Runtime Object Lifetime Profiler for Latency Sensitive Big Data Applications

Rodrigo Bruno*
rodrigo.bruno@inf.ethz.ch
Systems Group, Department of Computer Science, ETH Zurich
Switzerland

Duarte Patrício
dpatricio@gsd.inesc-id.pt
INESC-ID / IST - Técnico, ULisboa
Portugal

José Simão
jsimao@cc.isel.ipl.pt
INESC-ID / ISEL, IPL
Portugal

Luís Veiga
luis.veiga@inesc-id.pt
INESC-ID / IST - Técnico, ULisboa
Portugal

Paulo Ferreira
paulofe@ifi.uio.no
University of Oslo / INESC ID
Norway / Portugal

Abstract

Latency sensitive services such as credit-card fraud detection and website targeted advertisement rely on Big Data platforms which run on top of memory managed runtimes, such as the Java Virtual Machine (JVM). These platforms, however, suffer from unpredictable and unacceptably high pause times due to inadequate memory management decisions (e.g., allocating objects with very different lifetimes next to each other, resulting in severe memory fragmentation). This leads to frequent and long application pause times, breaking Service Level Agreements (SLAs). This problem has been previously identified, and results show that current memory management techniques are ill-suited for applications that hold in memory massive amounts of long-lived objects (which is the case for a wide spectrum of Big Data applications).

Previous works reduce such application pauses by allocating objects in off-heap, in special allocation regions/generations, or by using ultra-low latency Garbage Collectors (GC). However, all these solutions either require a combination of programmer effort and knowledge, source code access, off-line profiling (with clear negative impacts on programmer's productivity), or impose a significant impact on application throughput and/or memory to reduce application pauses.

We propose ROLP, a Runtime Object Lifetime Profiler that profiles application code at runtime and helps pretenuring GC algorithms allocating objects with similar lifetimes close to each other so that the overall fragmentation, GC effort, and application pauses are reduced. ROLP is implemented for the OpenJDK 8 and was evaluated with a recently proposed open-source pretenuring collector (NG2C). Results show long tail latencies reductions of up to 51% for Lucene, 85% for GraphChi, and 69% for Cassandra. This is achieved with negligible throughput (< 6%) and memory overhead, with no programmer effort, and no source code access.

CCS Concepts • **Software and its engineering → Memory management**; **Garbage collection**; **Runtime environments**;

Keywords Big Data, Garbage Collection, Pretenuring, Tail Latency, Profiling

1 Introduction

Big Data applications suffer from unpredictable and unacceptably high pause times due to bad memory management (Garbage Collection) decisions. This is the case of credit-card fraud detection or website targeted advertisement, for example, among others. These systems rely on latency sensitive Big Data platforms (such as graph-based computing or in-memory databases) to answer requests within a limited amount of time (usually specified in Service Level Agreements, SLAs). Such pauses in these platforms delay application requests which can easily break SLAs.

This latency problem has been previously identified [11, 19, 20] and results from a combination of factors. First, Big Data platforms keep in memory large volumes of data. Second, current Garbage Collector (GC) algorithms highly rely on object copying to compact memory. Third, object copying is bound to the physical memory bandwidth which has been increasing more slowly when compared to the number of cores and size of the memory available in current commodity hardware [3, 10, 13], resulting in memory bandwidth being the bottleneck for many parallel applications. In sum, although the widely accepted hypothesis that most objects

*Work done while at INESC-ID / IST - Técnico, ULisboa.

EuroSys '19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303988

die young [26, 39] is still valid for most platforms, the overhead that results from handling the minority of objects that live longer is not negligible and thus, these objects need to be handled differently. This problem has been described as a mismatch between the generational hypothesis and the epochal hypothesis [33].

