute address of the object in the second block, and the array length (if the allocated object is an array and the length does not fit into the first block).

To keep the events as small as possible, we omit all information that we can reconstruct offline. Consequently, we drop the type and the size in the normal events, and try to drop them in slow events.

Fast Allocation Event. Fast events do not include the type and the size because they can be reconstructed offline. Since the VM uses fast events only for TLAB allocations, we also drop the address and reconstruct it offline. The length block is only necessary for large arrays.

4.2.4 Move Events

Whenever an object is moved by the garbage collector, the VM must trigger a move event. Move events are performance-critical because they occur frequently during garbage collections. Figure 36 shows the formats of all move event variants.

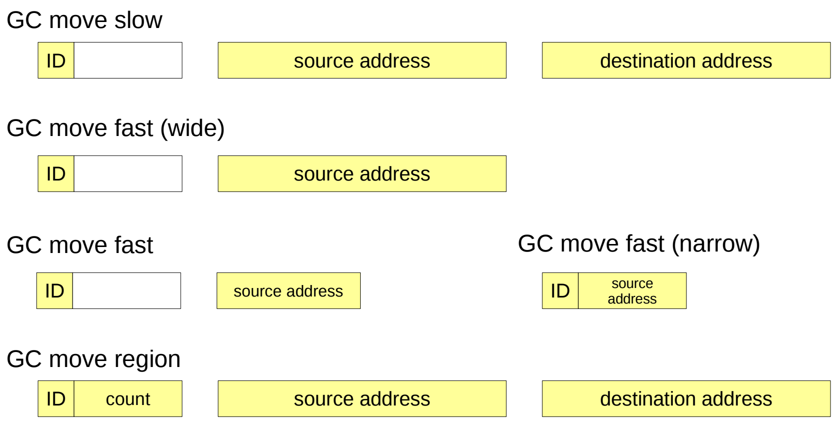


Figure 36: Event formats for move events

The *move slow* event is used to represent the generic case, i.e., a move of a single object from a source address to a destination address, consuming 20 bytes (4 bytes for the event block, and twice 8 bytes for the source address and the destination address). Using this event, the VM can represent arbitrary moves.

The *move fast* event family omits the destination address similarly to the allocation fast events. The VM usually uses PLABs for every GC thread in the same way Java threads use TLABs for allocations. Consequently, we can apply the same algorithm to reconstruct the destination address. Furthermore, as addresses make up one third of the event size, we try to fit the address into 4 bytes or 3 bytes if possible. In these cases, the address is not stored absolutely, but relative to the current base address that the GC start and GC end events defined.

Finally, the *move region* event represents moving up to 16777215 $((1 << (3 * 8)) - 1)$ neighboring objects by the same offset at once. The source address and the destination address represent the movement information for the first object. All subsequent objects are moved by the same offset. This event exploits the fact that objects survive and die in clusters.

4.2.5 Deallocation Events

Since the garbage collector triggers a move event for every object that is alive, we do not need explicit deallocation events. Instead, we can infer that an object is dead when we do not see a move event for it in a GC run.

4.3 Symbols

As described in Section 4.2, events contain only as little data as possible to keep them compact and efficient. Rather than storing symbol information explicitly in events, events contain IDs referencing entries in a dedicated symbols file that is generated on the fly. Consequently, whenever a symbol, e.g., a type or an allocation site, is referenced, the VM checks whether an ID has already been assigned to that symbol. If the symbol does not have an ID yet, a new ID is assigned to it and used in all subsequent events. Thus, only the first use of a symbol introduces additional overhead (because a corresponding entry must be written to the symbols file) and all subsequent uses are fast because the ID will be reused.

A symbols file starts with some general information about the execution environment, such as the alignment size, followed by an arbitrary number of entries for types and allocation sites.

4.3.1 Types

One of two major entries in the symbols file are types. A type entry contains the ID assigned to that type, the full name of that type, and the size of instances of that type. If the type is an array type (recognizable by its name) the size is replaced by the element size and the header size. Using these sizes in combination with the array length from the allocation event, the actual size of the object can be calculated.

The size of a type must be stored explicitly because it cannot be reconstructed offline. The size is influenced by the header size, the size of fields and elements, as well as their alignment, which is all implementation dependent. Consequently, the parser must not make any assumptions about that.

4.3.2 Allocation Sites

Similar to type entries, the symbol file also stores allocation site entries. Such an entry includes the ID assigned to that allocation site, the location (method, declaring class of the method, and BCI) as well as the *usually* allocated type at that location (as a reference to a type entry).

In principle, an allocation site is a single location in the code. However, the VM might associate an allocation site also with a (partial) call chain if it can be inferred at run-time. Thus, an allocation site entry is not just a single location, but rather a small stack trace, in which the topmost element is the location of the object allocation, and the other elements denote the current call chain at the time of the allocation. Therefore, a single allocating code location might occur in multiple allocation sites, with different call chains.

4.4 Efficient Trace Generation

Generating the trace efficiently is paramount for overall performance. Section 4.4.1 describes a buffering mechanism suitable for object tracing. Section 4.4.2 shows how to record the events described in Section 4.2 as efficient as possible.

4.4.1 Event Buffering

Writing every event to the trace file individually would result in millions of IO operations with very small data packages. Furthermore, the code would need to make a call to a native `write` method every time. This method would have to switch to the OS (or at least to the kernel) in order to complete its operation. Doing this whenever a new object is allocated or when an object is moved is unacceptable in terms of performance. Even making a call to an empty method (without switching to the OS) at every of the above mentioned points in the code would increase the run time by several orders of magnitude.

Global Buffer

A simple and well-known solution to the problem of having many small IO operations is buffering. We could use a global buffer and a static field pointing to the next free position in that buffer to store recorded events. If the buffer limit is reached, the buffer can be emptied into the trace file.

Writing a 32-bit event would result in the pseudo code shown in Figure 37. First, we have to lock the buffer to avoid concurrent writes to the same position. We can then check whether the buffer is full and flush it if necessary. Finally, we can write the event to the buffer, increment the top pointer, and unlock the buffer again.

```
lock buffer
if(buffer->top == buffer->end) {
  write(buffer);
  buffer->top = buffer->bottom;
}
*(buffer->top++) = event;
unlock buffer
```

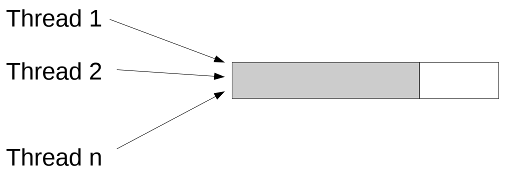


Figure 37: C-like pseudo code for storing a recorded event into a global buffer

Thread-local Buffers

Unfortunately, acquiring a global lock for every recorded event would defeat the purpose of lock-free allocations and lock-free moves (i.e., TLABs and PLABs respectively). Also, similarly to making calls at every allocation, it would increase the run time by several orders of magnitude. To counteract the locking problem, we can use thread-local buffers. Every thread has its own buffer which no other thread can write to. Consequently, the thread can write to its buffer without any locking. Locking is only necessary when flushing a thread-local buffer to the trace file to avoid concurrent writes.

Writing a 32-bit event to a thread-local buffer would result in the pseudo code shown in Figure 38. Like in Figure 37, we need to check whether the buffer is full and flush it if necessary. However, the lock instructions are only executed when the buffer is full. Consequently, we only need to acquire the lock rarely instead of at every event.

```
Thread* thread = Thread::current();
if(thread->buffer->top ==
    thread->buffer->end) {
  lock file
  write(thread->buffer);
  unlock file
  thread->buffer->top =
    thread->buffer->bottom;
}
*(thread->buffer.top++) = event;
```

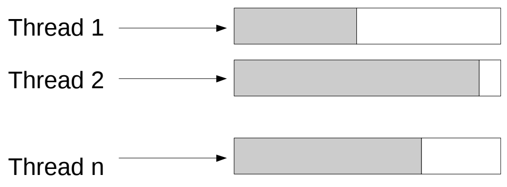


Figure 38: C-like pseudo code for storing a recorded event into a thread-local buffer

Buffering threads in a thread-local manner destroys the global temporal ordering of events, i.e., we cannot easily infer the temporal order of events that occurred in different threads based on their location in the trace file. This problem will be addresses in more detail in Section 4.5.

Thread-local Front & Back Buffers

Although thread-local buffers reduce the locking and IO overhead significantly, they introduce long and unexpected pauses. For example, in most cases, recording an allocation will be a fast operation because we just need to check whether the buffer is full (which is unlikely) and then store the event. However, if the buffer is full, flushing introduces an unwanted and long IO pause.

To reduce this pause, we introduced front and back buffers. This means, that every thread maintains two instead of just one thread-local buffer. Every thread always writes to its front buffer, keeping the back buffer in reserve. If the front buffer is full, it is swapped with the back buffer. The thread can continue submitting events to the new front buffer, while a dedicated VM-internal writer thread flushes the new back buffer into a file. The only case in which the recording thread is stalling now is when the front buffer is full, and the back buffer has not been completely flushed yet. Figure 39 shows C-like pseudo code implementing this buffer mechanism.

```
Thread* thread = Thread::current();
if(thread->front_buffer->top == thread->front_buffer->end) {
  lock writer_thread
  while(thread->back_buffer->top == thread->back_buffer->end) wait writer_thread
  writer_thread->submit(thread->front_buffer);
  unlock writer_thread
  swap(&thread->front_buffer, &thread->back_buffer);
}
*(thread->front_buffer->top++) = event;
```

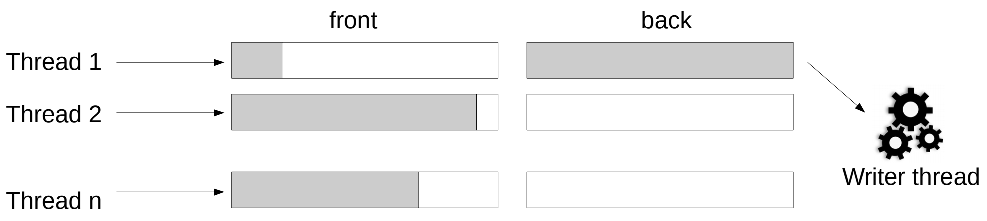


Figure 39: C-like pseudo code for storing a recorded event into a thread-local front buffer

Thread-local Buffers with Asynchronous Buffer Management

Even though the use of front and back buffers as well as a dedicated writer thread brought significant improvements, there were still a lot of cases where a thread had to wait. This was mainly due to the fact, that a thread can allocate objects (and consequently fill the buffer) much faster than any underlying medium can store the trace file. Furthermore, the overall memory used for the buffers was spread unfairly because every thread had the same amount of buffer memory at its disposal. However, some threads are allocating a bulk of the application's objects whereas others are rarely allocating anything. Also, application threads are completely inactive during a GC but keep hogging buffer memory although the GC threads could use it better during a GC. The same holds for GC threads while the mutator is active and the GC is not.

Therefore, we devised a more adaptive buffer management that can assign buffers to threads in need. This buffer management technique keeps a pool of unused buffers. When a thread tries to record an event, it can request a new buffer from that pool. When a thread has filled its buffer, it submits the buffer to the flush queue and replaces it immediately with a new buffer from the buffer pool. The flush queue is processed by a dedicated writer thread, which consumes buffers from the queue, writes them to the trace file, and returns the emptied buffers to the buffer pool. Figure 40 shows C-like pseudo code implementing this buffer mechanism.

```
Thread* thread = Thread::current();
if(thread->front_buffer->top == thread->front_buffer->end) {
  flush_queue->submit(thread->buffer);
  thread->buffer = buffer_pool->get();
}
*(thread->front_buffer->top++) = event;
```

Figure 40: C-like pseudo code for storing a recorded event into a thread-local buffer using asynchronous buffer management

Using this mechanism, the recording thread never has to stall, the only locks it needs to acquire are those for the flush queue and the buffer pool (implemented in the `submit()` and `get()`). Since these locks are usually uncontended, it can almost immediately proceed with its work. A detailed evaluation can be found in Sections 8.1.4 and 8.1.5.

Buffer Size Randomization

Using the thread-local buffers with asynchronous management, the only bottleneck is putting a buffer into the queue and taking a new one from the pool. Although these operations are very short, they may cause stalls if many threads try to submit their buffer simultaneously. However, when multiple threads are doing the same work (e.g., threads in a thread pool working on a big task split into similar small tasks), and thus execute the same code, their allocation frequency will also be similar. Consequently, every thread will submit its buffer approximately at the same time, resulting in unwanted contention at the lock guarding the queue.

To counteract this phenomenon, we randomized the buffer sizes. Thus, even if two threads are doing the exact same work, their buffers will get full at different points in time, reducing the probability for lock contention. A detailed evaluation can be found in Section 8.1.6.

4.4.2 Event Firing

Whenever an object is allocated or moved in the heap, an event must be fired. Firing events efficiently, i.e., writing them to the thread-local buffer, is paramount because this task has a direct performance impact. This section describes how the AntTracks VM fires events in order to achieve minimal run-time overhead. A detailed evaluation can be found in Section 8.1.1.

Allocations

While object movements are done by the GC itself, object allocations are mainly triggered by user-defined code. Figure 41 shows the transitions from source code to executable machine code as well as instrumentation points for firing allocation events. There are many languages that can be compiled to Java bytecode. Two prominent ones, i.e., Java and Scala, are shown in this example. This ahead-of-time compilation occurs before the code is loaded into the VM. Consequently, this is not a suitable point of instrumentation for AntTracks, because instrumenting in this step would have to be language-dependent and could not exploit VM internals. The resulting bytecode from the ahead-of-time compilation will be interpreted by the VM. The AntTracks VM must modify the interpreter to fire allocation events at every bytecode that allocates an object, i.e., `new`, `newarray`, `anewarray`, and `multianewarray`. However, since hot code will eventually be compiled to executable machine code, firing events efficiently in the interpreter is not as critical as in compiled code. Finally, when the bytecode is compiled by any JIT compiler, the code for firing an allocation event is added to the generated machine code. As the majority of objects will be allocated by compiled code (cf. Section 8), we will focus on optimizing this case.

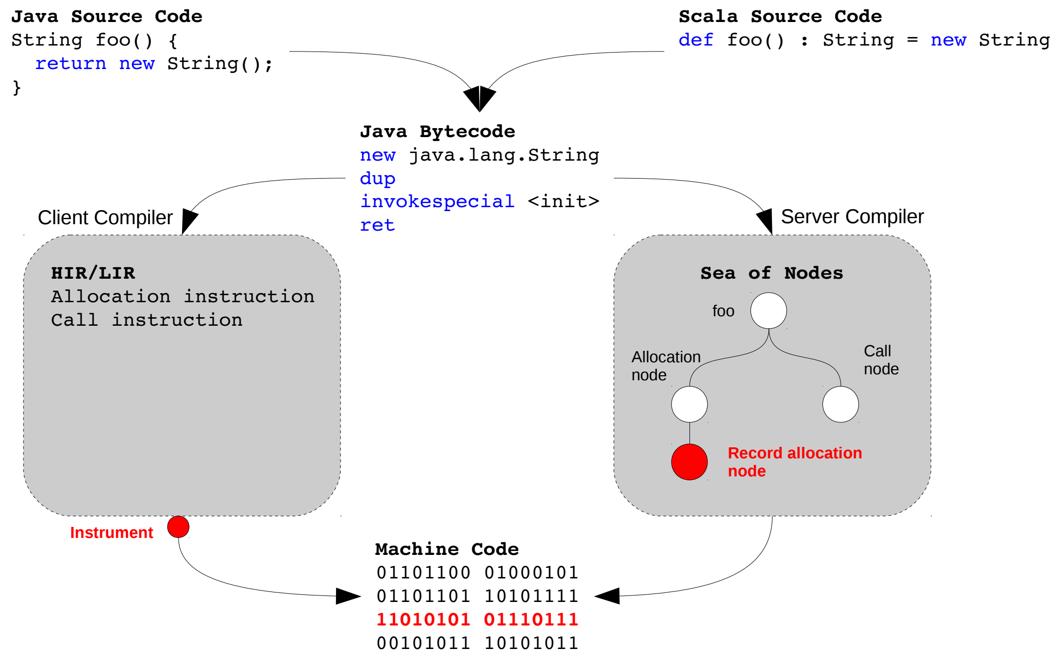

Java Source Code
```
String foo() {
  return new String();
}
```
Scala Source Code
```
def foo() : String = new String
```
Java Bytecode
```
new java.lang.String
dup
invokespecial <init>
ret
```
HIR/LIR
```
Allocation instruction
Call instruction
```
Machine Code
```
01101100 01000101
01101101 10101111
11010101 01110111
00101011 10101011
```

Figure 41: AntTracks instrumentations in the lifecycle of Java code

Ideally, firing an allocation event (fast version) is as easy as (1) checking whether there is still enough space in the thread-local buffer (`*top < *end`), (2) moving a 32-bit constant to a memory location (`*top = 0xABCDEF00`), and (3) incrementing a pointer (`top++`), as shown in Figure 42. This is due to the fact that the event encoding can be computed at JIT-time and can then be inlined as a constant into the code. Only for arrays, the array length must be combined with the event using a bit-wise `or` operation. If the thread-local buffer is full or the array length is too big to fit into the 32-bit fast allocation event, a fallback to a C function is executed. This function is capable of building more complex events and of fetching a new thread-local buffer if necessary. In the following sections we will call the generated code to fire the fast version of an allocation event the *fast path* and the fallback to the more generic function the *slow path*.

```
object obj = ...;
Thread* self = Thread::current_thread();
int** top = &self->buffer.top;
int** end = &self->buffer.end;
if(*top < *end) {
  *(*top++) = 0xABCDEF00;
} else {
  obj = Runtime::fire_allocation(obj, 0xCDEF);
}
```

```
array obj = ...;
Thread* self = Thread::current_thread();
int** top = &self->buffer.top;
int** end = &self->buffer.end;
if(*top < *end && obj->length <= 0xFF) {
  *(*top++) = 0xABCDEF00 | obj->length;
} else {
  obj = Runtime::fire_allocation(obj, 0xCDEF);
}
```

Figure 42: C-like pseudo code for firing an event for an instance allocation (left) and for an array allocation (right) (event = `0xABCDEF00`)

If an allocating method is inlined into another method, the allocation site stored into the symbols file comprises two stack frames. To do this, we redefine the term *allocation site* from a single code location allocating an object to an allocating code location, including an arbitrary number of callers. This way, we get a small stack trace of the allocation, i.e., a *mini stack*, for free, because the compiler knows in this case that the caller is always known. If the allocating method were inlined into some other caller, a different allocation site ID would be assigned Consequently, the same allocating code location can occur in multiple allocation sites, differing only in their call chains. A detailed analyses of the mini stacks can be found in 8.2.3.

Section A describes in more detail how to fire allocation events in virtual-machine code, interpreted code, or JIT-compiled code respectively.

Stack Traces. Although the mini stacks provide some limited information about the callers, a periodic full stack trace is necessary to infer the global context of allocations. But as taking a stack trace is a slow operation because the complete method stack must be walked, this cannot be done at every allocation. Also the code for walking the stack would be too complicated to be generated. Therefore, after every n allocations, a slow path is forced even if it is unnecessary. This slow path can then walk the stack and generate a new allocation site ID for it if that call chain is seen for the first time.

Usually when a stack trace is taken (for example via JVMTI), all application threads are suspended and forced into a safepoint. This is necessary because of two reasons: (1) a thread must not manipulate its method stack while it is walked, otherwise the stack might be unresolvable (e.g., if the thread is in the process of building or destroying a stack frame) and (2) we must know the exact stack layout of the code location the thread stopped at. Compiled code might push temporary values onto the stack or spill certain registers. Thus, the frame layout is highly dependent on the code that is currently being executed. As this information would be way too much to be stored for every instruction, it is only stored for safepoints. These safepoints are placed by the JIT compilers at their discretion. Only calls and allocations must always be safepoints. Calls must be safepoints so that the VM can walk the stack and resolve all frames properly and allocations must be safepoints because they might need to trigger a GC.

In the case of allocations, safepoints are implemented via a VM call. If the heap is full, the allocation's fast-path code will not succeed and make a slow-path call into the VM. Every call into the VM is a safepoint and checks whether a GC must be triggered.

When we want to walk the stack at an allocation site, we do not need to force every thread into a safepoint, but only the thread whose stack we want to walk. By forcing the event-firing code into a slow-path call, we automatically generate a safepoint at that location and we can walk the stack safely. Whenever a VM method is called, the VM remembers the location of the topmost java frame. From this location on, we can use the information stored for the safepoints to walk and decode the stack.

However, a safepoint also means that a GC might be triggered. Thus we need to make sure that we fire all events before that check is made. Otherwise, a GC might be triggered (recording other events) before the allocation event is fired, destroying the temporal ordering of events. However, because the check whether to trigger a GC is made on return of every VM call, the GC will only be triggered after we have done our work.

Another problem is that the allocation itself now contains an additional safepoint. If a GC is triggered, the object is already allocated, its header initialized but the content might not yet be initialized (i.e., it might contain invalid pointers that would make the GC crash). Therefore we must make sure that the object is completely cleared before the GC is invoked. This way the GC would only detect null pointers which is semantically correct at this point in time because the constructor, which could assign fields, has not been called yet.

Moves

Besides allocations, object move events are the second event category that occurs frequently and, consequently, must be recorded as efficiently as possible. As the HotSpot^{TM} GCs are implemented in `C++` and there is no generated code involved, we cannot apply the same techniques as for allocation events. Instead we call `C++` functions to record object move events at the appropriate places in the GC. Therefore, firing move events cannot be optimized beside using the most efficient (i.e., short) event format (cf. Section 4.2) and language-level optimizations (e.g., inline functions and macros to boost inlining).

ParallelOld GC. The ParallelOld GC uses the *Scavenge* algorithm for minor collections in the young generation, and the *Mark and Compact* algorithm for major collections. Both the minor and the major GC represent distinct GC phases in the trace file. Therefore, they both start by flushing all thread-local buffers and firing a GC start event. They end by flushing all buffers again and firing a GC end event.

Scavenge. The scavenge algorithm used for minor collections will copy all live objects from the Eden and the survivor from space to the survivor to space or to the old generation. Because the GC start event indicates a minor GC, the VM follows that event immediately with a closer specification about which spaces will be collected (in this case only Eden and survivor from) using the GC info event. Therefore, all objects in the survivor to and the old generation are expected to survive and stay in place. All objects in the Eden space and the survivor to space, on the other hand, are expected to be dead unless a move event is for them is sent.

The algorithm traverses the object graph and copies objects as it goes. Thus we fire a *GC move slow* event for every such move. If the object is moved into a PLAB, we fire a *GC move fast wide*, *GC move fast*, or *GC move fast narrow* instead depending on the address size. Whenever a new PLAB is allocated, we fire a *PLAB alloc* event.

Figure 43 shows the recording of the events during a Scavenge minor collection. At first a *GC start* is fired, followed immediately by *GC info* events specifying what spaces are going to be collected. In the case of the ParallelOld minor GC, this will always be the Eden space and exactly one of the survivor spaces. Subsequently, all pointers into the Eden and survivor space are followed (both root pointers and pointers from not collected spaces), detecting that object 4 is reachable. A new PLAB is allocated (recorded by a *PLAB alloc* event) and the object 4 is moved into it (recorded by a *GC move fast*, which omits the destination address because it can be reconstructed offline). Following the pointers of object 4, object 5 is detected as reachable and moved to the same PLAB because there is still enough space for it (recorded by a *GC move fast* event again). The pointer originating from object 2 defines object 3 as reachable. As there is no more space in the PLAB, a new PLAB is allocated and object 3 moved into it (recorded by a *GC move fast* event again). Object 3 points to object 8, but assuming that object 8 is old enough to be promoted to the old generation, a new PLAB is allocated in the old generation (recorded by a *PLAB alloc* event). In this situation, the GC thread holds two active PLABs, one for the survivor space and one for the old generation. Object 8 points to object 9, but assuming that object 9 is too big to fit into the rest of the currently active PLAB, it cannot be moved to it. Because there is also not enough space for a new PLAB, the object is moved without using a PLAB, recorded by a *GC move slow* event. To keep the heap unfragmented, the old PLAB is retired and filled with an integer array (recorded by an *allocation slow* event, denoted by an *f* for filler). Finally, as all live objects have been handled, all collected spaces are declared empty by the GC, all remaining PLABs are retired by filling them (recorded by *allocation slow* events again) and the *GC end* event is sent.

What variant of the *GC move fast* event will be used depends on the distance of the address to the base address that is set by the GC start event. For the scavenge algorithm, it is set to Eden start because most objects will be moved from near that location. A detailed evaluation of that claim can be found in Section 8.2.1.

After copying an object, a forwarding pointer to the new object is installed into the header of the old object in order to indicate that this object has already been copied. However, as multiple GC threads work in parallel, and the forwarding pointer can only be installed after the object has been copied, an object can be copied twice by two different threads. Nevertheless, only one will win the race of atomically installing the forwarding pointer. The thread that lost that the copy will be overwritten with an `int[]` that will be reclaimed at the next collection because no reference is pointing to it. Copying objects by accident is the reason why we need to wait and send the move event only for the thread that actually wins the race. If the other threads are able to the accidentally copied object, they do not fire a move event. If the copy is overwritten with an `int[]` because it cannot be unallocated, an allocation event is fired for that integer array.

A minor collection may fail if the survivor to space or the old generation is full. Since the scavenge algorithm has no marking phase, this situation cannot be detected in advance but rather occurs in the middle of a collection. As some objects (not all) have already been copied and some pointers (not all) have already been adjusted, these operations are hard to rollback. Instead, the scavenge algorithm finishes its work by ensuring that for all objects of which there are two variants now, all pointers either point to the new or to the old object. The GC resolves this issue by immediately following with a major GC, which cannot fail and which will collect duplicates. Thus, we just fire all move events of all copied objects, and indicate in the GC end event that the collection has failed. The post-processing will then handle this case correctly.

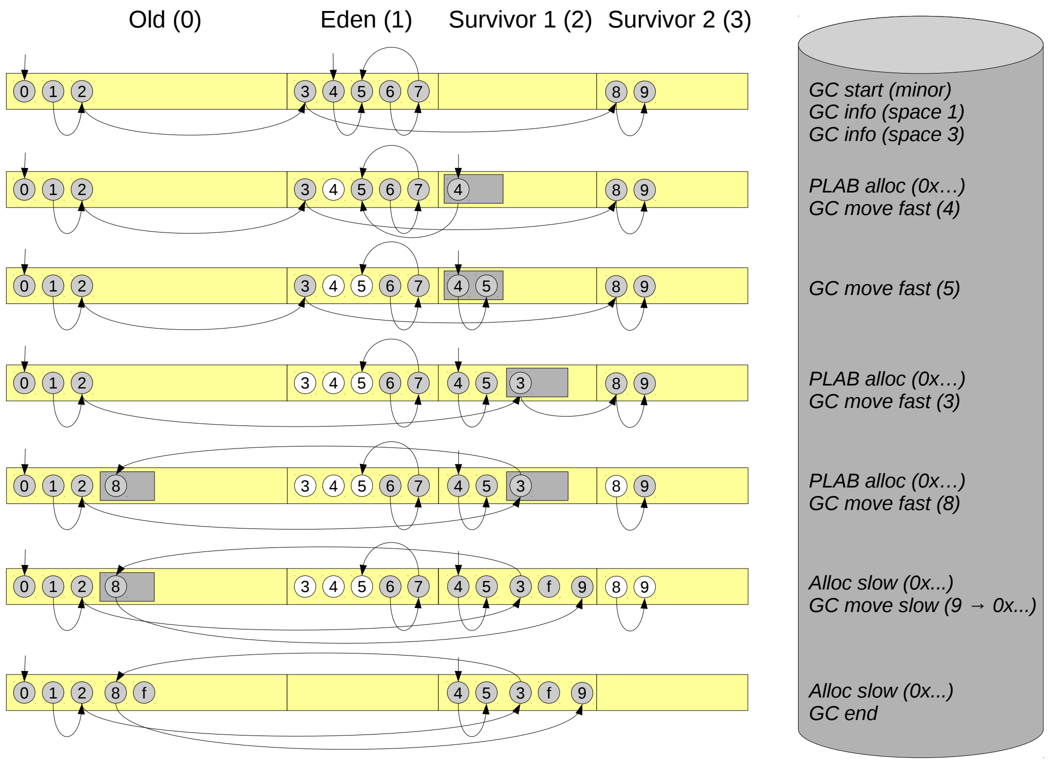


Figure 43: Creating move events for the ParallelOld minor GC

Mark and Compact. The mark and compact algorithm that is used for major collections will first mark all live objects by traversing the object graph and will then copy all marked objects towards the beginning of the heap, overwriting dead objects in the process. This algorithm is split into three phases: (1) marking all objects, (2) computing the new location for every object, and (3) copying all marked objects to their new locations.

The copy phase is the obvious phase to fire the appropriate move events. Basically, we fire a *GC move slow* event for every object to be moved. If multiple neighboring objects are moved by the same offset, we cluster those moves into a single *GC move region* event.

Figure 44 shows when move events are generated during a Mark and Compact major collection. A *GC start* event is fired before the first phase of the mark and compact algorithm. In the first phase, all live objects are marked starting with the roots. During that phase, no events are fired. In the next phase, the new locations of all marked objects are computed and the pointers are adjusted. This phase has been omitted in the figure since it does not generate any events. In the last phase, all live objects are moved towards the beginning of the heap. During that phase, at first a *GC keep alive* event is fired for object 0, because it is alive but does not move. Objects 4 and 5 are clustered into a *GC move region* event, that contains only the first move of object 4, and the number 2 because two objects are moved in this cluster. Finally, a *GC move slow* event is fired for object 7, followed by a *GC end* event.

Since multiple GC threads are performing this copy phase in parallel, we must be careful to not fire multiple move events for the same object because these threads do not copy object-wise but region-wise.

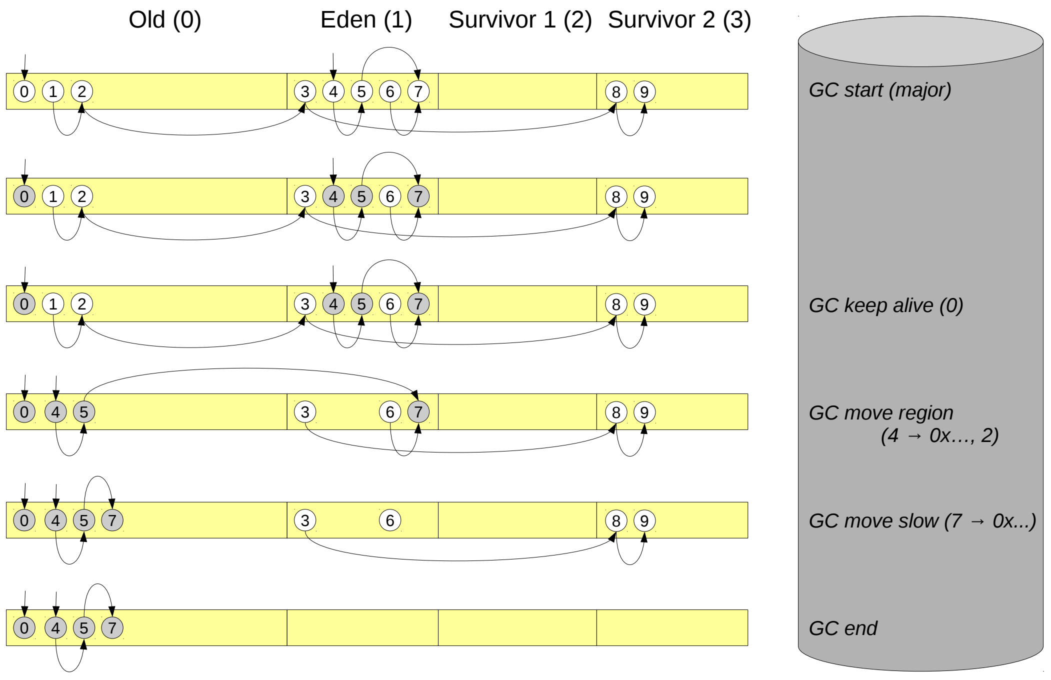


Figure 44: Creating move events for the ParallelOld major GC

In other words, every thread copies a specific region, and the region boundaries do not necessarily have to be object boundaries as well. Therefore, one thread might copy the beginning of an object, while another thread might copy the end of that object. If the object is a big array, there might even be an arbitrary number of threads involved in copying. To overcome this, we fire a move event only if the first byte of an object is copied. All threads copying some other part of the object will not fire a move event for that object.

G1 GC. Like the Parallel Old GC, the G1 GC uses the *Scavenge* algorithm for minor collections and the *Mark and Copy* algorithm for major collections. However, it does not use a fixed number of contiguous spaces but rather an extendable set of small fixed-sized regions. Every region is either empty or assigned to the abstract concept of an Eden space, a survivor space, or the old generation. Consequently, using the G1, the VM has no fixed number of spaces, but the *space* events will be used whenever a new space is added, a space is removed, or a space is reassigned to another purpose.

As the default region size is quite small (2 MB), it can easily happen that big arrays exceed that size. To overcome this issue, the G1 introduces the notion of *humongous objects*. Humongous objects are all objects that span more than one region. These regions will then be linked together and can only be collected as a unit.

Scavenge. A minor GC using the scavenge algorithm can collect both young generation regions and old generation regions in a single collection. It selects empty regions and declares them to be either survivor or old generation regions (depending on whether the source region is an Eden/survivor region or an old generation region).

The *GC info* event will specify what regions will be collected. Similar to the scavenge algorithm in the ParallelOld GC, *GC move slow* events will be sent for every move, but we try to replace them with *GC move fast wide*, *GC move fast*, and *GC move fast narrow* events whenever the object is moved into a PLAB.

Figure 45 shows how move events are recorded during a G1 minor collection. At first a *GC start* event is fired, followed by *GC info* events specifying the spaces that will be collected (in this case space 0 and space 3). Then, object 0 is found, which currently resides in an old space. Therefore, it must be evacuated into an old space again. A new space is selected from the free spaces (space 4) and is made an old space (recorded by a *Space alloc* event). The GC then allocates a PLAB and moves object 0 to it. (recorded by a *PLAB alloc* event and an *GC move fast* event because the destination address can be reconstructed). Following the pointers in object 0, object 3 is found, which is again in an old space. It is also copied and recorded via a *GC move fast* event. Object 3 points to object 4, which is in a survivor space. Thus it cannot be evacuated into the same space. A new free space is selected (space 5) and is made a survivor space. A PLAB is allocated in it and object 4 is moved (again recorded via a *PLAB alloc* and a *GC move fast* event). Similar to the ParallelOld GC, a single GC thread can hold one per PLAB per space type (i.e., one for Eden, survivor, and old each). Finally, all remaining PLABs are retired by filling them up (to keep the heap unfragmented) (recorded by *Alloc slow* events), the two collected spaces are added to the free list (recorded by *Space dealloc* events), and a *GC end* event is fired.

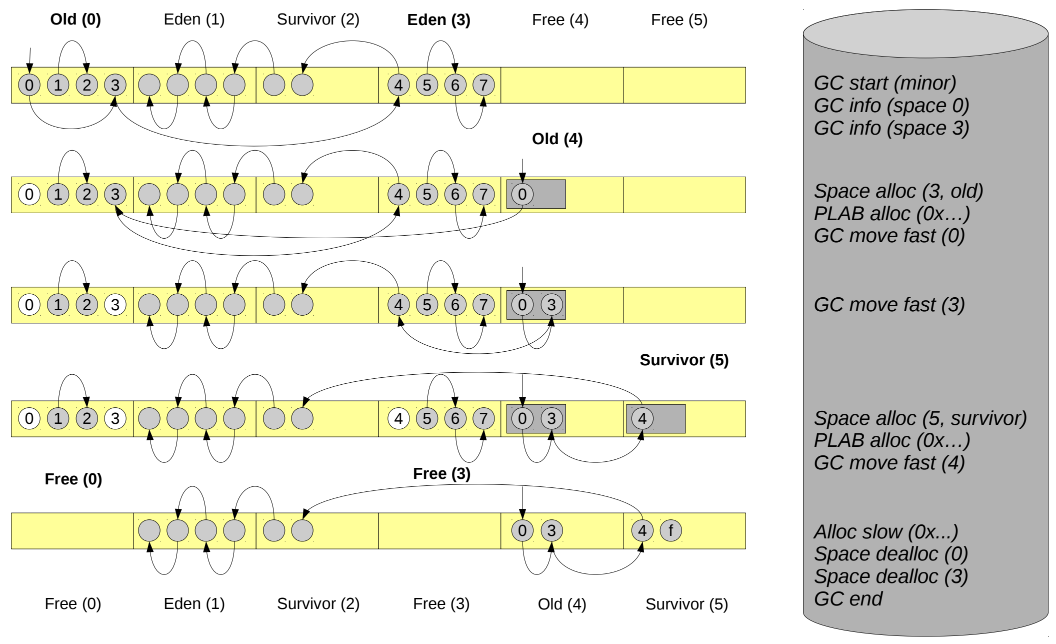


Figure 45: Creating move events for the G1 minor GC

Mark and Compact. The mark and compact algorithm used for major collections in G1 represents an emergency collection only as the minor collection should be able to handle most cases. If a major collection must be executed, all objects are compacted towards the beginning of the heap, possibly crossing generations in the process.

Figure 46 shows move events are recorded during a G1 major collection. At first a GC start event is fired. After that, all live objects are marked and their new locations are computed. No events are fired during these phases. Then, all marked objects are moved towards the start of the heap. Since all objects between object 3 and object 9 (inclusive) are marked, their moves can be clustered into a single *GC move region* event. Please note that a *GC move region* event can span multiple adjacent spaces like in this example. Object a is moved individually and is therefore recorded by a *GC move slow* event. After moving all live objects, the emptied spaces are added to the free list (in this case space 3) and all other spaces are made old spaces (in this case space 1 and 2). Finally, a *GC end* event is fired.

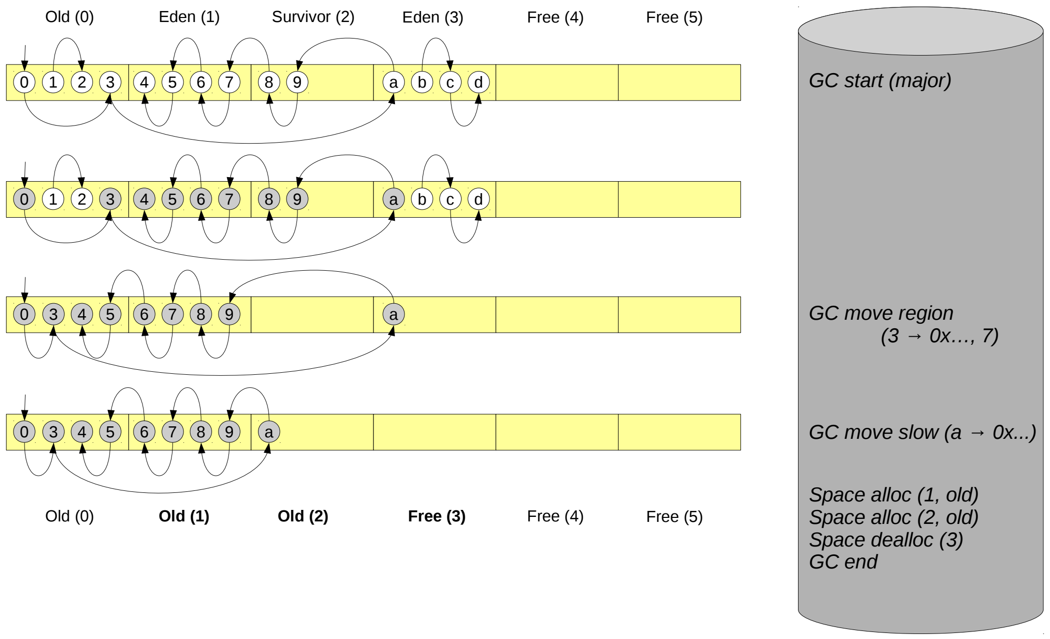


Figure 46: Creating move events for the G1 major GC

Concurrent Mark and Sweep GC. Like the Parallel Old GC, the Concurrent Mark and Sweep GC uses the *Scavenge* algorithm for minor collections in the young generation, but a concurrent version of the *Mark and Sweep* algorithm for the old generation. This means that a collection in the old generation may be interrupted by a minor collection in the young generation.

