Accurate and Efficient Object Tracing for Java Applications

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

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

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

ABSTRACT

Object allocations and garbage collection can have a considerable impact on the performance of Java applications. Without monitoring tools, such performance problems are hard to track down, and if such tools are applied, they often cause a significant overhead and tend to distort the behavior of the monitored application. In this paper we present a new light-weight memory monitoring approach in which we trace allocations, deallocations and movements of objects using VM-specific knowledge. We strive for utmost compactness of the trace by using a binary format with optimized encodings for different cases of memory events and by omitting all information that can be reconstructed offline when the trace is processed. Our approach allows us to reconstruct the heap for any point in time and to do offline analyses both on the heap and on the trace. We evaluated our tracing technique with more than 30 benchmarks from the DaCapo 2009, the DaCapo Scala, the SPECjvm 2008, and the SPECjbb 2005 benchmark suites. The average run-time overhead is 4.68%, which seems to be fast enough for keeping tracing switched on even in production mode.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory Management (Garbage Collection)*

General Terms

Performance, Measurement

Keywords

Tracing; Allocations; Garbage Collection; Java

1. INTRODUCTION

Automatic memory management, i.e., garbage collection, has gained wide-spread use because it relieves programmers from the error-prone task of freeing unused memory manually. Moreover, a compacting garbage collector (GC) produces a consecutive, i.e., unfragmented, heap. This makes object allocations simple because new objects are simply appended to the used portion of the heap without an expensive search for a fitting memory block.

However, in today's applications with millions of objects, allocating and collecting objects can easily become a performance bottleneck. Since the details of GC algorithms are hard to understand, developers find it difficult to predict the interaction between the application and the garbage collector. Therefore, diagnosing and fixing memory-related performance problems is a tedious and often futile task.

Some virtual machines (VMs), such as the Java Hotspot™ VM, support the logging of GC statistics, e.g., the collection time or the memory usage before and after garbage collection for individual spaces. While this may help in detecting memory anomalies, it does not help in locating and resolving them.

A common remedy for these issues is to analyze entire heap dumps, which tell developers what objects currently exist and thus give some clue on the reasons for a GC performance degradation. Although a dump contains structural information about the layout of the heap, it lacks information about the origin of the objects, i.e., their allocation site, their allocation time and the thread that allocated them. Furthermore, deallocations can only be detected by comparing two subsequent dumps and finding an object in one but not in the other. Identifying two objects from different dumps as the same is difficult because objects can be moved in the heap and VMs usually do not maintain unique object identifiers. For example, one object could have been reclaimed by the GC while another object of the same type could have been allocated at the same position. These two objects would be indistinguishable in a heap dump. Thus, in order to identify objects uniquely and to capture allocation-specific information we have to trace the actual object allocations.

Most Java VMs support dynamic bytecode instrumentation, enabling an external agent to inject code at each allocation site. However, instrumentation introduces a significant performance overhead because it impedes compiler optimizations, such as escape analysis and inlining. Furthermore, some information about objects cannot be obtained through mere bytecode instrumentation, e.g., the address and the size of an object or the reclamation of an object by the garbage collector. In order to obtain such information, one has to modify or extend the VM.

ICPE'15, January 31 – February 04, 2015, Austin, TX, USA.

http://dx.doi.org/10.1145/2668930.2688037.

Ricci et al. [11] describe an approach called *Elephant Tracks* in which they use instrumentation to trace memory events such as object allocations and deallocations. Since they aim for portability between different VMs and garbage collectors, efficiency is not their primary goal. In fact, the overhead of their tracing technique is so big that it changes the behavior of the instrumented program and the GC. One interesting part of their approach is that they compute estimated death times of objects, i.e., the points in time when objects are no longer reachable. Although this is interesting in general, it is not useful for performance monitoring, because even unreachable objects occupy heap space as long as they are not collected and thus influence the GC behavior.

Our approach integrates tracing into the Java Hotspot™ VM so that it has access to all information about objects, their allocation, and how they are treated by the garbage collector. We developed novel techniques for minimizing the tracing overhead. This includes a compact binary trace format in which frequent events are encoded more compactly than less frequent ones, in which certain event data is precomputed at compile time, and in which all information is omitted that can be reconstructed offline when the trace is processed. Based on our traces, we can rebuild the heap for arbitrary points in time, we can detect reoccurring patterns and anomalies in the event stream, and we can reproduce the GC behavior in order to diagnose performance problems.

Our scientific contributions are (1) a compact binary trace format that is largely precomputable at compile time, (2) an efficient tracing mechanism that is built into the Java Hotspot™ VM, and (3) algorithms for reconstructing omitted trace information offline (e.g., object addresses and deallocations). Moreover, we provide (4) an algorithm that is able to rebuild the heap layout based on our traces.

We conducted our research in cooperation with Compuware Austria GmbH. Compuware develops leading-edge performance monitoring tools for multi-tier Java and .NET applications. In their own applications as well as in applications of their customers, high GC times are a problem that currently cannot be resolved with Compuware's tools.

This paper is structured as follows: Section 2 provides an overview of memory management in the Java Hotspot™ VM; Section 3 describes our approach, i.e., the tracing mechanism and the event formats we use; Section 4 presents a detailed evaluation of our approach, including a validation of its soundness as well as a performance evaluation that compares our overhead with those of other tools; Section 5 discusses related work and the state of the art. Section 6 shows future work and potential further usage scenarios and Section 7 concludes the paper.

2. MEMORY MANAGEMENT IN THE HOTSPOT™ VM

The default collector of the Java Hotspot™ VM is the so-called Parallel GC. It is a stop-the-world collector, meaning that all application threads are halted when collecting and resumed afterwards. Using the Parallel GC, the heap is split into two regions: the young generation, which is divided into the eden space and two survivor spaces, and the old generation. Splitting the heap into two generations enables the GC to use a run-time-efficient collection strategy for objects that are likely to die (i.e., young objects) and a more memory-efficient strategy for objects that are unlikely to die (i.e., old objects). Thus, the run-time-efficient Scavenge algorithm (minor GC) is used for the young generation, whereas the Mark and Compact algorithm (major GC) is used if all spaces need collecting. Both algorithms produce an contiguous, i.e., unfragmented, heap and are highly parallelized.

2.1 Object Allocation

New objects are usually appended at the end of the eden space, which is possible because the eden space is never fragmented. To avoid multiple threads racing for the eden end, every thread uses a separate thread-local allocation buffer (TLAB) that resides within the eden space, and in which objects can be allocated by this thread without the need for synchronization. If a new object does not fit into the remainder of a TLAB, the TLAB is retired and a new one is allocated into the eden space. To avoid fragmentation, any remaining space in the old TLAB is filled with an `int[]` when it is retired. This filler object will be collected automatically at the next garbage collection because it is not referenced. Only under rare circumstances, e.g., when a new TLAB cannot be allocated, the object is put into the eden space without a TLAB.