Recent works (more details in Sections 2 and 9) try to alleviate this high latency problem by taking advantage of programmer's knowledge. To do so, proposed techniques resort to modifying code through: i) manual refactoring of the application code [21], ii) adding code annotations[11, 33], or iii) static bytecode rewriting [9, 34]. The modified code reduces the GC effort by either using off-heap memory,[1] or by redirecting allocations to scope limited allocation regions[2] or generations, leading to reduced GC effort to collect memory. However, these works have several drawbacks as they require: i) the programmer to change application code, and to know the internals of GC to understand how it can be alleviated; ii) source code access, which can be difficult if libraries or code inside the Java Development Kit needs to be modified; and iii) workloads to be stable and known beforehand, since different workloads might lead to completely different GC overheads.

Our work shares the same main goal with many previous works (reduce application pauses). However, we propose a number of additional goals that allow our solution to avoid the drawbacks present in previous solutions. In other words, our solution: i) requires no programmer knowledge and effort; ii) it works without any access to application source code, and iii) it copes with unknown/dynamic workloads. In addition, ROLP has no significant negative impact on throughput or on memory usage, and it works as a simple JVM command line flag.

Long application pauses caused by GC are mainly due to copying objects during object promotion and compaction. To reduce such copies, objects with different lifetimes should be allocated in different spaces, thus reducing fragmentation. To identify such objects with different lifetimes, we propose ROLP, an object lifetime profiler running inside the JVM that tracks object allocation and collection. The profiler has one main goal, to estimate the lifetime of objects based on their allocation context. Using this information, we take advantage the following hypothesis: if a high percentage of objects allocated through a particular allocation context are long-lived, then it is expected that future objects allocated through the same allocation context will also live for a long time. In other words, this hypothesis states that an object's allocation context can be used as an indicator of its lifetime [5]. According to our experience, indicators such as the object type do not provide accurate information due to the intensive use of factory methods, common in object oriented languages, that allocate objects used for very different use-cases (with different lifetimes).

Profiling information produced by ROLP makes it possible to instruct the JVM to allocate long-lived objects close to each other, in a separate space, thus avoiding the cost of copying them multiple times (which leads to long application pauses). ROLP is targeted for long-running cloud applications that hold massive amounts of objects in memory for a long time, and have low latency requirements. Target applications and motivation for this work is further explored in Section 2.

ROLP is implemented in the OpenJDK 8 HotSpot JVM, one of the most widely used industrial JVMs. To take advantage of the profiling information, ROLP is integrated with NG2C [11], a pretenuring GC (based on Garbage First [16]) that can allocate objects in different allocation spaces. Note that any other collector that supports multiple allocation spaces can also be used.

To the best of our knowledge, ROLP is the first object lifetime profiler that can categorize objects in multiple classes of estimated lifetime, implemented in a production JVM with negligible performance overhead. ROLP supports any application that runs on top of the JVM (i.e., it is not limited to the Java language) and users can benefit from reduced application pauses with no programmer effort or any need for off-line profiling. ROLP builds upon NG2C by providing automatic lifetime information that is used by NG2C to allocate objects in different memory locations (according to the estimated lifetime). ROLP also provides package filters that can be used to reduce the performance overhead introduced by profiling code in large applications. These package filters are easier and less error-prone to use when compared to hand-placed annotations necessary to use NG2C. As shown in the evaluation section (Section 8), when compared to other approaches, ROLP can be used to greatly reduce application pause times.

2 Motivation

This section presents the long pause time problem in current runtime systems, its importance, and why it cannot be solved using current GC algorithms and systems.

2.1 Tail Latency introduced by Garbage Collection

Partitioning allocated objects by their estimated lifetime is a fundamental design aspect in current GC design [25]. However, due to the high cost of estimating the lifetime of objects, most collectors simply rely on the weak generational hypothesis [39] that states that most objects die young, and therefore, allocate all objects in the same space and pay the cost of promoting the minority of objects that live longer than most objects.

While this design works well for applications that follow the generational hypothesis, it leads to long tail latencies for

[1] Off-heap is a manually managed backing store made available by the JVM.
[2] Objects' reachability is limited by the scope of allocation.

applications that handle many middle to long-lived objects (as it happens with many Big Data applications [10]). Such latencies come from long copy times (compacting live objects) that are bound to hardware memory bandwidth. In addition, these copy times will become longer and longer as the number of cores and memory capacity is increasing faster than memory bandwidth in current commodity hardware [10].