Figure 47 shows how move events are recorded during a concurrent mark and sweep collection in the old generation. In this example, the collection is interrupted by a minor collection in the young generation during its sweeping phase. At first, the VM is suspended to mark all roots, recorded via a *GC start* and a *GC end* event. Since we do not send a GC info event, the post-processing tool does not expect any space to be collected. We could omit these events, but we want to record the duration of the individual marking phases. We handle the concurrent marking and the final stop-the-world marking phase in the same manner. Finally, after the final marking, we start the concurrent sweeping phase by firing a *GC start* event and a *GC info* event specifying that the old generation is collected. In this phase, the GC sweeps from left to right, firing *GC keep alive* events when finding a live object. Dead objects are added to the free list, which is not represented by any event. However, after firing events for objects 0 and 3 and after reclaiming objects 1 and 2, a minor GC in the young generation is triggered. This collection is a stop-the-world collection and thus interrupts the other collection that is currently sweeping the old generation. The young generation is collected similarly to the ParallelOld GC, i.e., by copying all live objects to a survivor space or the old generation. While the objects 8, 9, and b are moved to a survivor space, object e is promoted to the old generation. It uses space that has just been reclaimed in the currently suspended collection of the old generation. After the minor collection in the young generation has finished, the collection in the old generation is resumed, sweeping the rest of the space. Please note that in this example, the IDs carried in the *GC start*, *GC end*, and *GC info* events are necessary to correlate events to each another.

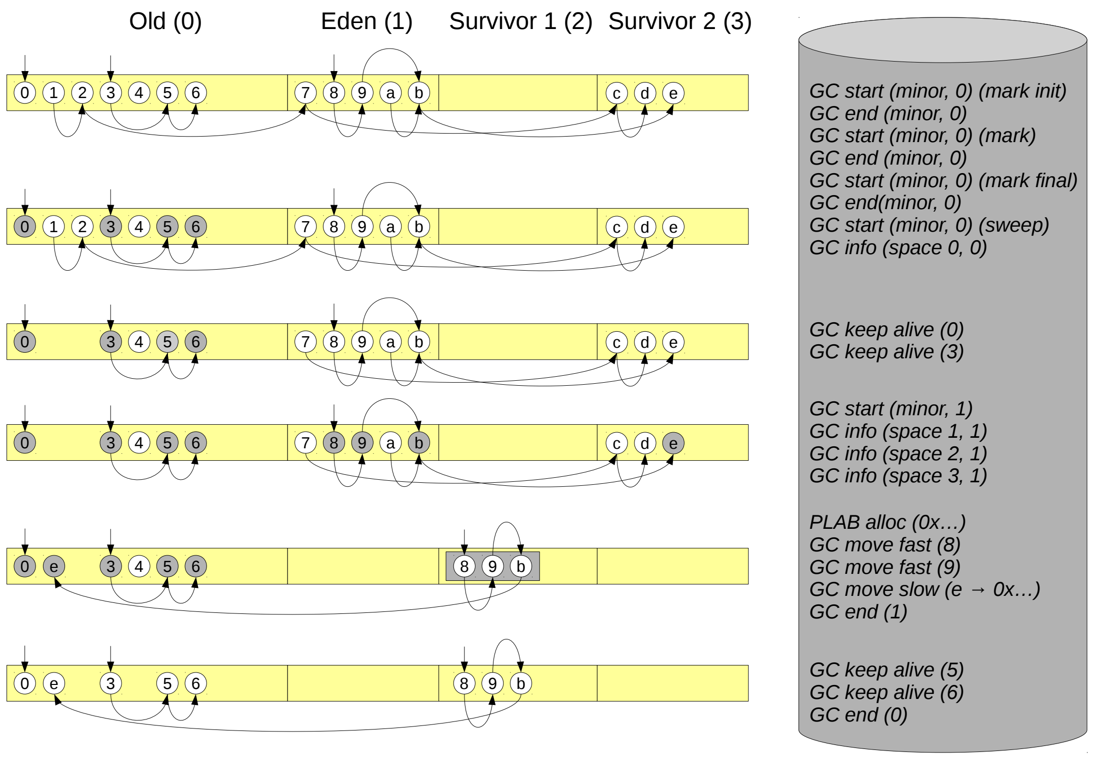


Figure 47: Creating move events for the Concurrent Mark and Sweep minor collection in the old generation

4.5 Temporal Ordering of Events

Thread-local event buffering destroys the overall temporal ordering among events. However, the events are in temporal order within one thread. Thus, if two events have been recorded by the same thread, we can determine which one came first. But if the events were recorded by different threads, we cannot determine their exact temporal ordering.

To get a global partial ordering, we introduced the possibility to make events synchronized. If such an event is in the trace, all events that are located before that event in the global trace have also been recorded before that event. The same must hold for all events after a synchronized event. Therefore, we can infer the temporal ordering of two events that were recorded by different threads if a synchronized event is between them, but not otherwise.

To guarantee that all events that are located before a synchronized event have also occurred before it, the buffers of all threads must be flushed before such an event is recorded, and the event must also be flushed immediately to the file. However, flushing buffers of other threads is a dangerous operation because the owning threads might be in the process of writing to it. Thus, such events may only be fired at safepoints, i.e., at points in time where all threads but one are suspended and no thread is in the process of writing to its buffer. Such events are only the *GC Start* event and *GC End* event. Consequently, within one mutator or GC phase, temporal ordering of events can only be established within one thread. Across multiple phases however, global temporal ordering of events can be inferred. We call this a *partial global temporal ordering* and we will show that no full temporal ordering is necessary in order to process and analyze the trace correctly in Section 5.

4.6 Trace Portability

We claim that the described trace format is portable in the sense that the post-processing tool does not need to make any assumption about the underlying VM implementation or the GC algorithm that generated the trace.

In theory, the trace format described in this thesis will work for every GC algorithm because the events do not necessarily have to represent actual operations in the VM. Thus, the VM can always introduce a stop-the-world pause, and fire all events so that they represent the changes made by the GC. Although this might not be efficient, it will always represent the correct heap state.

We implemented AntTracks for the *Scavenge*, *Mark and Compact*, and *Mark and Sweep* algorithm for both contiguous (ParallelOld GC, Concurrent Mark and Sweep GC) and non-contiguous generations (G1 GC). This shows that the trace can represent three different algorithms (Scavenge, Mark and Compact, Mark and Sweep) as well as it can represent four different approaches (stop-the-world vs semi-concurrent and contiguous generations vs non-contiguous generations).

4.7 Continuous Tracing

Many of todays Java applications are continuously running server applications. Even when the tracing overhead is low, tracing them continuously is not an option, because the size of the trace file will exceed the available storage capacity. We examined the growth rate of the trace file and found that some heavily-allocating benchmarks generate up to 250 MB/s. Thus we need to develop new approaches to deal with that amount of data. The following sections will introduce both compression and rotation as a solution to that problem (cf. Lengauer et al. [30]).

4.7.1 Compression

To reduce the overall size of the trace, we introduced *black-box compression.* This type of compression regards the trace file as a stream of bytes which can be compressed with any well-known compression algorithm and stands in contrast to white-box compression, where we know about the semantics and frequencies of event types and format them accordingly as described before. To better estimate the compression potential, we have run our traces through the zip algorithm at various compression levels shown in Figure 48. This experiment showed that event traces are highly compressible (down to 1-35% of their original size. However, these high compression rates come at a significant cost of up to 0.25 seconds per megabyte.

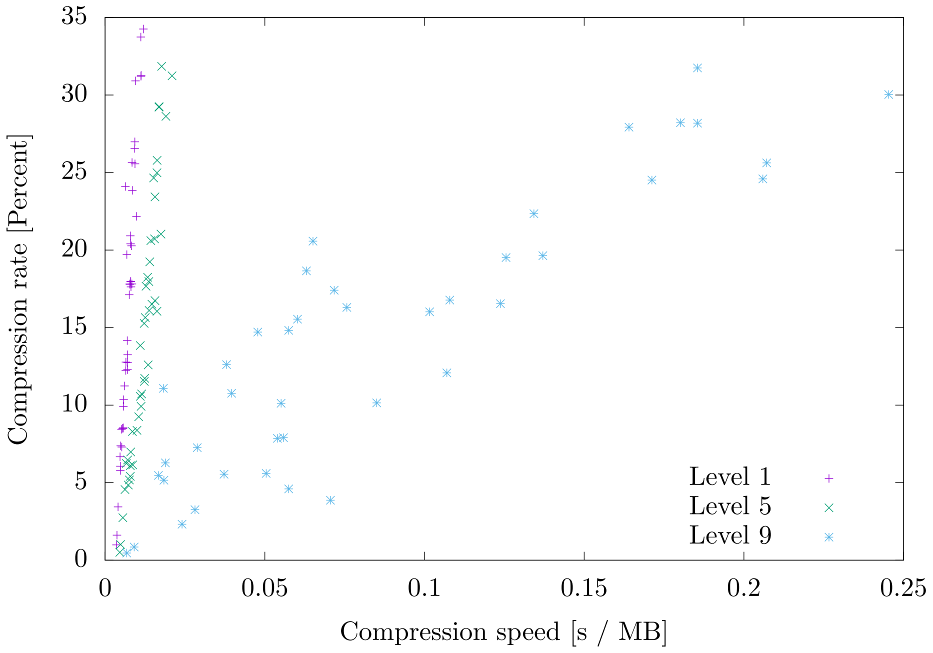


Figure 48: Compression rate vs. compression speed when zipping traces with compression level 1 (lowest), level 5 (medium), and level 9 (highest)

It is also interesting to see, that the traces seem to reach almost the same compression rate independent of the configured compression level. This might indicate that the already very dense event format cannot be compressed better at higher compression levels.

In order to relax the speed and capacity requirements of the underlying storage device, we introduce *on-the-fly compression*, i.e., AntTracks compresses the trace in the VM before it is written to disc. The goal of this approach is (1) to reduce size of big traces and (2) to reduce the run-time overhead if the time for compressing and writing the compressed trace is smaller than the time for writing the uncompressed trace.

In a first attempt, we applied the simple LZW compression algorithm in the writer thread. Although it significantly reduced the trace size, it increased the overhead by several orders of magnitude. The increased overhead was due to the fact that the writer thread was responsible for both compression and IO, although the time that was needed for compressing could have easily been used to write some other already compressed buffer. As a result, the flush queue overflowed and application threads were stalled. To overcome these issues, we introduced concurrent and parallel compression.

Concurrent Compression. Concurrent compression splits the flush queue into two distinct queues. Recording threads add buffers to the first queue. A dedicated compression thread consumes buffers from the first queue, compresses them, and adds them to the second queue. The writer thread then consumes the compressed buffers from the second queue and writes them to the trace file. By using separate queues and threads for compression and writing, the two tasks can be done in parallel and do not block each other.

Parallel Compression. Parallel compression uses several compression threads instead of just one. This extension allows multi-core machines to compress different parts of the trace in parallel. Although there is no global ordering of events or buffers but only a partial ordering within one thread and across phase change events (cf. Section 4.5), we have to take care not to destroy that partial ordering. For example, two compression threads might consume two buffers of the same application thread from the first flush queue. If the second thread finishes first and adds its compressed buffer to the second flush queue first, the partial temporal ordering is compromised and the post-processing cannot parse the trace properly.

Compression Heuristics. Compressing the entire trace might not be feasible due the big run-time overhead. So we looked for a heuristic to decide when to compress a buffer without imposing any overhead.

The AntTracks VM keeps track of how fast buffers can be compressed as well as how fast buffers can be written to disk. Using these speeds as well as information about the buffers in the queues, a compression thread can decide whether a buffer can be compressed while the writer thread is still busy. If so, the buffer is compressed, otherwise, it is passed to the writer queue without compression. In Section 8.1.8, we will show how this heuristic in combination with both concurrent and parallel compression will only introduce minimal overhead while still achieving a good compression rate.

4.7.2 Rotation

As compression only postpones the problem of exceeding the maximum storage capacity, the goal of *rotation* is to circumvent that problem entirely by only keeping the last part of the trace. This enabled the user to inspect the part of the trace before a certain problem occurred. Using rotation, the user can specify an absolute trace file limit, e.g., 16 GB, and a maximum deviation from that limit, e.g., 25%. With this configuration, the VM must guarantee that the trace does not exceed 16GB as well as that at least 12 GB (16 GB - 25%) are available at any point in time. How much absolute time this represents depends on the growth rate of the trace with the monitored application. We intentionally did not use absolute time for configuration because the limiting factor for rotation is the absolute size due to the capacity of the underlying storage medium.

To make overwriting old parts of the trace easier, we write the trace in multiple files instead of in one single file. When the limit is reached, the oldest file will be overwritten. Additionally, we store the index of the file in its header, so that the post-processing can restore the order of the files. Figure 49 shows a trace file rotation with a limit of 16 GB and a maximum deviation of 25%. With this configuration, the trace is split into 4 files, guaranteeing that at least 12 GB are available for analysis at any point in time. As we need at least 4 files to make that guarantee, a new trace file must be started every 4 GB.

Overwriting old trace data creates the problem of unresolvable dependencies in events. Many events rely on the fact that some previous event carries information that is necessary to reconstruct the full information of the current event. For example, the *fast allocation* and *fast move* events rely on previous *TLAB allocation* and *PLAB allocation* events respectively to reconstruct the correct address. Also *allocation* events in general depend on the allocation site entry in the symbols file and *move* events depend on all previous *move* events of the same object and on the corresponding *allocation* event.

However, we need to distinguish hard dependencies, i.e., information that must be available to reconstruct the heap, from soft dependencies, i.e., information that would be nice to have but that is not absolutely

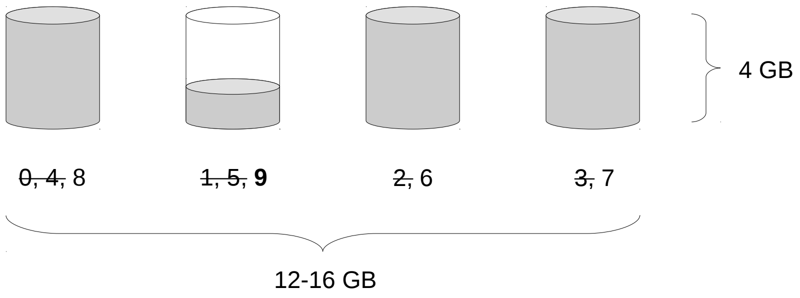


Figure 49: Rotating the trace with a 16 GB limit and a 25% deviation in its third round

necessary to reconstruct the heap. For example, the allocation site of an object for move events is a soft dependency because the allocation site is not absolutely necessary to reconstruct the heap. The type of an object, however, must be resolvable because without proper type information, specifically the size, we cannot calculate the destination address of *fast move* events. For the same reason, the dependencies of *allocation* and *move* events to *TLAB allocation* and *PLAB allocation* events are hard dependencies.

When we cut the trace, it is not a problem if we cannot reproduce the allocation site of an object (because the allocation event was cut off). We can still reconstruct the heap without knowing where the object came from. However, if we need to process a *fast allocation* event, and we miss the preceding *TLAB allocation* event, we have no basis for reconstructing the object's address.

To resolve hard dependencies and as many of the soft dependencies as possible, we cut the trace (i.e., move from one trace file to the next) only at the start of a garbage collection. Choosing these events as the only cut-off point, we resolve a lot of the hard dependencies automatically because due to the automatic retirement of all TLABs and PLABs before a GC starts, no *allocation* event and no *move* event misses the corresponding *TLAB allocation* or *PLAB allocation* event. The other hard dependency that needs to be resolved is the dependency of *move* events to the corresponding *allocation* events because *move* events do not carry the object's size which is necessary for address reconstruction.

Reconstructing Object Sizes

The object size is necessary to calculate the destination address in *move fast* events because the address is reconstructed as the previous address plus the size of the previous object. As every trace file starts with a garbage collection, all *move fast* events in that very first collection miss that information. Therefore, we need to introduce a mechanism to save the type information, including the object size.

To be able to restore the type information in the garbage collection events at the start of a trace file, we use a new kind of event called *GC move slow sync* and *GC move fast sync*. These events have the same format as the *GC move slow* and the *GC move fast*, except that they also carry the a type ID in the header. All *move* events in the first collection after a new trace file was started are replaced with these new event types. By doing this, we can reconstruct all types, even if the allocation event has been cut-off. We call those collections *synchronization collections*.

Figure 50 shows rotating the trace, with both normal collections and synchronization collections. As every trace file may eventually become the oldest trace file, all trace files must start with a synchronization collection.

As we can only switch to a new trace file right before a garbage collection, we cannot switch if no

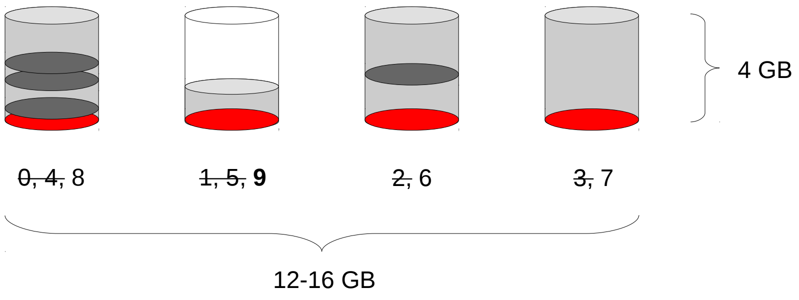


Figure 50: Rotating the trace with a 16 GB limit and a 25% deviation in its third round, including normal collections (grey disks) and synchronization collections (red disks)

collection is occurring, resulting in the trace file growing above the allotted limit. To overcome this issue, we use the configuration deviation to wait for a collection, i.e, a trace file may slightly exceed its alotted limit according to the configured deviation. If the limit including the deviation has been reached, we force a garbage collection. We call these garbage collections *emergency synchronization collections.* Although these collections distort the application and GC behavior, we claim that, given enough trace storage capacity, these collections will hardly occur. This is trivially true because only allocations generate trace data outside a garbage collection, and the more objects are allocated, the higher the probability that a garbage collection will occur. Section 8.3 shows a detailed evaluation of synchronization collections.

Reconstructing Allocation Sites

In contrast to type information, information about allocation sites is not absolutely necessary although nice to have. However, we cannot just add the allocation site ID (similarly to the type ID) to the move events because we cannot infer it from the object itself. The object header contains a pointer to the type descriptor, which makes it easy to determine the type ID of any live object, but the allocation site is not saved. This information exists only at the allocation itself (where we write it into the *allocation* event) and is lost afterwards because there is no need for the VM to track it. Thus we need save it at the allocation so that we can then add it to the *GC move sync* events later.

A naive solution would be to maintain a big map, using the address of the object as the key and the allocation site ID as the value. However, such a map would be difficult to keep up to date, because objects are created in parallel and also moved in parallel (the GC moves objects, changing the key in that map). Also, for big heaps with billions of object, this solution is not scalable.

Since an external data structure is unfeasible, the only remaining possibility is to store the allocation site inside the object itself. The typical object layout in the HotSpot^{TM} VM is shown in Figure 51. At the start of every object, there is the object header. It contains the mark word (8 bytes assuming a 64-bit architecture) and the class word (8 or 4 byte assuming a 64-bit architecture and depending on whether the VM is configured to compress pointers into 32-bit). The mark word can have several different formats, depending on the current state of the object.

Storing Allocation Sites as Fields. The first possibility we explored was extending the object header with an additional field (at least 4 bytes to keep the fields aligned, which is crucial for the JIT compiler). To evaluate in advance whether this approach is feasible, we examined the average object size in Figure 52.

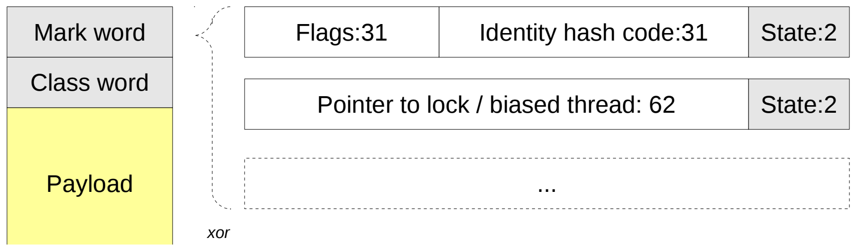


Figure 51: Typical object layout

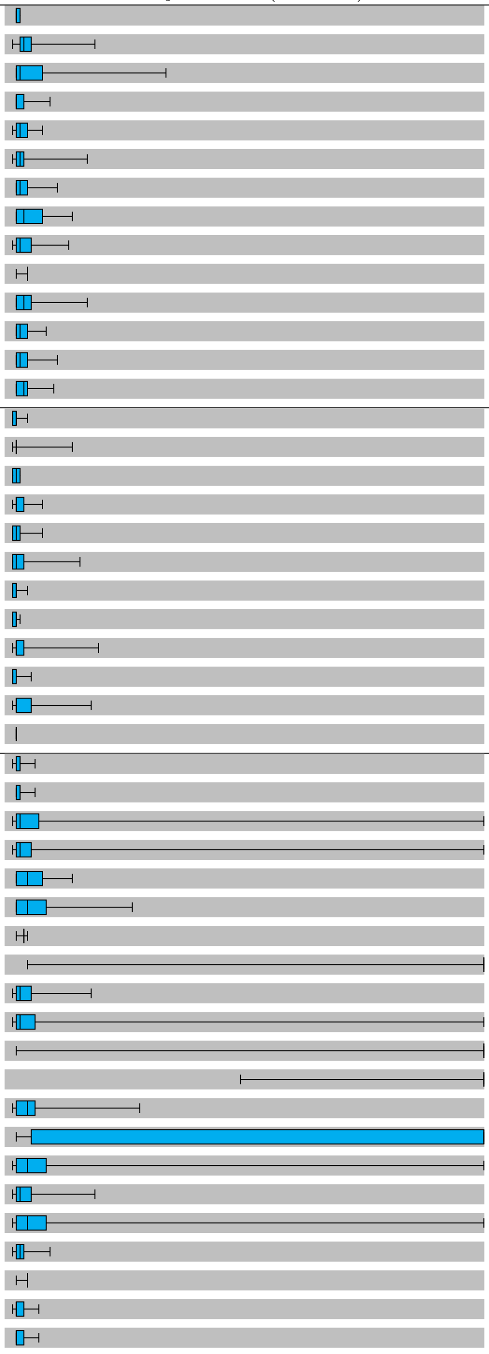
The relative change in object size can be seen in the last column of that figure. As most GC algorithms are of linear complexity depending on the amount of live memory, an increase in the object size will result in a similar increase in the GC time. Also, GCs would occur more often because the heap will fill faster.

Storing Allocation Sites in Identity Hashes. Since we do not want to add a header field, we need to store the allocation site somewhere else in the object header but as the layout shows, the object header is already densely packed. Inspired by Odaira et al. [37] we chose to exploit the identity hash code. The identity hash code is a 31-bit integer that should be as unique as possible for every object (disregarding the object's contents), but it does not necessarily have to be unique. Moreover, the identity hash code must not change during the object's lifetime. To ensure that this hash code remains unchanged, it is calculated lazily at the first access and stored into the header. How the hash is computed is at the discretion of the VM itself.

Since we can choose any value for the hash code, we can construct it from a 16 bit allocation site ID and a 15 bit real code. However, this produces two problems: (1) when storing something into the hash field, and thus making it part of the hash code, we must generate the identity hash for every object eagerly, and (2) the entropy of the hash code is significantly reduced, possibly making identity-hash-based data structures perform badly.

Eager Hash Code Generation. Generating the hash code for every object at the time of its allocation is a non-trivial task because the algorithm for calculating it is quite complex. Although this is already implemented in the VM, we cannot make a VM call at every object allocation because a call at every allocation would increase the run time significantly. Therefore, we implemented the hash code generation algorithm directly into the JIT compilers and inline it right after every allocation. This will, however, increase the code size depending on the algorithm used. To reduce the code size, we check (when compiling an allocation site) whether the object that is allocated is of a class with a overwritten *hashcode()* function. If this is the case, the identity hash code will only be generated if a special data structure, like the `IdentityHashMap`, is used. However, in most cases, we assume that the identity hash code will not be used and we just store the allocation site and do not initialize the rest of the hash field.

Hash Code Entropy. VM implementers have spent a lot of time thinking about good hash strategies for the identity hash code in order to increase the performance of hash-based data structures. The Hotspot^{TM}VM alone provides 6 different hash code algorithms (configurable with `-XX:hashCode=1..6`), i.e, the *Unguarded Global Park-Miller Random Number Generator* by Park and Miller [38], the *Stable Stop-the-world 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 object's address), and *Marsaglia's xor-shift Scheme*



	Benchmark	Avg. object size	Distribution object sizes (0...1024)	Rel. change
DaCapo	avrora	30.04		+13.32%
	batik	72.22		+5.54%
	eclipse	61.73		+6.48%
	fop	42.42		+9.43%
	h2	37.94		+10.54%
	jython	43.42		+9.21%
	luindex	48.50		+8.25%
	lusearch	100.95		+3.96%
	pmd	36.08		+11.09%
	sunflow	45.15		+8.86%
	tomcat	57.42		+6.97%
	tradebeans	42.00		+9.52%
	tradesoap	47.79		+8.37%
	xalan	48.52		+8.24%
DaCapoScala	actors	24.66		+16.22%
	apparat	32.25		+12.40%
	factorie	24.02		+16.65%
	kiama	34.86		+11.48%
	scalac	31.87		+12.55%
	scaladoc	43.00		+9.30%
	scalap	25.23		+15.85%
	scalariform	24.85		+16.10%
	scalatest	43.99		+9.09%
	scalaxb	24.78		+16.14%
	specs	47.42		+8.44%
	tmt	24.56		+16.29%
SPECjvm	compiler.compiler	33.39		+11.98%
	compiler.sunflow	35.20		+11.36%
	compress	1770.21		+0.23%
	crypto.aes	5027.60		+0.08%
	crypto.rsa	59.73		+6.70%
	crypto.signverify	207.26		+1.93%
	derby	38.97		+10.26%
	mpegaudio	504.20		+0.79%
	scimark.fft.large	74883.25		+0.01%
	scimark.fft.small	4150.65		+0.10%
	scimark.lu.large	1997.62		+0.20%
	scimark.lu.small	231.43		+1.73%
	scimark.monte carlo	105.34		+3.80%
	scimark.sor.large	1950.20		+0.21%
	scimark.sor.small	142.01		+2.82%
	scimark.sparse.large	45611.07		+0.01%
	scimark.sparse.small	6629.54		+0.06%
	serial	39.98		+10.01%
	sunflow	45.45		+8.80%
	xml.transform	37.99		+10.53%
	xml.validation	34.63		+11.55%

Figure 52: Average object sizes, object size distribution, and the relative change in object size if we would add a 4 byte field

(default) by Marsaglia et al. [35]. By reducing the number of possible values of the identity hash code (2^{31}) by a factor of 2^{16}, the entropy of the hash code is reduced to 0.0015%. To compensate for that, we take the lower 2 bytes for the hash (instead of the upper 2 bytes) of it because many data structures do modulo operations on the hash, making the lower bits more significant than the higher bits. We will show in Section 8.3 that on real-world benchmarks, the reduced entropy of the identity hash has practically no effect.

4.8 Pointer Tracing

So far, we have only recorded allocations and object movements. However, to track down memory leaks we may also need the values of all pointer fields.

Since recording every pointer modification is too expensive, we decided to capture pointers only during garbage collections. At this point in time, the garbage collector has to traverse all reachable references anyway. Although the JVM supports different GC algorithms, the concepts and the implementations of pointer traversal are quite similar. Tracing pointers only during garbage collections makes it impossible to notice immediately when an object becomes unreachable. However, we can still see which objects are live or dead at the time of a garbage collection. This is sufficient for memory leak detection.

Precision vs. Performance. Tracing pointers only at garbage collections reduces precision, because we do not trace every single change to a pointer field. In fact, tracing every change to a pointer field would impose enormous overhead, as shown by some related work (cf. Section 9). However, we argue that this amount of precision is not necessary for most use cases. For example, a memory leak is a big data structure that is kept alive unintentionally. This data structure does not change and can be found without tracing all pointer writes. Also, GC performance degradations can be tracked down by looking at the object graph. In this case, a precise and up-to-date object graph is necessary, but, since we trace pointer information at garbage collections, our information is precise when it needs to be. Furthermore, any other memory-related performance degradation (e.g., unnecessary allocations) will sooner or later fill up the heap and consequently trigger a garbage collection. Again, we can analyze at this point what was going wrong.

4.8.1 Move Events with Pointer Information

As discussed in Section 4.2.4 we record an event for every object movement, containing the object's source address and its destination address. All other object characteristics, e.g., its type, its size or its allocation site, can be derived from its allocation event. Since these GC move events are recorded during garbage collection, the pointers of a moved object can be simply appended to the existing event.

Figure 53 shows newly introduced events to include pointers in move events. The *move events w/ pointers* are based on the move events discussed in Section 4.2.4. They have the same semantics, but additionally contain the current values of the object's pointer fields. Due to the pointer encoding (see below), these events can hold only up to 12 pointers, which is a problem for big objects with many fields as well as for object arrays. Thus, the *GC Object Pointers* and the *GC Array Pointers* events are used to record an arbitrary number of additional pointers for an object.

All move events on which these new events are based have three unused bytes in their first block. The new events have exactly the same format, except that these three bytes are used to describe the number and encodings of the appended pointers.

	ID	Name	Semantics
Pointer	27	GC Move Slow w/ Pointers	an object has been moved (and pointers have been modified)
	28	GC Move Fast Wide w/ Pointers	an object has been moved (and pointers have been modified)
	29	GC Move Fast w/ Pointers	an object has been moved (and pointers have been modified)
	2A	GC Move Fast Narrow w/ Pointers	an object has been moved (and pointers have been modified)
	2B	GC Object Pointers	an object has modified pointers
	2C	GC Array Pointers	an array has modified pointers

Figure 53: Pointer events, including their IDs and a short description of their semantics, events with equal semantics have alternating formats for different use cases.

Pointer Encoding. We distinguish three pointer encodings: (1) Null pointers are encoded as binary 11. Since null pointers do not reference objects, no additional information needs to be written; (2) Relative pointers are encoded as binary 01. For every relative pointer, a 32 bit word denoting the offset of the referred object relative to its referrer (the previous reference of the referrer) is appended; (3) Absolute pointers are encoded as binary 10. Every absolute pointer is appended as a 64 bit value.

We introduced relative pointer encoding to reduce the amount of generated data. Objects that reference each other are likely to be located close to each other due to their sequential allocation and the depth-first traversal of common garbage collection algorithms. Consequently, their relative positions to each other, i.e., *(referrer address - pointer address) / heap word size* are likely to be encodable in 32 bits. The probability of being able to use 32-bit offsets increases if we do not use the offset relative to the referrer, but rather the offset to the previously referenced object. Since references may point to objects that are to the left or to the right of the referrer (or the previously referenced object), we use the most significant bit as sign bit. On a 64-bit system where the heap word size is 2^3 we can therefore encode an address space of up to $2^{31+3} - 1$ in either direction.

For every absolute pointer a 64-bit word denoting the pointer's absolute address in the heap is appended to the end of the GC move pointer event. We write absolute pointers only if they cannot be encoded relatively. This is especially the case for heap sizes larger than $2^{32+3} - 1$, where objects are located in different heap spaces, e.g., cross-generational pointers which point from the old generation to the young generation and vice versa.

The binary encoding 00 is used to declare that no further pointers are encoded in the event. It is only used as a sentinel when less than the maximum number of pointers (i.e., less than 12 pointers) are encoded in a GC move pointer event. For example, if a move event encodes 4 pointers in a relative way, the pointer encoding would be 010101010000000000000000, i.e., 4 times 01 to indicate a relative pointer. The rest of the 24 bit encoding is filled with 00 to indicate that no more pointers are contained.

Pointer Address Versions. The garbage collector moves all live objects from their source addresses to their destination addresses. For every moved object it emits a GC move event that also contains its pointers. However, the referenced objects may or may not have already been moved at the time when the referencing object is moved. In other words, some of the referenced objects are still at their old addresses while others have already been moved to their new addresses. Since it would be difficult to keep old and new addresses apart in the trace we decided to always write the old addresses of referenced objects. (If a referenced object has already been moved the reference still points to its old address and the stale copy there holds a forwarding pointer to the new address of this object.) Since the trace records all movements of live objects we can map all old addresses to the corresponding new addresses later in an offline processing step.

4.8.2 Dirty Pointer Tracing by Exploiting the Card Table

If an application triggers many garbage collections, we potentially record pointer values that have not changed between collections over and over again. Especially in-memory databases or applications with a large number of long-living objects would suffer from the GC overhead caused by tracing all pointers. In order to record only those pointers, which have been changed since the last garbage collection, we exploit the JVM's card table. As discussed in Section 2, the card table keeps track of all updates of pointer fields. Whenever a pointer is modified, the appropriate card is marked dirty. Although the VM uses this table for identifying pointers from the old generation to the young generation, it is maintained for the entire heap.

By checking the card table, it is possible to reduce the amount of recorded data by recording pointers only if they belong to objects that are (completely or partially) in dirty cards. We cannot preclude that we still emit unmodified pointers, because a single dirty card may relate to multiple objects.

Chapter 5

Efficient Heap Reconstruction and Processing

"Do what you can, with what you have, where you are." - *Theodore Roosevelt*

A trace file represents the memory behavior of the monitored application over time. Every event can be regarded as an incremental change to the heap. By replaying all events up to a specific point, we can recreate the heap layout for that point in time.

In this section we will discuss dependencies between events and how they can be resolved properly (cf. Bitto et al. [4]). We will show how we can recreate the heap in using parallel threads without violating dependencies and only using as little memory as possible to enable analysis on the same machine the trace has been recorded on. Furthermore, we will show how to compute a *heap diff*, i.e., how to compute changes of a heap over time.

5.1 Data Dependencies

Events are recorded per thread and are written to the trace file using thread-local buffers. This means that the trace file can be divided into chunks, where every chunk contains only events recorded by a single thread. Therefore, the temporal order of events can only be guaranteed within the context of a thread.

Also, the trace can be divided into *mutator phases*, i.e., phases when the application was active, and *GC phases*, i.e., phases when the GC was active. Both phases are alternating and may contain an arbitrary number of chunks. A *mutator phase* contains chunks recorded by application threads, whereas a *GC phase* contains chunks recorded by GC threads. The individual phases are separated by a chunk containing a single *GC start* or *GC end* event.

Mutator Phase. The mutator phase consists mainly of allocation events. An allocation usually does not contain the address of the allocated object. Rather, this address must be computed from the addresses of previously allocated objects in the same TLAB or from the TLAB address itself (cf. Section 4.2.3). As all TLABs are reset on phase changes, an allocation event can never depend on an allocation event of a previous phase.

Also, the dependency on previous allocation events introduces dependencies between chunks because the previous allocation event may not be in the same chunk. However, since such a dependency is always to events of the same thread, dependencies between chunks are still only between chunks of the same thread.

GC Phase. The GC phase contains move events as well as allocation events. An allocation event occurs when the GC allocates an object, e.g., because it needs to fill a region to ensure an unfragmented heap.

Like in the mutator phase, move events depend on previous move events and on PLAB allocation events. Additionally, move events also depend on the allocation event of the moved object because reconstructing the addresses requires the object's size.

Cross-phase Dependencies. Figure 54 shows the dependencies within chunks as well as across chunks and across phases. The allocation event in the first chunk depends on the previous TLAB allocation event because of the address reconstruction. The allocation event in the second chunk depends on the allocation event in the first chunk for the same reason. The first move event in the last chunk depends on the PLAB allocation event as well as on the object allocation event of the object it is moving. The second move event in the same chunk depends on the first move event as well as on its respective object allocation event.

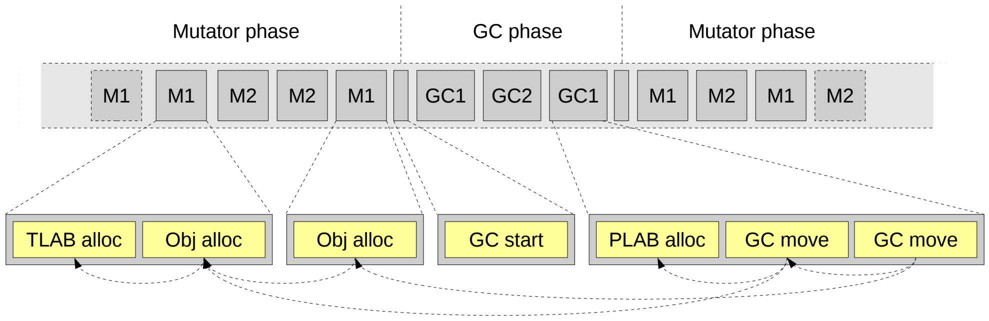


Figure 54: Event dependencies across multiple phases containing two mutator threads (M1 and M2) and two GC threads (GC1 and GC2)

As events are not in temporal order across multiple threads, recreating the entire heap within a phase is difficult. We can create parts of the heap (i.e., those objects that have been allocated by a single thread) but we can only merge them at phase changes. At phase changes, the VM ensures that all buffers are flushed into the trace file and all threads are synchronized. Thus, we can recreate the entire heap at any phase change, whereas in between, i.e., within a mutator phase or within a GC phase, we can only recreate thread-local portions.

5.2 Parallel Parsing

In principle, parsing a trace file is a trivial and well-understood task. Every event starts with an 8 bit event type, specifying the upcoming event and its format. In order to parse the trace as quickly as possible, we want to make use of multi-core machines. However, parsing the trace in parallel means that we need to split the trace into parts that can be processed in parallel without violating any dependencies described above.

A suitable portion of a trace that can be parsed in parallel to other portions is a single chunk recorded by a single application thread. Obviously, we cannot process chunks recorded by the same application thread in parallel. We need to keep them in order to resolve all dependencies correctly. However, we do not necessarily have to parse them with the same parsing thread. We just need to make sure that a chunk has been fully parsed, before another one of the same application thread is parsed. This means that the parsing parallelism is automatically limited by the number of concurrently recording threads in the monitored application.

Figure 55 shows our parallel and dependency-aware parsing infrastructure. At first, the IO thread reads chunks from the trace file an inserts them into an IO queue. By using a dedicated thread for reading the trace file, we decouple IO from parsing and can better utilize the speed of the underlying storage medium as well as the cores of the processor. A master thread consumes chunks from the IO queue and puts them in one of several queues. Per application thread, we maintain a queue that contains all chunks generated by this thread in the right order. Also, we keep a local heap per application thread, which represents all changes to the heap that this thread has made in the current phase. Every queue of an application thread is processed by exactly one parsing thread at a time, which has exclusive access to this queue and to the corresponding local heap. The queue can be in one of three states: (1) parked, i.e., we currently do not have any chunk in that queue and all previous chunks have already been processed, (2) queued, i.e., the queue contains at least one chunk that needs to be processed, but no parsing thread has claimed it yet, and (3) processing, i.e., a parsing thread is currently processing the first chunk of the queue. In the latter case no other parsing thread may parse any chunk of that queue. Thus we automatically guarantee that all dependencies are resolvable because only one parsing thread at a time can work on the chunks of a queue, which makes sure that all events of a particular application thread are processed in the right order.