When an object allocation fails, the VM halts all application threads and reclaims unused objects in order to make space for the new object. There are different levels of allocation failures with distinct actions, depending on how often the same allocation already failed.

- Level 1 allocation failures occur if the TLAB is full, and there is neither enough space for another TLAB, nor enough space for the object itself in the eden space. The VM triggers a minor GC (possibly followed by a major GC) and retries to allocate into the eden space.
- Level 2 allocation failures occur if there is still not enough space and a major GC has not been triggered at Level 1. The VM triggers a major GC and retries to allocate into the eden space.
- Level 3 allocation failures occur if there is still not enough space after the major GC or if the object is simply too big for the young generation. Instead of triggering another GC, the VM tries to allocate directly into the old generation.
- Level 4 allocation failures occur only if there is not enough space in the old generation. This means that the VM is running out of memory. Therefore, a more conservative major GC is triggered, clearing all soft references. The allocation is then retried in the eden space.
- Level 5 allocation failures trigger a final attempt to allocate into the old generation. If this fails, an `OutOfMemoryError` is thrown.

The memory manager provides allocation routines for different kinds of objects, i.e., for instances of any type and for arrays of any size. They allocate objects by first trying to allocate into a TLAB, and, if this fails, act according to the allocation failure level. There are four components which may allocate objects: the interpreter, compiled code, the garbage collector, and the VM itself. The interpreter

has a special fast path for allocating instances into a TLAB. The code of this fast path is generated during VM startup and relies on a non-full TLAB. If the TLAB is full or if an array must be allocated, the interpreter has to fall back to the slow path in which it calls one of the generic allocation routines provided by the memory manager. Similarly, compiled code pieces have fast paths for TLAB allocations (both for instances and for arrays). If the TLAB is full, the code falls back to the generic allocation routines (slow path). The garbage collector allocates filler objects (e.g., for retiring TLABs) by directly manipulating the heap. Finally, the VM uses the generic allocation routines for the allocation of so-called universe objects (i.e., objects that are created during VM startup) and for other special cases, e.g., for reflection and cloning.

2.2 Garbage Collection

During a minor GC, all live objects in eden and in one of the survivor spaces are copied into the other (empty) survivor space by following all root pointers into the young generation recursively. When an object has reached a certain age (measured in survived minor GCs), it is "tenured", i.e., copied into the old generation. An object might also be copied directly into the old generation if the survivor space is full or the object is too big. At the end of a minor GC, eden and the source survivor space are considered empty again. Any object that has not been copied is garbage. It is not deallocated explicitly but is implicitly freed when the containing spaces are declared empty. Therefore, the run time of a minor GC depends only on the number of live objects because they have to be evacuated into either one of the survivor spaces or the old generation.

To avoid locking, promotion-local allocation buffers (PLABs) are used for parallel copying into the survivor space and into the old generation. Like TLABs, PLABs are allocated and retired on demand, which might include filling the remainder with a filler object.

In order to avoid that an object is copied twice if two GC threads arrive at that object at the same time, a forward pointer is atomically installed into every copied object. The forward pointer points to the new location of the copied object and is used for detecting whether an object has already been copied as well as for fixing the references to it. If an object has been copied twice by accident, the slower thread then overwrites its copy with an `int[]` and uses the other copy obtained via the forward pointer.

A major GC consists of three phases: mark, summarize, and compact. In the mark phase, the heap is traversed starting with the root pointers and all reachable objects are marked as being alive. The summarize phase computes the new locations of live objects after compaction, whereas the compact phase moves all live objects accordingly and adjusts all pointers. For parallel marking, subsets of the root pointers are handled by different threads; for parallel compaction the heap is split into several small regions. As all objects are moved towards the beginning of the heap during compaction, very old objects, including the universe, tend to accumulate at the beginning of the old generation. As these objects are very unlikely to die in the future, the GC defines a dense prefix at the beginning of the old generation. In this special region, compaction is not performed every time, but dead objects are only overwritten with filler objects.

3. APPROACH

This section presents our tracing approach, i.e., the generation of raw traces as well as the offline post-processing steps required to reconstruct omitted information. Performance results and the impact of individual optimizations will be discussed in Section 4.2.

We modified the Java Hotspot™ VM to capture allocation and GC events. More specifically, we modified the interpreter, the just-in-time compiler, and the garbage collector so that every object allocation and every object move is treated as an event that is written to a dedicated trace file for subsequent analysis.

To reproduce an allocation from the trace, we need to know the object's address, its type, and its size (including the array length if the object is an array), the allocation site, the allocating thread, the allocator (i.e., interpreter, compiled code, GC, or VM), and the allocation mode (i.e., fast path or slow path). To capture object movements by the GC, we need to know the object's old and new address for every move. Furthermore, we need to know if an object was kept alive by the GC without moving it. Every object that is neither moved nor kept alive and is located in a collected space, is reclaimed. Therefore, we do not need dedicated deallocation events but can derive them from other events in the trace. The above information must either be explicitly stored in the trace, or must be reproducible from other events in the trace in combination with the context (i.e., the adjacent events).

Although we extended the Hotspot™ VM, our approach remains applicable to others, because modern VMs and GCs use very similar techniques and algorithms.

3.1 Symbols

Symbolic information such as the allocation site (i.e., class name and method name) and the type name of the allocated object would take up a lot of space if included in every event. Therefore, we map this symbol information to numeric IDs, which are used in the actual events instead. The information corresponding to the IDs is written to a separate symbols file. For every allocation site, the symbols file contains the name of the allocating class (represented by a 2-byte integer), the name of the allocating method, the bytecode index (BCI) of the allocation site, and the ID of the allocated type (represented as a 2-byte integer). For every type, the symbols file contains the name of the type and the size of objects belonging to this class. The size is the actual size for instances, and the element size as well as the header size for arrays. Encoding this information by a single ID keeps the trace file compact. The symbols file is generated on demand during tracing but stabilizes quickly.

3.2 Event Types and Formats

This section describes the format of allocation events and GC events in the trace file.

3.2.1 Allocation Events

Figure 1 shows the format of the most generic (and verbose) allocation event whereas every block is 4 bytes in size. The event type field (1 byte) indicates that the event describes an allocation and contains also the allocator and the allocation mode, e.g., compiled code fast (TLAB) allocation or interpreter slow (eden space) allocation. The allocation site (class, method and BCI) is mapped by an ID in the al-

location site field (2 bytes). Although, we support at most 65536 allocation sites, we have never reached this limit because we assign IDs lazily, i.e., only if we observe at least one allocation at that site. The object's address can be determined by the space into which the object was allocated and by an offset that is stored in the relative address field (4 bytes). Since addresses are word-aligned, the space can be encoded in the last 2 bits of the relative address field. If the object is an array, its length is stored in the optional array length / size field (4 bytes). Some instance objects do not have a fixed size, so their size can be encoded in that field instead of the array length. Finally, the optional class field (4 bytes) contains a reference to the type, if it cannot be inferred from the allocation site.