Recently proposed GC works [11, 14, 33] try to reduce the overhead of estimating the lifetime of objects. After integrating this new information into the GC, a better object partition by lifetime is possible, leading to reduced fragmentation and thus, reduced latency. However, these techniques are heavy, error-prone, require source code changes, are limited to simple workflows, or can only profile code during code warm-up. This topic is discussed in Section 9.

2.2 Trading Throughput and Memory for Latency

Other GC implementations such as C4 [37], Shenandoah [18], and ZGC [3] solve the latency problem by performing all GC tasks almost fully concurrent with the mutator (application threads). These collectors still require copying massive amounts of objects but most copies are done concurrently with the mutator. Thus, these GCs incur very short pauses; however, the drawback is the application throughput overhead that is caused by the heavy use of both read and write barriers, and the increased memory usage (see results in Section 8.5).

On the one hand, current GC algorithms, which heavily rely on object copying to compact memory inflict unpredictable and unacceptably long tail latencies on applications. This situation will not improve as memory bandwidth is a scarce resource, even more with the fast developments on the number of cores and memory capacity. On the other hand, concurrent GCs reduce overall GC latency but at a high throughput and memory cost.

The work now presented, ROLP, is a new solution in the Throughput-Memory-Latency trade-off as it reduces the overall latency, with special focus on long tail latency, and inflicts a negligible impact on throughput and memory usage. We envision that this solution is mostly beneficial for long-running cloud platforms that are latency sensitive, as it happens with many Big Data applications. There are additional use-cases for ROLP such as detecting memory leaks in applications by reporting object lifetime statistics per allocation context. Nevertheless, we will not explore use-cases in this paper however.

3 Object Lifetime Profiling

ROLP is built to answer one simple question: how long will an object live, based on its allocation context. To answer this question, we must first define the notion of time and allocation context. On one hand, time is measured in GC cycles, i.e., the GC cycle is the unit of time. Thus, the age of an object is the number of GCs that an object has survived. On the other hand, ROLP defines an allocation context as a tuple of: i) an allocation site identifier, which identifies the line of code where the object is allocated, and ii) a thread stack state, which describes the state of the execution stack upon allocation.

3.1 Solution Overview

ROLP uses different techniques to answer the proposed question. First, upon allocation, all objects are marked in their header with an allocation context that identifies both the allocation site (i.e., line of code) and the thread stack state. As detailed below, ROLP takes advantage of profiling code installed during Just-In-Time (JIT) compilation to accomplish this task. To know the age of objects, ROLP tracks both the number of allocated objects, and survivor objects during GC cycles. This information (number of allocated and survivor objects) is kept in a global Object Lifetime Distribution table (see Fig. 1). This table maintains the number of objects with a specific age organized by allocation context. In the following sections, we describe these techniques in detail.

3.2 Application Code Instrumentation

ROLP only profiles very frequently executed/hot application code. To that end, we take advantage of the JIT compilation engine in the HotSpot JVM to identify/define hot application code. There are two reasons why ROLP only profiles hot code. First, installing profiling code has a cost (e.g., for creating unique identifiers for allocation sites) and thus, it makes sense to pay this cost only for application code that is executed very frequently (note that only a small fraction of the application code is usually hot). Second, since most of the execution time is spent running hot code, not profiling code that is not executed frequently (i.e., cold code), does not lead to a significant loss of profiling information.

In short, the profiling code (added to the application code during JIT) is responsible for performing the following tasks: i) update the thread stack state (thread-local value that encodes the state of the execution stack) whenever a new frame is pushed or removed from the stack; ii) increment the number of allocated objects (in the Object Lifetime Distribution table) for the corresponding allocation context, upon object allocation; and iii) install the allocation context in the object header (see Fig. 2), upon object allocation. Note that, as ROLP does not profile cold methods (i.e., non JIT compiled), it does not record lifetime information of all objects. ROLP focuses on profiling the code that is executed more frequently in the hope of achieving the best trade-off of profiling overhead versus performance benefits. The next sections describe each one of these tasks in detail.