5.3 Fast-access and Low-memory Heap Data Structure

Recreating the heap of the monitored application poses a significant challenge with respect to memory usage. When parsing the allocations of a mutator phase, we will need to store a representation of every allocated object. This representation needs to be kept until the corresponding object is reclaimed in a GC phase.

We must be careful not to trigger a garbage collection while parsing a mutator phase. If we triggered a collection, meaning that we cannot allocate new objects, we would probably run out of memory because we cannot collect anything because we have to keep all object representations alive at least until we reach the next GC phase. Although we can overcome this problem by increasing the heap size of the post-processing tool, we do not want to require a bigger heap for analysis than for the monitored application. Our goal is to be able to parse and post-process any trace with at most the same heap size as the monitored application, so that the user can analyze the trace on the same machine and does not require a better machine with more memory.

We thus strive for a data structure with low memory overhead. Also, as we parse the trace in parallel, we want a data structure that can be modified concurrently.

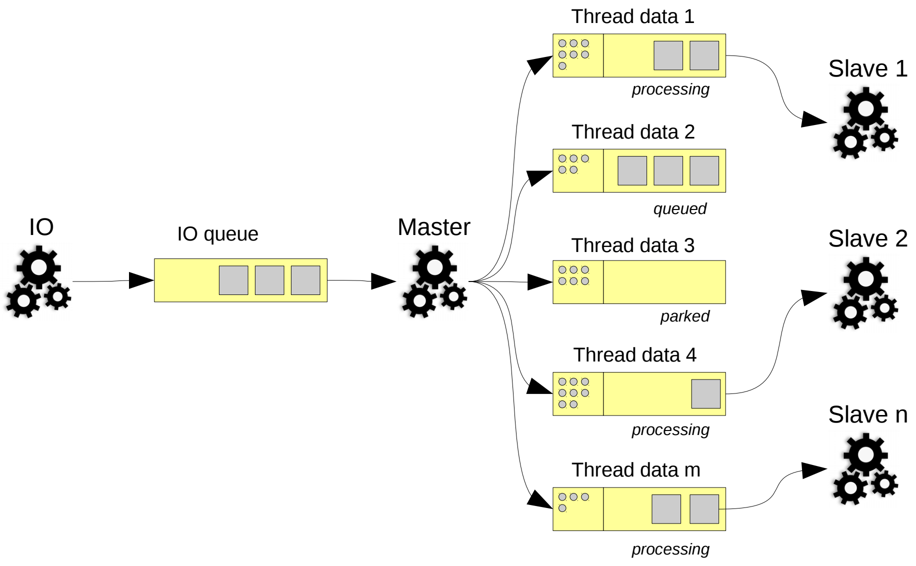


Figure 55: Architecture of our parallel dependency-aware parsing infrastructure

5.3.1 Naïve Heap Map

From an abstract perspective, a Java heap can be seen as a set of maps (one for every space) with Java objects as values and their respective addresses as keys. An intuitive solution would be to also allocate an object for every allocation event, store all information about the represented object into that new object and store it in a `java.util.HashMap`. The object to represent the allocation must store the address, the type, the size, the length in case of an array, the allocation site, as well as the event type. When a move event is parsed, the object with the corresponding address would be removed and re-inserted with the destination address.

Assuming all six properties of an object can be represented with 4 byte integers each, an object representation is at least 40 bytes (12 bytes for the header, 24 bytes for the properties as fields, 4 byte padding). Moreover, a `java.util.HashMap` uses `java.util.HashMap$Entry` objects to wrap key-value pairs. Every entry object holds a references to the key and the value (8 bytes each), as well as a link to the next element in the collision chain (another 8 bytes). Thus an entry object is also 40 bytes (12 bytes for the header, 24 bytes for the fields and 4 bytes padding). Furthermore, the map must maintain an array which is sized to be significantly bigger than absolutely necessary in order to avoid hash collisions (8 bytes per element). Consequently, storing a single object representation requires 88 bytes (40 bytes for the representation itself, 40 bytes for the entry object, and 8 bytes for the array element). Considering the fact that most applications have a relatively small average object size (cf. Figure 52), using this data structure is impractical in terms of memory overhead.

In terms of concurrent modifications, a `java.util.HashMap` can only be synchronized as a whole, making parallel processing impossible. Although there are some concurrent implementations, e.g., the `java.util.ConcurrentHashMap` which is almost lock free, these data structures require significantly more memory because they duplicate parts that are used concurrently. Thus, these data structures are not suitable for our use case.

5.3.2 Front Maps and Back Maps

To process a garbage collection based on the trace correctly, we need to keep two maps. A front map, holds the current state of the heap, and a the back map holds the state of the heap when the collection was starting. When a GC phase starts, the current front map becomes the new back map and a new empty front map is created. When the GC phase ends, we clear the back map and keep the front map. For every move, we read the object that is moved from the back map, and put it into the front map.

This is done for two reasons: (1) move events within a GC phase are potentially processed in a wrong order, and (2) in case of a failed collection, we might need to revert the heap to the the state in which it was before the GC.

The first case arises because move events are not in temporal order across multiple threads. For example, one GC thread moves an object to a new location. Subsequently, another GC thread moves another object to the old address of the previous object. This is no problem for the GC as the GC algorithms guarantee that the objects will be copied in the correct order. During parsing, however, we might parse the two move events in the wrong order. We would then replace the first object with the second object, before the second object is moved. We overcome this problem by keeping two maps, always reading from the back map, and writing to the front map.

The second case arises when a collection fails. Some GC algorithms, especially the copying algorithms, can fail if the heap gets too full or the survivor ratio is too high. If a GC failed, part of the work the GC has already performed must be reverted. We can then just clear the front map and fall back to the state we conserved in the back map.

5.3.3 Lightweight Objects

The naïve implementation creates one object to represent an object in the monitored application. However, using the address as key in the heap map, we do not have to store the address in the object representation. This leaves only the allocation site, the type, the size, and the length (if it is an array) which is all either constant or has only a limited range of values.

To optimize memory usage, we share the object representations. We cache all observed representations, and check that cache before we create a new representation. This way, we can share the representations of objects that are the same except for their address.

5.3.4 Lightweight Maps

Besides the object representation, we also need to optimize the map data structure. The map must (1) have a low memory footprint and must also (2) be (at least mostly) lock free for parallel parsing.

To achieve both goals we chose a data structure inspired by TLABs. A *space map* is a concurrent hash map, but instead of putting objects directly into the map, we store thread-local object buffers (TLOBs). Every TLOB contains only objects that are allocated or moved by the same application thread. Thus, we do not need to synchronize accesses to a TLOB. Also, we use TLAB and PLAB allocation events to create corresponding TLOBs. As every thread can only have one TLAB at a time, and only one PLAB per space at a time, we can store the corresponding TLOBs thread-locally and do not have to look them up in the map. Internally, a TLOB holds an array of all object representations, which are shared as discussed above.

This memory-efficient data structure comes at the cost of more expensive lookups. Using the naïve map, looking up an object at a specific address is as easy as retrieving the value at that address in the map. Now, we have to find the corresponding TLOB, i.e., find the element in the space map with the biggest address that is smaller or equal to the searched address (we use a map with a skip list implementation). Having found the TLOB, we need to find the correct index in the object array within the TLOB. Knowing the address of the TLOB and the size of every object in the object array we can iterate to the correct position.

To reduce the run-time complexity when searching inside a TLOB we keep an additional address array. This array contains the address of every n-th object. We can then use a binary search algorithm to find an object that is near the object we are looking for and start iterating there.

Figure 56 shows a space map. The space map references a number of TLOBs, each with an object array and an address array. The object array contains pointers to the global cache of object representations. The address array contains the address of every n-th object to speed up lookups by address.

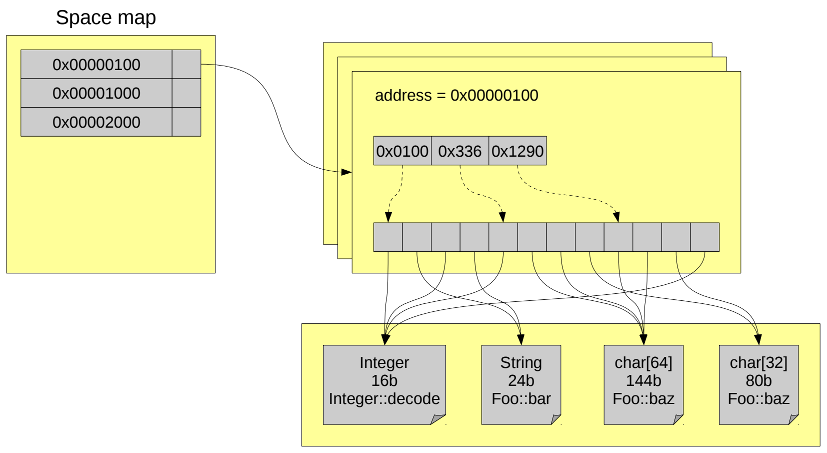


Figure 56: Low-memory data structure for shared object representations in TLOBs for almost lock-free access

5.4 Parsing Events

This section describes in detail how the events from Section 4.2 are processed.

5.4.1 Phase Events

The way in which a *GC start* event is processed depends on whether the event describes a minor or a major collection. We swap the front and the back map in every space, so that the back map contains all objects and the front space is empty. However, how the front map and the back map are handled at the corresponding *GC end* event depends on whether that particular space was collected or not. Whether a particular space is collected is determined by the minor/major flag in the *GC start* event. In case of a major collection, all spaces are collected, in case of a minor collection, the *GC info* events specify the set of collected spaces. If a space was collected, the *GC end* event commits all changes to that space by clearing the back map (all live objects must have been moved) and maintains the front map as is. If a space was not collected (all objects in the space stay alive without corresponding *move* events), we copy all objects from the back map into the front map before clearing the back map. The map will then contain all objects that have been in the map before the GC as well as the newly added objects. This case occurs often because many collection algorithms copy objects to not-collected spaces.

Also, as *GC start* and *GC end* events represent temporal synchronization points, the parser makes sure that all trace chunks located in front of such an event have been processed before processing the event itself.

5.4.2 Bookkeeping Events

The first kind of bookkeeping events are *space* events. A *space create* event creates a new space representation, including front and back maps, and a *space destroy* event destroys the space again. Between these events, an existing space can be allocated (i.e., marked) to be an Eden, Survivor, or Old space using the *space allocate* event or it can be freed (i.e., unmarked) with a *space release* event. A *space redefine* event redefines the bottom and the end address of the space. In general, a space may only contain objects if it is (1) created and (2) allocated first. The VM needs to ensure that the *space create* and *space allocate* events are fired before the first allocation event in that space. Releasing or destroying a space automatically frees all objects that are currently in that space. Thus, a GC may collect objects in a space by freeing the entire space if no object in that space is supposed to survive. However, we have to be careful about the temporal order when freeing a space. A space may be freed in the same GC phase as a GC thread evacuates its objects to another space. Therefore, we only clear the contents of freed spaces at a phase change.

The second kind of bookkeeping events are *thread* events We only use these events to create or destroy the parsing queues that are maintained per application thread to hold the chunks of that thread.

The last kind of bookkeeping events are *TLAB allocation* events and *PLAB allocation* events. Both events have the same effect, i.e., they allocate a new TLOB for the specified space type (i.e., Eden, Survivor, or Old). If a TLOB already exists for that space type, we retire it, i.e., we insert it into the front map of the corresponding space.

5.4.3 Allocation Events

For processing an *allocation* event, we only need to distinguish whether the used event format carries an explicit object address or whether it does not. In the latter case, we just take the current TLOB of the Eden space and append the object. It will automatically be at the correct address because the TLOB is owned by the same application thread and all events within that thread are in temporal order. If the address has been specified, we need to check whether a current TLOB exists and whether the address lies within that TLOB, because application threads do not use their TLABs for all objects, e.g., big objects above a specific threshold are allocated directly into the heap. If the address does not lie within the current TLOB, we allocate a new TLOB configured to hold exactly that one object and add it directly into the front map of the corresponding space. If the address lies within the TLOB, we append it just as if the address would not have been specified. In this case, the address must match the current fill level of the TLOB.

5.4.4 Move events

All *move* events are processed in the same manner. We determine the object at the source address (every format contains the source address) by looking it up in the back map and append it to the TLOB corresponding to the destination address. *Keep alive* events are handled in the same manner except that we assume that the destination address is equal to the source address.

5.5 Extending Stack Traces

As described in Section 4, an allocation site is not just defined as the allocating bytecode, but it may also include a chain of callers. We can determine these callers based on the inlining information provided by the JIT compiler without any additional run-time overhead.

However, in some cases, e.g., when a method is interpreted or when it is not inlined, we are unable to provide any callers. To counteract this problem, we try to extend the stack traces that we captured at run-time offline. We distinguish two techniques: (1) extending statically and (2) extending dynamically. The amount in which stack traces can be extended is discussed in detail in Section 8.2.4.

5.5.1 Extending Statically

Extending a stack trace statically means that we determine whether there is only one caller to a method. If so, we can extend the stack trace of every allocation site in this method with its caller.

As extending statically requires the entire code base for analysis, we extended the AntTracks VM to also serialize all loaded classes into a dedicated file. When parsing the symbol information, we also parse that file and try to extend all allocation sites as much as possible before even parsing the trace.

5.5.2 Extending Dynamically

Extending a stack trace dynamically means that we determine the calling method by the allocations that occurred before the allocation in question. Figure 57 shows an example where the allocation site in method `x()` allocating an object of type `A` can be extended dynamically. Because we see that `x()` is called by `y()` and `z()`, and we also see that they allocate an object of type `B` or `C` before the call respectively, we can extend the stack trace with a frame of `y()` or `z()` depending on the type of the previous allocation event.

```
void x() {
  new A();
}
void y() {
  new B();
  new x();
}
void z() {
  new C();
  new x();
}
```

Figure 57: Dynamic extension of stack traces

Extending stack traces dynamically has several limitations. For example, when there are two callers allocating the same type before the actual call, we cannot distinguish them. Also, calls that allocate other objects between the allocation and the call may interfere.

We address the first issue by expanding the context to several allocations, i.e., instead if having a single allocation that identifies a caller we use multiple allocations if available. The second issue can be addressed heuristically. We use a buffer to store the last n allocations and try to find an identifying allocation in there. However, if the call in between allocates too many objects, we cannot extend the stack trace.

5.6 Computing Heap Diffs

When analyzing changes in the heap over time is vital to understand the development of memory leaks or the development of situations that can degrade GC performance. Therefore, we need to support analyzing the changes between two defined points in time, i.e., a start point and an end point. When analyzing such an interval, every object can be assigned to one of four categories: (1) *permanent*, i.e., an object that was allocated before the start point and survived until after the end point, (2) *born*, i.e., an object that was allocated within the interval and survived until after the end point, (3) *died*, i.e., an object that was allocated before the start point, but died within the interval, and (4) *temporary*, i.e., an object that was allocated within the interval and also died within the interval.

Most memory monitoring tools do not have a concept of object identity over time. For example, dump-based tools can only compare aggregated metrics. When the number of objects of a specific type rises from one dump to the next, they cannot tell whether just the difference has been allocated or whether all objects have been deallocated and the objects in the second dump are all completely new. Such tools only know that the absolute number of objects of that type changed. Also, they cannot tell what happened between the dumps. After all there might have been a large quantity of objects allocated and deallocated again. Although sampling-based tools could in theory provide enough data to detect these cases, they distort the application behavior and the GC behavior so that it does not reflect the unmonitored application anymore (cf. Section 3).

Using the four categories defined above, we can provide precise information about any interval within the application's lifetime.

5.6.1 Iterative Computation

Given an interval for analysis, i.e., a start point and an end point, we can iteratively compute the heap diff. For performance reasons, we do not compute the categories from the events of the trace one by one, but rather compute them only at phase changes (using the parsing algorithm described above) and then use these categories at the phase changes to compute the categories of the interval using fast set operations. We start with the heap state at the start point and initialize the four categories as shown in Figure 58. Every category is represented by a set of objects. The permanent (*perm*) category is initialized with all objects of the initial heap state, the other categories are empty. During the iterative computation, the subsequent heap states at the phase changes will be added one by one until the end point is reached. In that process, all sets are modified so that they represent all heap changes in that interval.

$$\begin{aligned} perm &= init \\ born &= \emptyset \\ died &= \emptyset \\ temp &= \emptyset \end{aligned}$$

Figure 58: Initializing the heap diff computation with the heap state at the start point (*init*)

Advancing a Heap State. When advancing to the next heap state (i.e., to the next phase change), we need to modify the sets so that they represent the now extended interval, as shown in Figure 59. We first compute the *next* heap state parsing the trace from the current heap state to the next phase change. Then, we compute all objects that have not survived since the last heap state by checking which objects exist in the *perm* and *born* sets (these two sets contain all live objects) but are missing in the *next* heap state (1,2). We can then remove those freed objects from the *perm* set (3) and the *born* set (4). While the *perm* set can only get smaller, we also need to add all new objects to the *born* set. We introduce new objects by adding all objects of the *next* heap state except those that are also in the *perm* set. Although this operation will contain too many objects, i.e., it may also contain objects that are already in the *born* set because they have been allocated before the last heap state, this problem is solved by using the set union operation, adding an object only if it is not already contained in the set. This is more efficient and easier to implement than actually computing or tracking what objects have been allocated between the last heap state and the next heap state. Finally, we need to modify the *died* and *temp* sets by adding the corresponding freed objects computed above (5,6). We do not use the union operator here because there is a subtle difference between the sets containing live objects (*perm* and *born*) and the sets containing dead objects (*died* and *temp*): As the first sets contain only live objects, every object can be uniquely identified by its address, making $\setminus$ and $\cup$ operations very efficient. This is also true for the $freed_{perm}$ and $freed_{born}$ sets because they contain objects freed within a single iteration, making objects identifiable by address. The *died* and *temp* sets, however, contain objects that have died over the course of multiple iterations, possibly many objects at the same address. Therefore, we really need to add objects to the set (denoted by the + operator), and cannot perform a fast union operation as we could do if we would have at most one object per address.

$$\begin{aligned} freed_{perm} &= perm \setminus next && (1)\\ freed_{born} &= born \setminus next && (2)\\ perm &= perm \setminus freed_{perm} && (3)\\ born &= (born \setminus freed_{born}) \cup (next \setminus perm) && (4)\\ died &= died + freed_{perm} && (5)\\ temp &= temp + freed_{born} && (6) \end{aligned}$$

Figure 59: Extending a heap diff iteratively by adding the subsequent heap state (*next*)

Handling Moved Objects. When a garbage collector moves objects, their address, and thus their key in the *perm* and *born* sets, changes. Therefore, when parsing to the next heap state, we also check whether a moved object is in the *perm* or the *born* set (it must be in either one of them) remove it, and reinsert it with its new address.

As removing and inserting potentially involves reorganizing the data structure used to implement the *perm* and *born* set, and as we parse in parallel, we would need to synchronize these operations. To keep the parsing threads from racing for the same lock, we keep thread-local versions of the *perm* and the *born* set. Every lookup is done in the old global version of the respective set, but the object is not removed. Instead, all moved objects are added to the thread-local versions and an additional set of removed thread-local *perm* and *born* objects is maintained. Finally, new versions of the global *perm* and *born* sets are created by incrementally merging the old versions (except objects in the removed set) and the thread-local versions.

Handling Overwritten Objects. The *perm* and *born* sets operate on addresses only. For memory efficiency reasons, these sets do not store the actual object information because it is stored in the current heap state anyway.

However, if an object is moved to the same address as another object, overwriting it in the process, we need to update the *perm* and *born* sets accordingly. If an object in the *perm* set overwrites an object in *born* (or vice versa), we would mix up objects and would either miss deallocations or store a wrong deallocation. Therefore, we introduced a dedicated step to detect overwrites, and add them to the *died* and *temp* set, before we add a new heap state. This is shown in Figure 60.

First, we compute overwritten objects for the *perm* set and *born* set by examining which addresses occur both in the old and in the new version of the set and considering which objects survived without being moved. We then add the overwritten objects to the *died* and *temp* set depending on which set they have originated from (1,2). Finally, we remove the overwritten objects from the *perm* and *born* set (3,4).

$$died = died + overwritten_{perm} \tag{1}$$
$$temp = temp + overwritten_{born} \tag{2}$$
$$perm = perm \cup (perm_{old} \setminus overwritten_{perm}) \tag{3}$$
$$born = born \cup (born_{old} \setminus overwritten_{born}) \tag{4}$$

Figure 60: Detection and handling of cross-set overwritten objects

5.6.2 Complexity

In addition to parsing the trace, the set operations described above must be executed at every phase change. Although some operations will be trivial in many cases (e.g., when the trace changes from a mutator phase to a GC phase, the $freed_*$ sets in Figure 59 will be empty, simplifying many subsequent operations) the $\setminus$ and $\cup$ operations are of linear complexity, depending on the amount of live objects. This makes calculating a heap diff inherently more expensive than just parsing through the trace and computing a heap state.

In terms of memory complexity, the sets grow linearly depending on the amount of objects in the heap because the *perm* and *born* collections contain every live object exactly once. The *died* and *temp* sets have exactly one entry for every object description associated with a counter, making their size negligible.

Chapter 6

Visualization and Analysis

"Any fool can know. The point is to understand." - *Albert Einstein*

Every monitoring tool is just a means for finding non-functional defects, such as performance problems. However, finding non-functional defects is a tedious task that can easily frustrate the user if he is missing proper tool support. Thus, an intuitive and useful visualization is crucial for every monitoring tool.

This chapter gives an overview of how the prototype implementation of AntTracks employs common visualization techniques. Please note, that the main contributions of this thesis do not include visualization techniques. We therefore provide only a rough overview of how the user can use this tool to help resolving performance defects and to show how the data can be used in a memory monitoring tool.

6.1 Application Lifetime Visualization

To get a rough overview, the AntTracks tool first presents the application's overall memory and GC behavior. Figure 61 shows several different metrics, all using time (in seconds, relative to application start) on the x-axis. The following sections will discuss every plot in detail.

6.1.1 Memory Behavior

The first two plots show the memory consumption both in terms of objects and bytes over time. Furthermore, the consumption is divided into the individual spaces.

In this example, we can see how the Eden space is constantly pulsating due to many minor GCs. Additionally, as every minor GC promotes some objects, the Old space is slowly filled up. When the Old space reaches approximately 30MB, a major GC is triggered (in this example) and the memory consumption in the old space drops.

These plots are useful for getting a first idea of the memory behavior. The user can easily see the memory consumption of the application and can identify anomalies that require closer investigation.

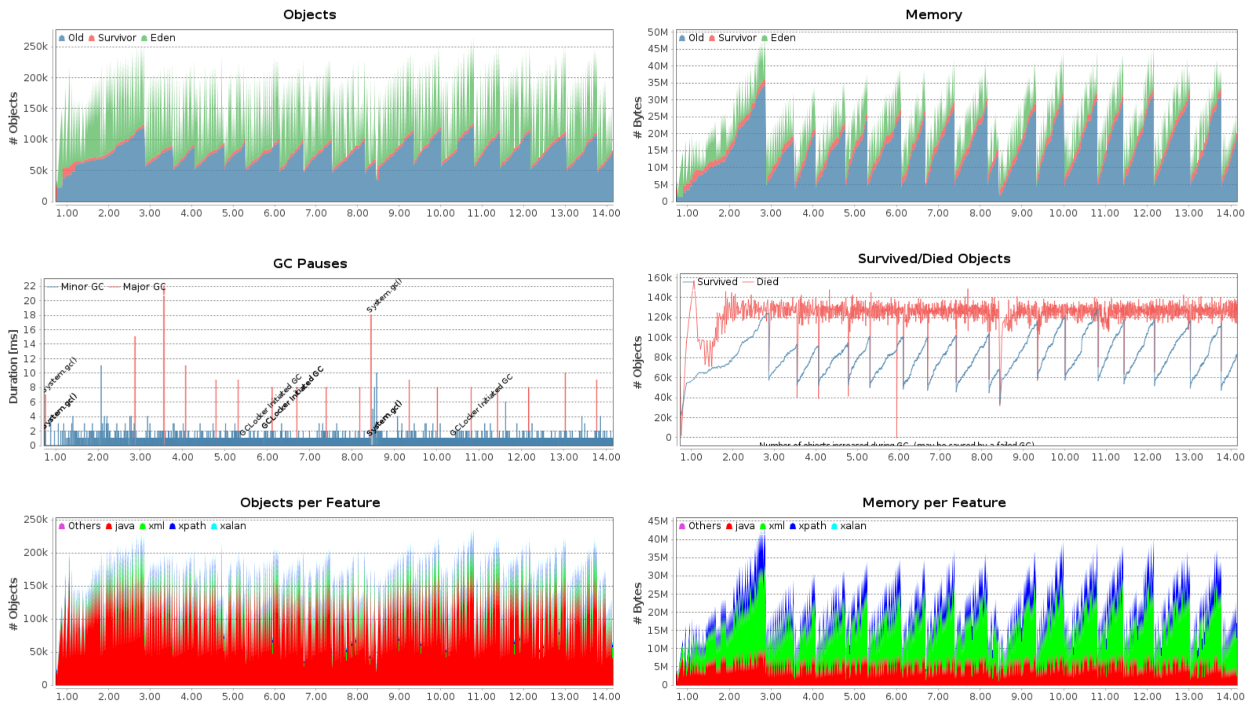


Figure 61: Overview of the memory behavior and the GC behavior of the xalan benchmark (large input, 2 iterations)

6.1.2 GC Behavior

The second set of plots show the GC behavior. The tool shows the GC pauses as well as the number of survived and dead objects for every collection.

In this example, most collections have been very fast (2ms or faster), only the major collections take up to 22ms. As the VM does not collect the heap arbitrarily but rather for a specific reason (a normal reason would be an allocation failure due to a full Eden space), the reasons for uncommon GCs are shown as additional captions in the first plot. In this example, we have some calls to `System.gc()` and some postponed collections due to native locks (`GCLocker Initiated GC`). The number of surviving objects remain within the same range for most of time and drops only on some occasions when a major collection is performed.

These plots are useful for identifying abnormal GC reasons, for identifying spikes in the GC pauses (that could have been the causes of unresponsive applications), or for detecting memory leaks which would cause the survived objects metrics to constantly rise.

6.1.3 Feature Behavior

The last set of plots show the memory consumption (in terms of objects and bytes) per feature, inspired by Lengauer et al. [27]. Within AntTracks, a feature is defined as a collection of classes, methods, and statements responsible for a single goal from the perspective of a developer (e.g., logging or database storage). An object is associated with a feature if it has been allocated by code that is part of that feature, i.e., a `String` object allocated by a method in the `Logger` class is associated with the *logging* feature, whereas a `String` object allocated by the database is associated with the *database* feature.

The feature definitions can be imported as an external configuration file into the AntTracks tool (cf. Figure 62). The config file in Figure defines the features *xml*, *xpath*, and *xalan*, each associated with a color encoded as an RGB value. Arbitrary code can be mapped to these features. For example, all classes of the package `org.apache.xml` are mapped to the feature *xml*. If code is not mapped, it will be associated with a special *Others* feature. If code is mapped more than once, it will be associated with both features because we prefer seeing memory correctly per feature over a clear association. We can map packages, classes, methods or individual bytecode ranges to specific features. When assigning features manually, only the mapping of classes or methods is feasible.

```
java (255 0 0) {
  java.*
}

xml (0 255 0) {
  org.apache.xml.*
}

xpath (0 0 255) {
  org.apache.xpath.*
}

xalan (0 255 255) {
  org.apache.xalan.*
}
```

Figure 62: A list of all features defined for the xalan benchmark (including RGB coding) as well as their mapping to code

We also provide an Eclipse plugin to automatically map features. In this plugin, the user defines a number of features and associates source code by selecting a class, a method or a range of lines in the source code and typing the feature's name. The appropriate mapping file will be generated automatically every time the code is compiled. This way, the user does not have to tediously adjust associated bytecode ranges every time the source code is modified.

In this example, we can see that the majority of objects are associated with the *java* feature (red). The configuration file defines the *java* feature as all classes in the package `java` or in any subpackage. However, we also see that that feature makes up only a small portion of the memory used. Most of the memory used is actually associated with the *xml* feature (green), defined as all classed in the `org.apache.xml` package or any subpackage. As we can see, the memory usage of the *java* and the *xpath* feature is always fluctuating around the same amount. Thus, we can conclude that `java` code and `xpath` code just allocate objects that can be deallocated within the next few minor collections. However, the *xml* feature rises until the next major collection occurs. This indicates that those objects are kept alive for too long and are therefore promoted to the old generation, in which only a major collection can reclaim them.

6.1.4 Allocation Modes and Object Distributions

For advanced users, we also provide another set of charts describing the distribution of objects with certain allocation modes and object kinds (cf. Figure 63). The first chart shows how many objects were allocated by the VM, by interpreted code, and by compiled code, whereas the second chart shows how many objects were instances, small arrays (length ¡= 255), and big arrays (length ¿= 255). Please note, that in the first chart, a large red area does not mean that many objects have been allocated by the interpreter, but rather that many objects allocated by the interpreter at some point in the past are still in the heap.

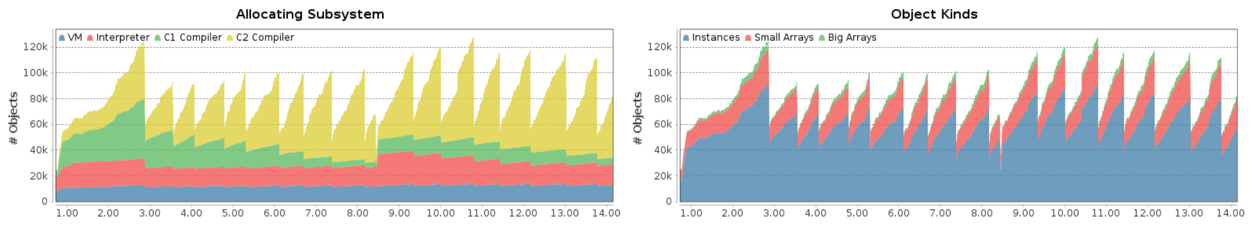


Figure 63: Allocation modes and object distributions of the xalan benchmark (large input, 2 iterations)

6.2 Heap State Visualization

Having identified a point in time that deserves closer investigation, the AntTracks tool will compute and visualize the heap state for that specific point in time. This section shows how that heap state is visualized and inspected.

6.2.1 Classification

Before visualizing a heap state, the user can define by what properties the objects in the heap will be grouped (cf. Weninger et al. [48]). This process is called object classification. By default, objects are first grouped by type, and then by allocation site, as shown in Figure 64. The tool offers a number of out-of-the-box classifiers, e.g., age (e.g., 1 collection, 2 collections), type (e.g., `java.lang.String`), allocating subsystem (e.g,, interpreter, c2-compiled code), object kind (e.g., instance, array), space (e.g., Eden, survivor, old), feature (as described above), and allocation site (e.g., `java.lang.String::substring(II):22`).

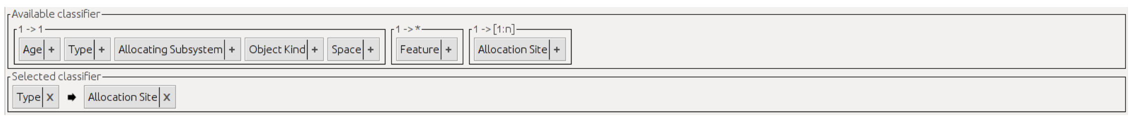


Figure 64: Out-of-the-box classifiers and default classification by type and allocation site

The user can also define custom classifiers by implementing the interface shown in Figure 65. The `space` parameter object provides access to the space the object is located in as well as to the entire heap. The `address` parameter represents the absolute address and the `pointsTo` parameter contains all pointers of the objects if available. The `obj` parameter contains meta information such as allocation site, type, and size. The user can then return any value to classify an object, e.g., the `String ''small''` or `''big''` depending on the size of the object. Finally, the user must annotate her custom classifier with the `@Classifier` annotation to specify whether it is a one-to-one, a one-to-many, or one-to-hierarchy classifier. The first classifier type associates an object with exactly one value, e.g., its type or its age. The second classifier type associates an object with a number of values (`T` must be an array), i.e., a number of features. Finally, the third classifier type associates an object with a sorted hierarchy of values (`T` must be an array), e.g., an allocation site whereupon every calling frame is an element in the hierarchy.

```
public interface ObjectClassifier<T> {
  public abstract T classify(Space space, long address, long[] pointsTo, ObjectInfo obj);
}

@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE, ElementType.TYPE_USE })
public @interface Classifier {
  String name();
  String desc();
  ClassifierType type();
}

public enum ClassifierType {
  ONE, MANY, HIERARCHY
}
```

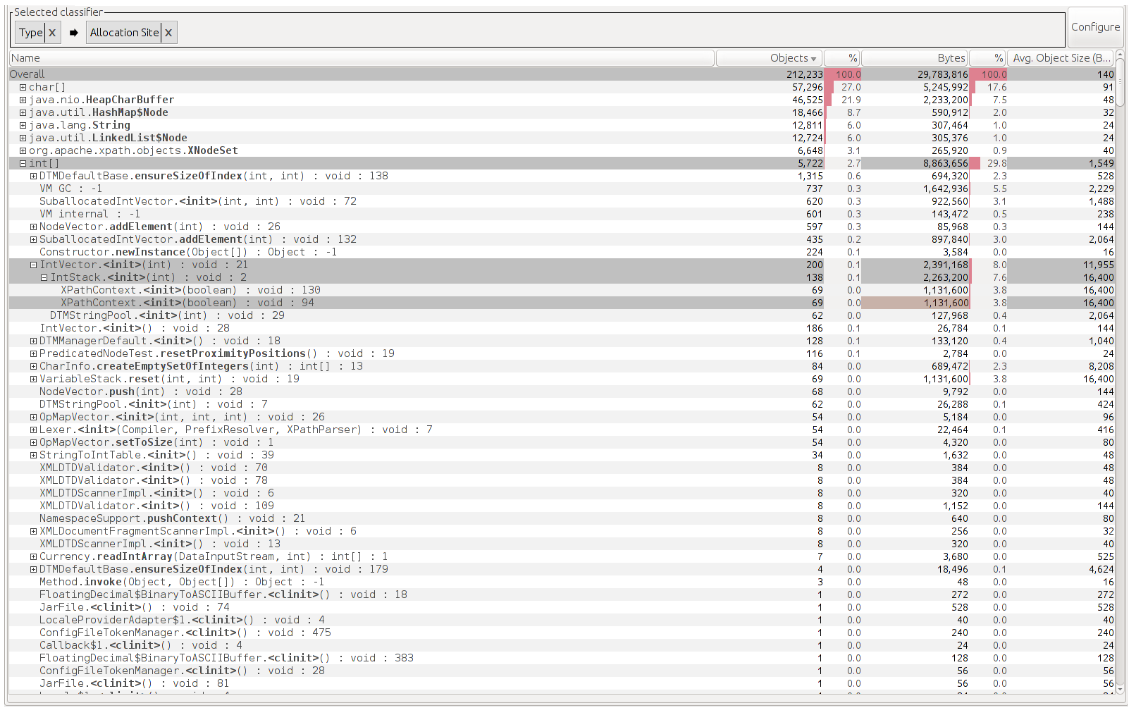
Figure 65: Classifier interface for a custom classifier)

Having selected a chain of classifiers, we can classify all objects and show their distribution in a tree. Figure 66 shows a heap state with the default classifier chain, i.e., type and allocation site. The root item in the tree represents all objects within the current heap state in number of objects as well as in bytes. The children of every tree element shows the distribution among the specified properties. In this example, every child of the root item represents all objects of a specific type because we classified by type primarily. The children of those elements are all allocation sites for objects of that type because we classified by allocation site secondarily. Further children are call sites of the parent allocation site.

6.3 Heap Diff Visualization

Beside single heap states, it might also be necessary to examine changes to the heap over some period of time. To do this, the user can specify an interval by selecting a start time and end time. The AntTracks tool will then compute the changes within this interval. As described in more detail in Section 5.6, objects are divided into four sets, i.e., permanent (*perm*) objects, died objects, born objects, and temporary (*temp*) objects. Figure 67 shows the visualization of a heap diff. The top charts show the development of the four object sets over the selected interval, while the table shows permanent (blue), died (red), born (green), and temporary (gray) objects within that interval. Similar to the heap state visualization, objects can be grouped with an arbitrary combination of classifiers.