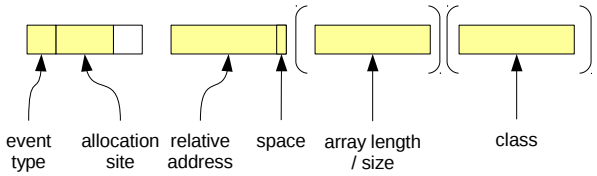


Figure 1: Event format for allocation events

Some of the fields are necessary because their contents cannot always be inferred automatically. For example, the type of the allocated object can usually be inferred from the allocation site, because most sites, can allocate just a single type of objects. However, if a single valid type cannot be determined statically, we use the optional class field to store the type. This can happen if there is no valid allocation site or if different types can be allocated at a specific allocation site. The former occurs when the universe is built by the VM, whereas the latter occurs in multidimensional array allocations. For example, `new int[21][42]` is actually one `new int[21][]` and 21 allocations of kind `new int[42]`. Based on the type information and optionally the array length, we can calculate the size of an object. If the size cannot be derived from the type of the allocated object, the explicit size field is used. This is necessary because the `Class` class contains all static fields of the class it describes. Therefore, the size of a `Class` object, although not being an array, is not fixed but varies depending on the amount of static members. Finally, the allocating thread can be determined from the context, which will be shown in Section 3.3.

This generic allocation event format has potential for optimizations: (1) 16 bytes per allocation event is too big when tracing allocation-intensive applications. (2) Calculating the fields again and again is redundant, because their values usually do not change for a specific allocation site; Figure 2 shows our optimized allocation event format that we use whenever possible, falling back to the generic format only if necessary.

We observed (cf. Section 4.2) that most objects are instances and small arrays, i.e., arrays with a length smaller than 255, which are allocated into the TLAB. Correspondingly, the above mentioned corner cases (i.e., multidimensional arrays and `Class` objects) account only for a small fraction of all objects.

In the optimized format, the first word remains the same, except that we use the previously unused byte for storing the length of small arrays. Moreover, considering only TLAB

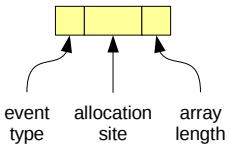


Figure 2: Optimized allocation event format by exploiting continuous TLAB allocations

allocations, there is no need for an explicit object address, since the address can be computed from the TLAB address and previous TLAB allocation events, i.e., the sum of the individual object sizes already allocated into this TLAB (the algorithm to reconstruct the object address is discussed in more detail in Section 4.1). As this event format is not used for large arrays, `Class` objects, multidimensional arrays, and allocations with an unknown allocation site, we can encode everything in a single 4-byte word, without losing any information.

Given this optimized event format and an arbitrary allocation site, the first 3 bytes of the event word can be statically precomputed. This characteristic is used to minimize the run-time performance overhead of allocation events in compiled code, as described in Section 3.4.

3.2.2 GC Events

Figure 3 shows an event denoting that a single object has been moved by the GC, as indicated by the event type field. The from-address and space represent the current object position, whereas the to-address and space are its new location. Although every object movement can be described with such an event, there are several optimized event formats that we use whenever possible in order to keep the trace small and to minimize IO.

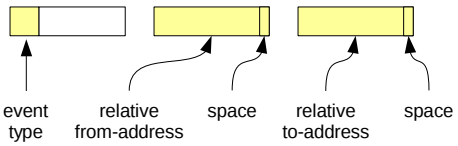


Figure 3: Event format for GC events

During minor GCs, PLABs are placed into the survivor space and the old space to avoid synchronization when moving live objects (cf. Section 2.2). As PLABs are similar to TLABs, we can employ the same technique to calculate the to-address based on the PLAB address and previous object moves into this PLAB. The first optimized event format, shown in Figure 4, takes advantage of this fact. We omit the to-address and only need to know into which space (i.e., survivor space or old space) objects are moved to correctly reconstruct their addresses. If the relative from-address is small enough to fit into the unused 22 bits of the first word, we further compress the event into another 4-byte format. Only if the object is moved without using a PLAB (e.g., because no more PLABs can be allocated or because the object is too big for a single PLAB) we fall back to the above described format.

During major GCs, live objects are moved towards the beginning of the heap (cf., Section 2.2). As the Mark and Compact algorithm does not use PLABs, we cannot omit the

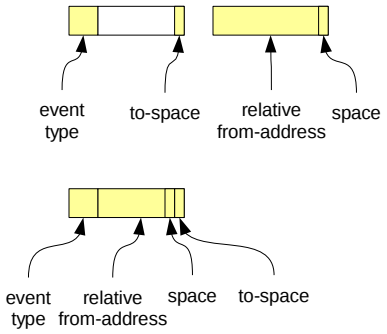


Figure 4: Optimized event format for minor GC events by exploiting continuous PLABs and small addresses

to-address in the same manner as described above. However, we can exploit the fact that objects survive in clusters, because referenced objects are often located next to each other due to their sequential allocation and compaction. Thus, these clusters are moved by the same offset and can therefore be handled more efficiently by the event format shown in Figure 5. This format differs from Figure 3 only in making use of the remaining 24 bits to store the number of adjacent objects moved by the same offset. In combination with the object sizes from the allocation events, we can then reproduce every single move correctly.

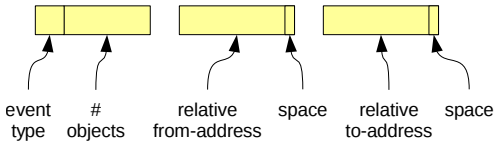


Figure 5: Optimized event format for major GC events by aggregating similar events

3.2.3 *Other Events*

To support the omission of information in the above mentioned event formats, we capture a number of additional events, such as TLAB and PLAB allocations as well as the start and end of GC runs. TLAB and PLAB allocation events carry the absolute address and the size of the respective thread-local buffer. This information enables computing the relative address of each object allocation or GC move if the address has been omitted. GC start and GC end events carry the absolute address and the size of each space as well as the type, i.e., minor or major. By means of the absolute addresses for each space, we can convert relative object addresses to the correct absolute address. As these events occur infrequently compared to allocation or GC move events, there was no need for compressing the contents or other further optimizations.