[3] http://openjdk.java.net/projects/zgc

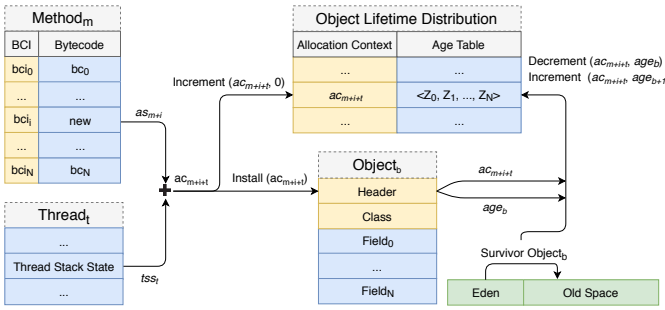


Figure 1. ROLP Profiling Object Allocation and GC Cycles

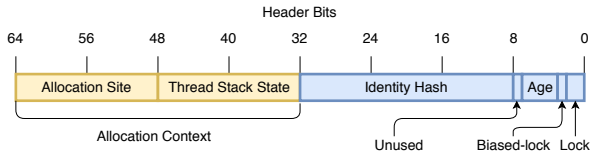


Figure 2. Object Header in HotSpot JVM using ROLP

3.2.1 Context of Allocation

The number of allocated objects per allocation context is maintained in the Object Lifetime Distribution table (see Fig. 1). As depicted, upon each object allocation, the allocation context ac_{m+i+t} is generated by combining both: i) the allocation site identifier (as_{m+i}), which identifies the specific code location where the allocation is taking place (method m, bytecode index i), and ii) the thread stack state ss_t, which identifies the state of the execution stack of the thread t (which is allocating the object). The resulting allocation context is installed in the header of the newly allocated object.

3.2.2 Marking Objects with the Allocation Context

ROLP associates each allocated object with an allocation context by storing the corresponding allocation context in the object's header. Note that adding more information to application objects (for example, increasing the header size) is undesirable as it increases the memory footprint by adding extra bytes to every object. Therefore, ROLP reuses spare bits that already exist in an object header.

Figure 2 presents the 64-bit object header used for each object in the HotSpot JVM. As depicted, for each object, ROLP installs the corresponding allocation context in the upper 32 bits of the 64-bit header. These 32 bits are currently only used when an object is biased locked towards a specific thread,[4] and using them does not compromise the semantics of biased locks. Given that ROLP installs an allocation context upon an object allocation, if the object becomes biased locked, the profiling information will get overwritten. In addition, biased locking is controlled by the JVM using a specific bit in an object header (bit number 3). Thus, if the object is biased locked (i.e., if bit number 3 is set) or if the allocation context is corrupted (i.e., it does not correspond to any entry in the Object Lifetime Distribution table), the object is simply discarded for profiling purposes. Profiling information can be mistakenly used if the upper 32 bits were used recently for biased locking and the OLD table contains an allocation context which matches the same 32 bits. This is a very rare scenario as the upper 32 bits of the object header (which store the pointer to the thread that owns the biased lock) must match the same exact 32 bit of an allocation context.

Using the space dedicated to biased locks means that ROLP looses some profiling information. However, through our experience and based on previous evaluation results, we argue that: i) the number of biased locked objects in Big Data applications is not significant; ii) data objects are usually not used as locks (and therefore are not biased locked); iii) not profiling control (non-data) objects does not lead to a significant loss of important information since these control objects are usually small both in size and number.

3.2.3 Allocation Context Tracking

As already mentioned, the allocation context is a tuple of: i) allocation site identifier that identifies a specific line of code, and ii) thread stack state. The later is necessary to distinguish two object allocations that, although using the same allocation site identifier (i.e., the same code location), use different call paths to reach the allocation site. This is a very common scenario when object allocation and initialization is delegated to libraries or frameworks.