As described in Section 5.6, heap diffs, and the four object sets, are computed iteratively, i.e., by advancing one heap state after the other until the diff covers the requested interval. The charts showing the entire



Name	Objects ▾	%	Bytes	%	Avg. Object Size (B...
Overall	212,233	100.0	29,783,816	100.0	140
⊞ char[]	57,296	27.0	5,245,992	17.6	91
⊞ java.nio.**HeapCharBuffer**	46,525	21.9	2,233,200	7.5	48
⊞ java.util.**HashMap$Node**	18,466	8.7	590,912	2.0	32
⊞ java.lang.**String**	12,811	6.0	307,464	1.0	24
⊞ java.util.**LinkedList$Node**	12,724	6.0	305,376	1.0	24
⊞ org.apache.xpath.objects.**XNodeSet**	6,648	3.1	265,920	0.9	40
⊟ int[]	5,722	2.7	8,863,656	29.8	1,549
⊞ DTMDefaultBase.**ensureSizeOfIndex**(int, int) : void : 138	1,315	0.6	694,320	2.3	528
VM GC : -1	737	0.3	1,642,936	5.5	2,229
SuballocatedIntVector.**<init>**(int, int) : void : 72	620	0.3	922,560	3.1	1,488
VM internal : -1	601	0.3	143,472	0.5	238
⊞ NodeVector.**addElement**(int) : void : 26	597	0.3	85,968	0.3	144
⊞ SuballocatedIntVector.**addElement**(int) : void : 132	435	0.2	897,840	3.0	2,064
Constructor.**newInstance**(Object[]) : Object : -1	224	0.1	3,584	0.0	16
⊟ IntVector.**<init>**(int) : void : 21	200	0.1	2,391,168	8.0	11,955
⊟ IntStack.**<init>**(int) : void : 2	138	0.1	2,263,200	7.6	16,400
XPathContext.**<init>**(boolean) : void : 130	69	0.0	1,131,600	3.8	16,400
XPathContext.**<init>**(boolean) : void : 94	69	0.0	1,131,600	3.8	16,400
DTMStringPool.**<init>**(int) : void : 29	62	0.0	127,968	0.4	2,064
IntVector.**<init>**() : void : 28	186	0.1	26,784	0.1	144
⊞ DTMManagerDefault.**<init>**() : void : 18	128	0.1	133,120	0.4	1,040
⊞ PredicatedNodeTest.**resetProximityPositions**() : void : 19	116	0.1	2,784	0.0	24
⊞ CharInfo.**createEmptySetOfIntegers**(int) : int[] : 13	84	0.0	689,472	2.3	8,208
⊞ VariableStack.**reset**(int, int) : void : 19	69	0.0	1,131,600	3.8	16,400
NodeVector.**push**(int) : void : 28	68	0.0	9,792	0.0	144
DTMStringPool.**<init>**(int) : void : 7	62	0.0	26,288	0.1	424
⊞ OpMapVector.**<init>**(int, int, int) : void : 26	54	0.0	5,184	0.0	96
⊞ Lexer.**<init>**(Compiler, PrefixResolver, XPathParser) : void : 7	54	0.0	22,464	0.1	416
⊞ OpMapVector.**setToSize**(int) : void : 1	54	0.0	4,320	0.0	80
⊞ StringToIntTable.**<init>**() : void : 39	34	0.0	1,632	0.0	48
XMLDTDValidator.**<init>**() : void : 70	8	0.0	384	0.0	48
XMLDTDValidator.**<init>**() : void : 78	8	0.0	384	0.0	48
XMLDTDScannerImpl.**<init>**() : void : 6	8	0.0	320	0.0	40
XMLDTDValidator.**<init>**() : void : 109	8	0.0	1,152	0.0	144
NamespaceSupport.**pushContext**() : void : 21	8	0.0	640	0.0	80
⊞ XMLDocumentFragmentScannerImpl.**<init>**() : void : 6	8	0.0	256	0.0	32
XMLDTDScannerImpl.**<init>**() : void : 13	8	0.0	320	0.0	40
⊞ Currency.**readIntArray**(DataInputStream, int) : int[] : 1	7	0.0	3,680	0.0	525
⊞ DTMDefaultBase.**ensureSizeOfIndex**(int, int) : void : 179	4	0.0	18,496	0.1	4,624
Method.**invoke**(Object, Object[]) : Object : -1	3	0.0	48	0.0	16
FloatingDecimal$BinaryToASCIIBuffer.**<clinit>**() : void : 18	1	0.0	272	0.0	272
JarFile.**<clinit>**() : void : 74	1	0.0	528	0.0	528
LocaleProviderAdapter$1.**<clinit>**() : void : 4	1	0.0	40	0.0	40
ConfigFileTokenManager.**<clinit>**() : void : 475	1	0.0	240	0.0	240
Callback$1.**<clinit>**() : void : 4	1	0.0	24	0.0	24
FloatingDecimal$BinaryToASCIIBuffer.**<clinit>**() : void : 383	1	0.0	128	0.0	128
ConfigFileTokenManager.**<clinit>**() : void : 28	1	0.0	56	0.0	56
JarFile.**<clinit>**() : void : 81	1	0.0	56	0.0	56

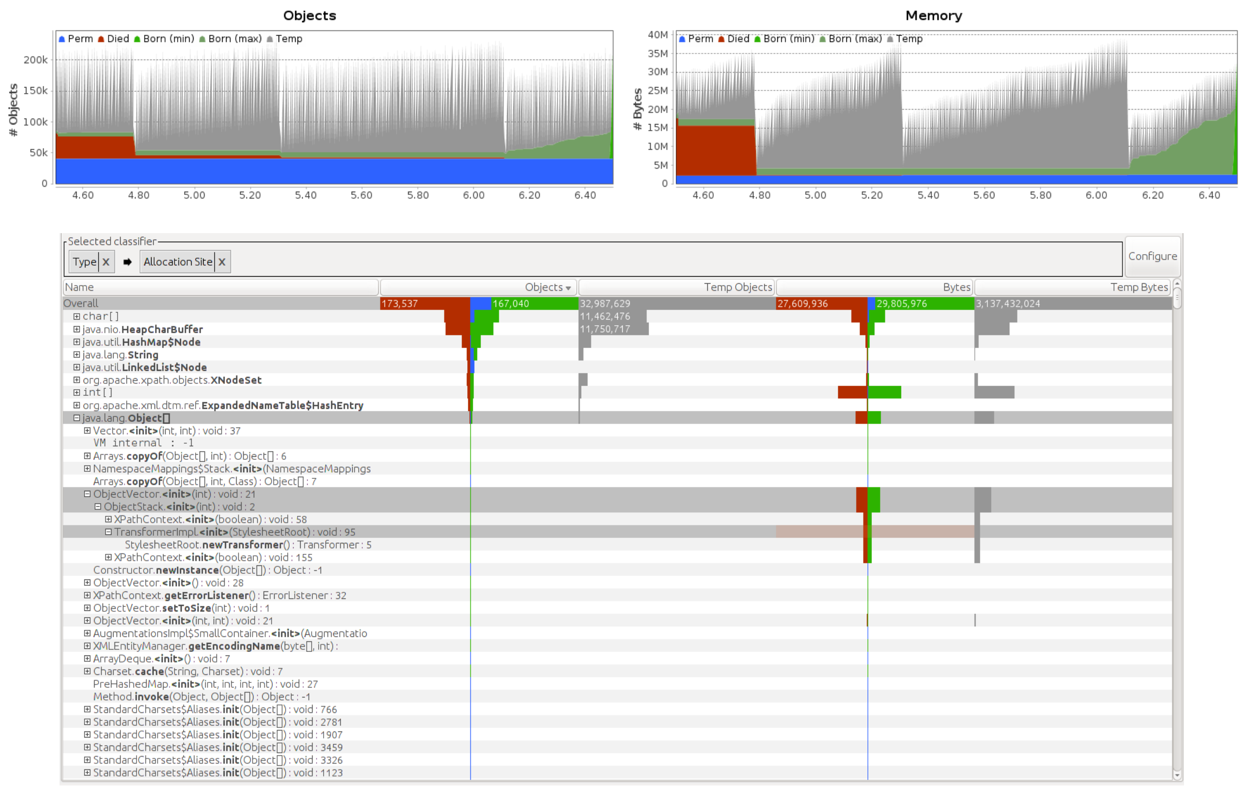
Figure 66: Heap state of the xalan benchmark (large input, 2 iterations) at second 4.5

interval has the same shape as the overview charts, i.e., at every point, the hight determines how many objects or how many bytes are in the heap at that point in time. Considering these charts, and the zoomed versions in Figure 68, we see four invariants that always hold: (1) The perm set must be constant; (2) The died set can only shrink and must be empty at the end; (3) The born set can only grow and must start empty; (4) The temp set can grow and shrink arbitrarily, but it must start and end empty.

When we just sample and plot the size of every set during the iterative computation in the example in Figure 69, we see that the invariants do not hold. For example, died objects are categorized as perm while they are still alive, making the perm set drop and the died set rise when they die. The same holds for the born set and temp set. Furthermore, the temp set and died set actually contain objects that have already died, consequently distorting the chart because it no longer presents the currently live objects for every point in time.

Since the heap diff's set sizes do not satisfy our requirements for the chart, we need to perform additional computations based on every point i in the heap diff of length n. We take the size of every set, and compute a new count that satisfies our invariants described above for every point in the chart. From here on, we will denote a set's size in the heap diff as *size*, and the newly computed value for the chart as *count*. Figure 70 shows all additional computations necessary.

The died size is the opposite of what should be shown in the chart: whenever an object dies, the perm size drops by one and the died size rises by one. This violates the invariants, that the died set may only shrink and that the perm set must be constant. We therefore compute the died count by subtracting the died size from the died size of the final heap diff (1). We then compute the perm count by subtracting the



```
Name                                              Objects ▾     Temp Objects   Bytes                    Temp Bytes
Overall                                   173,537  167,040  32,987,629     27,609,936  29,805,976  3,137,432,024
⊞ char[]                                                     11,462,476
⊞ java.nio.HeapCharBuffer                                    11,750,717
⊞ java.util.HashMap$Node
⊞ java.lang.String
⊞ java.util.LinkedList$Node
⊞ org.apache.xpath.objects.XNodeSet
⊞ int[]
⊞ org.apache.xml.dtm.ref.ExpandedNameTable$HashEntry
⊟ java.lang.Object[]
  ⊞ Vector.<init>(int, int) : void : 37
    VM internal : -1
  ⊞ Arrays.copyOf(Object[], int) : Object[] : 6
  ⊞ NamespaceMappings$Stack.<init>(NamespaceMappings
    Arrays.copyOf(Object[], int, Class) : Object[] : 7
  ⊟ ObjectVector.<init>(int) : void : 21
    ⊟ ObjectStack.<init>(int) : void : 2
      ⊞ XPathContext.<init>(boolean) : void : 58
      ⊟ TransformerImpl.<init>(StylesheetRoot) : void : 95
          StylesheetRoot.newTransformer() : Transformer : 5
      ⊞ XPathContext.<init>(boolean) : void : 155
    Constructor.newInstance(Object[]) : Object : -1
  ⊞ ObjectVector.<init>() : void : 28
  ⊞ XPathContext.getErrorListener() : ErrorListener : 32
  ⊞ ObjectVector.setToSize(int) : void : 1
  ⊞ ObjectVector.<init>(int, int) : void : 21
  ⊞ AugmentationsImpl$SmallContainer.<init>(Augmentatio
  ⊞ XMLEntityManager.getEncodingName(byte[], int) :
  ⊞ ArrayDeque.<init>() : void : 7
  ⊞ Charset.cache(String, Charset) : void : 7
    PreHashedMap.<init>(int, int, int, int) : void : 27
    Method.invoke(Object, Object[]) : Object : -1
  ⊞ StandardCharsets$Aliases.init(Object[]) : void : 766
  ⊞ StandardCharsets$Aliases.init(Object[]) : void : 2781
  ⊞ StandardCharsets$Aliases.init(Object[]) : void : 1907
  ⊞ StandardCharsets$Aliases.init(Object[]) : void : 3459
  ⊞ StandardCharsets$Aliases.init(Object[]) : void : 3326
  ⊞ StandardCharsets$Aliases.init(Object[]) : void : 1123
```

Figure 67: Heap diff of the xalan benchmark (large input, 2 iterations) over two seconds (from 4.5 to 6.5)

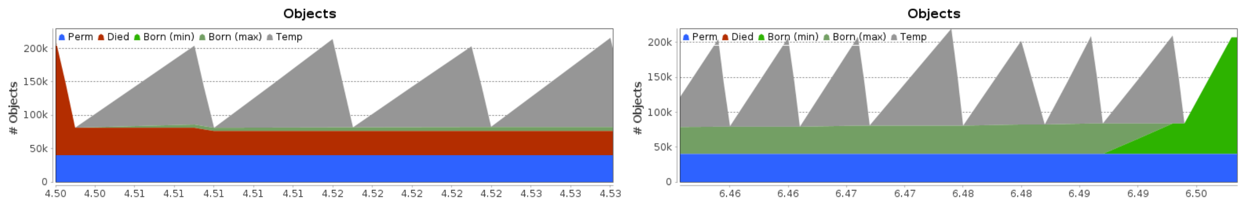
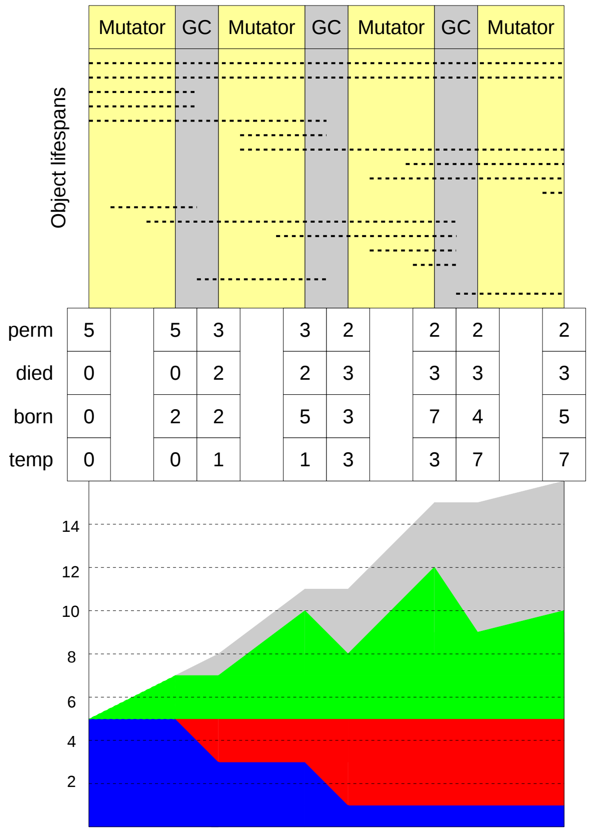


Figure 68: Heap diff of the xalan benchmark (large input, 2 iterations) over two seconds (from 4.5 to 6.5) zoomed to the very start and the very end respectively

died count for that point (2).

Similarly, we need to compute the born count and the temp count. However, these counts are more complicated to compute. In case of perm and died, the perm size can only drop, forcing the died size to rise by the same amount. The born size, however, may fall as well as rise. To make the computation easier, we calculate the allocations and deallocations in an intermediate step. We can compute the deallocations directly, based on the rise of the temp size (3). Then, we can compute the allocations indirectly by considering



perm	5	5	3	3	2	2	2	2
died	0	0	2	2	3	3	3	3
born	0	2	2	5	3	7	4	5
temp	0	0	1	1	3	3	7	7

Figure 69: Diffing example with alternating mutator phases and GC phases, object lifespans, sampled set sizes during iterative diff computation and the resulting chart

the deallocations and the change in the born size (4). We cannot compute the allocations directly because there might be allocations as well as deallocations at the same time. Assuming that the born count is zero, we can compute an intermediate temp count ($temp_{noborn}$) based on the allocations and deallocations (6).

At this point, the sum of the perm count, the died count, the born count, and the temp count already represents the exact amount of objects at every point i. However, we assumed the born count to be always zero as described above because we do not know exactly when a born object was allocated. Born objects could be allocated very early or very late, the only value we know that is correct is the born count of the latest sample (i.e., at the interval end) (6). The only thing we can do, without any concept of object identity, is computing the earliest point (born max count) and the latest point (born min count) the born objects are allocated. To compute the born min count, we use the amount of allocations we need to execute to get the final born count. The born min count can then be computed as the minimum of the current allocations, the sum of all allocations already assigned to the born min count subsequently, i.e., how many born objects are still missing, and the temp counts until the end is reached (we cannot reassign more temp objects than how many there are in the first place) (7). In other words, we run from the end to the start and reassign as many allocations as possible from the temp count to the born min count until the final born count for the interval end is fully satisfied. Finally, we can compute the born max count by doing the same thing from left to right. We start by reassigning allocations from the temp count to the born max count (9). We only have to be careful not to assign too many because the heap may shrink later. Also, we subtract born min from born max and adjust the temp count accordingly to satisfy all invariants and because born max is actually a subset of born min (10).

$$(1)\ died_i = died_n^{set} - died_i^{set}$$

$$(2)\ perm_i = perm_i^{set} - died_i$$

$$(3)\ deallocs_i = temp_i^{set} - temp_{i-1}^{set}$$

$$(4)\ allocs_i = born_i^{set} + deallocs_i - born_{i-1}^{set}$$

$$(5)\ temp_i^{no\ born} = temp_{i-1}^{set} + allocs_i - deallocs_i$$

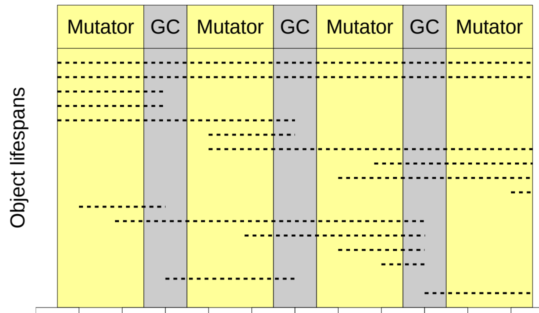
$$(6)\ born_n^{min} = born_n^{set}$$

$$(7)\ born_i^{min} = born_{i+1}^{min} - min \begin{cases} allocs_{i+1} \\ born_n - \sum_{j=i+1}^{n}(born_{j+1}^{min} - born_j^{min}) \\ temp_i^{no\ born} \\ temp_{i+1}^{no\ born} \\ \ldots \\ temp_n^{no\ born} \end{cases}$$

$$(8)\ temp'_i = temp_i^{no\ born} - born_i^{min}$$

$$(9)\ born_i^{max} = min \begin{cases} born_n^{set} \\ temp'_i + born_i^{min} \\ temp'_{i+1} + born_{i+1}^{min} \\ \ldots \\ temp'_n + born_n^{min} \end{cases} - born_i^{min}$$

$$(10)\ temp_i = temp'_i - born_i^{max}$$



$perm^{set}$	5		5	3		3	2		2	2		2
$died^{set}$	0		0	2		2	3		3	3		3
$born^{set}$	0		2	2		5	3		7	4		5
$temp^{set}$	0		0	1		1	3		3	7		7
allocs	0		2	1		3	0		4	1		1
deallocs	0		0	1		0	2		0	4		0
perm	2		2	2		2	2		2	2		2
died	3		3	1		1	0		0	0		0
$temp^{no\ born}$	0		2	2		5	3		7	4		5
$born^{min}$	0		0	0		0	0		3	4		5
temp'	0		2	2		5	3		4	0		0
$born^{max}$	0		2	2		3	3		1	0		0
temp	0		0	0		2	0		3	0		0

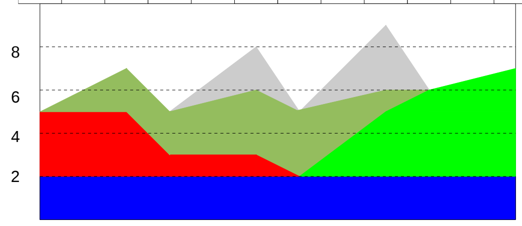


Figure 70: Example showing how to compute perm, died, born max, born min, and temp counts based on the original set sizes (denoted by the superscript *set*)

Chapter 7

Functional Evaluation

"To err is human, but to really foul things up you need a computer." - Paul R. Ehrlich

This chapter shows several real-world memory-related performance problems, how they can be detected as well as how they can be resolved using the techniques presented in this thesis. The first example shows a memory leak in the beta version of the the well-known DaCapo benchmark suite. The second example shows unnecessarily large memory consumption in the AntTracks tool. Finally, the third example shows an unexpectedly high GC ratio due to a large number of unnecessary temporary objects.

7.1 Memory Leak

This example shows a memory leak in the beta version (beta0) of the tradebeans and tradesoap benchmark[1]. As this problem has already been detected and fixed, this example is ideal to determine whether the AntTracks tool can find it as well.

To reproduce the memory leak, we ran the tradesoap benchmark continuously with the small input size. Also, to trigger an `OutOfMemoryError` early, we limited the size of the heap to 100 MB, which is more than enough to execute the fixed version of the benchmark. Running the benchmark with that heap limit, produced an `OutOfMemoryError` within 14 iterations, where the last complete iteration took already twice the time of the previous iteration.

Tracing this memory leak, AntTracks produced an overview of the memory behavior shown in Figure 71. We can easily tell that there must be a memory leak somewhere based on the data presented by the four plots. The top two, showing the number of objects and bytes per space, tell us that the heap gets almost entirely full. At every garbage collection, the GC is able to free less memory than before. In the second half of the program execution, the GC is even unable to evacuate the entire Eden space. The third plot, showing the GC pauses, presents us with the fact that the GC has even given up on using minor collections but

[1]Memory leak in the tradebeans and tradesoap benchmark: https://sourceforge.net/p/dacapobench/bugs/9/

only performs major collections (introducing at least 100ms pauses) in a desperate attempt to free memory. Finally, the fourth plot, showing the amount of surviving and dying objects, also tells us that the number of surviving objects rises and the number of dying objects falls as a function of time. Thus, using these four out-of-the-box overview plots, we can easily diagnose memory leaks.

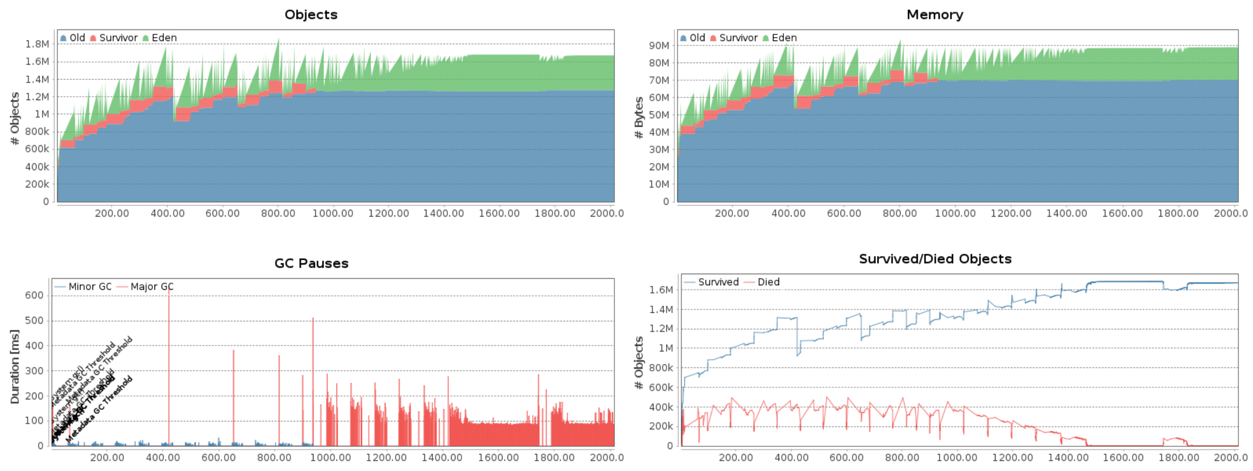


Figure 71: Overview of the memory behavior and the GC behavior of the tradesoap benchmark beta version with memory leak (small input, continuous)

Diagnosing the memory leak is only half the work. We now also need to find the root cause of the problem to be able to properly repair it. As a first step, we want to find the feature that leaks memory. Since the source code and the development teams are often organized based on different features of an application, associating a problem with a feature helps in narrowing down its location in the code as well as finding the best-suited person to fix it. Figure 72 shows the memory behavior split into four features that have been defined to be the four major packages that allocate objects. The bottom four plots show the memory usage for every individual feature. These plots show us that we can definitely exclude the *derby* feature, and probably also the *activemq* feature. We can exclude *derby* because this feature shows a constant-ish memory usage after some initialization period. The *activemq* feature rises, but also drops in the end, indicating that objects allocated by that feature will eventually be garbage collected. The *others* feature constantly rises, but this feature only includes objects allocated by code that is used by other features. However, the remaining features, i.e., *geronimo* and *axis*, show a constant rise in memory usage, indicating that the memory leak is located within one of them.

The next step in narrowing down the memory leak is associating it with specific objects, or at least with properties that all objects involved in the memory leak have in common. Thus, we used the heap diffing mechanism to look at the changes of the heap over time from about 400 seconds after start up to the very end of the application's life time. The plots in Figure 73 show us what we already expected, namely that new objects are allocated that cannot be deallocated. This clearly confirms the memory leak. Looking at the types of the objects that are currently in the heap, we see many `HashMap$Node`s, `String`s, and `char[]` objects that have been allocated by the `geronimo` package. Examining the callers of the method that allocated most of the `HashMap$Node` shows us that these callers are indeed part of the geronimo package. The same is true for the other types.

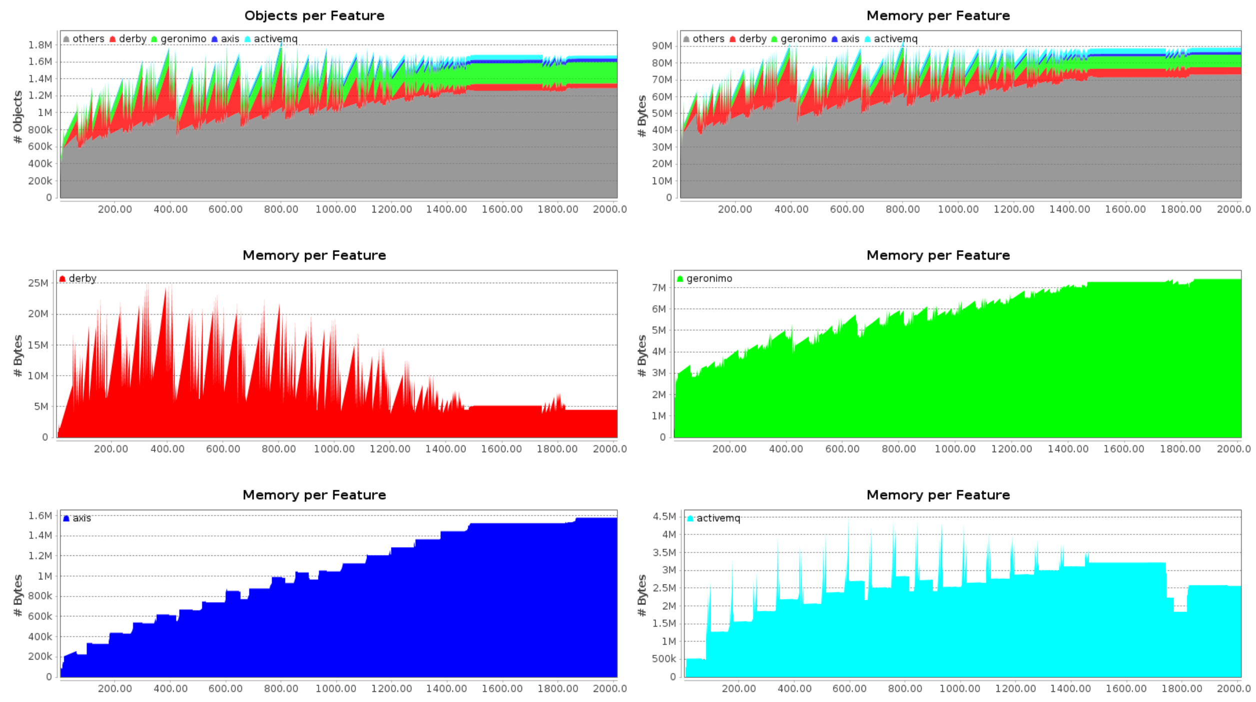
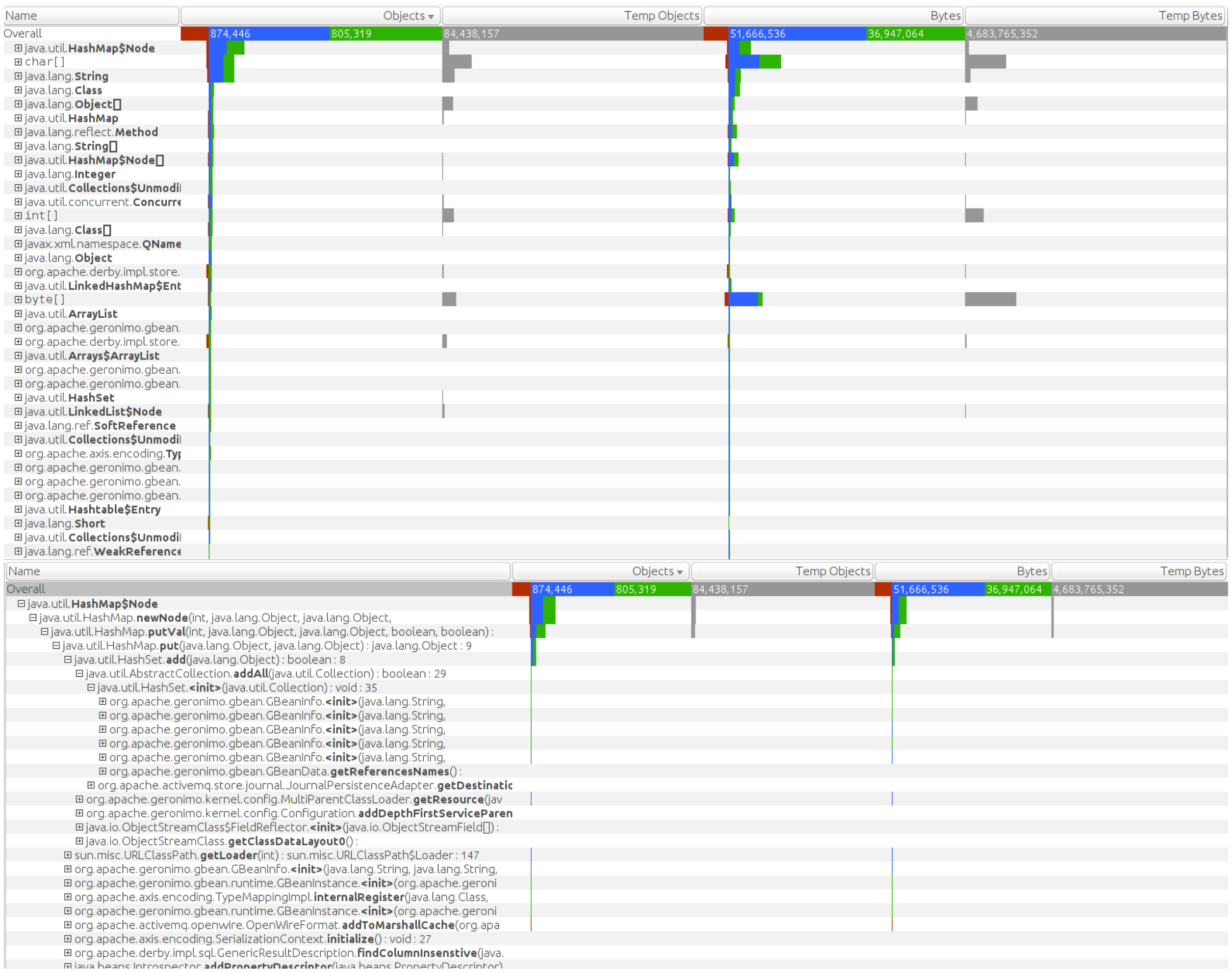


Figure 72: Overview of the feature-based memory behavior of the tradesoap benchmark beta version with memory leak (small input, continuous)

To further drill down to the root cause, we classified by type, then by types and allocation sites pointed to, and finally by allocation sites (cf. Figure 74). Please note that the *points to* and the *pointed to* classifiers contains subclassifiers, e.g., classifying all pointed to objects by type. These subclassifications are prefixed with a dash (-). The *allocation site* classifier after the *pointed from* classifier classifies the original objects (please note the different coloring and ratios in the columns on the right). Thus, the expanded elements show the allocation sites of all `Strings`, that are pointed to by `HashMap$Nodes` that have been allocated mostly by the `add` call, which is called by classes of the `geronimo` package.

When we looked up the fix for the leak we learned that the `geronimo` package was indeed keeping `String` objects (and consequently also `char[]` objects) needlessly by adding but not removing them in a `HashMap`.

While diagnosing the existence of a memory leak is possible with state-of-the-art tools, narrowing it down to specific objects is not as easy. Sampling-based approaches are only able to determine the types of objects that are part of the memory leak. They are however unable to determine the origin of the objects, i.e., their allocation sites. State-of-the-art instrumentation-based approaches are able to determine the object's origins, but they are impractical in terms of run-time overhead. Furthermore, when using instrumentation without deallocation detection, these approaches cannot even determine the existence of a memory leak because they cannot narrow down what objects constitute the leak. Also, using deallocation detection distorts the memory behavior in a way that the memory leak might be hidden in the introduced overhead.



```
Name                                  Objects ▾    Temp Objects    Bytes    Temp Bytes
Overall                   874,446  805,319  84,438,157  51,666,536  36,947,064  4,683,765,352
⊞ java.util.HashMap$Node
⊞ char[]
⊞ java.lang.String
⊞ java.lang.Class
⊞ java.lang.Object[]
⊞ java.util.HashMap
⊞ java.lang.reflect.Method
⊞ java.lang.String[]
⊞ java.util.HashMap$Node[]
⊞ java.lang.Integer
⊞ java.util.Collections$Unmodi
⊞ java.util.concurrent.Concurre
⊞ int[]
⊞ java.lang.Class[]
⊞ javax.xml.namespace.QName
⊞ java.lang.Object
⊞ org.apache.derby.impl.store.
⊞ java.util.LinkedHashMap$Ent
⊞ byte[]
⊞ java.util.ArrayList
⊞ org.apache.geronimo.gbean.
⊞ org.apache.derby.impl.store.
⊞ java.util.Arrays$ArrayList
⊞ org.apache.geronimo.gbean.
⊞ org.apache.geronimo.gbean.
⊞ java.util.HashSet
⊞ java.util.LinkedList$Node
⊞ java.lang.ref.SoftReference
⊞ java.util.Collections$Unmodi
⊞ org.apache.axis.encoding.Typ
⊞ org.apache.geronimo.gbean.
⊞ org.apache.geronimo.gbean.
⊞ org.apache.geronimo.gbean.
⊞ java.util.Hashtable$Entry
⊞ java.lang.Short
⊞ java.util.Collections$Unmodi
⊞ java.lang.ref.WeakReference
```

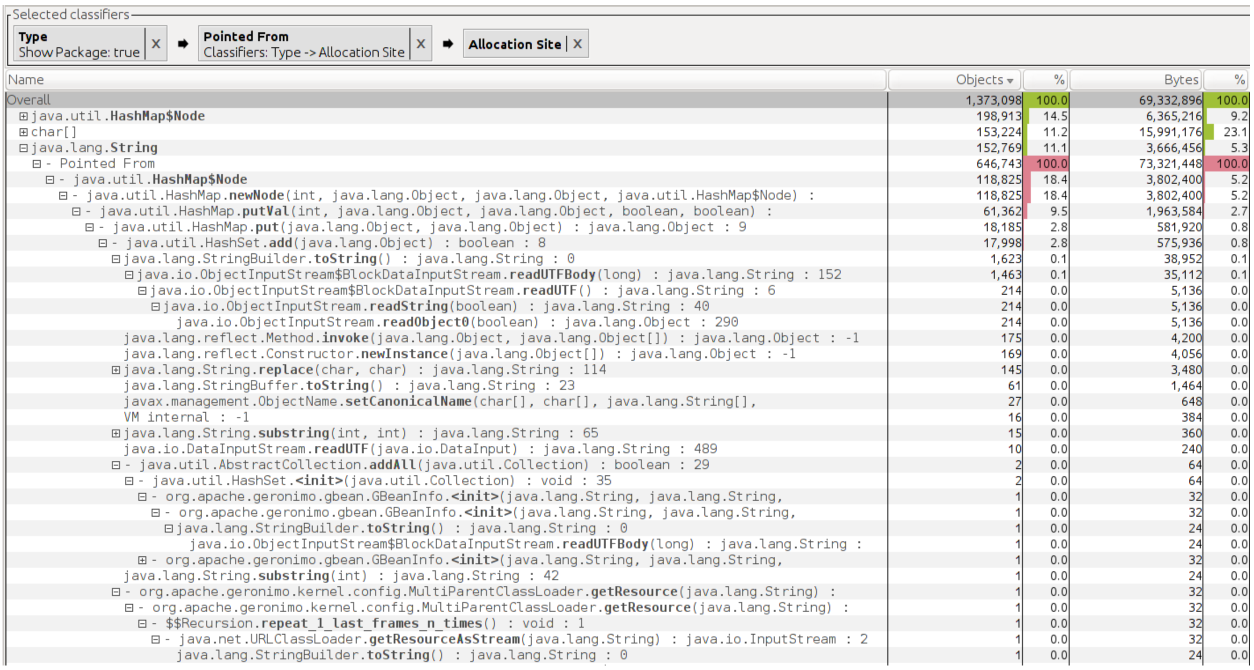
```
Name                                  Objects ▾    Temp Objects    Bytes    Temp Bytes
Overall                   874,446  805,319  84,438,157  51,666,536  36,947,064  4,683,765,352
  ⊟ java.util.HashMap$Node
    ⊟ java.util.HashMap.newNode(int, java.lang.Object, java.lang.Object,
      ⊟ java.util.HashMap.putVal(int, java.lang.Object, java.lang.Object, boolean, boolean) :
        ⊟ java.util.HashMap.put(java.lang.Object, java.lang.Object) : java.lang.Object : 9
          ⊟ java.util.HashSet.add(java.lang.Object) : boolean : 8
            ⊟ java.util.AbstractCollection.addAll(java.util.Collection) : boolean : 29
              ⊟ java.util.HashSet.<init>(java.util.Collection) : void : 35
                ⊞ org.apache.geronimo.gbean.GBeanInfo.<init>(java.lang.String,
                ⊞ org.apache.geronimo.gbean.GBeanInfo.<init>(java.lang.String,
                ⊞ org.apache.geronimo.gbean.GBeanInfo.<init>(java.lang.String,
                ⊞ org.apache.geronimo.gbean.GBeanInfo.<init>(java.lang.String,
                ⊞ org.apache.geronimo.gbean.GBeanInfo.<init>(java.lang.String,
                ⊞ org.apache.geronimo.gbean.GBeanData.getReferencesNames() :
              ⊞ org.apache.activemq.store.journal.JournalPersistenceAdapter.getDestinatic
            ⊞ org.apache.geronimo.kernel.config.MultiParentClassLoader.getResource(jav
            ⊞ org.apache.geronimo.kernel.config.Configuration.addDepthFirstServiceParen
            ⊞ java.io.ObjectStreamClass$FieldReflector.<init>(java.io.ObjectStreamField[]) :
            ⊞ java.io.ObjectStreamClass.getClassDataLayout0() :
          ⊞ sun.misc.URLClassPath.getLoader(int) : sun.misc.URLClassPath$Loader : 147
          ⊞ org.apache.geronimo.gbean.GBeanInfo.<init>(java.lang.String, java.lang.String,
          ⊞ org.apache.geronimo.gbean.runtime.GBeanInstance.<init>(org.apache.geroni
          ⊞ org.apache.axis.encoding.TypeMappingImpl.internalRegister(java.lang.Class,
          ⊞ org.apache.geronimo.gbean.runtime.GBeanInstance.<init>(org.apache.geroni
          ⊞ org.apache.activemq.openwire.OpenWireFormat.addToMarshallCache(org.apa
          ⊞ org.apache.axis.encoding.SerializationContext.initialize() : void : 27
          ⊞ org.apache.derby.impl.sql.GenericResultDescription.findColumnInsenstive(java.
          ⊞ java.beans.Introspector.addPropertyDescriptor(java.beans.PropertyDescriptor)
```

Figure 73: Diff showing the memory leak and its involved types as well as exemplary the allocation site of one type

7.2 Large Memory Consumption

Similar to memory leaks, an application may also suffer from a large unnecessary memory consumption. The difference to a memory leak is that the objects are actually used, although very inefficiently.

This example shows an unnecessarily large memory consumption in the AntTracks tool itself. We found it when we traced the AntTracks tool with a 4 GB heap while it was parsing the tradesoap memory leak described above. Figure 75 shows the memory behavior of the AntTracks tool in this case. We can see a rather high overall heap usage, many GC pauses, and frequent major collections. Examining the number of objects per feature, we see that there are up to 50 million objects just representing the heap. In terms of memory, they need almost 2 GB.