3.3 Writing Events to the Trace File

All events described in Section 3.2 have to be written to the trace file. However, writing them one by one would result in an unnecessarily large number of IO operations, stalling application threads. Therefore, we use buffers to collect events before they are written.

Event Buffers.

One simple approach is to store all events in a global event buffer, as shown in Figure 6. However, as today's applications are heavily multi-threaded, the application threads would have to race for the buffer top. This would result in stalling application threads at every event.

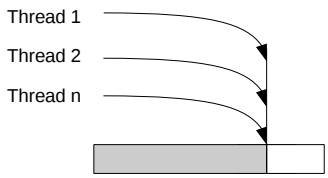


Figure 6: Global buffer for all threads

To avoid multiple threads racing for a single buffer, we assigned a buffer to every thread, as shown in Figure 7. Therefore, every thread can fire events, i.e., store them into its own buffer, without having to synchronize with the other threads. When the buffer is full, it must be written to the trace file, stalling the application thread again.

Figure 7: Thread-local buffers

Figure 8 shows how two buffers per thread can be used to avoid stalling application threads for most of the time. A thread always writes to its front buffer until this buffer gets full. Then it swaps its front buffer with its back buffer and continues to write on the new front buffer. The full back buffer is appended to a flush queue. A dedicated worker thread consumes buffers from this queue and writes them to the trace file. The application thread stalls only if the front buffer gets full while the back buffer has not yet been processed by the worker thread. However, our observations showed that only a small number of threads allocate frequently. Therefore, the majority of threads hog buffer space needlessly.

Our final approach, shown in Figure 9, solves this problem by assigning one buffer to each thread on demand, using a flush queue and a worker thread for asynchronous output, as well as a pool of empty buffers. When a thread's buffer gets full, it is appended to the flush queue and a new buffer is requested from the pool of empty buffers. Thus, no application thread consumes buffer space needlessly and threads never have to stall due to full buffers. Moreover, they only have to synchronize on the flush queue and the buffer pool. However, this occurs rarely i.e., only when their own buffer gets full.

Using thread-local event buffers allows us to associate all events in a buffer with a particular thread, avoiding the need for a dedicated thread ID in every event. When a buffer is written to the trace file, it is preceded by a thread ID, thus allowing us to recover this association when the trace file is processed.

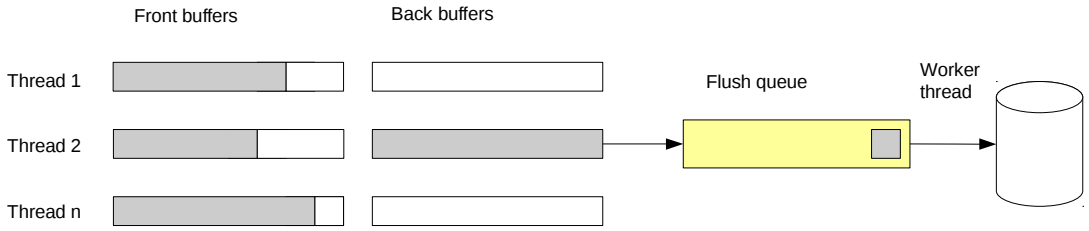


Figure 8: Thread-local front and back buffers

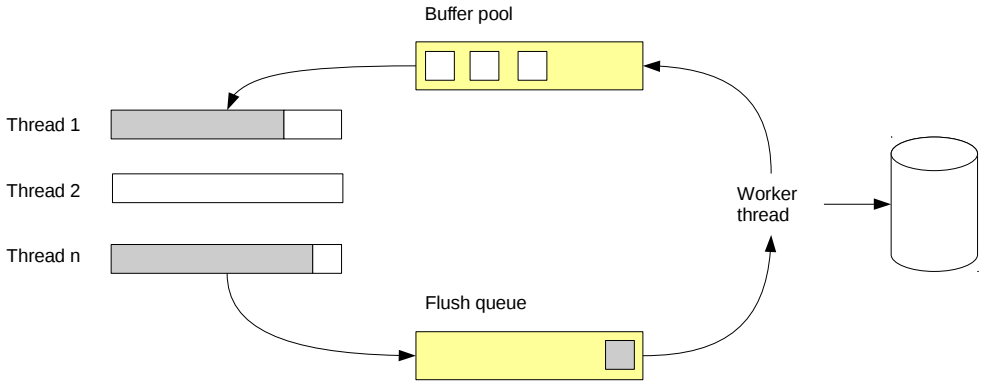


Figure 9: Thread-local buffers with a buffer pool

Buffer Sizes.

We observed that, if threads execute similar tasks, they also have similar allocation frequencies. This can lead to buffer overflows at the same time, and consequently, to performance degradations, because (1) the locks of the flush queue and the buffer pool are contended and (2) the flush queue might overflow itself because the worker thread may not be able to process the buffers fast enough. This problem was overcome by randomizing the individual buffer sizes while leaving the overall size of all buffers the same. As the overall size is the same, buffers are flushed as often as before, but because of the different individual sizes, they are flushed at different points in time.

Event Times.

Since events are collected in thread-local event buffers, their chronological order cannot be recorded if they occur in different threads. We accept that as the price for not having to synchronize the recording of every event. However, we flush all event buffers (even those that are not yet full) before and after every GC run so that the chronological order of events is at least maintained across GC runs. In other words: all events that happended before a GC run occur before the corresponding GC events in the trace file, and all events that happened after a GC run occur after the corresponding GC events. Furthermore, all events that happened in the same thread occur in the correct chronological order.

3.4 Event Capturing

We designed a new VM component (the *Event Runtime*), that is responsible for (1) building an event, (2) writing it to a buffer as well as for (3) appending a full buffer to the flush queue and (4) fetching a new buffer. This section describes how it is used by other VM components such as the memory manager, the interpreter, compiled code, and the GC.

3.4.1 Allocations in the Memory Manager

Objects allocated by the memory manager are the universe objects (at VM startup) and filler objects (e.g., when retiring TLABs or PLABs). These objects are allocated in native C methods, which we instrumented to call the proper methods of the Event Runtime. Since these objects are not allocated by Java code, no allocation site can be defined. Hence, we use a predefined `VM_INTERNAL` allocation site and append the type ID in the optional field.

3.4.2 Allocations in the Interpreter

As discussed in Section 2.1, the interpreter allocates instance objects in the fast path. When it processes the allocating bytecode it calls into the Event Runtime to record the allocation event. Since interpreted code is never "hot" (otherwise it would have been compiled) there was no need for optimizing this call.

When the TLAB is full or when the allocated object is an array, the interpreter goes into the slow path and again makes a call into the Event Runtime. In this case, however, the Runtime additionally extracts the method and current BCI from the call stack and records them in the event.