Selected classifiers

Type Show Package: true ➡ **Pointed From** Classifiers: Type -> Allocation Site ➡ **Allocation Site**

Name	Objects	%	Bytes	%
Overall	1,373,098	100.0	69,332,896	100.0
⊞ java.util.**HashMap$Node**	198,913	14.5	6,365,216	9.2
⊞ char[]	153,224	11.2	15,991,176	23.1
⊟ java.lang.**String**	152,769	11.1	3,666,456	5.3
⊟ - Pointed From	646,743	100.0	73,321,448	100.0
⊟ - java.util.**HashMap$Node**	118,825	18.4	3,802,400	5.2
⊟ - java.util.HashMap.**newNode**(int, java.lang.Object, java.lang.Object, java.util.HashMap$Node) :	118,825	18.4	3,802,400	5.2
⊟ - java.util.HashMap.**putVal**(int, java.lang.Object, java.lang.Object, boolean, boolean) :	61,362	9.5	1,963,584	2.7
⊟ - java.util.HashMap.**put**(java.lang.Object, java.lang.Object) : java.lang.Object : 9	18,185	2.8	581,920	0.8
⊟ - java.util.HashSet.**add**(java.lang.Object) : boolean : 8	17,998	2.8	575,936	0.8
⊟ java.lang.StringBuilder.**toString**() : java.lang.String : 0	1,623	0.1	38,952	0.1
⊟ java.io.ObjectInputStream$BlockDataInputStream.**readUTFBody**(long) : java.lang.String : 152	1,463	0.1	35,112	0.1
⊟ java.io.ObjectInputStream$BlockDataInputStream.**readUTF**() : java.lang.String : 6	214	0.0	5,136	0.0
⊟ java.io.ObjectInputStream.**readString**(boolean) : java.lang.String : 40	214	0.0	5,136	0.0
java.io.ObjectInputStream.**readObject0**(boolean) : java.lang.Object : 290	214	0.0	5,136	0.0
java.lang.reflect.Method.**invoke**(java.lang.Object, java.lang.Object[]) : java.lang.Object : -1	175	0.0	4,200	0.0
java.lang.reflect.Constructor.**newInstance**(java.lang.Object[]) : java.lang.Object : -1	169	0.0	4,056	0.0
⊞ java.lang.String.**replace**(char, char) : java.lang.String : 114	145	0.0	3,480	0.0
java.lang.StringBuffer.**toString**() : java.lang.String : 23	61	0.0	1,464	0.0
javax.management.ObjectName.**setCanonicalName**(char[], char[], java.lang.String[],	27	0.0	648	0.0
VM internal : -1	16	0.0	384	0.0
⊞ java.lang.String.**substring**(int, int) : java.lang.String : 65	15	0.0	360	0.0
java.io.DataInputStream.**readUTF**(java.io.DataInput) : java.lang.String : 489	10	0.0	240	0.0
⊟ - java.util.AbstractCollection.**addAll**(java.util.Collection) : boolean : 29	2	0.0	64	0.0
⊟ - java.util.HashSet.**<init>**(java.util.Collection) : void : 35	2	0.0	64	0.0
⊟ - org.apache.geronimo.gbean.GBeanInfo.**<init>**(java.lang.String, java.lang.String,	1	0.0	32	0.0
⊟ - org.apache.geronimo.gbean.GBeanInfo.**<init>**(java.lang.String, java.lang.String,	1	0.0	32	0.0
⊟ java.lang.StringBuilder.**toString**() : java.lang.String : 0	1	0.0	24	0.0
java.io.ObjectInputStream$BlockDataInputStream.**readUTFBody**(long) : java.lang.String :	1	0.0	24	0.0
⊞ - org.apache.geronimo.gbean.GBeanInfo.**<init>**(java.lang.String, java.lang.String,	1	0.0	32	0.0
java.lang.String.**substring**(int) : java.lang.String : 42	1	0.0	24	0.0
⊟ - org.apache.geronimo.kernel.config.MultiParentClassLoader.**getResource**(java.lang.String) :	1	0.0	32	0.0
⊟ - org.apache.geronimo.kernel.config.MultiParentClassLoader.**getResource**(java.lang.String) :	1	0.0	32	0.0
⊟ - $$Recursion.**repeat_1_last_frames_n_times**() : void : 1	1	0.0	32	0.0
⊟ - java.net.URLClassLoader.**getResourceAsStream**(java.lang.String) : java.io.InputStream : 2	1	0.0	32	0.0
java.lang.StringBuilder.**toString**() : java.lang.String : 0	1	0.0	24	0.0

Figure 74: Heap overview showing the memory leak and what types are keeping objects alive

Since the amount of objects representing the heap as well as the frequency of major GCs seemed rather high, we investigated further and examined the heap approximately 2000 seconds after the start. Figure 76 shows the heap objects at that point in time, classified by type and by allocation site. The top four types in the heap are `Long`, `ConcurrentSkipList$Node`, `SingleObjectLab`, and `ConcurrentSkipListMap$Index`, together making up almost 77% of all objects and almost 68% of the used memory. Looking at their allocation sites shows that they are all allocated while parsing *GC keep alive* events. The implementation at that time allocates a LAB for just one object and inserts it into the heap. This results in the exorbitant number of objects described above.

Our next step was to determine why so many *GC keep alive* events are fired in the first place. We found out that this is because while parsing the trace containing a memory leak, most objects do not move because the GC can hardly free any objects. This causes the old space to be filled up with live objects that cannot be moved further to the front so that a large amount of *GC keep alive* events are being fired for them. To solve this problem, we chose to optimize the tracing of this dense prefix of the old space in the AntTracks VM by also firing *GC move region* events rather than *GC keep alive* events.

We reran the memory leaking benchmark again with the adjusted AntTracks VM and traced the parsing of the new trace. Figure 77 shows the new memory behavior. We can easily see that the memory consumption is much lower (please note the different scaling). Instead of about 100 million objects at a time, we now need only about 25 million, and instead of about 3.5 GB of memory we now need only 2.5 GB. Also we have much shorter GC pauses and much fewer major GCs. Looking at the memory behavior per feature, we also see the drastically reduced activity in the *heap* feature.

Like in the case of memory leaks, sampling-based approaches would only be able to determine the types of

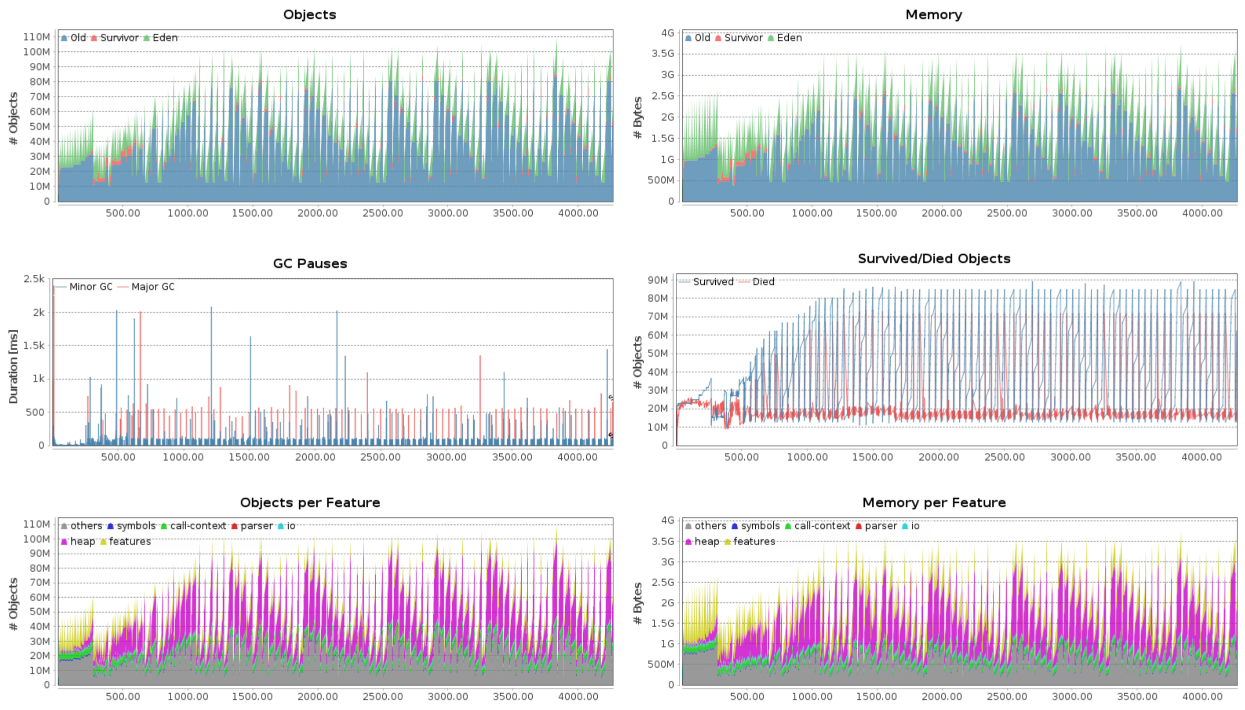


Figure 75: Memory behavior of the AntTracks analysis tool parsing through the memory leaking tradesoap benchmark

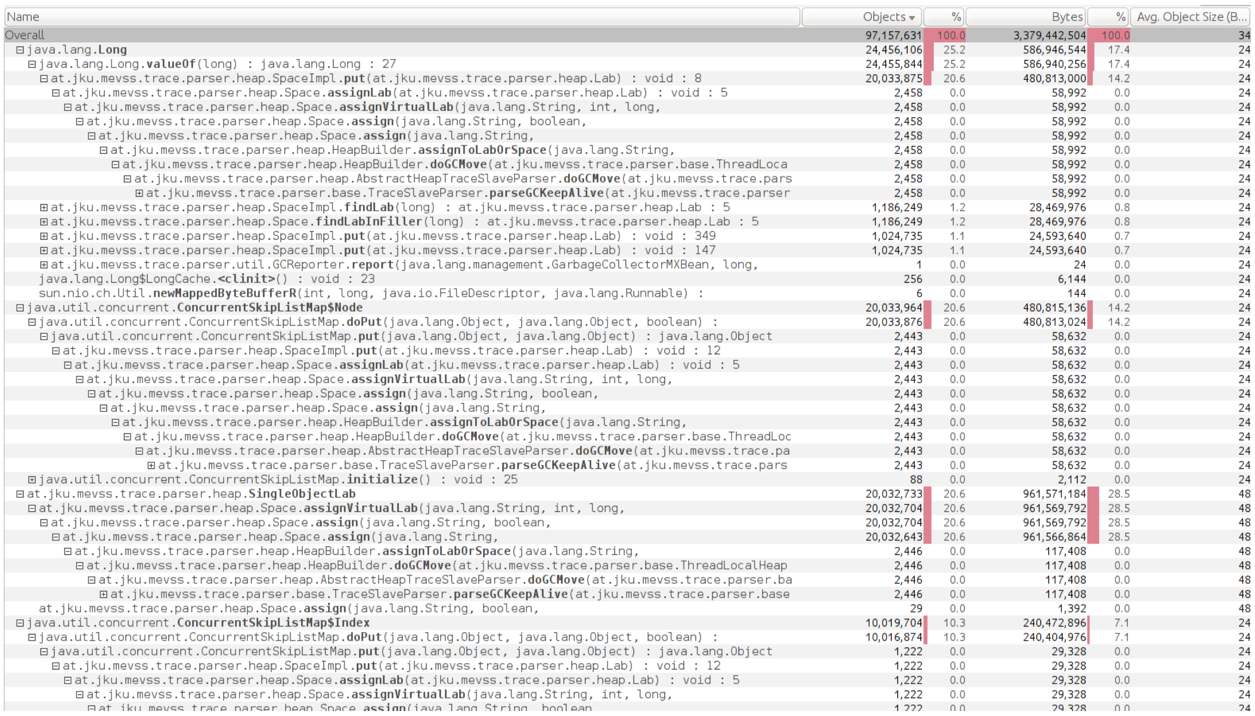
the objects in the heap. State-of-the-art instrumention-based approaches could also determine the allocation site as well as the callers. However, walking the stack from outside the VM imposes significant overhead because the entire VM must be suspended. This makes such tools impractical for production use.

7.3 Long GC Times

Another class of memory-related performance problems are long GC times. The following example is based on the previous section, i.e., the already fixed AntTracks VM in combination with the AntTracks analysis tool. In the last plot of Figure 77 we saw that the tool still shows some rather long GC times. Although there are lots of GCs that are very quick, some take up to two seconds.

Looking at the objects per feature and the memory per feature plots in Figure 77 we see that most of the heap is made up by the *feature* feature, i.e., the feature that is responsible for assigning objects to features. We can also see, that, for in contrast to the *call-context* or the io feature, the objects assigned to the *feature* feature can be collected. Thus, they might be responsible for the long GC pauses.

To further investigate the problem, we created a heap diff over 3 seconds in the middle of the application's life time (cf. Figure 78). We grouped the objects by feature (due to our suspicion above), then by type, and finally per allocation site. The diff shows that almost the entire heap is made up of permanent objects (mostly in the *heap*, and the *call-context* features), and of temporary objects (mostly in the *feature* feature. These



Name	Objects ▾	%	Bytes	%	Avg. Object Size (B...
Overall	97,157,631	100.0	3,379,442,504	100.0	34
⊟ java.lang.**Long**	24,456,106	25.2	586,946,544	17.4	24
⊟ java.lang.Long.**valueOf**(long) : java.lang.Long : 27	24,455,844	25.2	586,940,256	17.4	24
⊟ at.jku.mevss.trace.parser.heap.SpaceImpl.**put**(at.jku.mevss.trace.parser.heap.Lab) : void : 8	20,033,875	20.6	480,813,000	14.2	24
⊟ at.jku.mevss.trace.parser.heap.Space.**assignLab**(at.jku.mevss.trace.parser.heap.Lab) : void : 5	2,458	0.0	58,992	0.0	24
⊟ at.jku.mevss.trace.parser.heap.Space.**assignVirtualLab**(java.lang.String, int, long,	2,458	0.0	58,992	0.0	24
⊟ at.jku.mevss.trace.parser.heap.Space.**assign**(java.lang.String, boolean,	2,458	0.0	58,992	0.0	24
⊟ at.jku.mevss.trace.parser.heap.Space.**assign**(java.lang.String,	2,458	0.0	58,992	0.0	24
⊟ at.jku.mevss.trace.parser.heap.HeapBuilder.**assignToLabOrSpace**(java.lang.String,	2,458	0.0	58,992	0.0	24
⊟ at.jku.mevss.trace.parser.heap.HeapBuilder.**doGCMove**(at.jku.mevss.trace.parser.base.ThreadLoca	2,458	0.0	58,992	0.0	24
⊟ at.jku.mevss.trace.parser.heap.AbstractHeapTraceSlaveParser.**doGCMove**(at.jku.mevss.trace.pars	2,458	0.0	58,992	0.0	24
⊞ at.jku.mevss.trace.parser.base.TraceSlaveParser.**parseGCKeepAlive**(at.jku.mevss.trace.parser	2,458	0.0	58,992	0.0	24
⊞ at.jku.mevss.trace.parser.heap.SpaceImpl.**findLab**(long) : at.jku.mevss.trace.parser.heap.Lab : 5	1,186,249	1.2	28,469,976	0.8	24
⊞ at.jku.mevss.trace.parser.heap.Space.**findLabInFiller**(long) : at.jku.mevss.trace.parser.heap.Lab : 5	1,186,249	1.2	28,469,976	0.8	24
⊞ at.jku.mevss.trace.parser.heap.SpaceImpl.**put**(at.jku.mevss.trace.parser.heap.Lab) : void : 349	1,024,735	1.1	24,593,640	0.7	24
⊞ at.jku.mevss.trace.parser.heap.SpaceImpl.**put**(at.jku.mevss.trace.parser.heap.Lab) : void : 147	1,024,735	1.1	24,593,640	0.7	24
⊞ at.jku.mevss.trace.parser.util.GCReporter.**report**(java.lang.management.GarbageCollectorMXBean, long,	1	0.0	24	0.0	24
java.lang.Long$LongCache.**<clinit>**() : void : 23	256	0.0	6,144	0.0	24
sun.nio.ch.Util.**newMappedByteBufferR**(int, long, java.io.FileDescriptor, java.lang.Runnable) :	6	0.0	144	0.0	24
⊟ java.util.concurrent.**ConcurrentSkipListMap$Node**	20,033,964	20.6	480,815,136	14.2	24
⊟ java.util.concurrent.ConcurrentSkipListMap.**doPut**(java.lang.Object, java.lang.Object, boolean) :	20,033,876	20.6	480,813,024	14.2	24
⊟ java.util.concurrent.ConcurrentSkipListMap.**put**(java.lang.Object, java.lang.Object) : java.lang.Object	2,443	0.0	58,632	0.0	24
⊟ at.jku.mevss.trace.parser.heap.SpaceImpl.**put**(at.jku.mevss.trace.parser.heap.Lab) : void : 12	2,443	0.0	58,632	0.0	24
⊟ at.jku.mevss.trace.parser.heap.Space.**assignLab**(at.jku.mevss.trace.parser.heap.Lab) : void : 5	2,443	0.0	58,632	0.0	24
⊟ at.jku.mevss.trace.parser.heap.Space.**assignVirtualLab**(java.lang.String, int, long,	2,443	0.0	58,632	0.0	24
⊟ at.jku.mevss.trace.parser.heap.Space.**assign**(java.lang.String, boolean,	2,443	0.0	58,632	0.0	24
⊟ at.jku.mevss.trace.parser.heap.Space.**assign**(java.lang.String,	2,443	0.0	58,632	0.0	24
⊟ at.jku.mevss.trace.parser.heap.HeapBuilder.**assignToLabOrSpace**(java.lang.String,	2,443	0.0	58,632	0.0	24
⊟ at.jku.mevss.trace.parser.heap.HeapBuilder.**doGCMove**(at.jku.mevss.trace.parser.base.ThreadLoc	2,443	0.0	58,632	0.0	24
⊟ at.jku.mevss.trace.parser.heap.AbstractHeapTraceSlaveParser.**doGCMove**(at.jku.mevss.trace.pa	2,443	0.0	58,632	0.0	24
⊞ at.jku.mevss.trace.parser.base.TraceSlaveParser.**parseGCKeepAlive**(at.jku.mevss.trace.pars	2,443	0.0	58,632	0.0	24
⊞ java.util.concurrent.ConcurrentSkipListMap.**initialize**() : void : 25	88	0.0	2,112	0.0	24
⊟ at.jku.mevss.trace.parser.heap.**SingleObjectLab**	20,032,733	20.6	961,571,184	28.5	48
⊟ at.jku.mevss.trace.parser.heap.Space.**assignVirtualLab**(java.lang.String, int, long,	20,032,704	20.6	961,569,792	28.5	48
⊟ at.jku.mevss.trace.parser.heap.Space.**assign**(java.lang.String, boolean,	20,032,704	20.6	961,569,792	28.5	48
⊟ at.jku.mevss.trace.parser.heap.Space.**assign**(java.lang.String,	20,032,643	20.6	961,566,864	28.5	48
⊟ at.jku.mevss.trace.parser.heap.HeapBuilder.**assignToLabOrSpace**(java.lang.String,	2,446	0.0	117,408	0.0	48
⊟ at.jku.mevss.trace.parser.heap.HeapBuilder.**doGCMove**(at.jku.mevss.trace.parser.base.ThreadLocalHeap	2,446	0.0	117,408	0.0	48
⊟ at.jku.mevss.trace.parser.heap.AbstractHeapTraceSlaveParser.**doGCMove**(at.jku.mevss.trace.parser.ba	2,446	0.0	117,408	0.0	48
⊞ at.jku.mevss.trace.parser.base.TraceSlaveParser.**parseGCKeepAlive**(at.jku.mevss.trace.parser.base	2,446	0.0	117,408	0.0	48
at.jku.mevss.trace.parser.heap.Space.**assign**(java.lang.String, boolean,	29	0.0	1,392	0.0	48
⊟ java.util.concurrent.**ConcurrentSkipListMap$Index**	10,019,704	10.3	240,472,896	7.1	24
⊟ java.util.concurrent.ConcurrentSkipListMap.**doPut**(java.lang.Object, java.lang.Object, boolean) :	10,016,874	10.3	240,404,976	7.1	24
⊟ java.util.concurrent.ConcurrentSkipListMap.**put**(java.lang.Object, java.lang.Object) : java.lang.Object	1,222	0.0	29,328	0.0	24
⊟ at.jku.mevss.trace.parser.heap.SpaceImpl.**put**(at.jku.mevss.trace.parser.heap.Lab) : void : 12	1,222	0.0	29,328	0.0	24
⊟ at.jku.mevss.trace.parser.heap.Space.**assignLab**(at.jku.mevss.trace.parser.heap.Lab) : void : 5	1,222	0.0	29,328	0.0	24
⊟ at.jku.mevss.trace.parser.heap.Space.**assignVirtualLab**(java.lang.String, int, long,	1,222	0.0	29,328	0.0	24
⊟ at.jku.mevss.trace.parser.heap.Space.**assign**(java.lang.String, boolean,	1,222	0.0	29,328	0.0	24

Figure 76: Heap state at a GC spike of the AntTracks analysis tool parsing through the memory leaking tradesoap benchmark

temporary objects all originate (either directly or indirectly) from the `FeatureMap.match(...)` method.

A closer examination of the implementation showed that every feature is internally denoted by an ID (i.e., an `int`). The `match` method is called for every object and is supposed to return an `int[]` containing all feature IDs the object can be associated with. To do this, the method walks the call chain, splits the callers' signatures into `String`s, denoting the caller's package, type, method name, and method signature, and finally creates the `int[]`. Therefore, the `match` method allocates an `int[]` as well as several `char[]` and `String` objects for every object that is allocated or otherwise seen while iterating through the heap.

We optimized this method by introducing a feature cache. What features an object is assigned to depends on the call chain. As there is only a limited number of call chains that are all defined in the symbols file, we can cache the features for every possible call chain.

Figure 79 shows the memory behavior with the new feature cache in place. We see that the memory usage is drastically reduced from 2.5 GB to about 1.7 GB. Interestingly, the heap can now contain more, but smaller objects, showing an increase from 35 million to 60 million objects. However, looking at the GC pauses, we see that after starting up, we do not need any major GC at all. Please note, that the figure shows again the entire execution, the interval is just different because the execution with the fixed tool is much faster. We still observe some long collections, but they are rarer as well as shorter.

Sampling-based as well as Instrumentation-based approaches would not be able to find this defect. Sampling-based approaches would not be able to detect where the allocated objects originated from, and even if they could, they could not detect whether the objects are permanent or temporary. Instrumentation-

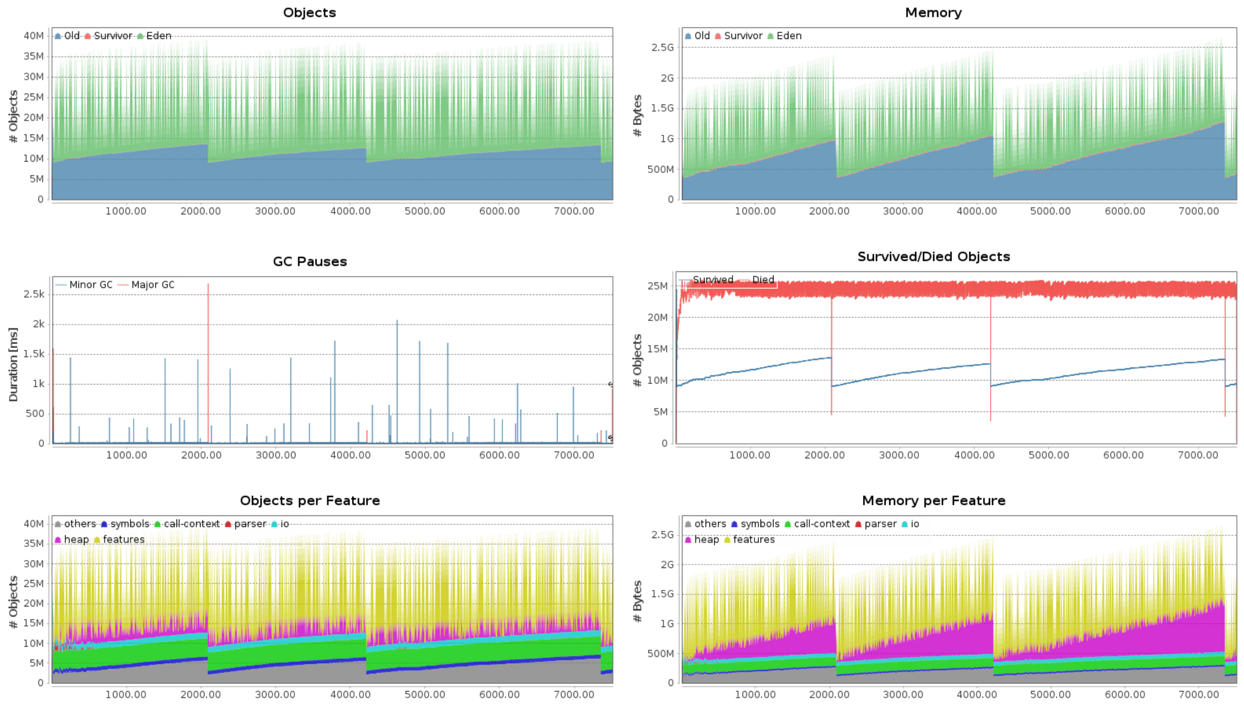
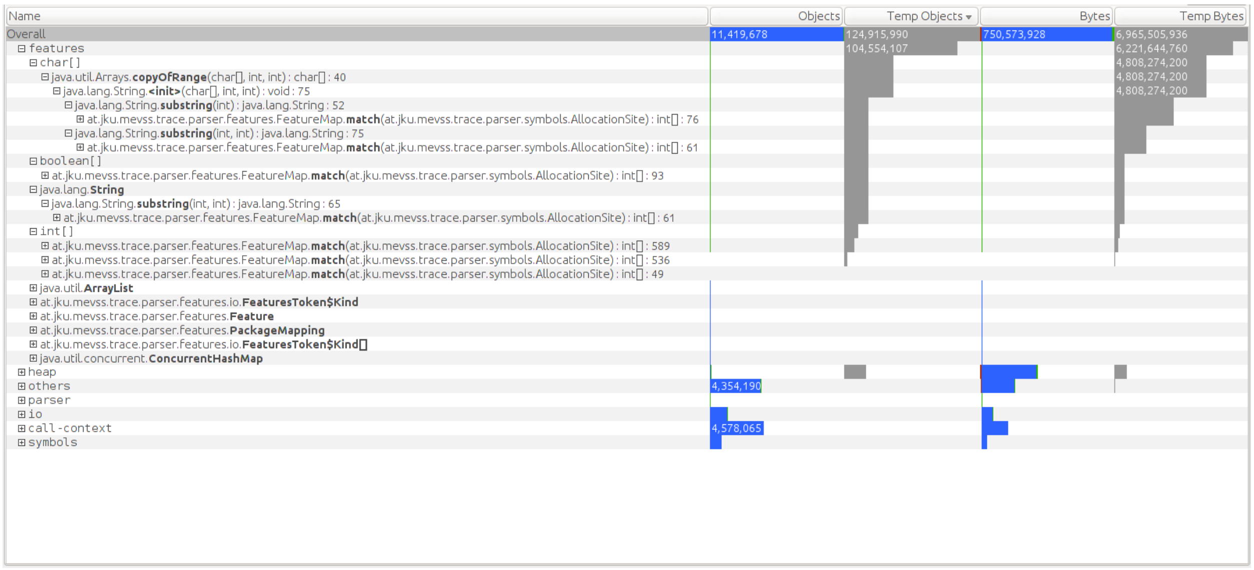


Figure 77: Memory behavior of the AntTracks Tool parsing through the tradesoap memory leak with *move region* events instead of *keep alive* events

based approaches could associate the `char[]`, `String` and `int[]` arrays with the `match()` method. However, it would be very costly because they need to suspend the VM to walk the stack. Also, they still lack the capability to tell whether these objects are temporary or not.



Name	Objects	Temp Objects ▾	Bytes	Temp Bytes
Overall	11,419,678	124,915,990	750,573,928	6,965,505,936
⊟ features		104,554,107		6,221,644,760
⊟ char[]				4,808,274,200
⊟ java.util.Arrays.**copyOfRange**(char[], int, int) : char[] : 40				4,808,274,200
⊟ java.lang.String.**<init>**(char[], int, int) : void : 75				4,808,274,200
⊟ java.lang.String.**substring**(int) : java.lang.String : 52				
⊞ at.jku.mevss.trace.parser.features.FeatureMap.**match**(at.jku.mevss.trace.parser.symbols.AllocationSite) : int[] : 76				
⊟ java.lang.String.**substring**(int, int) : java.lang.String : 75				
⊞ at.jku.mevss.trace.parser.features.FeatureMap.**match**(at.jku.mevss.trace.parser.symbols.AllocationSite) : int[] : 61				
⊟ boolean[]				
⊞ at.jku.mevss.trace.parser.features.FeatureMap.**match**(at.jku.mevss.trace.parser.symbols.AllocationSite) : int[] : 93				
⊟ java.lang.**String**				
⊟ java.lang.String.**substring**(int, int) : java.lang.String : 65				
⊞ at.jku.mevss.trace.parser.features.FeatureMap.**match**(at.jku.mevss.trace.parser.symbols.AllocationSite) : int[] : 61				
⊟ int[]				
⊞ at.jku.mevss.trace.parser.features.FeatureMap.**match**(at.jku.mevss.trace.parser.symbols.AllocationSite) : int[] : 589				
⊞ at.jku.mevss.trace.parser.features.FeatureMap.**match**(at.jku.mevss.trace.parser.symbols.AllocationSite) : int[] : 536				
⊞ at.jku.mevss.trace.parser.features.FeatureMap.**match**(at.jku.mevss.trace.parser.symbols.AllocationSite) : int[] : 49				
⊞ java.util.**ArrayList**				
⊞ at.jku.mevss.trace.parser.features.io.**FeaturesToken$Kind**				
⊞ at.jku.mevss.trace.parser.features.**Feature**				
⊞ at.jku.mevss.trace.parser.features.**PackageMapping**				
⊞ at.jku.mevss.trace.parser.features.io.**FeaturesToken$Kind[]**				
⊞ java.util.concurrent.**ConcurrentHashMap**				
⊞ heap				
⊞ others	4,354,190			
⊞ parser				
⊞ io				
⊞ call-context	4,578,065			
⊞ symbols				

Figure 78: Diff over five seconds of the AntTracks Tool parsing through the tradesoap memory leak

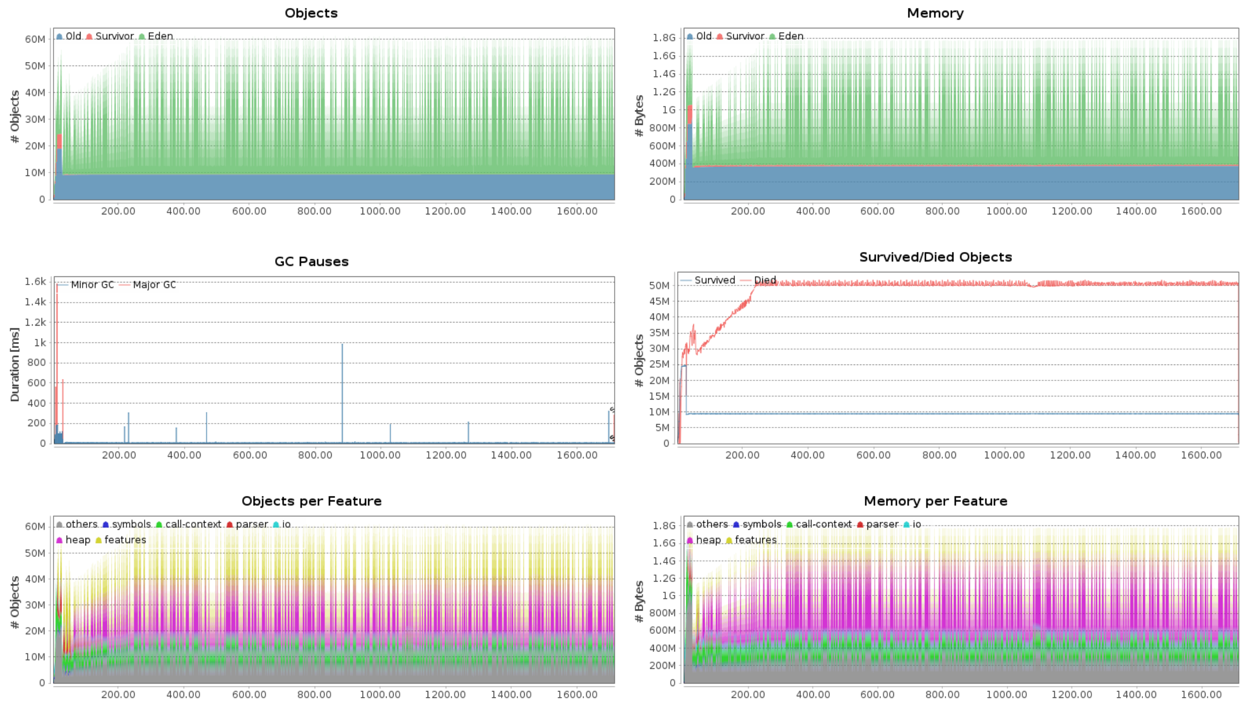


Figure 79: Memory behavior of the AntTracks analysis tool parsing through the memory leaking tradesoap benchmark

Chapter 8

Performance Evaluation

"Don't lower your expectations to meet your performance. Raise your level of performance to meet your expectations." - Ralph Marston

This section provides a detailed evaluation of the AntTracks VM and the AntTracks Tool. We present the tracing overhead on state-of-the-art benchmarks in terms of selected metrics such as run time, GC count, GC time, and memory usage. We also provide detailed measurements of the VM's internals to justify optimizations discussed in previous sections.

Methodology. We evaluated our approach with state-of-the-art methodology based on Georges et al. [17], Blackburn et al. [7], and Horký et al. [22]. When we report a single value for a metric, it is the median of 50 runs. When we report data as boxplots, we define the box as the middle 50% of the data (the line in the middle being the median), and the area between the whiskers as the middle 95% of the data.

Hardware. The evaluation machine uses an Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz with 4 cores (2 hardware threads each) with 8192 KB primary cache and 32 GB of underlying RAM. As underlying storage device it uses a Samsung SSD 850 with 512 GB storage capacity and a 6 GB/s SATA interface. According to the vendor, the internal read and write rate are 67.5 MB/s and 65 MB/s respectively.

Software. The evaluation machine runs Ubuntu 16.04 Xenial Xerus Server Edition with the Kernel 4.4.0-31-generic. AntTracks has been compiled with `g++` and `gcc` version 5.4.0 respectively and linked and assembled with `ld` and `as` version 2.16.1 respectively.

Benchmarks. We have selected three distinct benchmark suites to cover a wide variety of application behavior, i.e., the DaCapo, the DaCapo Scala, and the SPECjvm benchmark suites.

DaCapo. The DaCapo[1] benchmark suite by Blackburn et al. [6] contains 14 real-world applications including fixed workloads. Every benchmark uses multiple threads to process the given workload, making run time (lower is better) the primary metric. The suite can be configured to run the same benchmark several times (iterations) before starting measurement. These warmup iterations execute the same workload as the final measurement iteration. The suite comes with workloads of different sizes, i.e., *tiny*, *small*, *default*, *large*, and *huge*. Although every benchmark has a default workload, the other sizes are not supported for every benchmark. As we want to measure allocation-intensive behavior with big heaps, we always chose the biggest available workload.

Figure 80 shows all benchmarks, including their used workload size, the number of warmups necessary to reach a stable state, the approximate run time of the measurement iteration, and the live size. Workload sizes marked with a star (*) are not the highest available size and are discussed below. The live size is defined as the smallest heap size setting with which the benchmark is still able to execute without triggering an `OutOfMemoryError`. We determined this size by binary searching the lowest heap limit stopping at a 1KB granularity.

Name	Workload	Warmups	Run time [s]	Live size [MB]	Allocations [10^3]
avrora	large	20	6.82	7.49	6710
batik	small*	80	0.14	24.74	462
eclipse	small*	80	0.09	28.53	676
fop	default	40	0.121	29.58	2511
h2	huge	10	141.68	1300.67	388081
jython	large	20	7.19	27.94	180707
luindex	default	40	0.32	5.03	217
lusearch	large	20	1.99	2.64	21904
pmd	large	20	0.99	38.06	15660
sunflow	large	20	4.15	11.07	138674
tomcat	huge	10	0.77	17.02	177454
tradebeans	huge	10	50.54	278.43	925570
tradesoap	huge	10	35.53	110.82	944162
xalan	large	20	5.65	5.17	103069

Figure 80: DaCapo benchmarks and their vital properties

The DaCapo benchmark suite has a number of problems which must be discussed in order to better understand the results:

- batik workload: Although batik has a *large* and a *default* workload, these workloads crash on any non-closed-source Oracle JVM because they use internal classes from the `com.sun.image.codec.jpeg` package. If this package, which is not part of the JDK specification, is not available, this benchmark will fail with those workloads. Consequently, we use the *small* workload, which is the biggest working workload.

- eclipse workload: The eclipse benchmark uses a project as its workload, which will be loaded, indexed, built, and analyzed with eclipse Java IDE functionality. However, this benchmark will use the currently running Java instance as a JDK to build and analyze the project. The configured code level in the

[1]DaCapo 2009: http://www.dacapobench.org/

project is below 1.8, thus not allowing default methods in interfaces[2]. As the referenced JDK classes are Java 1.8 and thus contain default methods, this makes the eclipse benchmark fail. We therefore use only the *small* workload.

- tomcat: For some unknown reason, the tomcat benchmark does not execute properly due to an HTTP error. This occurs on all Java 8 VMs and is therefore not considered a problem we can deal with.
- tradebeans and tradesoap: The tradebeans and tradesoap benchmarks tend to fail with a 404 error if they are executed too fast one after another. This is most likely due to their heavy usage of the network loopback interface and that this interface needs some time to reset properly. Thus, when using these benchmarks, we introduce a 15 minute pause afterwards.

DaCapo Scala. The DaCapo Scala[3] benchmark suite by Sewe et al. [44] is based on the DaCapo benchmark suite. It uses the same infrastructure but adds 10 benchmarks implemented in the Scala language. As Scala is a functional language, it exhibits different allocation behavior. Also, it adds the *gargantuan* input size for some of its benchmarks. Figure 81 shows all new benchmarks, including their used workload size, the number of warmups necessary to reach a stable state, the approximate run time of the measurement iteration, and the live size.

Name	Workload	Warmups	Run time [s]	Live size [MB]	Allocations [10^3]
actors	gargantuan	5	12.33	17.02	245235
apparat	gargantuan	5	72.01	66.68	399087
factorie	gargantuan	5	82.19	558.27	5716589
kiama	default	40	0.17	25.83	11384
scalac	large	20	2.01	71.87	42252
scaladoc	large	20	1.83	68.44	38065
scalap	large	20	0.14	5.77	3454
scalariform	huge	10	1.55	19.16	50745
scalatest	huge	10	0.52	19.36	3824
scalaxb	huge	10	11.02	109.06	99651
specs	large	20	1.34	10.81	6426
tmt	huge	10	32.02	39.23	266357

Figure 81: DaCapo Scala benchmarks and their vital properties

The DaCapo Scala benchmark suite has a problem with warming up the scalatest and the specs benchmarks. These benchmarks use custom class loaders and consequently reload a lot of classes in every iteration. Thus, the VM has no choice but to throw the compiled code of those classes away and to recompile them when the are loaded again. This makes warming these benchmarks up impossible. To warm up the rest of the classes as good as possible, we use the same number of warmup iterations as with other benchmarks of the same workload size.

SPECjvm. The SPECjvm [4] benchmark suite provides 21 benchmarks. In contrast to the DaCapo suites, every benchmark is single threaded internally and executed in parallel on the same workload depending

[2]DaCapo eclipse failure: http://stackoverflow.com/questions/24301986/
[3]DaCapo Scala 2012: http://www.scalabench.org/
[4]SPECjvm 2008: https://www.spec.org/jvm2008/

on the number of cores. This means that although the benchmarks appear to be multi-threaded, the threads will not interact. Also, this benchmark suite is throughput-based, meaning that every benchmark is executed for a fixed amount of time repeating the same operation over and over again. Finally, based on the number of completed operations, a score (higher is better) is calculated. To make it comparable to the other benchmarks, we used the *lagom* configuration, which reconfigures the benchmark to run a fixed number of operations (every operation is the same again) and measure the run time (lower is better). The benchmarks, including the number of operations defined by the *lagom* workload are shown in Figure 82.

Name	Operations	Warmups	Run time [s]	Live size [MB]	Allocations [10^3]
compiler.compiler	40	20	11.32	229.11	471781
compiler.sunflow	40	20	30.64	144.32	1195208
compress	10	20	5.78	85.20	36
crypto.aes	4	20	8.14	41.76	73
crypto.rsa	30	20	3.37	3.19	32880
crypto.signverify	24	20	6.70	18.33	4009
derby	60	20	16.75	435.11	2001219
mpegaudio	10	20	8.64	4.49	561
scimark.fft.large	2	20	7.84	612.17	1
scimark.fft.small	20	20	6.84	16.47	87
scimark.lu.large	1	20	37.19	583.21	34
scimark.lu.small	24	20	9.24	10.78	4023
scimark.monte_carlo	18	20	9.63	2.93	13
scimark.sor.large	2	20	6.98	294.85	17
scimark.sor.small	14	20	7.36	6.70	11
scimark.sparse.large	2	20	11.77	446.93	1
scimark.sparse.small	4	20	2.81	11.45	6
serial	50	20	43.87	367.18	3361406
sunflow	30	20	54.39	19.26	2047771
xml.transform	14	20	6.00	35.13	253054
xml.validation	80	20	17.09	92.80	807942

Figure 82: SPECjvm benchmarks and their vital properties

8.1 Tracing Overhead

This section describes in detail what overhead is introduced by the AntTracks VM when tracing an application.

8.1.1 Run Time

Two of the primary metrics we need to consider when evaluating overhead is run time and CPU time. The run time is the time a benchmark needs to fully process its workload. It is thus the most intuitive metric for a human observer. The CPU time is the time the processors actually spent working. Assuming single-core machines, this metric will never be higher than the run time, because the CPU cannot spend more time working than actual time that has passed. Usually it is lower, because in many cases a processor has to wait, e.g., for an IO operation to complete or for a lock. As this time is aggregated for all CPUs, it can be up to n times of the run time on an n-core machine.

Figure 83 shows the run-time overhead as well as the CPU-time overhead when tracing the DaCapo, DaCapo Scala and SPECjvm benchmarks. All values are normalized relative to the run time and CPU time of the same benchmarks without tracing (gray bars). The red or green offsets represent the respective overheads.

We can see that some benchmarks show a smaller run time or a smaller CPU time when tracing is on. This is due to the fact that even small modifications to the code, affect the scheduling and the overall performance. However, in all cases in which the run time or CPU time is smaller with tracing on, the difference is marginal.

Also, we see that the run-time overhead does not always correlate with the CPU-time overhead. On the one hand, there are some benchmarks, e.g., *kiama*, *scalap*, *scaladoc*, *scalap*, and *scalariform*, where the CPU-time overhead is significantly higher than the run-time overhead. This can be the result of idle times in the benchmarks. For example, if a CPU is idle because most threads are blocked doing some other work (e.g., processing event queues), the CPU time increases but not the run time. The *derby* benchmark shows the opposite behavior, i.e., the CPU-time overhead is much lower than the run-time overhead. This is the result of running into the IO boundary. The benchmark generates so many events that the underlying storage medium cannot keep up serializing them and the AntTracks VM has to repeatedly stall application threads. Thus, the CPU-time overhead is small but the run-time overhead is up to 75%. However, this overhead is not introduced by the virtual machine's monitoring infrastructure but rather by the underlying storage medium.

The average (arithmetic mean) run-time overhead and the CPU-time overhead using the ParallelOld GC is 8.07% and 11.48%. Using the G1 GC, it is 7.17% and 8.18%. As expected, the run-time overhead is slightly smaller than the CPU-time overhead because the AntTracks VM can exploit stalls when application threads are waiting anyway to serialize data. Comparing this overhead to the overhead of state-of-the-art approaches (cf. Section 3 and Section 2.4), the overhead of AntTracks is small enough to use AntTracks in production environments , although AntTracks provides even more data than other approaches.

8.1.2 GC Behavior

Since many events are recorded during garbage collections and since we wanted to know whether AntTracks distorts the garbage collection behavior, we looked at both the changes in the number of collections as well as at changes in the overall collection time (cf. Figure 84).

The results show that the number of garbage collections is unaffected with the exception of the *scimark.sparse.small* benchmark using the ParallelOld GC and the *h2*, *tmt*, and *derby* benchmarks using the G1 GC. Also, some benchmarks show a significant drop in garbage collections using the G1 GC, e.g., *avrora*, *lusearch*, *scimark.sor.small*, and *serial*. For the other benchmarks, this metric shows that we hardly introduce additional garbage collections, and thus do not distort the GC behavior. The GC time overhead is a result of the existing collections taking longer (due to the recording of the GC move events), but this overhead does not change the overall GC behavior. The above mentioned benchmarks show distorted behavior due to the GC starting to apply different thresholds and heuristics because of the introduced overhead.

The average distortion in number of garbage collection cycles (i.e., the arithmetic mean of the absolute differences in the number of garbage collections) using the ParallelOld GC and the G1 GC is 5.54% and 5.07% respectively. Comparing our distortion with state-of-the-art approaches shows that our distortion is negligible (cf. Section 3). The average GC-time overhead (arithmetic mean) is 24.4% and 11.98% respectively.

		ParallelOld GC		G1 GC	
	Benchmark	Run time	CPU time	Run time	CPU time
DaCapo	avrora	0.67%	1.22%	−0.76%	−0.42%
	batik	8.05%	33.33%	3.77%	7.69%
	eclipse	5.38%	10.00%	7.92%	8.33%
	fop	10.74%	25.00%	8.15%	18.18%
	h2	0.79%	4.00%	2.43%	2.79%
	jython	10.11%	33.08%	6.75%	10.52%
	luindex	0.31%	0.00%	1.76%	0.00%
	lusearch	4.77%	10.97%	−10.19%	−6.54%
	pmd	9.66%	9.50%	9.77%	7.35%
	sunflow	6.54%	8.90%	6.80%	4.99%
	tomcat	4.94%	6.10%	4.36%	5.07%
	tradebeans	5.44%	1.72%	2.38%	−0.17%
	tradesoap	9.22%	7.37%	10.62%	9.05%
	xalan	8.93%	8.53%	7.57%	7.69%
DaCapoScala	actors	6.36%	6.55%	10.02%	9.62%
	apparat	3.02%	10.33%	7.05%	9.98%
	factorie	14.36%	30.90%	17.50%	25.38%
	kiama	25.58%	62.96%	17.65%	24.14%
	scalac	13.18%	25.06%	10.44%	20.81%
	scaladoc	12.02%	29.60%	9.12%	20.57%
	scalap	17.14%	45.83%	6.52%	13.64%
	scalariform	12.18%	39.62%	10.68%	31.53%
	scalatest	13.66%	18.57%	12.95%	16.19%
	scalaxb	3.45%	8.61%	5.62%	9.22%
	specs	14.70%	18.86%	12.88%	16.67%
	tmt	11.89%	11.20%	13.73%	15.94%
SPECjvm	compiler.compiler	13.22%	9.75%	8.56%	6.19%
	compiler.sunflow	16.10%	12.07%	10.05%	6.51%
	compress	−0.50%	−0.39%	−0.17%	−0.07%
	crypto.aes	0.20%	0.50%	5.19%	5.21%
	crypto.rsa	3.76%	6.75%	1.48%	2.29%
	crypto.signverify	0.21%	0.90%	0.27%	0.71%
	derby	75.11%	7.12%	88.30%	10.09%
	mpegaudio	0.20%	1.33%	0.54%	0.68%
	scimark.fft.large	0.09%	0.21%	0.05%	0.13%
	scimark.fft.small	−0.48%	−0.02%	−0.07%	0.02%
	scimark.lu.large	0.26%	0.36%	−0.33%	1.60%
	scimark.lu.small	1.07%	3.91%	2.24%	−1.27%
	scimark.monte carlo	−0.10%	−0.14%	−7.93%	−8.15%
	scimark.sor.large	0.07%	0.52%	0.31%	−0.02%
	scimark.sor.small	0.10%	0.02%	0.14%	−0.13%
	scimark.sparse.large	−0.29%	0.27%	0.43%	49.61%
	scimark.sparse.small	5.68%	2.22%	−3.12%	−2.08%
	serial	9.52%	5.62%	10.25%	8.51%
	sunflow	5.12%	3.77%	8.53%	5.37%
	xml.transform	5.34%	7.12%	7.39%	4.87%
	xml.validation	11.74%	10.23%	9.75%	6.53%

Figure 83: Run-time overhead and CPU-time overhead when tracing using the ParallelOld GC and the G1 GC

		ParallelOld GC		G1 GC	
	Benchmark	GC count	GC time	GC count	GC time
DaCapo	avrora	0.00%	36.84%	−28.85%	−2.13%
	batik	0.00%	42.86%	0.00%	12.50%
	eclipse	0.00%	0.00%	0.00%	0.00%
	fop	0.00%	20.00%	0.00%	−6.25%
	h2	7.14%	36.82%	12.50%	14.69%
	jython	0.00%	74.39%	0.65%	10.54%
	luindex	0.00%	0.00%	0.00%	−45.45%
	lusearch	−0.53%	2.42%	−16.18%	−16.31%
	pmd	0.00%	25.29%	9.09%	21.28%
	sunflow	−0.19%	14.66%	0.39%	26.67%
	tomcat	0.00%	10.87%	0.00%	21.21%
	tradebeans	0.00%	26.46%	−2.17%	26.17%
	tradesoap	−0.34%	17.68%	4.63%	20.37%
	xalan	−5.54%	14.40%	0.40%	4.71%
DaCapoScala	actors	2.24%	14.56%	−1.15%	29.60%
	apparat	−0.57%	36.60%	−6.00%	17.94%
	factorie	1.59%	25.77%	−0.88%	28.90%
	kiama	0.00%	81.82%	0.00%	21.05%
	scalac	−3.33%	30.16%	0.00%	15.60%
	scaladoc	−2.13%	33.64%	5.56%	16.28%
	scalap	3.70%	53.85%	0.00%	5.56%
	scalariform	0.00%	54.70%	0.00%	52.63%
	scalatest	0.00%	11.54%	0.00%	35.71%
	scalaxb	0.00%	47.13%	0.00%	37.50%
	specs	1.72%	17.27%	0.00%	4.69%
	tmt	0.31%	26.44%	18.80%	9.44%
SPECjvm	compiler.compiler	−6.25%	19.88%	−3.70%	15.50%
	compiler.sunflow	0.26%	32.95%	−0.42%	16.08%
	compress	0.00%	14.29%	0.00%	0.00%
	crypto.aes	−1.30%	0.00%	−2.94%	5.11%
	crypto.rsa	0.62%	10.31%	−3.86%	17.73%
	crypto.signverify	1.10%	8.95%	−3.61%	7.37%
	derby	0.00%	13.84%	25.00%	32.51%
	mpegaudio	−0.77%	6.25%	−9.31%	5.93%
	scimark.fft.large			0.00%	−100.00%
	scimark.fft.small	0.71%	11.11%	2.41%	5.26%
	scimark.lu.large	0.00%	0.00%	0.00%	−24.00%
	scimark.lu.small	−0.09%	3.81%	2.45%	1.60%
	scimark.monte carlo	0.00%	0.00%	0.00%	0.00%
	scimark.sor.large	0.00%	5.56%	9.09%	58.82%
	scimark.sor.small	0.00%	100.00%	−16.67%	−25.00%
	scimark.sparse.large			0.00%	−100.00%
	scimark.sparse.small	14.29%	23.81%	0.00%	24.49%
	serial	3.13%	79.67%	−50.65%	−22.40%
	sunflow	0.00%	33.85%	0.00%	39.81%
	xml.transform	0.94%	2.50%	1.07%	14.45%
	xml.validation	−1.89%	24.04%	0.00%	27.25%

Figure 84: GC-count overhead and GC-time overhead when tracing using the ParallelOld GC and the G1 GC

8.1.3 Heap Usage

Figure 85 shows the heap usage and the code size with and without tracing. We define the heap usage as the maximum amount of used bytes at any point in time during the execution. The code size is defined as the amount of bytes used for JIT-compiled machine code.

We can see that the heap usage is very stable with the exception of the *avrora* and *serial* benchmarks, indicating that the heap is sized and garbage collected in the same manner as before. The two exceptional benchmarks show also a significant drop in their GC count, which is a side effect of their increased heap size.

The code size shows slight increases which is due to the additional machine code generated for recording allocation events. However, the additional code makes up only 3.61% and 3.18% respectively on average.

8.1.4 Buffer Management

In Section 4.4.1 we explained how we put considerable effort into buffering events, managing buffers and writing data to disk asynchronously. This section presents an experiment to evaluate these optimizations. Figure 86 shows the run time, the lock time, and io time of the benchmarks when using the buffer management described in Section 4.4.1 compared to when not using this optimization, i.e., when every thread writes its own buffer to the disk as soon as the buffer is full. While the run time is defined in the same way as in the previous section, the lock time is defined as the accumulated wall clock time, which all threads working on event buffers need to wait on a lock to synchronize their data before serialization. If buffer management is in use, this lock is only used to synchronize on the end of the queue from which the IO thread is consuming. If buffer management is not in use, this lock ensures that only one thread is writing to the trace file at a time.

This experiment shows that although buffer management hardly affects the run time of some benchmarks, others benefit greatly, e.g., *factorie*. Looking at the lock time, we see a drastic decrease when buffer management is used, i.e., hardly any application thread needs to stall when its buffer is full. Those benchmarks that show a significant increase in lock time, are benchmarks that hardly allocate anything and the lock times are within milliseconds at best.

8.1.5 Buffer Size

The AntTracks VM uses a default buffer size of 16KB to record events. When these 16KB are full, the owning thread submits the buffer to the flush queue and fetches a new buffer (cf. Section 4.4.1). Figure 87 shows the run-time impact of different buffer sizes compared to the default size.

As expected, choosing smaller buffer sizes increases the overall run time. However, choosing bigger buffer sizes hardly brings any additional benefit.

8.1.6 Buffer Randomization

In Section 4.4.1 we discussed randomizing buffer sizes in order to avoid contention on the flush queue and the IO thread. Figure 88 shows the performance increase (lower is better) when using randomized buffer sizes (+/- 50%) in relation to fixed buffer sizes.

Looking at the overall run time, we see that randomizing buffer sizes only provides 0.2% average run time reduction (arithmetic mean). The lock time (i.e., the time an application thread needs to wait for the flush queue lock) and the wait time (i.e., the time an application thread has to wait because the flush queue is currently full) shows that this optimization is somewhat unpredictable. Although some benchmarks show

		ParallelOld GC		G1 GC	
	Benchmark	Heap usage	Code size	Heap usage	Code size
DaCapo	avrora	−0.45%	2.54%	33.83%	1.84%
	batik	2.72%	3.01%	−0.91%	3.43%
	eclipse	0.32%	5.02%	2.91%	4.20%
	fop	0.06%	2.59%	4.08%	3.32%
	h2	0.34%	3.32%	−1.92%	4.49%
	jython	0.43%	2.93%	0.10%	2.87%
	luindex	−2.65%	3.06%	1.65%	3.49%
	lusearch	−0.75%	−0.09%	−4.46%	6.79%
	pmd	7.15%	2.62%	1.54%	3.32%
	sunflow	0.27%	4.14%	−0.95%	5.93%
	tomcat	−0.18%	3.06%	1.49%	1.69%
	tradebeans	−0.21%	7.98%	5.67%	4.66%
	tradesoap	−1.46%	1.46%	4.73%	1.33%
	xalan	0.15%	−1.89%	0.00%	0.58%
DaCapoScala	actors	−0.98%	2.33%	−1.58%	0.32%
	apparat	−1.60%	6.49%	5.49%	4.74%
	factorie	1.68%	3.34%	4.16%	3.77%
	kiama	0.22%	5.67%	−0.86%	5.42%
	scalac	−1.22%	5.03%	0.04%	0.98%
	scaladoc	2.64%	5.16%	−0.62%	5.14%
	scalap	−1.03%	2.22%	0.65%	3.55%
	scalariform	3.42%	2.92%	−0.13%	4.31%
	scalatest	9.15%	9.40%	−0.76%	2.93%
	scalaxb	−0.59%	3.65%	−9.80%	3.84%
	specs	0.40%	2.09%	−1.48%	4.92%
	tmt	−0.36%	2.18%	−6.63%	1.68%
SPECjvm	compiler.compiler	4.03%	4.15%	−0.75%	0.05%
	compiler.sunflow	−0.35%	7.67%	−2.18%	0.87%
	compress	−0.05%	7.79%	0.05%	5.46%
	crypto.aes	−3.73%	4.76%	−0.56%	2.45%
	crypto.rsa	0.11%	2.87%	4.79%	4.08%
	crypto.signverify	−3.19%	1.94%	1.66%	4.30%
	derby	−0.04%	7.54%	−14.75%	3.79%
	mpegaudio	−0.04%	4.65%	5.48%	1.19%
	scimark.fft.large	3.16%	−0.97%	0.01%	0.97%
	scimark.fft.small	−1.41%	−0.98%	1.09%	−0.24%
	scimark.lu.large	0.24%	3.22%	−9.49%	1.34%
	scimark.lu.small	−0.04%	4.05%	0.44%	2.43%
	scimark.monte carlo	0.82%	1.92%	1.09%	1.46%
	scimark.sor.large	−0.15%	1.41%	6.35%	5.37%
	scimark.sor.small	0.48%	3.04%	1.12%	3.92%
	scimark.sparse.large	1.54%	4.23%	0.02%	6.00%
	scimark.sparse.small	1.51%	2.30%	0.76%	3.01%
	serial	1.96%	4.63%	47.56%	2.41%
	sunflow	0.13%	5.76%	0.15%	6.59%
	xml.transform	0.14%	5.13%	−0.58%	1.61%
	xml.validation	−0.38%	4.75%	−4.83%	3.13%

Figure 85: Heap-usage overhead and code-size overhead when tracing using the ParallelOld GC and the G1 GC

	Benchmark	Run time	Lock time	IO time
DaCapo	avrora	−0.26%	−59.22%	−9.77%
	batik	5.23%	−96.97%	9.20%
	eclipse	−2.00%	−2.71%	13.12%
	fop	−2.90%	−98.64%	−12.68%
	h2	−1.43%	−94.75%	−4.01%
	jython	−0.34%	−7.30%	9.49%
	luindex	0.62%	13.59%	43.52%
	lusearch	0.77%	−1.28%	19.74%
	pmd	−2.15%	−85.15%	−0.72%
	sunflow	−0.54%	−95.40%	−9.36%
	tomcat	−0.49%	−6.26%	1.21%
	tradebeans	−0.90%	−98.71%	−12.32%
	tradesoap	−2.26%	−92.07%	−4.46%
	xalan	1.25%	−29.17%	4.79%
DaCapoScala	actors	−2.34%	−35.91%	−4.30%
	apparat	0.20%	−80.93%	7.06%
	factorie	−9.52%	−76.75%	4.10%
	kiama	−3.14%	−89.45%	−6.75%
	scalac	−0.09%	−97.07%	−5.27%
	scaladoc	−1.62%	−97.90%	−5.09%
	scalap	0.61%	−91.69%	−4.11%
	scalariform	−0.63%	−93.59%	1.63%
	scalatest	−1.16%	8.85%	−3.61%
	scalaxb	−1.84%	−97.01%	18.39%
	specs	0.07%	−49.85%	9.75%
	tmt	−4.42%	−98.11%	8.41%
SPECjvm	compiler.compiler	−2.40%	−98.58%	−12.52%
	compiler.sunflow	−0.78%	−98.67%	−8.76%
	compress	0.21%	11.32%	344.13%
	crypto.aes	−0.62%	57.61%	23.09%
	crypto.rsa	2.94%	−39.56%	−6.70%
	crypto.signverify	−0.93%	−28.41%	32.22%
	derby	1.22%	−98.51%	−12.70%
	mpegaudio	0.02%	0.72%	11.39%
	scimark.fft.large	−2.48%	139.17%	−11.27%
	scimark.fft.small	0.22%	−1.11%	16.91%
	scimark.lu.large	0.48%	−50.22%	−6.93%
	scimark.lu.small	−1.57%	−7.30%	20.18%
	scimark.monte carlo	0.05%	3.69%	59.45%
	scimark.sor.large	−0.13%	−5.31%	−4.67%
	scimark.sor.small	−0.15%	32.66%	19.87%
	scimark.sparse.large	0.02%	63.14%	−22.29%
	scimark.sparse.small	−0.57%	559.35%	84.52%
	serial	−1.56%	−98.60%	−4.12%
	sunflow	−2.66%	−99.47%	−5.62%
	xml.transform	0.06%	−93.26%	−9.75%
	xml.validation	−1.96%	−98.85%	−13.95%

Figure 86: Performance increase in run time, lock time, and IO time when using buffer management and asynchronous IO

	Benchmark	1K	2K	4K	8K	16K	32K	64K
DaCapo	avrora	0.49%	0.49%	0.20%	0.12%	0.00%	−0.26%	−0.22%
	batik	4.29%	14.72%	0.00%	−4.91%	0.00%	−5.52%	−6.75%
	eclipse	7.22%	5.15%	3.09%	2.06%	0.00%	1.03%	−1.03%
	fop	5.63%	4.23%	−0.70%	−4.23%	0.00%	−4.23%	−5.63%
	h2	2.58%	2.96%	0.51%	1.14%	0.00%	1.64%	0.15%
	jython	10.57%	6.64%	3.84%	1.52%	0.00%	0.01%	−0.58%
	luindex	0.61%	0.00%	−1.53%	−1.23%	0.00%	−0.61%	−1.84%
	lusearch	4.78%	2.65%	1.54%	1.69%	0.00%	0.10%	−0.53%
	pmd	6.99%	2.57%	1.56%	2.02%	0.00%	−1.38%	−0.83%
	sunflow	2.47%	1.32%	0.23%	−0.48%	0.00%	−0.36%	1.00%
	tomcat	2.25%	1.13%	1.63%	0.50%	0.00%	0.00%	0.88%
	tradebeans	6.73%	3.18%	1.95%	0.86%	0.00%	0.16%	−0.83%
	tradesoap	6.66%	4.45%	1.89%	0.99%	0.00%	−0.22%	−0.96%
	xalan	5.63%	0.08%	−1.21%	−0.71%	0.00%	−1.95%	−0.99%
DaCapoScala	actors	5.71%	3.31%	2.45%	0.95%	0.00%	−0.17%	−0.65%
	apparat	5.07%	−0.90%	0.98%	−1.24%	0.00%	−4.51%	15.42%
	factorie	27.79%	15.15%	7.94%	4.13%	0.00%	−2.69%	−4.12%
	kiama	23.00%	11.27%	1.88%	1.41%	0.00%	−0.94%	0.94%
	scalac	14.32%	6.28%	3.95%	−0.13%	0.00%	0.83%	−1.19%
	scaladoc	13.70%	8.34%	4.63%	3.51%	0.00%	−0.15%	−0.49%
	scalap	6.71%	0.61%	−1.83%	−0.61%	0.00%	−2.44%	−2.44%
	scalariform	11.78%	5.64%	1.06%	−0.89%	0.00%	−3.29%	−1.34%
	scalatest	2.63%	1.81%	1.15%	−0.16%	0.00%	−0.33%	−0.82%
	scalaxb	5.84%	1.77%	2.11%	0.54%	0.00%	−0.55%	−0.37%
	specs	5.75%	3.20%	2.75%	1.05%	0.00%	0.13%	−1.05%
	tmt	20.22%	10.77%	8.00%	4.53%	0.00%	7.99%	7.81%
SPECjvm	compiler.compiler	9.43%	9.80%	4.41%	2.83%	0.00%	−0.68%	−1.41%
	compiler.sunflow	10.90%	6.85%	4.53%	2.03%	0.00%	0.06%	−0.11%
	compress	0.42%	0.23%	0.16%	0.26%	0.00%	0.09%	0.10%
	crypto.aes	1.32%	−1.13%	2.06%	1.90%	0.00%	−1.57%	0.96%
	crypto.rsa	1.94%	−0.80%	1.03%	0.94%	0.00%	−3.02%	−3.70%
	crypto.signverify	0.67%	−0.19%	0.40%	0.12%	0.00%	0.25%	−0.13%
	derby	12.00%	9.03%	7.82%	−0.61%	0.00%	−2.54%	−4.25%
	mpegaudio	−0.14%	0.05%	−0.01%	0.74%	0.00%	−0.15%	0.19%
	scimark.fft.large	1.30%	−0.03%	0.23%	−0.08%	0.00%	−0.09%	−0.08%
	scimark.fft.small	0.09%	0.10%	0.06%	−0.07%	0.00%	−0.12%	−0.28%
	scimark.lu.large	−0.98%	−0.15%	−0.18%	0.17%	0.00%	0.27%	0.12%
	scimark.lu.small	−0.34%	0.92%	0.42%	0.08%	0.00%	0.05%	0.19%
	scimark.monte carlo	0.42%	0.75%	−5.29%	0.23%	0.00%	0.28%	0.25%
	scimark.sor.large	−0.33%	−0.03%	−0.06%	−0.09%	0.00%	0.10%	−0.17%
	scimark.sor.small	−0.01%	−0.01%	0.22%	0.31%	0.00%	−0.01%	−0.05%
	scimark.sparse.large	0.09%	0.17%	0.53%	0.22%	0.00%	0.31%	0.64%
	scimark.sparse.small	−0.13%	−2.55%	−8.99%	−9.02%	0.00%	−10.65%	−5.00%
	serial	11.44%	6.91%	4.41%	1.16%	0.00%	−1.20%	−1.50%
	sunflow	6.42%	2.80%	0.62%	0.27%	0.00%	−0.38%	−0.23%
	xml.transform	7.65%	3.60%	1.19%	−0.25%	0.00%	−0.51%	−0.47%
	xml.validation	8.07%	4.74%	2.90%	1.56%	0.00%	−0.87%	−0.16%

Figure 87: Impact of buffer size on run time compared to 16K default size

	Benchmark	Run time	Lock time		Wait time	
DaCapo	avrora	−0.39%	−1.78%	−127	−0.68%	−11
	batik	3.87%	−2.51%	−4	1.64%	8
	eclipse	−1.01%	2.26%	2	5.50%	6
	fop	0.00%	2.31%	4	−31.44%	−240
	h2	−0.66%	0.29%	42	11.74%	59253
	jython	1.20%	−5.78%	−294	−1.19%	−93
	luindex	−0.92%	10.08%	8	−10.41%	−62
	lusearch	1.11%	5.09%	4560	−2.34%	−747
	pmd	−0.37%	−33.10%	−8785	28.47%	1036
	sunflow	0.25%	0.88%	365	4.42%	1346
	tomcat	0.12%	4.20%	270	−0.27%	−5
	tradebeans	1.02%	−17.19%	−9027	3.91%	89345
	tradesoap	−0.35%	−0.60%	−2099	−79.05%	−227890
	xalan	0.92%	−3.79%	−12041	−2.90%	−2203
DaCapoScala	actors	−0.19%	−0.91%	−452	−3.30%	−130
	apparat	1.44%	7.51%	13793	−25.84%	−10162
	factorie	−0.81%	−3.85%	−4269	−5.68%	−126079
	kiama	0.00%	1.57%	5	4.20%	46
	scalac	−0.52%	−0.56%	−27	−4.67%	−1244
	scaladoc	0.24%	1.43%	26	−16.78%	−3777
	scalap	0.61%	2.97%	8	0.95%	17
	scalariform	0.17%	7.94%	120	6.18%	274
	scalatest	0.84%	7.58%	20745	−10.11%	−72
	scalaxb	0.45%	2.29%	51	−5.54%	−360
	specs	−0.32%	2.22%	78	−0.60%	−28
	tmt	−2.44%	5.25%	6662	−15.70%	−172886
SPECjvm	compiler.compiler	0.81%	4.87%	3377	220.59%	2043501
	compiler.sunflow	0.69%	6.29%	12381	−19.58%	−1745928
	compress	−0.28%	3.27%	14	42.83%	78
	crypto.aes	1.82%	40.51%	429	−0.96%	−17
	crypto.rsa	2.04%	0.52%	129	8.69%	520
	crypto.signverify	−0.21%	4.76%	166	−9.26%	−705
	derby	−2.36%	10.79%	9304	−20.45%	−853887
	mpegaudio	−0.22%	5.97%	2633	0.20%	6
	scimark.fft.large	−0.34%	−98.22%	−19453		0
	scimark.fft.small	−0.06%	−0.49%	−5	7.59%	124
	scimark.lu.large	−0.24%	−81.69%	−2991	2.35%	1
	scimark.lu.small	0.42%	18.14%	2065	−1.35%	−780
	scimark.monte carlo	0.67%	−38.73%	−81	45.81%	41
	scimark.sor.large	−0.11%	2772.95%	7720	5.78%	4
	scimark.sor.small	−0.05%	15.74%	13	6.57%	6
	scimark.sparse.large	0.05%	−54.33%	−4664		0
	scimark.sparse.small	1.36%	2871.06%	27280	−11.44%	−175
	serial	0.10%	32.21%	57270	−0.39%	−34533
	sunflow	1.31%	−9.94%	−101685	33.62%	266242
	xml.transform	−0.14%	−8.80%	−7185	−0.92%	−193
	xml.validation	0.72%	28.15%	20287	−6.87%	−36673

Figure 88: Performance increase when randomizing buffer sizes in terms of run time, lock time, and wait time (in microseconds)

a slight gain, *scimark.sor.large* and *scimark.sparse.small* show a drastic increase in lock time. However, looking at the absolute numbers, we see that the outlying benchmarks show only differences in the range of a few milliseconds. Nevertheless, the average change in lock time and the average change in wait time are only +34 microseconds and +16 milliseconds respectively, making this optimization not worth the effort.

8.1.7 Stacks

As described in Section 4.4.2, we capture an entire stack trace periodically. Figure 89 shows the performance impact in terms of overall run time and walk time (i.e., time needed to walk and resolve the stack, measured as the percentage of the overall CPU time) when capturing the stack every 4096, 8192, or 16384 allocations respectively.

The results show that even a relatively high interval can introduce overheads of up to 30% in the *tmt* benchmark. However, the average overall run time overheads (arithmetic means) are 4.3%, 3.49%, and 3.57% respectively.

8.1.8 Compression

As described in Section 4.7.1, we try to compress as much data on the fly as possible without imposing much additional overhead. Figure 90 shows the additional run-time overhead introduced when compressing data on the fly. It also shows the reduction in trace size and IO time, as well as the relative amount of buffers the mechanism described in Section 4.7.1 chose to compress.

The results show that compressing on the fly introduces an average run-time overhead (arithmetic mean) of 4.12%. On average, the trace can be reduced to 79% of its original size, resulting in an average reduction of IO time to 92%.

8.2 Trace Properties

This section evaluates qualitative trace properties, such as event distributions, the size and the growth rates of traces as well as the depth of allocation stack traces.

8.2.1 Events

Figure 91 shows the distribution of different event kinds in the trace. The first column presents the ratio of allocation events (yellow, north east lines), move events (blue, north west lines), and all other events (red, no lines). The second column shows the distribution of different kinds of allocation events, i.e., slow allocation events (dark yellow, north east lines), normal allocation events (yellow, north west lines), and fast allocation events (light yellow, no lines). The third column shows the distribution of different kinds of move events, i.e., slow move events (dark blue, north east lines), fast move events (blue, north west lines), and region move events (light blue, no lines).

The results show that, as expected, allocation events make up the largest portion of every trace. Almost the entire rest is composed of move events. Only a small portion are other bookkeeping events.

Except for some *scimark* benchmarks, almost all allocation events are fast allocation events. The *scimark* benchmarks use many slow allocation events because they mainly allocate large multi-dimensional arrays, forcing a slow path in the VM and consequently a slow allocation event.

		4096		8192		16384	
	Benchmark	Run time,	walk time	Run time,	walk time	Run time,	walk time
DaCapo	avrora	0.89%	0.05%	−0.07%	0.03%	0.58%	0.02%
	batik	1.24%	0.24%	−4.97%	0.15%	16.77%	0.06%
	eclipse	2.04%	1.09%	2.04%	0.60%	0.00%	0.41%
	fop	9.70%	1.82%	9.70%	1.19%	8.96%	0.64%
	h2	1.22%	0.18%	1.62%	0.10%	0.86%	0.06%
	jython	3.78%	1.83%	1.44%	1.00%	0.93%	0.58%
	luindex	−0.31%	0.05%	0.93%	0.02%	1.24%	0.02%
	lusearch	0.62%	0.21%	0.58%	0.11%	0.19%	0.06%
	pmd	1.19%	2.12%	1.38%	1.11%	0.37%	0.64%
	sunflow	7.27%	2.32%	7.76%	1.14%	6.14%	0.65%
	tomcat	2.10%	0.83%	2.23%	0.49%	2.23%	0.30%
	tradebeans	1.20%	0.77%	1.11%	0.42%	0.95%	0.24%
	tradesoap	3.00%	1.59%	3.04%	0.88%	2.86%	0.49%
	xalan	3.10%	0.73%	2.27%	0.37%	3.41%	0.20%
DaCapoScala	actors	4.12%	0.51%	3.05%	0.28%	4.20%	0.15%
	apparat	5.93%	0.60%	3.17%	0.34%	−1.89%	0.21%
	factorie	6.79%	3.24%	8.20%	1.73%	5.69%	0.99%
	kiama	14.35%	4.30%	11.11%	2.75%	10.19%	1.78%
	scalac	12.26%	2.91%	9.36%	1.74%	5.05%	0.97%
	scaladoc	8.84%	2.81%	6.27%	1.57%	4.81%	0.91%
	scalap	6.10%	1.80%	4.88%	1.17%	−2.44%	0.98%
	scalariform	10.91%	3.74%	6.38%	2.31%	6.66%	1.35%
	scalatest	2.34%	1.95%	2.67%	1.03%	1.84%	0.50%
	scalaxb	2.05%	0.86%	2.23%	0.48%	1.41%	0.28%
	specs	2.86%	1.40%	1.24%	0.87%	2.41%	0.46%
	tmt	30.72%	1.55%	24.65%	1.36%	29.63%	0.68%
SPECjvm	compiler.compiler	6.44%	2.74%	5.02%	2.86%	3.85%	1.10%
	compiler.sunflow	3.13%	1.52%	3.22%	0.90%	2.41%	0.55%
	compress	0.45%	0.00%	0.66%	0.00%	1.02%	0.00%
	crypto.aes	−5.26%	0.00%	−1.89%	0.00%	0.54%	0.00%
	crypto.rsa	0.69%	0.13%	−0.66%	0.07%	−0.09%	0.04%
	crypto.signverify	−0.55%	0.02%	−0.71%	0.01%	−0.45%	0.01%
	derby	18.32%	2.66%	16.74%	2.22%	16.12%	1.34%
	mpegaudio	−0.17%	0.00%	−0.17%	0.00%	−0.35%	0.00%
	scimark.fft.large	0.08%	0.00%	−0.09%	0.00%	0.20%	0.00%
	scimark.fft.small	0.13%	0.00%	−0.01%	0.00%	0.09%	0.00%
	scimark.lu.large	−0.32%	0.00%	−0.66%	0.00%	−0.12%	0.00%
	scimark.lu.small	−0.52%	0.00%	−0.40%	0.00%	0.03%	0.00%
	scimark.monte carlo	1.54%	0.00%	0.41%	0.00%	0.97%	0.00%
	scimark.sor.large	−0.01%	0.00%	0.10%	0.00%	−0.01%	0.00%
	scimark.sor.small	0.04%	0.00%	−0.05%	0.00%	−0.22%	0.00%
	scimark.sparse.large	0.74%	0.00%	−0.07%	0.00%	0.06%	0.00%
	scimark.sparse.small	4.50%	0.00%	8.56%	0.00%	6.28%	0.00%
	serial	10.72%	2.26%	9.51%	1.06%	11.15%	0.52%
	sunflow	9.51%	21.11%	6.37%	9.40%	7.42%	4.01%
	xml.transform	4.39%	1.44%	3.16%	0.86%	2.84%	0.44%
	xml.validation	4.32%	1.30%	3.14%	0.74%	3.18%	0.43%

Figure 89: Performance overhead when capturing stacks every 4096, 8191, or 16384 allocations

	Benchmark	Run time	Trace size	Compressed buffers	IO time
DaCapo	avrora	1.69%	−68.75%	80.07%	−52.79%
	batik	3.11%	−45.87%	57.14%	−28.02%
	eclipse	5.10%	−64.45%	96.83%	−26.38%
	fop	7.46%	8.83%	0.00%	27.59%
	h2	0.77%	−35.82%	52.56%	−14.97%
	jython	5.46%	−0.64%	0.12%	22.64%
	luindex	2.48%	−68.49%	80.29%	−38.92%
	lusearch	53.04%	−11.72%	15.83%	−8.20%
	pmd	2.75%	0.35%	0.20%	1.74%
	sunflow	2.08%	−0.58%	1.17%	−3.19%
	tomcat	2.35%	−0.23%	0.03%	15.77%
	tradebeans	−0.47%	−1.77%	2.47%	−3.68%
	tradesoap	3.92%	−0.53%	1.38%	−4.76%
	xalan	13.76%	−0.15%	2.39%	2.74%
DaCapoScala	actors	1.68%	2.05%	0.05%	2.26%
	apparat	9.20%	−6.93%	7.55%	−0.60%
	factorie	1.76%	−0.04%	0.49%	7.28%
	kiama	3.24%	0.17%	0.00%	9.42%
	scalac	7.25%	1.29%	2.87%	14.16%
	scaladoc	7.63%	−1.13%	1.18%	26.22%
	scalap	1.22%	−1.43%	0.00%	6.57%
	scalariform	6.89%	−0.43%	0.00%	19.94%
	scalatest	2.84%	−0.27%	0.00%	9.38%
	scalaxb	4.58%	−36.80%	45.23%	−10.76%
	specs	2.80%	−0.72%	0.00%	35.99%
	tmt	4.75%	−0.27%	0.35%	1.07%
SPECjvm	compiler.compiler	1.51%	−3.26%	2.73%	−3.06%
	compiler.sunflow	3.67%	−2.27%	3.25%	−2.04%
	compress	0.40%	−51.45%	81.82%	−6.10%
	crypto.aes	−1.14%	−46.77%	73.36%	−37.09%
	crypto.rsa	1.69%	−2.00%	2.53%	−6.96%
	crypto.signverify	0.88%	−60.41%	79.24%	0.27%
	derby	0.56%	−4.36%	5.15%	0.42%
	mpegaudio	0.98%	−66.03%	77.58%	−8.45%
	scimark.fft.large	0.37%	−25.90%	100.00%	−33.53%
	scimark.fft.small	0.31%	−39.82%	78.97%	−52.86%
	scimark.lu.large	0.16%	−66.91%	95.74%	−56.66%
	scimark.lu.small	3.48%	−58.75%	81.32%	−6.32%
	scimark.monte carlo	−0.06%	−56.61%	85.96%	−53.16%
	scimark.sor.large	0.06%	−65.94%	93.65%	−41.69%
	scimark.sor.small	0.01%	−55.88%	89.47%	−28.79%
	scimark.sparse.large	0.13%	−22.68%	100.00%	0.96%
	scimark.sparse.small	14.04%	−10.85%	83.69%	−23.01%
	serial	3.37%	0.28%	0.34%	−3.59%
	sunflow	1.63%	−1.20%	1.82%	−4.95%
	xml.transform	2.47%	1.05%	0.59%	−4.29%
	xml.validation	2.26%	−1.29%	0.96%	−3.26%

Figure 90: Performance increase when compressing buffer on the fly

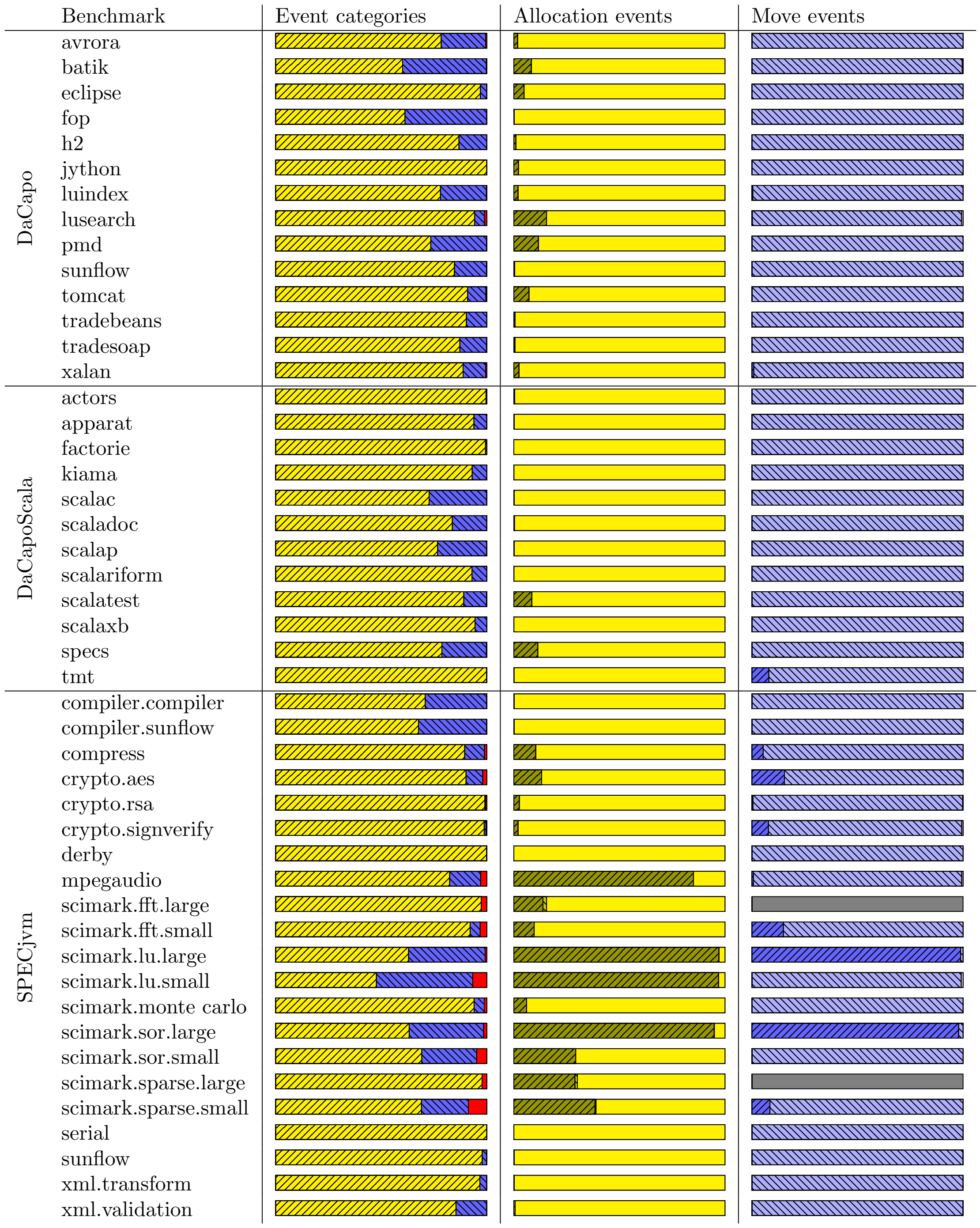


Figure 91: Distribution of allocation events and move events, of allocation events (slow (dark yellow, north east lines), normal (yellow, north west lines), and fast(light yellow, no lines)), and of move events (slow (dark blue, north east lines), fast (blue, north west lines), region (light yellow, no lines))

Similar to the allocation events, almost all move events are fast move events. Although a large portion of objects are moved with region events, there are so few of these events that they are hardly visible.