3.4.3 Allocations in Compiled Code

Since hot code is compiled, most objects will be allocated by compiled code (cf. Section 4.2). These allocations are fast because there is a fast path for both instances and arrays of arbitrary lengths. Only if the TLAB is full, a VM routine is called that allocates objects using the generic allocation routines. This is why we designed the event format in a way that we can capture events in the compiled code as fast as possible.

Since the just-in-time compiler knows which allocation site it is currently compiling (i.e., which method and BCI), the value of the optimized allocation event can be inlined into the generated code as a precomputed constant. If the allocation site is an array allocation with less than 255 elements, we just have to add the array length to the precomputed event word. Therefore, the overhead of most allocation events in compiled code is as small as checking whether the event buffer is full (i.e., comparing two pointers), firing the event (i.e., assigning a 4 byte constant to a memory location) and incrementing the top of the event buffer (i.e., incrementing a pointer).

Figure 10 shows C-like pseudo code representing the generated code at every allocation site of an instance object (excluding `Class` allocation sites), whereas Figure 11 shows the generated code for array allocation sites (excluding allocation sites for multiddimensional arrays).

Please note that this code is generated during the translation of an intermediate program representation to machine code, i.e., after optimizations have been applied. Therefore, this code does not impede any optimizations, such as escape analysis or control/data flow analysis. If the buffer is full (or if the array is too big), the Event Runtime is called which is able to handle these cases appropriately. This might include fetching a new buffer or, in the worst case, falling back to the generic allocation event format.

```
Object* o = ...; //allocation
if(buf->top < buf->end) {
  *(buf->top++) = 0xABCDEF00;
} else {
  allocation_event_slow_path(o);
}
```

Figure 10: Generated code for firing an optimized allocation event for instances

```
Object** a = ...; //allocation
if(buf->top < buf->end && a->len < 255) {
  *(buf->top++) = 0xABCDEF00 | a->len;
} else {
  allocation_event_slow_path(a);
}
```

Figure 11: Generated code for firing an optimized allocation event for arrays

3.4.4 Allocations in the GC

To avoid fragmentation, the GC allocates objects to fill holes in the heap. This is done by directly manipulating the memory location where the object header will be. We instrumented all sites where the GC creates such filler objects so that they fire proper allocation events. Although filler objects will never survive the next collection (because they are not referenced), they must be tracked in order to calculate the absolute addresses of neighboring objects correctly and to keep our rebuilt heap unfragmented.

3.4.5 Moves in the GC

Whenever the GC moves an object, an appropriate GC move event is fired. If a PLAB is used, e.g., during a minor GC, one of the fast event formats is used. During a major GC, most GC move events are suppressed and grouped into region events.

4. EVALUATION

In this section we validate our approach by showing how we can reconstruct omitted information from the trace and consequently rebuild the heap layout. Furthermore, we show that our approach outperforms similar approaches significantly in terms of performance.

4.1 Validation

In order to validate our approach, we defined two categories of tests: Correctness tests are (self-written) applications whose allocation behavior is predictable. For these applications we defined tests in which we check if every single allocation event is recorded in the trace and carries the correct data; Consistency tests check whether the trace is consistent in itself, i.e., whether the heap can be reconstructed. We applied these tests to well-known Java benchmarks described in detail in Section 4.2.

4.1.1 Reconstructing the Heap Layout

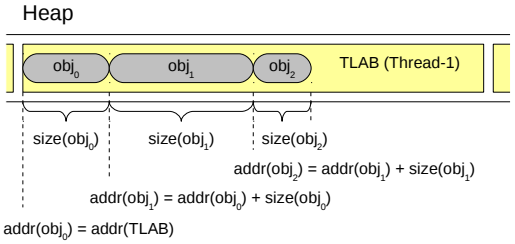
Based on the trace file, we can reconstruct the heap layout offline for any point in the execution of the application. This includes information about which objects were alive at that time, as well as where they were located in the heap. To do this, we first parse the symbols file and start with an empty heap before parsing the actual trace file. Each trace file contains an alternating sequence of two phases: the mutator phase and the GC phase. The GC phase starts at a GC start event and ends at a GC end event, whereas the mutator phase is active at all other times.

Mutator Phase.

The first phase is the mutator phase, which consists of object allocations and TLAB allocations. During this phase, we (logically) create a new object for each allocation and store it in a map using the object address as its key. If an event uses the unoptimized event format, this is trivial because the object address is stored in the event explicitly. If an event uses the optimized format, i.e., if the object was allocated into a TLAB and the address was omitted from the trace, the object address must be reconstructed. Thus, we also maintain a map of currently active TLABs using the thread ID as the key. When the address of such an event is reconstructed, we look up the correct TLAB and, consequently, can determine the range in which the object must lie (somewhere in between the TLAB start and the TLAB end).

As objects are allocated sequentially into TLABs, the address of the first object (obj_0) is equal to the address of its TLAB. The address of all subsequent objects (obj_n) can be calculated by adding the size of the previous objects to its address. Figure 12 shows how to incrementally reconstruct object addresses based on previously reconstructed addresses.

As object allocations into a TLAB might also be described by the generic allocation event format, their address must be checked if it lies within the current active TLAB (there might be other reasons for falling back to this format than a non-TLAB allocation). If it lies within the TLAB, the



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

Figure 12: Algorithm for reconstructing object addresses of TLAB allocations

counters are adjusted as described above, if not, the object is just added to the map with the explicitly stored address.

This algorithm assumes that (1) all allocations are in temporal order within the scope of one thread and that (2) only the owner thread can allocate into a TLAB. Although the former is trivially true because all events are written sequentially to a thread-local buffer and the buffers are flushed in the order they are committed to the flush queue, the latter is false due to filler objects when retiring TLABs in preparation for a GC run.

In this case, all Java threads are suspended and their respective TLABs are retired by a dedicated VM thread. Retiring a TLAB might include allocating a filler object to fill the current TLAB, but, as the VM thread is retiring the TLABs, this allocation is accounted to that thread instead of the owner thread. To make it even worse, due to the lack of temporal ordering among multiple threads, the allocation event of that filler object might be located in the trace before or after the last allocation into the corresponding TLAB.

To deal with filler objects correctly, we detect them while parsing a mutator phase and postpone their processing until the mutator phase is complete. As this filler objects are always of type `int[]` with a `VM_INTERNAL` allocation site allocated by the GC and thus carry an explicit object address, they are easily detectable. Therefore, we process the postponed filler objects when the mutator phase is complete by searching the correct TLAB and assigning the filler to it as described above.

GC Phase.