This analysis validates the usefulness of our optimizations for the common cases when allocating or moving objects because the fast event versions are dominant in every trace.

8.2.2 Size

Figure 92 shows the total size of the symbols file, the trace size of the measurement iteration, as well as the growth rate of the trace.

The data shows that the symbols file is kept relatively small in relation to the trace file. Also, we see that we need trace rotation when tracing continuously, because some traces are generated with up to 270 MB/sec. It is obvious that even a large storage medium can quickly be completely filled.

8.2.3 Mini Stacks

Figure 93 shows the static depth and the dynamic depth of all mini stacks. The static stack depth is computed over all allocation sites, whereas the dynamic stack depth is computed over all dynamically occurring allocations.

The results show, that most benchmarks have a median static stack depth of two, meaning that 50% of all objects have an allocation site where at least two stack frames can be recovered. The *scimark* benchmarks are an exception again, because most allocations are still interpreted and mini stacks are only available for compiled allocations.

The median dynamic stack depth is significantly higher because hot allocations are likely to be compiled and thus are also likely to be inlined. This results in a higher dynamic stack depth, and in turn, enabling better analysis.

8.2.4 Stack Trace Extensions

Figure 94 shows the amount of stack frames that can be added offline by static and dynamic analysis. Static analysis adds stack frames if a method is found at the bottom of a call stack that has only one caller. Dynamic analysis adds stack frames if the caller can be identified based on previous allocations. Both values are weighted with the number of allocations at the respective allocation sites.

The results show that the static analysis performs very well, i.e., it can almost double the amount of stack frames in the case of *scimark.sparse.small*. However, the dynamic extension can only add a few stack frames, hardly making any difference.

8.3 Tracing Overhead with Rotation

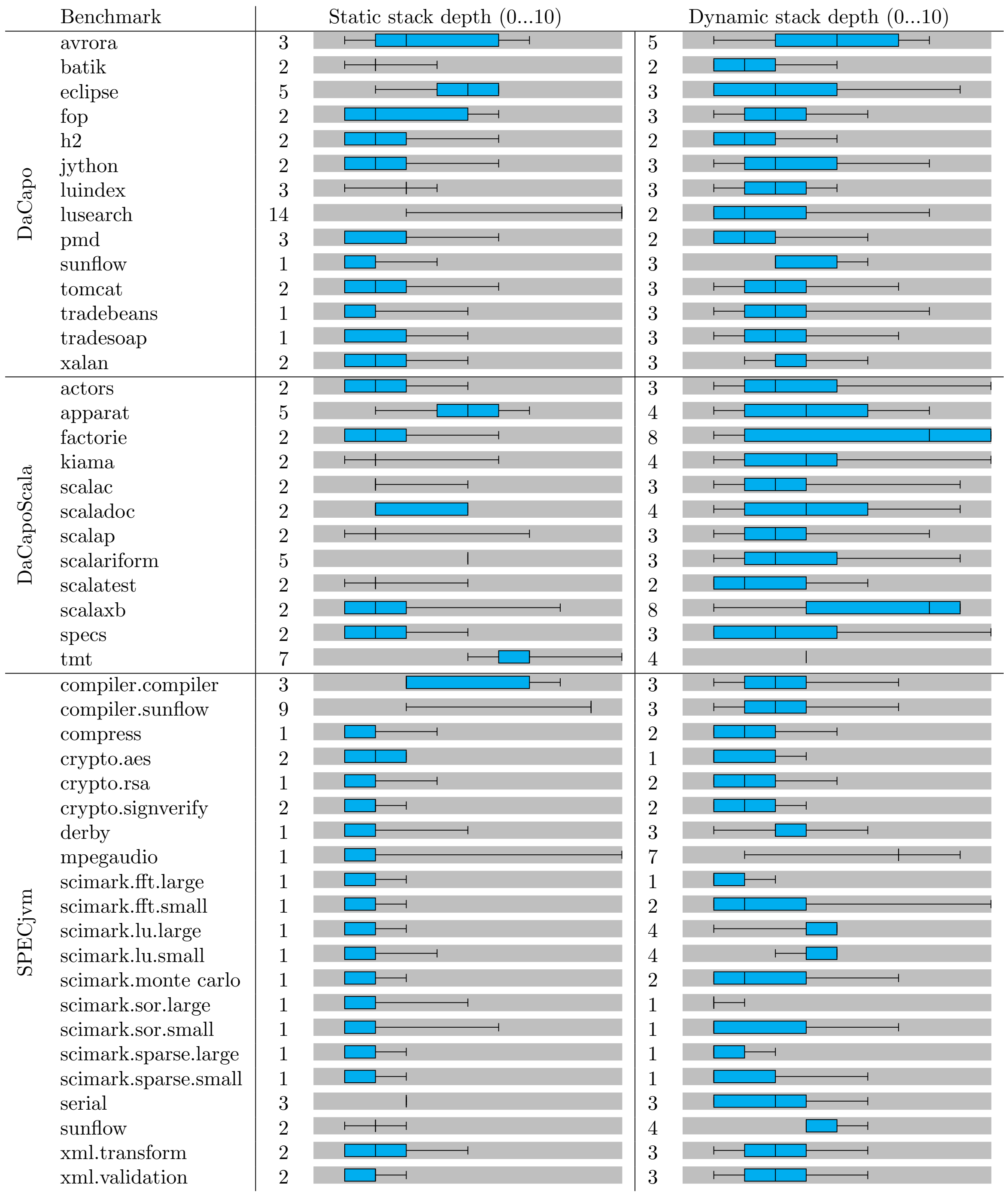
This section examines the overhead that is introduced when rotating the trace with a maximum size of 16GB and 10% deviation (cf. Section 4.7.2). This configuration will generate 10 trace files, where every trace file will be at most 1.6GB in size. All results are presented in relation to tracing without rotation.

8.3.1 Run Time

Figure 95 shows the run-time overhead and the CPU-time overhead compared to tracing without rotation. The figure also shows how often a new trace file has been started. This metric is needed to set the overhead

	Benchmark	Symbols file size [KB]	Trace file size [KB]	Trace file growth [KB/s]
DaCapo	avrora	1652	40516	5872.37
	batik	2768	3672	19357.82
	eclipse	5038	3329	31308.59
	fop	3306	18448	125753.57
	h2	1893	2106046	14249.36
	jython	5055	795622	93247.56
	luindex	1342	1187	3643.26
	lusearch	1884	141401	60845.47
	pmd	2898	120659	99511.20
	sunflow	1479	663596	147275.73
	tomcat	6189	37480	45178.66
	tradebeans	9519	4536167	83286.22
	tradesoap	10867	4965048	118268.29
	xalan	2071	506610	83942.93
DaCapoScala	actors	2585	962715	72063.98
	apparat	2847	1788143	22058.00
	factorie	1454	23489431	232359.35
	kiama	3268	51088	207933.03
	scalac	15630	287637	121777.07
	scaladoc	10018	236928	110163.06
	scalap	1921	18273	105809.18
	scalariform	5136	217886	120022.99
	scalatest	135551	21410	33758.40
	scalaxb	3183	441681	38075.85
	specs	58270	42098	26690.93
	tmt	1812	10703288	287524.77
SPECjvm	compiler.compiler	4228	3333945	257876.08
	compiler.sunflow	3823	9235067	251796.36
	compress	1168	239	41.71
	crypto.aes	1306	524	64.24
	crypto.rsa	1331	144731	41552.76
	crypto.signverify	1297	17444	2562.49
	derby	3200	8027290	263053.77
	mpegaudio	1414	8923	1027.05
	scimark.fft.large	1145	5	0.68
	scimark.fft.small	1163	557	81.93
	scimark.lu.large	1145	1073	28.77
	scimark.lu.small	1166	83786	8694.44
	scimark.monte carlo	1157	71	7.33
	scimark.sor.large	1143	546	78.33
	scimark.sor.small	1158	117	15.79
	scimark.sparse.large	1143	6	0.49
	scimark.sparse.small	1151	80	29.41
	serial	1347	13537593	272142.09
	sunflow	1799	8447169	148007.55
	xml.transform	3899	1119685	170802.45
	xml.validation	2256	4353470	221207.42

Figure 92: Symbols file size, trace file size, and trace file growth per second



	Benchmark	Static stack depth (0...10)	Dynamic stack depth (0...10)
DaCapo	avrora	3	5
	batik	2	2
	eclipse	5	3
	fop	2	3
	h2	2	2
	jython	2	3
	luindex	3	3
	lusearch	14	2
	pmd	3	2
	sunflow	1	3
	tomcat	2	3
	tradebeans	1	3
	tradesoap	1	3
	xalan	2	3
DaCapoScala	actors	2	3
	apparat	5	4
	factorie	2	8
	kiama	2	4
	scalac	2	3
	scaladoc	2	4
	scalap	2	3
	scalariform	5	3
	scalatest	2	2
	scalaxb	2	8
	specs	2	3
	tmt	7	4
SPECjvm	compiler.compiler	3	3
	compiler.sunflow	9	3
	compress	1	2
	crypto.aes	2	1
	crypto.rsa	1	2
	crypto.signverify	2	2
	derby	1	3
	mpegaudio	1	7
	scimark.fft.large	1	1
	scimark.fft.small	1	2
	scimark.lu.large	1	4
	scimark.lu.small	1	4
	scimark.monte carlo	1	2
	scimark.sor.large	1	1
	scimark.sor.small	1	1
	scimark.sparse.large	1	1
	scimark.sparse.small	1	1
	serial	3	3
	sunflow	2	4
	xml.transform	2	3
	xml.validation	2	3

Figure 93: Static (per allocation site) and dynamic (per allocation) stack depths (median and distribution)

	Benchmark	ParallelOld GC Base / Static / Dynamic			G1 GC Base / Static / Dynamic		
DaCapo	avrora	87.85%	12.14%	0.01%	86.22%	13.77%	0.01%
	batik	94.26%	5.43%	0.31%	94.26%	5.66%	0.08%
	eclipse	90.72%	8.68%	0.60%	91.03%	8.42%	0.55%
	fop	90.24%	9.75%	0.01%	89.54%	10.45%	0.01%
	h2	96.13%	3.87%	0.00%	96.57%	3.43%	0.00%
	jython	90.47%	9.53%	0.00%	91.66%	8.34%	0.00%
	luindex	91.65%	8.27%	0.08%	88.22%	11.64%	0.14%
	lusearch	87.88%	11.48%	0.65%	89.13%	10.87%	0.00%
	pmd	94.36%	5.63%	0.01%	93.75%	5.53%	0.72%
	sunflow	67.48%	32.52%	0.00%	60.02%	39.98%	0.00%
	tomcat	93.29%	6.71%	0.00%	91.89%	8.10%	0.01%
	tradebeans	97.99%	2.01%	0.00%	96.84%	3.16%	0.00%
	tradesoap	95.77%	4.22%	0.00%	89.08%	10.92%	0.00%
	xalan	94.37%	5.46%	0.17%	94.93%	4.91%	0.15%
DaCapoScala	actors	89.09%	10.91%	0.00%	91.92%	7.89%	0.19%
	apparat	98.60%	1.35%	0.05%	98.35%	1.59%	0.06%
	factorie	96.02%	3.98%	0.00%	96.00%	4.00%	0.00%
	kiama	90.53%	9.47%	0.00%	89.74%	10.22%	0.04%
	scalac	92.52%	7.44%	0.04%	89.29%	10.67%	0.04%
	scaladoc	97.39%	2.17%	0.44%	92.89%	6.09%	1.02%
	scalap	97.02%	2.92%	0.06%	98.53%	1.39%	0.08%
	scalariform	88.64%	7.77%	3.58%	86.59%	10.49%	2.92%
	scalatest	94.02%	5.38%	0.60%	90.25%	3.83%	5.92%
	scalaxb	99.67%	0.33%	0.00%	99.43%	0.57%	0.00%
	specs	96.86%	3.08%	0.06%	95.29%	4.67%	0.04%
	tmt	99.92%	0.07%	0.00%	99.98%	0.01%	0.00%
SPECjvm	compiler.compiler	79.38%	20.62%	0.00%	80.75%	19.23%	0.02%
	compiler.sunflow	81.46%	18.54%	0.01%	0.00%	0.00%	0.00%
	compress	78.97%	21.02%	0.00%	79.07%	20.92%	0.00%
	crypto.aes	81.88%	17.13%	0.99%	86.83%	12.18%	0.99%
	crypto.rsa	95.99%	4.01%	0.00%	93.93%	6.07%	0.00%
	crypto.signverify	92.83%	7.04%	0.13%	91.99%	8.01%	0.00%
	derby	99.97%	0.03%	0.00%	95.24%	4.76%	0.00%
	mpegaudio	58.62%	41.38%	0.00%	59.00%	40.74%	0.25%
	scimark.fft.large	79.12%	20.86%	0.02%	78.96%	21.02%	0.02%
	scimark.fft.small	92.00%	8.00%	0.00%	91.12%	8.88%	0.00%
	scimark.lu.large	58.83%	41.16%	0.00%	59.30%	40.69%	0.00%
	scimark.lu.small	57.72%	42.28%	0.00%	55.05%	44.95%	0.00%
	scimark.monte carlo	74.82%	25.17%	0.01%	74.75%	25.24%	0.01%
	scimark.sor.large	91.53%	8.34%	0.13%	91.73%	8.17%	0.10%
	scimark.sor.small	59.99%	39.77%	0.24%	59.52%	40.24%	0.24%
	scimark.sparse.large	77.69%	22.29%	0.02%	77.76%	22.22%	0.02%
	scimark.sparse.small	56.24%	43.75%	0.01%	57.86%	42.13%	0.01%
	serial	85.40%	14.60%	0.00%	84.37%	15.63%	0.00%
	sunflow	73.72%	26.28%	0.00%	73.72%	26.28%	0.00%
	xml.transform	96.44%	3.47%	0.10%	97.12%	2.79%	0.09%
	xml.validation	88.86%	11.14%	0.00%	82.93%	17.07%	0.00%

Figure 94: Amount of stack frames added statically as well as dynamically

into perspective, because some benchmarks generate so little data that rotation has no effect and the first trace file is never exhausted.

The results show that only some benchmarks generate a second trace file. Only the *factorie* benchmark rotates the trace fully (more than 10 new trace files). Also, we can see that significant overhead is only introduced when a new trace file is started (because of the synchronization GC at the beginning of every trace file). Benchmarks with no new trace file hardly show any additional overhead. The overhead in benchmarks that do not need additional trace files (e.g., in *kiama*) comes from the fact we have to save the allocation site into every object, for the case that we need a new trace file later. Not a single benchmark needed an emergency synchronization collection that would distort the application behavior.

8.3.2 Allocation Site Saving

Previous experiments showed that some overhead is introduced, even when no additional trace file is required. This is because the allocation site still needs to be saved in every object, in case the trace will be rotated later. To measure this overhead, we traced with and without storing the allocation site in every object. Figure 96 shows the performance increase when not saving the allocation site in every object.

The results show, that in allocation-intensive benchmarks such as *h2* and *factorie*, we can save up to 70% by not storing the allocation site. This is due to two reasons: (1) saving the allocation site for every object is expensive, because it has to be done at every allocation, and (2) as we store the allocation site in the identity hash, identity-hash-based data structures are not as efficient as before.

8.3.3 Hash Code Elimination

As described in Section 4.7.2, we optimistically did not generate the identity hash code when we determined that the corresponding class has its own implementation of the method `hashcode()`, assuming that the identity hash will not be used. Figure 97 shows the performance increase when eliminating as many hash code computations as possible.

The results show that this optimization is unpredictable, some benchmarks perform better, others worse, depending on their use of the identity hash code.

8.4 Parsing Performance

Figure 98 shows the performance of the offline processing and analysis tool. Before analysis can be started, the tool must first process the trace and reconstruct all omitted information. This figure also shows which traces could be parsed with the same heap size as the monitored application.

As the data shows, parsing the trace can take some time depending on the memory behavior of the monitored application. However, as parsing is done offline, it does not impede the monitored application.

All benchmarks could be parsed with the same heap size as the monitored application, with the exception of *batik*, *eclipse*, *jython*, *tradesoap*, *scalatest*, and *specs*. Closer examination showed that all these benchmarks are not parsable with the same heap size because of one of two reasons: (1) their heap size was so small that the static overhead of the analysis tool is too large, or (2) they reload all classes leading to too much symbol information. Every analysis tool will have some static overhead, even if it is very small. Also, applications reload their classes again and again (as the *scalatest* and *specs* benchmarks do), this will eventually lead to huge amounts of loaded classes. However, as our goal is to not require better machines for parsing than for

		ParallelOld GC			G1 GC		
	Benchmark		Run time	CPU time		Run time	CPU time
DaCapo	avrora	0	1.18%	−0.23%	0	2.32%	2.32%
	batik	0	19.25%	53.57%	0	3.03%	3.57%
	eclipse	0	1.02%	0.00%	0	2.75%	7.69%
	fop	0	20.90%	12.00%	0	19.86%	7.69%
	h2	2	108.43%	67.43%	2	89.67%	53.68%
	jython	0	8.75%	9.07%	1	11.69%	11.07%
	luindex	0	0.93%	0.00%	0	0.00%	2.94%
	lusearch	0	1.01%	3.05%	0	−4.15%	0.31%
	pmd	0	2.94%	4.15%	0	1.93%	0.76%
	sunflow	1	0.90%	2.09%	0	1.58%	1.16%
	tomcat	0	1.73%	6.19%	0	3.08%	8.33%
	tradebeans	3	200.37%	69.63%	2	195.94%	67.24%
	tradesoap	3	32.12%	4.29%	4	28.60%	1.28%
	xalan	0	−0.36%	1.43%	0	−0.26%	0.42%
DaCapoScala	actors	1	−0.94%	−1.31%	1	−0.95%	−1.21%
	apparat	1	5.88%	0.72%	1	−1.95%	−3.31%
	factorie	15	406.58%	283.85%	14	371.38%	266.60%
	kiama	0	97.22%	50.00%	0	89.55%	66.67%
	scalac	0	23.03%	2.58%	0	9.20%	11.00%
	scaladoc	0	4.71%	−1.44%	0	5.88%	1.76%
	scalap	0	2.44%	5.71%	0	2.04%	−8.00%
	scalariform	0	5.05%	7.77%	0	0.77%	−2.25%
	scalatest	0	−1.84%	−2.41%	0	−0.83%	1.64%
	scalaxb	0	−0.42%	0.63%	0	0.33%	−0.45%
	specs	0	2.86%	11.06%	0	2.16%	1.65%
	tmt	6	53.62%	3.45%	6	50.93%	0.53%
SPECjvm	compiler.compiler	2	51.84%	6.05%	2	44.42%	4.02%
	compiler.sunflow	5	43.90%	2.87%	5	36.87%	0.91%
	compress	0	0.31%	−0.02%	0	1.05%	0.68%
	crypto.aes	0	−2.82%	−3.29%	0	0.38%	0.36%
	crypto.rsa	0	0.06%	−0.42%	0	1.31%	−0.25%
	crypto.signverify	0	4.69%	4.67%	0	5.36%	5.35%
	derby	4	7.87%	3.77%	4	2.30%	3.52%
	mpegaudio	0	−0.12%	0.22%	0	−0.41%	−0.46%
	scimark.fft.large	0	0.98%	1.10%	0	1.27%	0.32%
	scimark.fft.small	0	0.22%	−0.09%	0	−0.30%	0.04%
	scimark.lu.large	0	−0.94%	−5.48%	0	1.34%	7.85%
	scimark.lu.small	0	2.40%	2.49%	0	−0.56%	1.31%
	scimark.monte carlo	0	0.75%	0.71%	0	6.68%	7.04%
	scimark.sor.large	0	0.94%	0.66%	0	1.27%	1.45%
	scimark.sor.small	0	0.69%	1.20%	0	0.51%	−0.17%
	scimark.sparse.large	0	−0.03%	−0.10%	0	0.89%	0.14%
	scimark.sparse.small	0	6.15%	1.51%	0	2.12%	−2.38%
	serial	7	60.40%	18.61%	7	62.00%	17.33%
	sunflow	4	22.20%	2.89%	5	26.41%	1.77%
	xml.transform	0	7.76%	9.82%	0	5.75%	6.68%
	xml.validation	2	30.88%	−0.32%	2	27.33%	0.45%

Figure 95: Number of additional trace files needed, run-time overhead and CPU-time overhead when tracing using the ParallelOld GC and the G1 GC, limiting the trace size to 16GB

	Benchmark	Run time	CPU time	GC count	GC time
DaCapo	avrora	−1.06%	−0.75%	0.00%	3.92%
	batik	−17.71%	−41.86%	−14.29%	−62.50%
	eclipse	0.00%	9.09%	0.00%	−50.00%
	fop	−14.81%	−10.71%	0.00%	−9.09%
	h2	−50.98%	−39.99%	7.69%	−6.44%
	jython	−8.30%	−5.83%	0.00%	0.63%
	luindex	−0.31%	3.33%	0.00%	33.33%
	lusearch	−1.47%	−3.98%	0.00%	0.18%
	pmd	−3.92%	−2.90%	0.00%	−6.03%
	sunflow	−2.15%	−2.89%	0.19%	−0.46%
	tomcat	−1.82%	−8.33%	0.00%	−3.70%
	tradebeans	−63.07%	−40.18%	2.83%	1.70%
	tradesoap	−6.33%	−2.65%	0.17%	0.84%
	xalan	−4.14%	−3.34%	0.63%	−9.39%
DaCapoScala	actors	0.70%	0.91%	−0.88%	−2.73%
	apparat	−4.41%	3.96%	0.00%	4.47%
	factorie	−70.93%	−72.79%	−26.17%	−18.15%
	kiama	−50.47%	−39.39%	0.00%	−11.76%
	scalac	−16.58%	11.22%	0.00%	9.27%
	scaladoc	−3.11%	1.71%	0.00%	10.26%
	scalap	−4.17%	−8.11%	3.70%	−2.56%
	scalariform	−3.12%	−6.27%	0.00%	−3.47%
	scalatest	1.36%	0.41%	0.00%	−6.67%
	scalaxb	−0.18%	0.24%	8.00%	3.15%
	specs	−3.29%	−10.39%	−1.69%	−8.45%
	tmt	−2.02%	0.69%	0.00%	−0.15%
SPECjvm	compiler.compiler	−2.14%	−4.56%	−2.13%	−1.60%
	compiler.sunflow	−1.41%	−2.26%	0.26%	−0.26%
	compress	0.31%	0.02%	9.09%	0.00%
	crypto.aes	0.86%	1.21%	0.00%	0.00%
	crypto.rsa	−0.31%	−0.09%	0.00%	0.00%
	crypto.signverify	−4.52%	−4.94%	0.00%	2.16%
	derby	−0.46%	−0.36%	0.00%	1.00%
	mpegaudio	0.03%	0.18%	0.52%	2.35%
	scimark.fft.large	0.61%	−0.06%		
	scimark.fft.small	−0.20%	−0.02%	−0.71%	−3.45%
	scimark.lu.large	0.79%	5.55%	0.00%	0.00%
	scimark.lu.small	−2.36%	−2.72%	0.18%	−1.06%
	scimark.monte carlo	−0.84%	−0.81%	0.00%	0.00%
	scimark.sor.large	−0.74%	−0.93%	0.00%	0.00%
	scimark.sor.small	−0.84%	−1.25%	0.00%	100.00%
	scimark.sparse.large	0.22%	−0.32%		
	scimark.sparse.small	−0.28%	0.27%	0.00%	0.00%
	serial	−10.62%	−15.16%	−4.86%	−11.84%
	sunflow	−2.17%	−3.04%	−0.08%	−0.19%
	xml.transform	−8.01%	−8.44%	0.93%	1.14%
	xml.validation	−1.06%	−0.60%	8.01%	0.44%

Figure 96: Performance increase when not saving the allocation sites for the ParallelOld GC

	Benchmark	Run time	CPU time	GC count	GC time
DaCapo	avrora	−0.76%	0.00%	0.00%	1.96%
	batik	−18.23%	−44.19%	−14.29%	−75.00%
	eclipse	2.02%	0.00%	0.00%	−50.00%
	fop	2.47%	3.57%	0.00%	4.55%
	h2	0.97%	0.89%	7.69%	−0.50%
	jython	2.91%	2.06%	0.00%	2.03%
	luindex	0.92%	0.00%	0.00%	16.67%
	lusearch	0.28%	−0.41%	−0.03%	0.62%
	pmd	−0.62%	0.00%	0.00%	−5.17%
	sunflow	−0.49%	−0.67%	0.00%	0.00%
	tomcat	−1.09%	−2.92%	0.00%	−9.26%
	tradebeans	0.07%	0.83%	0.00%	1.45%
	tradesoap	0.47%	1.62%	0.00%	0.96%
	xalan	3.37%	2.25%	−7.60%	6.57%
DaCapoScala	actors	1.38%	1.26%	0.88%	0.55%
	apparat	−0.41%	−0.34%	1.74%	−0.41%
	factorie	−61.90%	−62.87%	−22.27%	−13.75%
	kiama	−44.60%	−33.33%	7.69%	−14.71%
	scalac	1.21%	0.97%	0.00%	8.63%
	scaladoc	3.06%	1.46%	2.17%	0.00%
	scalap	1.79%	0.00%	0.00%	5.13%
	scalariform	0.44%	−1.25%	1.64%	−7.92%
	scalatest	0.85%	−1.23%	0.00%	−6.67%
	scalaxb	−0.41%	−0.95%	0.00%	4.72%
	specs	−1.52%	−6.93%	0.00%	−5.63%
	tmt	0.75%	2.70%	−0.15%	0.23%
SPECjvm	compiler.compiler	−1.07%	−1.96%	0.00%	−1.29%
	compiler.sunflow	−0.26%	−0.26%	1.59%	0.80%
	compress	0.09%	0.00%	9.09%	0.00%
	crypto.aes	3.32%	3.43%	1.32%	0.00%
	crypto.rsa	−0.91%	−0.17%	0.00%	−0.47%
	crypto.signverify	0.24%	0.30%	0.55%	0.72%
	derby	2.30%	4.70%	0.00%	1.00%
	mpegaudio	−0.08%	−0.24%	0.65%	0.00%
	scimark.fft.large	−0.13%	−0.32%		
	scimark.fft.small	−0.32%	−0.06%	0.71%	0.00%
	scimark.lu.large	0.14%	0.42%	0.00%	0.00%
	scimark.lu.small	−2.54%	−2.76%	0.09%	−0.48%
	scimark.monte carlo	−0.27%	−0.26%	0.00%	0.00%
	scimark.sor.large	−0.17%	−0.22%	0.00%	0.00%
	scimark.sor.small	−0.94%	−1.30%	0.00%	100.00%
	scimark.sparse.large	0.88%	0.11%		
	scimark.sparse.small	−2.15%	6.04%	−4.65%	−5.56%
	serial	−6.77%	−9.52%	−4.59%	−3.95%
	sunflow	−0.72%	−1.17%	−0.15%	−0.87%
	xml.transform	−0.89%	−0.85%	4.18%	4.36%
	xml.validation	0.01%	0.54%	0.00%	0.30%

Figure 97: Performance increase when optimistically not generating the identity hash code

		ParallelOld GC			G1 GC		
	Benchmark	Parse time	Parse ratio	Parsable	Parse time	Parse ratio	Parsable
DaCapo	avrora	8.00	115.96%	yes	9.05	129.03%	yes
	batik	19.80	10476.19%	yes	17.72	10610.78%	no
	eclipse	8.45	7897.20%	yes	7.27	6321.74%	no
	fop	10.75	7363.01%	yes	13.96	9306.67%	yes
	h2	257.24	174.05%	yes	2241.04	1414.44%	yes
	jython	179.42	2102.66%	yes	65.97	700.91%	no
	luindex	1.64	504.62%	yes	3.52	997.17%	yes
	lusearch	20.51	882.91%	yes	88.36	2283.20%	yes
	pmd	23.48	1937.29%	yes	28.82	2379.85%	yes
	sunflow	106.52	2364.48%	yes	121.82	3030.35%	yes
	tomcat	9.24	1113.25%	yes	15.20	1826.92%	yes
	tradebeans	481.92	884.84%	yes	617.34	1038.86%	yes
	tradesoap	605.22	1441.65%	yes	114.63	191.97%	no
	xalan	84.85	1405.97%	yes	136.35	1491.96%	yes
DaCapoScala	actors	45.72	342.24%	yes	57.58	407.39%	yes
	apparat	116.10	143.22%	yes	129.88	184.46%	yes
	factorie	1791.77	1772.45%	yes	1990.43	1730.36%	yes
	kiama	30.93	12573.17%	yes	22.97	9375.51%	no
	scalac	67.85	2872.57%	yes	56.16	2322.58%	yes
	scaladoc	67.91	3157.14%	yes	61.22	2756.42%	yes
	scalap	5.55	3208.09%	yes	5.61	3666.67%	yes
	scalariform	31.66	1744.35%	yes	33.52	1790.60%	yes
	scalatest	7.29	1151.66%	no	7.10	1074.13%	no
	scalaxb	51.51	444.05%	yes	61.36	510.53%	yes
	specs	7.07	448.32%	no	7.04	450.42%	no
	tmt	1119.37	3006.96%	yes	1283.78	4183.47%	yes
SPECjvm	compiler.compiler	746.88	5777.23%	yes	749.38	4517.06%	yes
	compiler.sunflow	1989.66	5424.96%	yes	1910.99	4396.52%	yes
	compress	0.75	13.10%	yes	1.28	22.19%	yes
	crypto.aes	1.05	12.87%	yes	2.91	36.74%	yes
	crypto.rsa	22.61	649.15%	yes	27.41	811.91%	yes
	crypto.signverify	3.78	55.53%	yes	9.47	134.63%	yes
	derby	1383.48	4533.77%	yes	1769.78	5338.38%	yes
	mpegaudio	3.12	35.91%	yes	4.79	55.27%	yes
	scimark.fft.large	0.59	7.49%	yes	1.33	16.98%	yes
	scimark.fft.small	1.03	15.15%	yes	3.42	46.33%	yes
	scimark.lu.large	0.94	2.52%	yes	5.49	15.36%	yes
	scimark.lu.small	22.11	229.48%	yes	33.15	399.83%	yes
	scimark.monte carlo	0.68	7.07%	yes	1.05	11.69%	yes
	scimark.sor.large	0.78	11.18%	yes	2.43	34.16%	yes
	scimark.sor.small	0.67	9.06%	yes	1.06	13.62%	yes
	scimark.sparse.large	0.60	5.11%	yes	1.23	10.44%	yes
	scimark.sparse.small	0.82	30.19%	yes	1.36	36.47%	yes
	serial	2470.82	4967.07%	yes	2994.89	6064.37%	yes
	sunflow	1479.35	2592.12%	yes	2226.03	3636.71%	yes
	xml.transform	161.83	2469.93%	yes	223.27	3398.84%	yes
	xml.validation	875.57	4449.71%	yes	916.26	3956.22%	yes

Figure 98: Parse time (seconds) and parse ratio when parsing the benchmark's traces

monitoring and these problems only occur with very small heaps (with only a few megabytes) rather than with large heaps, we consider that goal to be met.

Chapter 9

Related Work

"If I have seen further it is by standing on the shoulders of giants that came before me."
- Isaac Newton

9.1 Finding Performance Degradations

Chis et al. [13] present 11 patterns of memory inefficiency that can be used to detect memory-related performance problems. They claim that these patterns are sufficient to detect most memory-related performance problems and evaluate them with several case studies.

Blackburn et al. [5] present a study on the performance impact of garbage collection on the application using several different garbage collection algorithms. They performed their study using the Jikes RVM [1], originally published in Alpern et al. [2]. That VM is for research purposes only. It is implemented in Java, designed for clarity and simplicity to support quick experimentation, but is not suitable for production use. Therefore, their study is difficult to compare to real-world production VMs.

Jones et al. [24] provide a study on object lifetime behavior. They show that allocation sites alone are not a good predictor of object lifetimes, but rather allocate objects of a small range of life times that are robust with respect to benchmark's input size.

Bu et al. [10] describe bloat-aware design patters as well as performance-degrading patterns for GC-oriented languages such as Java.

9.2 Memory Leak Detection

Xu et al. [50] present *LeakChaser*, a tool that enables programmers at different knowledge levels about the application to track down memory leaks. On the one hand, their tool allows low-knowledge programmers,

[1] Jikes RVM: http://www.jikesrvm.org/

i.e., programmers that just know about the basic components and APIs, to define transactions, which the tool will use to automatically infer lifetime constraints. On the other hand, high-knowledge programmers, i.e., programmers familiar with the inner workings and algorithms of the application, can define fine-grained lifetime constraints on individual pairs of objects. They show that their tool could resolve several memory leaks in real-world applications and that the run-time overhead of LeakChaser is about 10% (geometric mean, max 210%) and the GC-time overhead is about 230% (geometric mean, max 370%). Although their tier-based approach for different knowledge levels of the user is interesting for a scientific point of view, they use the Jikes RVM, making the overhead numbers difficult to compare.

Aftandilian et al. [1] introduce *GC assertions*, an interface that can be used to check invariants and find memory leaks. Such assertions include *assert-dead* (checks whether an object can be collected), *assert-instances* (checks whether the number of instances do not exceed a given limit), and *assert-unshared* (checks whether the given object has only one incoming pointer). The evaluation of these assertions are piggybacked onto the GC because the GC has to traverse the heap anyway and knows most about object lifetimes and connections. *GC assertions* can be very useful for both finding memory leaks and bugs as well as for in-code documentation, i.e., a well-placed assertion can explain a lot about the code to a new reader. However, the authors have again implemented their approach into the Jikes RVM, making their overhead measurement of 2.57% (geometric mean) difficult to compare. Also, the experiments where conducted on a machine with only 2 GB RAM, making the actual heap of the Java application even smaller.

Pauw and Sevitsky [39] propose defining object lifetime constraints and use those constraints to detect memory leaks. This work can be regarded as a simplified version of the *GC assertions* described above.

Xu et al. [51] propose to profile containers only, i.e., collection data structures (e.g., lists and maps), to keep track of growing containers and to even identify which elements are accessed frequently and which are not accessed at all. Due to the context of a container, and the calls to their retrieval methods, they can identify the users of elements, which puts objects in an easy-to-understand context. The authors implemented their approach using JVMTI and thus do not need a custom VM. Their approach introduces an overhead of 88.5% (geometric mean).

Printezis et al. [41] present *GC Spy*, an adaptable heap visualization framework. They provide a powerful API to visualize heap data. However, *GC spy* can only visualize blocks, not individual objects.

9.3 Tracing

Chilimbi et al. [12] propose a new generic trace format for heap allocation events to enable simple exchange and definition of garbage collector workloads. They also define a new meta language to describe this trace format. On average, their approach generates 5.6 bytes per event (excluding compression). By compression and by extracting the addresses into a separate stream, they can reduce the average event size to 1.54 bytes. However, they do not describe the run-time overhead when generating such a trace at full compression. Also, their trace does not contain garbage collection events, so that it cannot be used to reconstruct the heap at certain points in an application.

Harkema at al. [18] present a tool that instruments Java applications and generates a stream of events that the user specified, i.e., thread creations, method invocations, or lock contentions. They are also recording object allocations and deallocations using the old Java Virtual Machine Profiling Interface (JVMPI) (now replaced by JVMTI). However, they seem to have not evaluated the generation of traces in terms of overhead.

Hertz et al. [19, 20] present the *Merlin* algorithm for generating fine-granular traces of object allocations, object deaths, and pointer updates. Whenever a pointer is removed from an object, a timestamp is written into that object that can later be used to determine the time of its death. *Merlin* is more precise than accumulates trace. Moreover, the algorithm does not need a whole-heap GC at every possible GC point to detect object deaths exactly. They do claim that this approach is 800 times faster than their previous whole-heap GC approach, but they do not provide any data about the absolute overhead. Also, they have implemented the algorithm into the Jikes RVM.

Ricci et al. [42, 43] describe *Elephant Tracks*, a profiling tool that can record method invocations as well as object allocations and deallocations. It uses JVMTI and bytecode instrumentation to record events and to insert custom monitoring code into the application code. *Elephant Tracks* builds onto the *Merlin* algorithm to detect the time of death. They introduce an run-time overhead of 24,580% (cf. Section 2.4).

Brown et al. [9] introduce *STEP*, a generic trace definition language as well as a flexible and compact format for trace events. The goal of this work is to ease the definition, understand, and processing of tracing data. They also show that their trace formats can easily be compressed down to about 4% of the original size. However, they do not include any performance numbers with respect to compression-time.

Burtscher et al. [11] present a new compression algorithm, specifically designed for execution traces. The authors show that their algorithm outperforms all other well-known general-purpose compression algorithms in both compression rate and compression time. They partially achieve this by exploiting *local value locality* and a simple value predictor.

Mohror and Karavanic [36] try to reduce the size of traces based on similarity. The authors search patterns in the trace and keep only one occurrence of that pattern. However, finding such patterns is very costly and may introduce errors, i.e., change the semantics of the trace. Thus, this technique is not suitable for on-the-fly compression.

Wagner and Nagel [47] present strategies for real-time event reduction. Their goal is not to keep all events but to reduce the number of events to keep all in memory. Although interesting, this technique is not applicable for AntTracks, as we need to keep all events.

Janjusic and Kavi [23] present the memory profiling tool *Gleipnir*. This tool is a plugin for valgrind, a well-known memory analysis tool for binary-compiled applications. They admit that their tool is not 100% accurate due to fancy pointer manipulations they are not capable to observe properly. Also, they do not provide any overhead information, most likely because this tool, like valgrind, is more for debugging than for profiling.

Marathe et al. [34] propose to detect cache inefficiencies by dynamic binary rewriting. They generate a trace of memory accesses and extract symbol information from the executable binary to associate events with locations in the source code. Using this information they can pinpoint the cause of a memory inefficiency. Since their focus seems to lie on recording events such as cache misses, they do not provide any overhead information.

Peiris and Hill [40] present the *Excessive Memory Allocation Detector* (EMAD). This tool is able to rewrite native binaries and insert code to automatically detect allocations (`malloc` calls) and deallocations (`free` calls). It then applies a variety of analysis algorithms to detect excessive memory allocations. Since they focus on native applications without a garbage collector, the authors do not have to deal with the problem that deallocations cannot be instrumented. Also, the authors seem to be only interested in finding the excessive allocations in a test setup. No overhead is reported.

Horky et al. [21] present an analysis about the effects of adding and removing instrumentations at run-time. Although their focus is on run-time monitoring of method executions, they also need to deal with the problem that too many instrumentations distort application behavior. Therefore, they add and remove instrumentations as needed by retransforming Java classes at run-time. They report observations similar to ours, e.g., that instrumentation may both speed up as well as slow down the execution. While this paper focuses on run-time monitoring, one could also imagine applying the presented method to memory monitoring.

Chapter 10

Conclusion

"I may not have gone where I intended to go, but I think I have ended up where I needed to be." - Douglas Adams

This thesis addresses the distortions and overheads state-of-the-art memory monitoring tools introduce in the monitored systems. Being well aware of the overheads introduced, every monitoring tool chooses one of two approaches: (1) collecting only coarse-grained information, thus being feasible to be used in production environments but hardly providing enough data to track down the root cause of a performance degradation, or (2) collecting fine-grained information, thus collecting enough data to track down the root cause but disqualifying the tool for production use due to its overhead. Furthermore, the second approach may also distort the application's behavior, i.e., the observed behavior may not represent the application's behavior in an unmonitored state.

In this thesis, we have quantified the behavioral distortion in state-of-the-art memory monitoring techniques. We have shown that these techniques either do not collect enough data to track down the root causes of memory leaks, high memory consumption, or long GC times, or they introduce so much overhead that, in the worst case, applications crash due to internal timeouts, or, in the better case, do not crash but completely change the GC behavior. Also, this thesis presented AntTracks, a mechanism built into a state-of-the-art production virtual machine, that is able to record fine-grained information at object-level, with only very little overhead and without distorting the application's behavior and thus combining the advantages of both techniques. AntTracks writes a trace, i.e., a sequence of events, where every event represents a single memory operation, e.g., an object allocation. The trace format is very compact, omitting all information that can be reconstructed offline. This makes individual events very small, even JIT-compile-time constant, enabling us to record memory operations very fast. We then use an offline tool to reconstruct omitted information. This tool can then reproduce the heap for any point in the application's lifetime. While existing tools always require a multiple of the monitored heap's size for analysis (if they can reproduce the heap at all), our tool only requires as much heap as the monitored application. This property makes AntTracks even applicable for monitoring memory behavior on high-performance machines, because no *better* machine is needed for analysis. To evaluate all these claims, we provide a two-part evaluation: (1) an overhead evaluation showing that AntTracks' overhead is small enough for production environments and showing that there is almost no distortion, and (2) a by-example evaluation of how the undistorted data AntTracks is collecting can be used to track down the root causes of memory-related performance problems that state-of-the-art approaches cannot easily find.

The contributions of this thesis are (1) the design and implementation of a specialized Java VM that is able to record memory events at object-level with minimal overhead and that does not distort the application's memory behavior, (2) an efficient technique for multi-threaded parsing and analysis of such a memory trace, (3) a detailed evaluation of the performance of the AntTracks VM as well as of its capabilities, and (4) an extensive analysis of the state of the art in memory monitoring.

Since todays applications as well as their underlying execution environments get more complex, we need tooling to help developers resolve performance problems. We have shown that AntTracks, while producing only very little overhead, can detect a wide variety of memory-related performance problems in real-world applications.

"Je n'ai fait celle-ci plus longue que parce que je n'ai pas eu le loisir de la faire plus courte." - Blaise Pascal

Appendix A

Firing Allocation Events

This section describes how to fire allocation events efficiently in the virtual machine, the interpreter, client-compiled code, and server-compiled code respectively.

Virtual Machine. The VM provides C functions to allocate new objects. These functions are used primarily when the VM itself needs to allocate objects, e.g., for system classes. As these functions can also trigger GCs if necessary or allocate a new TLAB, they are also used as a fallback by compiled code if the generated code fails to allocate the object.

All allocation functions are instrumented so that they write the according allocation event (with the special allocation site `VM_INTERNAL` to the thread-local buffer. Because these fallbacks are slow and rare, no special effort must be taken to optimize them.

Interpreter. The interpreter in the Hotspot^{TM} VM is a *template interpreter*. This means, that instead of having a loop over a big switch with all bytecode implementations, this interpreter generates a piece of code at VM startup for every bytecode. This implementation has two major advantages: (1) VM developers do not have to rely on the compiler to generate efficient interpreter code but instead they can define it themselves, and (2) the generated code can be adjusted depending on the VM configuration, making expensive run-time checks for specific configuration options superfluous.

Examining the interpreter implementation has shown that only for instances (bytecode `new`), the allocation code is generated directly. The bytecodes `newarray`, `anewarray`, and `multianewarray` are implemented as calls to the VM-internal C functions described above. For instances, we thus add a call after the allocation to fire the according allocation event. For arrays, the method called to allocate the array already fires an allocation event, so we only need to pass the correct allocation site.

Resolving the correct allocation site in the interpreter must be done dynamically because the same bytecode template is executed for different methods. The interpreter frame, i.e., the method frame layout that is used to execute interpreted methods, contains a slot for the current method and the current BCI. Using this information, we need to resolve the allocation site just before the allocation event is fired.

Although resolving the allocation site over and over again, as well as not making use of any fast path for instances imposes some overhead, we do not optimize these cases because they are slow anyway and hot code will eventually be JIT compiled so that frequent allocations will finally be executed in machine code.

Client-compiled Code. The HotSpot^{TM} client compiler transforms bytecode into a high-level intermediate representation (HIR), lowers the HIR into a low-level intermediate representation (LIR), and finally compiles the LIR into executable machine code. All optimizations are done on the HIR and LIR by replacing and reordering instructions.

We insert code for firing allocation events during the compilation of an LIR allocation instruction to executable machine code. Consequently, we do not impede any compiler optimizations because all optimizations have already been carried out at that point.

The client compiler uses a *macro assembler* to translate LIR instructions into machine code. We have instrumented this assembler so that for allocation instructions, additional code is inserted. Figure 99 shows the fast path code as well as the slow path code generated to fire an allocation event (fast version) with the pseudo code in Figure 42 as reference.

In that example, the encoded event (`0x1701e400` is moved into a temporary register first (the actual register allocation may vary). Then, top (`0x1b8(%r15)`) and end (`0x1c0(%r15)`) of the thread-local buffer are loaded into temporary registers as well. These pointers can be accessed relatively to `r15`, because all compiled code always keeps the thread pointer in that register for performance reasons. If the top plus the event size (4) is greater than the end, i.e., if the thread-local buffer is full, the code jumps to the slow path. Otherwise, the event loaded before is written to the location top is pointing to, top is incremented and stored back into the thread. The generated code for the array is the same except for an additional check of the array size (and a jump to the slow path if it is too big) and the combination of the event and the length.

The slow path moves the allocation site to the stack as an argument to the subsequent call of a C method which is able to fetch a new thread-local buffer if necessary or to write a more complex event with a dedicated 32-bit length field (for arrays). After the call, the code jumps back to the next instruction after the fast path.

The slow path has several special characteristics one has to keep in mind: (1) The code for the slow path is not placed right after the fast path because it is executed rarely and the common (fast path) case should be able to execute without a jump. Consequently, all slow paths are located at the very end of the method and include two jumps; (2) The compiler determines the stack frame layout and has already allocated the slot for the argument of the slow path call at method entry. Thus, the argument must only be moved to the stack and not pushed onto it; (3) Every call to the VM automatically transitions the thread from `in_java` to `in_vm` and back. It also contains a safepoint check because the VM may want to suspend itself. Therefore, a GC may occur at the slow path call, and an an `OutOfMemoryError` might be thrown. As the exception needs to walk the stack, we need to associate the call with debug information, i.e., method and bytecode index.

Server-compiled Code. The HotSpot^{TM} server compiler transforms the bytecode into a *sea of nodes*. Every node represents a typed operation, e.g., an integer `add` or the loading of a constant, and contains its operands (if any) as children. Additionally, every node has a *control flow parent* as well as a *memory parent*. These parents indicate which node is the dominator in terms of control flow, i.e., which node is the last node that must be executed before this one, and which node changed the memory before this one.

All optimizations are executed on the sea of nodes by reordering nodes (with respect to their control flow and memory parents) and replacing nodes. Some high-level operations, e.g., allocations, are represented as a single *macro node*. These nodes simplify optimizations such as escape analysis and scalar replacement (an optimization that can remove unnecessary object allocations by moving the fields of the allocated object as variables onto the method stack). In a later phase, these macro nodes are expanded into the actual low-level nodes (e.g., loads, stores, adds) to allocate an object.

```
0x7fad5d122931: mov     $0x1701e400,%esi
0x7fad5d122936: mov     0x1b8(%r15),%rcx
0x7fad5d12293d: mov     0x1c0(%r15),%rdi
0x7fad5d122944: sub     $0x4,%rdi
0x7fad5d122948: cmp     %rdi,%rcx
0x7fad5d12294b: jg      0x7fad5d122aaf
0x7fad5d122951: mov     %esi,(%rcx)
0x7fad5d122953: add     $0x4,%rcx
0x7fad5d122957: mov     %rcx,0x1b8(%r15)

0x7fad5d122aaf: movq    $0x1e4,(%rsp)
0x7fad5d122ab7: callq   0x7fad5d0fe860
; OopMap{rbx=Oop r9=Oop off=860}
; *new
; java.lang.String::substring@65 (line 1969)
; {runtime_call}
0x7fad5d122abc: jmpq    0x7fad5d12295e
```

```
0x7ffadd1438c1: mov     $0x17024100,%edi
0x7ffadd1438c6: cmp     $0xff,%ebx
0x7ffadd1438cc: jge     0x7ffadd143c36
0x7ffadd1438d2: or      %ebx,%edi
0x7ffadd1438d4: mov     0x1b8(%r15),%rcx
0x7ffadd1438db: mov     0x1c0(%r15),%rsi
0x7ffadd1438e2: sub     $0x4,%rsi
0x7ffadd1438e6: cmp     %rsi,%rcx
0x7ffadd1438e9: jg      0x7ffadd143c36
0x7ffadd1438ef: mov     %edi,(%rcx)
0x7ffadd1438f1: add     $0x4,%rcx
0x7ffadd1438f5: mov     %rcx,0x1b8(%r15)

0x7ffadd143c36: movq    $0x241,(%rsp)
0x7ffadd143c3e: callq   0x7ffadd0fe8e0
; OopMap{r11=Oop off=1475}
; *newarray
; java.lang.String::replace@48 (line 2078)
; {runtime_call}
0x7ffadd143c43: jmpq    0x7ffadd1438fc
```

Figure 99: Client-compiled code for firing an allocation event (fast version, id = 0x17) for an instance (`new String @ java.lang.String::substring(II)`, allocation site 0x1e4) (left) and for an array (`new char[] @ java.lang.String::replace(CC)`, allocation site 0x241) (right)

We added the code to fire the allocation event as a subgraph of nodes during the expansion of the allocation macro node to low-level nodes. Therefore, we do not impede compiler optimizations such as scalar replacement because if this optimization would apply, the entire allocation node would have been removed in the first place. Moreover, embedding our code into the sea of nodes enables the compiler to automatically optimize it for that special allocation.

Figure 100 shows the subgraph of nodes we need to add to fire the allocation event. The start block is connected to the actual allocation whereas the end block is connected to whatever follows the allocation. Every circle is a node, every box is a group of nodes defined outside the graph (*obj* and *length* are external input nodes defined depending on the allocation). The black edges show the children of a node (usually located right below), whereas the red and blue edges represent the control flow parent and the memory parent respectively. Although, every node has a control flow and memory parent, we omitted the edges if the parent is the same as the parent of the node that is referencing this node as a child. Finally, the turquoise parts are only necessary if the allocation is an array allocation.

At first, the invocation counter is increased (below the start block). This is accomplished by loading the address as a constant (`ConP` node and `LoadL` node) incrementing it (`AddL` node, and writing it back (`StoreL` node.

The next part of the graph checks whether the thread-local buffer is full. To do this, it first loads the `top` and the `end` pointer. The top pointer is incremented and compared (`CompP` node) with the end pointer. Subsequently, the compare node is embedded into a `Bool` node, checking whether the left operand of the `CompP` node is greater than (`gt`) the right operand. This boolean node can then be used in an `If` node, which represents the branching at that point. Additionally, this node can also store profiling information, i.e., the

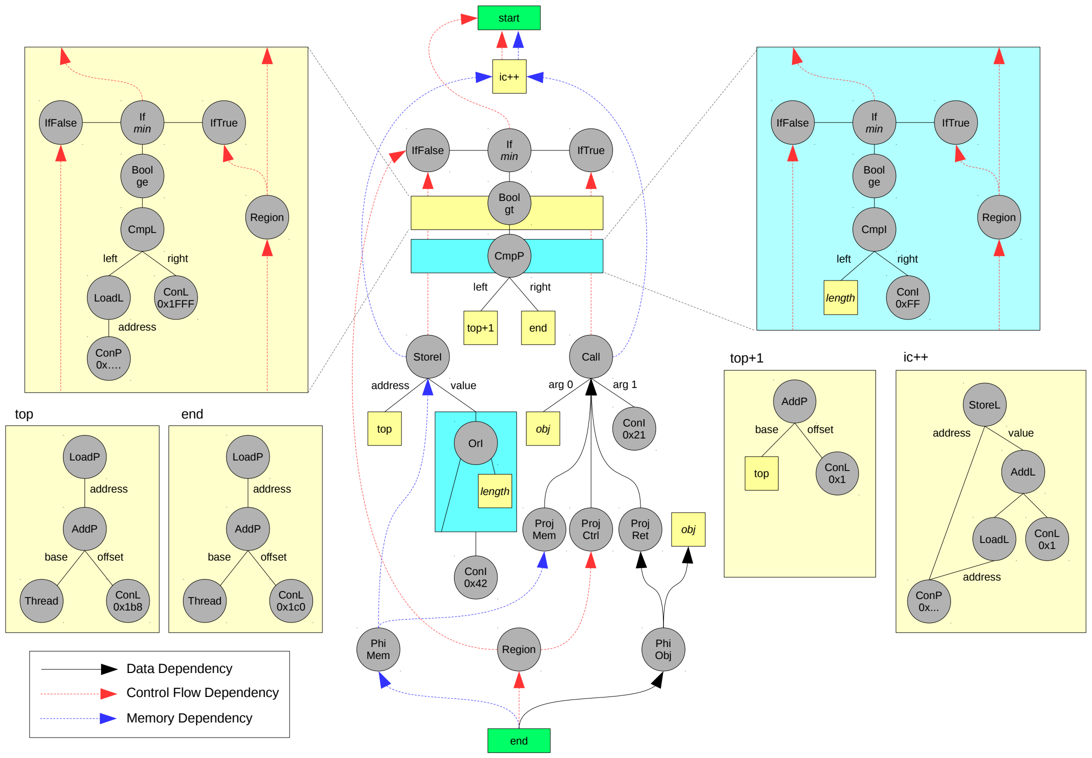


Figure 100: Server compiler nodes for recording an allocation (id = `0x21`, event = `0x42`)

likelihood of every branch. In our case we assume that the fast path is the common case, and, consequently, assume a *minimal probability* that the true branch will be taken. Finally, a `IfTrue` and a `IfFalse` node represent the control flow parents for every branch respectively.

The second check we always need to do is to determine whether we need to take a stack trace. This is done by checking whether the invocation counter has reached a specific threshold (in this case 0x1FFF). In this case, we add another `If` node, comparing the length with a constant long (`ConL`) node representing 0x1FFF. If that check yields false we may continue with the fast path, in which case this `IfFalse` node becomes the new control flow parent for that path. If the check yields true (again, we assume that this is very unlikely), we must continue with the slow path. In this case, we will override the allocation site with 0 (`UNKNOWN`) which forces the slow path to take a stack trace. The control flow parent of the slow path is now either the `IfTrue` node of this check, or the `IfTrue` node of the previous check, depending on which check resulted in jumping to the slow path. Therefore, we merge both control flows into one using a `Region` node and use that as the control flow parent in the slow path.

If the allocation was an array allocation, we also have to check whether the array length fits into our fast event version (big turquoise block). In this case, we add another `If` node, comparing the length with a constant integer (`ConI`) node representing 0xFF. If that check yields false we may continue with the fast path, in which case this `IfFalse` node becomes the new control flow parent for that path. If the check yields true (again, we assume that this is very unlikely), we must continue with the slow path. The control flow parent of the slow path is now either the `IfTrue` node of this check, or the `IfTrue` node of the previous check, depending on which check resulted in jumping to the slow path. Therefore, we merge both control flows into one using a `Region` node and use that as the control flow parent in the slow path.

The *fast path* is just a `StoreI` node with the top pointer as address and the event as value. The event is a constant for instance allocation and a bit-wise `or` of that same constant and the length for an array allocation.

The *slow path* is a *Call* node with the just allocated object and the allocation site as parameters. The object must be passed to the call for two reasons: (1) the slow path may need the object, e.g., to determine the type or the array length, and (2) this call (as every call) executes an automatic transition of the executing thread from `in_java` to `in_vm`, including a safepoint check. As a garbage collection could occur while making the transition in this call, the object could be moved by the GC. The easiest way to handle this case, is to pass the object as parameter, and return it again for subsequent use. This way, the compiler will automatically update the address of the object.

Finally, before continuing with whatever code follows the allocation, we need to merge the control flow of the fast path and the slow path again using a `Region` node. Additionally, we need a `Phi` node to merge the memory modifications both paths have executed. In case of the fast path, the memory parent is the `StoreI` node, in case of the slow path it is the call itself (more specifically, a `Proj` node of the memory modifications made by the `Call` node. As we have passed the object through the slow path call, we also need a `Phi` node for the object, merging (in case of the fast path) the original obj with the `Proj` node of the return value for the `Call` node.

Figure 101 shows the generated code for an instance allocation and an array allocation based on the tree defined in Figure 100. First, the invocation counter for that allocation is incremented, by loading the address as a constant, loading the value at that address, incrementing it, and finally writing it back. Next, the top pointer is loaded (`r15` always holds the pointer to the current thread, top and end of the thread-local buffer can be accessed relatively to that register), incremented, and compared to the end of the thread-local buffer. If the thread-local buffer is full, the code jumps to the slow path. The slow path is also jumped to if the invocation counter has reached the configurable threshold (0x1FFF in this example). If neither check triggers a jump to the slow path, the pre-compiled event is written to top, and the incremented top value is written back.

The slow path is relocated to the end of the method, because we have specified that it is very unlikely that it will be taken. Therefore, the fast path can be executed without a single jump. Additionally, the allocation site is set to zero if the invocation counter has reached the specified threshold to indicate to the slow path that a stack trace must be taken. Before, the call, the compiler automatically takes care of spilling all registers onto the stack and restoring them after the call.

Please note, that the register allocation is highly volatile, and many values are only loaded once. For example, the top pointer is only incremented once, before the check whether the buffer is full, and stays unchanged in the register it is loaded into. It can the be re-used to write that pointer back after storing the event. Moreover, the allocation site is loaded into a register during the actual allocation, although it is only needed when jumping to the slow path. The compiler can do this, because the register is not needed, and this is an instruction that can be executed completely in parallel by the processor. These optimizations are not specified explicitly in the tree, but introduced automatically by the compiler.

Figure 102 shows optimized code for firing an allocation event for an array with constant size. The array is allocated in `java.lang.AbstractStringBuilder::<init>(I)`, whereas the length is determined by the `int` parameter. However, in this case, the method is inlined into `java.lang.String::<init>()`, which always calls this method with the constant parameter 16. Therefore, the subtree representing the length is a constant node. Thus, the compiler knows that the array length check will always succeed and the slow path will never have to be taken. Consequently, this check is automatically removed from the code. Moreover,

```
0x7fad5d2e17fb: mov     $0x1e4,%edx



0x7fad5d2e1831: mov     $0x7fad6c00ee60,%r10
0x7fad5d2e183b: mov     $0x1,%r11d
0x7fad5d2e1841: add     (%r10),%r11
0x7fad5d2e1844: mov     %r11,(%r10)
0x7fad5d2e1847: mov     0x1b8(%r15),%r10
0x7fad5d2e184e: mov     %r10,%r8
0x7fad5d2e1851: add     $0x4,%r8
0x7fad5d2e1855: cmp     0x1c0(%r15),%r8
0x7fad5d2e185c: ja      0x7fad5d2e1b31
0x7fad5d2e1862: test    $0x1fff,%r11
0x7fad5d2e1869: je      0x7fad5d2e1b2f
0x7fad5d2e186f: movl    $0x1c01e400,(%r10)
0x7fad5d2e1876: mov     %r8,0x1b8(%r15)



0x7fad5d2e1b2f: xor     %edx,%edx
0x7fad5d2e1b31: mov     %rbx,%r10
0x7fad5d2e1b34: mov     %rsi,0x10(%rsp)
0x7fad5d2e1b39: mov     %rax,-0x8(%rsp)
0x7fad5d2e1b3e: mov     0x8(%rsp),%eax
0x7fad5d2e1b42: mov     %eax,(%rsp)
0x7fad5d2e1b45: mov     -0x8(%rsp),%rax
0x7fad5d2e1b4a: mov     %r10,%rsi
0x7fad5d2e1b4d: xchg    %ax,%ax
0x7fad5d2e1b4f: callq   0x7fad5d0ff060
; OopMap{[16]=Oop off=980}
; *new
; java.lang.String::substring@65 (line 1969)
; {runtime_call}
0x7fad5d2e1b54: rex.W pushq   0x10(%rsp)
0x7fad5d2e1b59: rex.W popq    (%rsp)
0x7fad5d2e1b5d: mov     %rax,%rbx
0x7fad5d2e1b60: jmpq    0x7fad5d2e187d
```

```
0x7ffadd1ba25d: mov     $0x241,%edx


0x7ffadd1ba2c5: mov     $0x7ffaec00ec20,%rcx
0x7ffadd1ba2cf: mov     $0x1,%eax
0x7ffadd1ba2d4: add     (%rcx),%rax
0x7ffadd1ba2d7: mov     %rax,(%rcx)
0x7ffadd1ba2da: mov     0x1b8(%r15),%rcx
0x7ffadd1ba2e1: mov     %rcx,%rdi
0x7ffadd1ba2e4: add     $0x4,%rdi
0x7ffadd1ba2e8: cmp     0x1c0(%r15),%rdi
0x7ffadd1ba2ef: ja      0x7ffadd1ba69f
0x7ffadd1ba2f5: cmp     $0xff,%r13d
0x7ffadd1ba2fc: jge     0x7ffadd1ba69f
0x7ffadd1ba302: test    $0x1fff,%rax
0x7ffadd1ba309: je      0x7ffadd1ba69d
0x7ffadd1ba30f: mov     %r13d,%ebx
0x7ffadd1ba312: or      $0x1c024100,%ebx
0x7ffadd1ba318: mov     %ebx,(%rcx)
0x7ffadd1ba31a: mov     %rdi,0x1b8(%r15)


0x7ffadd1ba69d: xor     %edx,%edx
0x7ffadd1ba69f: mov     %rbp,%rcx
0x7ffadd1ba6a2: mov     %r8d,0x14(%rsp)
0x7ffadd1ba6a7: mov     %r9d,0x10(%rsp)
0x7ffadd1ba6ac: mov     %esi,0xc(%rsp)
0x7ffadd1ba6b0: mov     %r13d,0x8(%rsp)
0x7ffadd1ba6b5: mov     %r11d,(%rsp)
0x7ffadd1ba6b9: mov     %r10d,%ebp
0x7ffadd1ba6bc: mov     %rcx,%rsi
0x7ffadd1ba6bf: mov     %rbx,0x18(%rsp)
0x7ffadd1ba6c4: xchg    %ax,%ax
0x7ffadd1ba6c7: callq   0x7ffadd06f1a0
; OopMap{[0]=NarrowOop [24]=Oop off=1292}
; *newarray
; java.lang.String::replace@48 (line 2078)
; {runtime_call}
0x7ffadd1ba6cc: mov     %ebp,%r10d
0x7ffadd1ba6cf: mov     (%rsp),%r11d
0x7ffadd1ba6d3: mov     0x8(%rsp),%r13d
0x7ffadd1ba6d8: mov     0xc(%rsp),%esi
0x7ffadd1ba6dc: mov     0x10(%rsp),%r9d
0x7ffadd1ba6e1: mov     0x14(%rsp),%r8d
0x7ffadd1ba6e6: mov     %rax,%rbp
0x7ffadd1ba6e9: jmpq    0x7ffadd1ba321
```

Figure 101: Server-compiled code for firing an allocation event (fast version, id = 0x1c) for an instance (`new String @ java.lang.String::substring(II)`, allocation sit 0x1e4) (left) and for an array (`new char[] @ java.lang.String::replace(CC)`, allocation sit 0x241) (right)

the encoded event is constant like in the instance case because the length can be combined with the rest of the event at compile-time.

```
0x7f3441822a95: mov     $0x1,%r10d
0x7f3441822a9b: mov     $0x7f3400687040,%r11
0x7f3441822aa5: add     (%r11),%r10
0x7f3441822aa8: mov     %r10,(%r11)
0x7f3441822aab: mov     0x1b8(%r15),%r11
0x7f3441822ab2: mov     %r11,%r8
0x7f3441822ab5: add     $0x4,%r8
0x7f3441822ab9: cmp     0x1c0(%r15),%r8
0x7f3441822ac0: ja      0x7f3441822b47
0x7f3441822ac6: test    $0x1fff,%r10
0x7f3441822acd: je      0x7f3441822b45
0x7f3441822acf: movl    $0x1c04fe10,(%r11)
0x7f3441822ad6: mov     %r8,0x1b8(%r15)
```

Figure 102: Optimized server-compiled code for firing an optimized allocation event for an array with a constant size (`new char[16] @ java.lang.AbstractStringBuilder::<init>(I) inlined into java.lang.String::<init>()`)

Bibliography

[1] E. E. Aftandilian and S. Z. Guyer. Gc assertions: Using the garbage collector to check heap properties. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 235–244, New York, NY, USA, 2009. ACM.

[2] 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.

[3] V. Bitto and P. Lengauer. Building custom, efficient, and accurate memory monitoring tools for java applications. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 321–324, 2016.

[4] 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, 2015.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 25–36, New York, NY, USA, 2004. ACM.

[6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[7] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, Aug. 2008.

[8] G. Blaschek and P. Lengauer. Time matters: Minimizing garbage collection overhead with minimal effort. In *Proceedings of the Symposium on Software Performance*, SSP '15, 2015.

[9] 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. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '02, pages 27–34, New York, NY, USA, 2002. ACM.

[10] Y. Bu, V. Borkar, G. Xu, and M. J. Carey. A bloat-aware design for big data applications. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 119–130, New York, NY, USA, 2013. ACM.

[11] 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.

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

[13] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, and J. Murphy. Patterns of memory inefficiency. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 383–407, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM.

[15] D. Detlefs, R. Knippel, W. D. Clinger, and M. Jacob. Concurrent remembered set refinement in generational garbage collection. In *Proceedings of the 2Nd Java&#153; Virtual Machine Research and Technology Symposium*, pages 13–26, Berkeley, CA, USA, 2002. USENIX Association.

[16] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16, pages 13:1–13:9, New York, NY, USA, 2016. ACM.

[17] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[18] M. Harkema, D. Quartel, B. M. M. Gijsen, and R. D. van der Mei. Performance monitoring of java applications. In *Proceedings of the 3rd International Workshop on Software and Performance*, WOSP '02, pages 114–127, New York, NY, USA, 2002. ACM.

[19] 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 *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, pages 140–151, New York, NY, USA, 2002. ACM.

[20] 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.

[21] V. Horký, J. Kotrč, P. Libič, and P. Tůma. Analysis of overhead in dynamic java performance monitoring. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 275–286, New York, NY, USA, 2016. ACM.

[22] V. Horký, P. Libič, A. Steinhauser, and P. Tůma. Dos and don'ts of conducting performance measurements in java. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 337–340, New York, NY, USA, 2015. ACM.

[23] T. Janjusic and K. Kavi. Gleipnir: A memory profiling and tracing tool. *SIGARCH Comput. Archit. News*, 41(4):8–12, Dec. 2013.

[24] R. E. Jones and C. Ryder. A study of java object demographics. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 121–130, New York, NY, USA, 2008. ACM.

[25] K. C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, Oct. 1965.

[26] P. Lengauer. Vm-level memory monitoring for resolving performance problems. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, &#38; Applications: Software for Humanity*, SPLASH '13, pages 29–32, 2013.

[27] P. Lengauer, V. Bitto, F. Angerer, P. Grünbacher, and H. Mössenböck. Where has all my memory gone?: Determining memory characteristics of product variants using virtual-machine-level monitoring. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, pages 13:1–13:8, 2013.

[28] P. Lengauer, V. Bitto, S. Fitzek, M. Weninger, and H. Mössenböck. Efficient memory traces with full pointer information. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16, pages 4:1–4:11, New York, NY, USA, 2016. ACM.

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

[30] P. Lengauer, V. Bitto, and H. Mössenböck. Efficient and viable handling of large object traces. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 249–260, 2016.

[31] P. Lengauer, V. Bitto, and H. Mössenböck. A comprehensive java benchmark study on memory and garbage collection behavior of dacapo, dacapo scala, and specjvm2008. In *Proceedings 8th ACM/SPEC International Conference on Performance Engineering (ICPE 2017)*, L'Aquila, Italy, 2017.

[32] P. Lengauer and H. Mössenböck. The taming of the shrew: Increasing performance by automatic parameter tuning for java garbage collectors. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 111–122, 2014.

[33] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983.

[34] J. Marathe, F. Mueller, T. Mohan, S. A. Mckee, B. R. De Supinski, and A. Yoo. Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Trans. Program. Lang. Syst.*, 29(2), Apr. 2007.

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

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

[37] 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 International Conference on Virtual Execution Environments*, VEE '10, pages 109–120, 2010.

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

[39] W. D. Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in java. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 116–134, London, UK, UK, 1999. Springer-Verlag.

[40] M. Peiris and J. H. Hill. Automatically detecting "excessive dynamic memory allocations" software performance anti-pattern. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 237–248, New York, NY, USA, 2016. ACM.

[41] T. Printezis and R. Jones. Gcspy: An adaptable heap visualisation framework. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 343–358, New York, NY, USA, 2002. ACM.

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

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

[44] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 657–676, New York, NY, USA, 2011. ACM.

[45] G. Tene, B. Iyengar, and M. Wolf. C4: The continuously concurrent compacting collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 79–88, New York, NY, USA, 2011. ACM.

[46] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 157–167, New York, NY, USA, 1984. ACM.

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

[48] M. Weninger, P. Lengauer, and H. Mössenböck. User-centered offline analysis of memory monitoring data. In *Proceedings 8th ACM/SPEC International Conference on Performance Engineering (ICPE 2017)*, L'Aquila, Italy, 2017.

[49] P. R. Wilson and T. G. Moher. A &ldquo;card-marking&rdquo; scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *SIGPLAN Not.*, 24(5):87–92, May 1989.

[50] G. Xu, M. D. Bond, F. Qin, and A. Rountev. Leakchaser: Helping programmers narrow down causes of memory leaks. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 270–282, New York, NY, USA, 2011. ACM.

[51] G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 151–160, New York, NY, USA, 2008. ACM.

Acknowledgments

"I don't know half of you half as well as I should like; and I like less than half of you half as well as you deserve." - Bilbo Baggins (J.R.R. Tolkien)

I want to thank my supervisor *Hanspeter Mössenböck* for his support and supervision. This thesis was supported by the Christian Doppler Forschungsgesellschaft, and by Dynatrace Austria GmbH.

I want to thank *Verena Bitto* for supporting me in developing the AntTracks virtual machine as well as the first version of the AntTracks tool. We had countless fruitful and critical discussions that continuously questioned and improved my work. Moreover, our mutual co-authorship of our respective research areas has lead to a successful publication streak. I want to also thank *Markus Weninger* for his recent contributions to the AntTracks tool. Also, together with *Stefan Fitzek*, we had a lot of memorable moments in the office. Additionally, *Günther Blaschek* always made some time available when I needed an outside perspective, be it about personal or professional matters.

In *Tanja Reder*, I've found a friend for over 10 years now. She has continuously supported me with advice whenever I needed it. Although being far away and conversations being rare these days, *Franziska Öllerer* always found time for me when I needed it most. She helped me through difficult times and gave me courage to move on. *Miriam Derenbach* is a relatively recent addition to my friends but quickly made it to the front of the line nevertheless. In the last year, she was a friend for every situation. On the one hand, she always lightened my mood and took my mind off after a hard day of work. Be it in summoners rift, the howling abyss, or the murder bridge, she always battled at my side while making jokes based primarily on the opponent's cyclic family tree. On the other hand, she always made me feel appreciated on rainy days.

Finally, I want to thank my long-term partner *Monika Göweil* for her love and support in my studies.

Curriculum Vitae

Personal Information	
Name	Philipp Lengauer
Date of Birth	21.03.1988
Nationality	Austria
E-Mail	p.lengauer@gmail.com

School	
1998 - 2006	Lycée Danube, Linz

Studies	
2013 February - today	Doctoral Studies (Engineering Sciences) at the Johannes Kepler University
2010 October - 2012 June	Master Studies (Software Engineering) at the Johannes Kepler University, Graduation with Distinction, Thesis: *A Trace-based Debugger for Dynamically Composed Applications*
2006 October - 2010 July	Bachelor Studies (Computer Science) at the Johannes Kepler University, Thesis: *Erweiterung des NetBeans Plugins für Coco/R um syntaktische und semantische Codevervollständigung in Attributed Grammar Dateien*

Publications	
2017 April	*A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008* in the Proceedings of the 8th International Conference on Performance Engineering (ICPE'17), L'Aquila, Italy
2017 April	*User-centered Offline Analysis of Memory Monitoring Data* in the Proceedings of the 8th International Conference on Performance Engineering (ICPE'17), L'Aquila, Italy
2017 April	*DuckTracks: Path-based Object Allocation Tracking* in the Proceedings of the 8th International Conference on Performance Engineering (ICPE'17), L'Aquila, Italy

2016 August	*Efficient Memory Traces with Full Pointer Information* in the Proceedings of the 13th International Conference on Principles and Practice of Programming on the Java Platform: Virtual machines, Languages, and Tools (PPPJ'16), Lugano, Switzerland
2016 March	*Efficient and Viable Handling of Large Object Traces* in the Proceedings of the 7th International Conference on Performance Engineering (ICPE'16), Delft, Netherlands
2016 March	*Building Custom, Efficient and Accurate Memory Monitoring Tools for Java Applications* in the Proceedings of the 7th International Conference on Performance Engineering (ICPE'16), Delft, Netherlands
2015 October	*Time Matters: Minimizing Garbage Collection Overhead with Minimal Effort* in the Proceedings of the 6th Symposium on Software Performance (SSP'15), Munich, Germany
2015 June	*Efficient Rebuilding of Large Java Heaps from Event Traces* in the Proceedings of the 12th International Conference on Principles and Practice of Programming on the Java Platform: Virtual machines, Languages, and Tools (PPPJ'15), Melbourne, FL, USA
2015 February	*Accurate and Efficient Object Tracing for Java Applications* in the Proceedings of the 6th International Conference on Performance Engineering (ICPE'15), Austin, TX, USA
2014 March	*The Taming of the Shrew: Increasing Performance by Automatic Parameter Tuning for Java Garbage Collectors* in the Proceedings of the 5th International Conference on Performance Engineering (ICPE'14), Dublin, Ireland
2014 January	*Where Has All My Memory Gone? Determining Memory Characteristics of Product Variants using Virtual-Machine-Level Monitoring* in the Proceedings of the 8th International Workshop on Variability Modeling of Software-intensive Systems (VaMoS'14), Nice, France
2013 October	*VM-Level Memory Monitoring for Resolving Performance Problems* in the Proceedings of the 4th International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH'13), Indianapolis, IN, USA

Work Experience

2013 February - today	*University Assistant* at the Institute for System Software, Johannes Kepler University, Lectures include Software Development in Java, Compiler Construction, Java Performance Monitoring and Benchmarking, Supervision of Bachelor's Theses and Master's Theses
	Research Associate at the Christian Doppler Laboratory for Monitoring and Evolution of Very-Large-Scale Software Systems
2012 August - 2013 February	*Software Engineer* at Compuware Austria GmbH
2012 February - 2012 August	*Software Engineer* at Wolfinger Software

2010 October - 2012 August	*Research Assistant* at the Christian Doppler Laboratory for Automated Software Engineering
2008 October - 2011 February	*Tutor* at the Johannes Kepler University
2008 August	*Intern* at Aktivsoftware GmbH

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references. This printed thesis is identical with the electronic version submitted.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Dipl.-Ing. Philipp Lengauer (Linz, February 17, 2017)