The second phase is the GC phase, which consists of GC move events, PLAB allocations, and a few object allocations (for filler objects when PLABs are retired). At the end of the mutator phase, the data structure representing the heap can be interpreted as a map of object addresses to object-specific information. When processing a GC move event, the object is copied from the old heap map to a new heap map. In the end, the maps of all collected spaces are cleared and replaced with the new ones (the kind of spaces depend on the kind of GC). Every object which is in the old map but not in the new map has been deallocated.

To reconstruct missing addresses, we use the same technique for PLABs as for TLABs. (cf. Mutator Phase).

When the GC phase has been processed completely (indicated by a GC end event), the heap map is in a valid state and can thus be used for the next mutator phase.

4.1.2 Consistency Tests

The consistency tests are performed while the heap layout is reconstructed from the trace of an application as described above. The tests check certain invariants, e.g., whether the sum of all object sizes in a TLAB matches the TLAB size and whether there are no holes in the heap at each phase change where the heap should be valid. Extensive tests with well-known Java benchmarks (cf. Section 4.2) showed that we can reconstruct the heap for every Java application without restrictions or inconsistencies.

4.2 Performance

In order to measure the overhead of our tracing mechanism we performed measurement on a large number of well-known Java benchmarks such as the DaCapo[1] 2009 benchmark suite [1], the DaCapo Scala[2] benchmark suite [12], the SPECjvm[3] 2008 benchmark suite, and the SPECjbb[4] 2005 benchmark. These suites contain a large variety of programs exhibiting different kinds of allocation and garbage-collection behavior. The DaCapo Scala benchmark suite contributes benchmarks that are not implemented in Java but in Scala, which usually results in a more allocation-intensive behavior. For every benchmark, we chose the largest input available to put more pressure on the memory manager and the garbage collector.

In addition to that, we implemented a benchmark that prints "Hello World" to the standard output. We use this benchmark to measure the overhead we introduce during VM startup and teardown.

Every result we show is the median of 50 runs. For every run, we executed enough warmups in order to JIT-compile all hot methods and to stabilize the caches before measurement.

We ran all measurements on an Intel® Core™ i7-3770 CPU @ 3.40GHz×4 (8 Threads) on 64-bit with 18GB RAM running Ubuntu 13.10 Saucy Salamander with the Kernel Linux 3.11.0-23-generic. All unnecessary services were disabled and the experiments were always executed in text-only mode.

4.2.1 Overhead

Figure 13 shows the run-time overhead of our approach (i.e., lower is better). Every group of bars is one benchmark, whereas the left/red/dark bar is the run time without the tracing mechanism enabled and the right/green/light bar is the run time with the tracing mechanism enabled. Due to large absolute differences in their run times, every benchmark has been normalized with respect to the run time without the tracing mechanism. The crypto, compress, scimark, mpegaudio, factorie, specs and scalatest benchmarks were excluded for space reasons and because they do not show any measurable run-time overhead anyway. The size of a single event buffer has been fixed at 16KB for this experiment.

The results show that the average run time overhead is about 4.68%. Some benchmarks such as tomcat, tradebeans, tradesoap and actors even show a slight speedup. We tracked this reproducible behavior down to the fact that

[1] `http://www.dacapobench.org/`
[2] `http://www.dacapo.scalabench.org/`
[3] `http://www.spec.org/jvm2008/`
[4] `http://www.spec.org/jbb2005/`

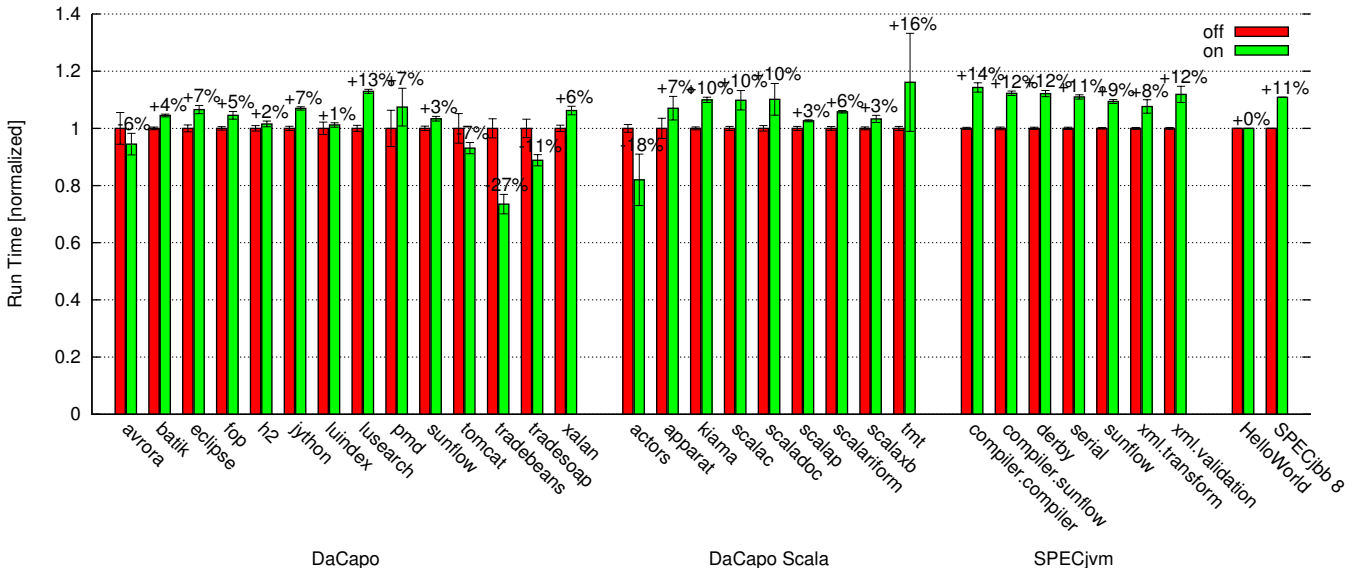


Figure 13: Run time overhead

the slight overhead we introduce when tracing allocations and GC moves causes the compiler to make different optimization decisions, that have an accidental albeit positive influence on the run time.

The overhead observed is significantly lower than similar tracing mechanisms. Figure 14 shows the overhead of *Elephant Tracks* on the DaCapo benchmarks based on [11] (time for tracing allocation and deallocation events without method entry and exit events). Although Elephant Tracks was not built for performance monitoring, it is the tool that is most similar to ours and can be configured to collect almost the same amount of data as we do. Please note the different scale on the y axis and that the overhead on the tradebeans and tradesoap benchmarks is so high that they crash due to internal timeouts with *Elephant Tracks*. The results show that our tracing mechanism outperforms Elephant Tracks significantly.

Although there was no overhead measurable in Java heap size, the non-Java heap rose by 1% due to the event buffers. Furthermore, the code cache (holding all compiled code) grew by 3% due to the additional code generated. These numbers are the same for all benchmarks.

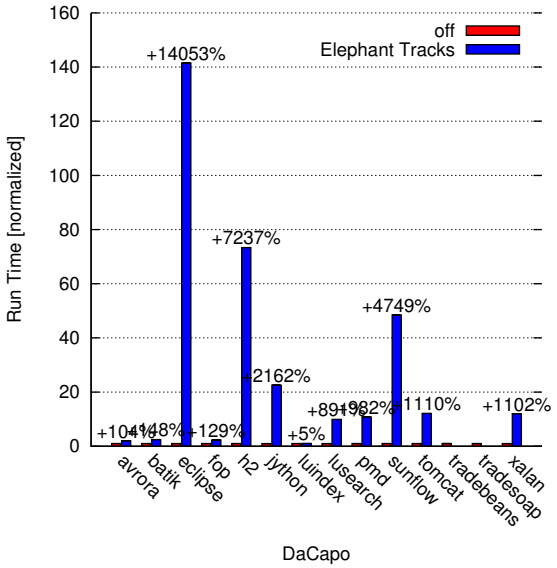


Figure 14: Run time overhead of elephant tracks

4.2.2 *Impact of Optimizations*

In order to determine the gain of individual trace optimizations, we compared the benchmark results with and without these optimizations. Here, we will only discuss the most import optimizations in terms of performance impact.

Optimized Allocation Event Formats.

Figure 15 shows the distribution of the types of allocated objects, i.e., instances, small arrays ($length < 255$), and big arrays, of a reduced albeit representative set of benchmarks. As expected, instances and small arrays comprise the by far largest amount of heap objects. Therefore, handling them by a more compact event format makes sense.

Figure 16 shows the distribution of the allocators, i.e., compiled code, interpreter, and the VM. Again, as expected, after the application is warmed up, almost all allocations were performed by compiled code. The only exception is the HelloWorld benchmark, where all allocations are done during startup by the VM itself or by the interpreter. Therefore, our decision to optimize the tracing of allocations in compiled code pays off.

To summarize, it is obvious that optimizing compiled code for instances and small arrays is of utmost importance. For this reason, we defined optimized and precompilable event formats for these cases as described in Section 3.2.1. Disabling these optimizations and falling back to the unoptimized allocation event format increases the trace size by a factor of 2 to 3 (depending on the ratio between instances and arrays), roughly doubling the overall run-time overhead.

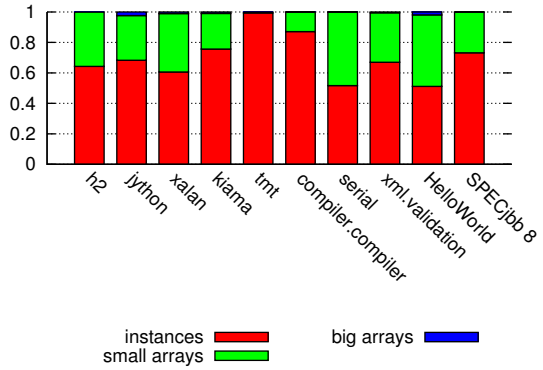


Figure 15: Object type distribution of instances, small arrays, and big arrays (from bottom to top)

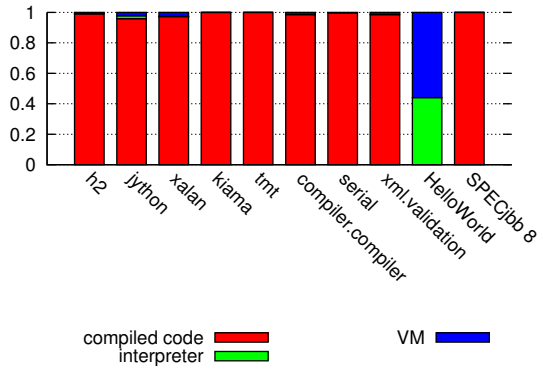


Figure 16: Allocator distribution of compiled code, interpreter, and VM (from bottom to top)

Optimized GC Event Formats.

Figure 17 shows the distribution of the GC event formats used in the benchmarks. We can observe that most events use the optimized event format without the source address; only a few events use the generic (unoptimized) event format (mostly originating from major GCs). The region move event format as well as the generic move event format are used by less than 0.1% of the events in the trace. However, observations have shown that on average at least 300 objects are aggregated into a single region event, meaning that 300 events (900 words) have been replaced by a single event (3 words as well). Furthermore, a significant amount of optimized events can be replaced by their narrow version, cutting the event size in half and consequently reducing buffer overflows and IO.

Asynchronous IO.

The performance impact of asynchronous IO strongly depends on the chosen buffer size. When using synchronous IO, we observed that the run time overhead is multiplied by at least a factor of 5 without changing the buffer size (16KB by default). When the buffers are of much larger size (several MB at least), synchronous IO performs almost as good as asynchronous IO. This is due to the fact that buffer overflows occur only rarely. However, whenever a garbage collection starts or ends, all buffers must be flushed, which increases

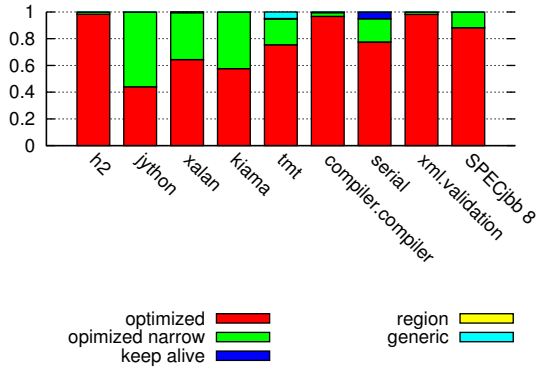


Figure 17: GC event format distribution of optimized, opimized narrow, keep alive, region and generic event formats (from bottom to top)

the pause time of the GC significantly with synchronous IO and large buffers. Furthermore, with large buffer sizes, the non-Java heap size overhead can easily grow to several GBs.

Buffer Management.

Using only a single thread-local buffer produces the same overhead and the same space trade-off as described above (cf. Paragraph Asynchronous IO) because even with asynchronous IO, the thread cannot continue without waiting for the buffer to be flushed. However, using front and back buffers doubles the average overhead compared to our final implementation. Our investigation showed that this is due to the fact that almost every benchmark has allocation-intensive bursts. During these bursts, objects are allocated faster than the buffers can be flushed. In these cases, application threads must be stalled because both, the front buffer is full, and the back buffer has not yet been flushed. As these bursts occur rarely and only affect a limited number of threads at a time, stalling can be avoided by our more flexible buffer management described in Section 3.3.

Randomized Buffer Sizes.

To evaluate the performance impact of randomized buffer sizes, we measured the lock and wait times when accessing the flush queue. The lock time is the time a thread has to wait for the lock, whereas the wait time is the time spent waiting because the flush queue became full. A high lock time indicates that multiple threads wanted to flush their buffers at the same time, whereas a high wait time suggests that the worker thread could not process the flush queue fast enough. Using randomized buffer sizes reduced the overall run time overhead by 2% on average and the lock time by 5% and wait time by 21% on average.

5. RELATED WORK

There is ample work on how to track down memory-related performance degradations in Java and other managed execution environments. Chilimbi et al. [2] define a binary format for recording memory event traces with the intention to use them as simulation workloads for garbage collector performance tests. However, as their trace format is very flexible and allows for variable-sized fields and custom fields, the trace is not encoded as efficiently as it could

be. Harkema et al. [3] create events (not only memory events) by instrumentation and provide an event trace API. Furthermore, they implemented a number of tools on top of this API. Although their events include allocations they lack the granularity of our approach. For example, they do not trace GC events. Hertz et al. [4, 5] generate traces including allocation, deallocation and pointer-update events. They invoke a full GC every n milliseconds to calculate the approximate time of death for an object, whereas the precision depends on n. However, they cannot tell how an object was moved by the GC. Moreover, frequent full collections degrade performance as well as they distort GC behavior. Printezis et al. [10] provide an adaptable framework and API for collecting memory management behavior, including GC behavior. However, they record the behavior for blocks instead of for objects and thus lack object-specific information. Ricci et al. [11] use a combination of bytecode rewriting and JVMTI to build a shadow heap and to generate traces with events such as allocations, deallocations, and pointer updates. However, they produce an overhead of a factor of 3000 for some benchmarks, making this approach unfeasible in production environments. Furthermore, they compute estimated object death times, i.e., the time the last reference to an object is cleared, and not actual deallocations, i.e., the time an object is actually reclaimed by the GC. Although the former might be interesting, it ignores the fact that even unreferenced objects have an impact on the GC behavior. Considering the actual point in time, where they are reclaimed, is therefore more useful for performance monitoring. Finally, Jones et al. [7] offer a detailed study about the correlation of object age, allocation site, program input, and GC performance.

There is also some work on tracing objects in native environments. Although these approaches are only marginally relevant for us, they deserve being mentioned: Janjusic et al. [6] implemented a memory profiling tool that is able to associate each memory access to source-level data structures using binary instrumentation. This enables the programmer to refactor code based on memory access patterns and to simulate cache behavior. Marathe et al. [9] also provide a framework for partial access traces as well as a novel compression algorithm for these traces.

6. FUTURE WORK

This section discusses future work, as well as further potential usage scenarios.

6.1 Compression

As event traces can grow large, we plan to compress them. The event stream can either be compressed on the fly inside the VM (before writing it to the trace file) or by an external phase. Compressing the event stream inside the VM has the advantage that the VM has to write less data which in turn reduces the IO overhead. However, the compression itself will take some time as well, resulting in a trade-off between the quality of the compression (i.e., the resulting size and the run time) and IO time. Additional research and experiments are needed to determine whether an internal compression is beneficial in our case.

6.2 Cyclic Traces

When applications run indefinitely, the trace becomes too large to be kept in a single file or even on a single disk. Therefore, we plan to overwrite the trace cyclically, so that the last n minutes of the execution are always available for analysis. In other words, the trace must be cut off and restarted in regular intervals. However, cutting off the trace will result in loss of data and, therefore, impede subsequent analysis. Information about previously allocated objects will no longer be available. To solve this problem, we plan to collect all relevant information about objects that are alive at cut points and write it to the beginning of the new trace file. The contents of this information as well as the best suitable cut points are undefined as yet and therefore considered to be future work.

6.3 Pointer Updates

In addition to object allocations and GC moves, we consider tracing also pointer updates. This can either be done at every assignment to a pointer field (which would augment the trace with valuable information about the dynamic behavior of the program but is probably too expensive) or only at major GCs where we could record the current values of pointer fields. Tracing the connections between objects on the heap would open new possibilities for advanced offline analyses such as memory leak detection or the discovery of "tenured garbage" (i.e., dead objects in the old generation that keep other objects in the young generation alive).

6.4 GC Configuration Comparisons

In previous work, we described a technique for automatically tuning the GC by finding optimal GC parameter setting in a black box manner (Lengauer et al. [8]). Object traces will help us to understand why a specific GC configuration is superior to others by providing detailed insights into dynamic behavior of the memory manager and the GC.

6.5 Mining Recurring Patterns in Transactions

Large-scale server applications have to handle a continuous stream of similar requests, i.e., transactions. Transactions can be split into different phases, some of which exhibit exactly the same allocation behavior at every execution. Other phases might follow specific patterns, e.g., the number of allocations of a certain type of objects might depends on a certain transaction parameter. Object traces can be extended with information about transaction boundaries and then mined for recurring patterns of object allocations in relation to transaction parameters. This information can help the developer to understand the performance impact of certain parameters on transactions.

7. CONCLUSIONS

In this paper, we presented the design and the implementation of an efficient tracing mechanism for Java objects, creating a trace file which contains all object allocations and GC moves that occur during the execution of a Java application. We also described a novel and compact event format as well as its efficient generation based on the omission of redundant event data, precomputed event information, and a sophisticated buffering scheme.

Extensive evaluation showed that our approach has a very low run time overhead (4.68%) that is significantly smaller than the overhead reported for other techniques. Furthermore, we showed that our approach can track the full life

cycle of every object (i.e., allocations, moves, and deallocations) and allows us to reconstruct the heap layout (i.e., the location of every object) at any point during the execution of an application.

When monitoring applications with large GC pauses or when tuning GC parameters, a trace that reflects the exact behavior of an application can lay the foundation for new tools that help developers in solving memory and GC-related problems.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

[1] 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 *Proc. of the Annual ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages, and Applications*, pages 169–190, 2006.

[2] T. Chilimbi, R. Jones, and B. Zorn. Designing a trace format for heap allocation events. In *Proc. of the 2nd Int'l Symp. on Memory Management*, pages 35–49, 2000.

[3] M. Harkema, D. Quartel, B. M. M. Gijsen, and R. D. van der Mei. Performance monitoring of java applications. In *Proc. of the 3rd Int'l Workshop on Software and Performance*, pages 114–127, 2002.

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

[5] 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.

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

[7] R. E. Jones and C. Ryder. A study of java object demographics. In *Proc. of the 7th Int'l Symp. on Memory Management*, pages 121–130, 2008.

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

[9] 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.

[10] 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*, pages 343–358, 2002.

[11] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant tracks: Portable production of complete and precise gc traces. In *Proc. of the 2013 Int'l Symp. on Memory Management*, pages 109–118, 2013.

[12] 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, 2011.