ROLP uses arithmetic operations (sum and subtraction) to incrementally update the 16 bit thread stack state of each thread. Thus, before each method call, the thread-local stack state is incremented with a unique method call identifier. The same value is subtracted when the execution exits the method.

This technique relies on the following. First, for allocation tracking purposes, it suffices that the thread stack state differentiates two different call paths. Hence, the order of the method calls that compose each call path is not required to be contained in the thread stack state. Second, this state must be incrementally maintained as the application execution goes through the call path and enters and leaves methods.

However, adding two arithmetic operations for each method call can lead to throughput penalties as method calls are very common in high level languages. To cope with this problem, ROLP is able to dynamically turn on and off the execution stack tracking for each method call. Hence, method call profiling code is only enabled for method calls that can differentiate call paths leading to the same allocation site. This process is discussed in Section 5.

Finally, it is also possible to have collisions in the thread stack state, i.e., if two or more different call paths lead to the same execution stack state. This problem is greatly reduced by two factors. First, we only profile hot code, thus greatly

[4] Biased Locking is a locking technique available for the HotSpot JVM which allows locking an object towards a specific thread [17].

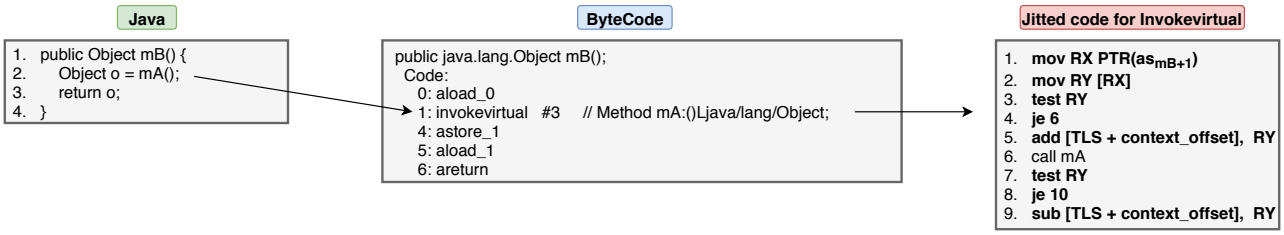


Figure 3. Method Call Code Sample: from Java (left) to Bytecode (middle) to x86 Assembly (right)

reducing the number of method calls that can contribute to a collision. Second, a collision would only be harmful if the allocation site is the same for the values that are colliding. Execution stack states that collide in different allocation sites are not a problem (i.e., they correspond to different lines in the Object Lifetime Distribution table). Nevertheless, we show in Section 8.3 that conflicts are very rare.

3.2.4 Code Profiling Example

We now analyze a snippet of code and see how ROLP installs the profiling code. Figure 3 presents a simple snippet of Java code (left), the result of its compilation to Bytecode using the Java compiler `javac` (center), and the `x86` Assembly code for the `invokevirtual` instruction produced by the OpenJDK HotSpot Opto JIT compiler (right), which also contains the profiling code introduced by ROLP. Both the Bytecode and the Assembly code presented in this figure are simplified for clarity reasons. We do not present an example with the `new` instruction as it is more complex and would require more space to illustrate with almost no benefit compared to analyzing the `invokevirtual` Assembly code. We now analyze the Assembly code generated for the `invokevirtual` instruction (right side of Figure 3).

Looking at the generated Assembly code, lines 1 to 5 and 7 to 9 correspond to profiling instructions introduced by ROLP(lines in bold). These instructions are meant to increment (lines 1 to 5) and to decrement (lines 7 to 9) the thread stack state by the specific amount that was calculated for this specific line of code (as_{mB+1}). The increment or decrement Assembly instructions (`add` and `sub`) are executed on the condition that the value of as_{mB+1} is non-zero (note the `test` and `je` Assembly instructions in lines 3, 4, 7, and 8).

This conditional execution of the thread stack state update, enables ROLP to turn on and off the profiling of method calls. By doing so, ROLP avoids the execution of the `add` and `sub` instructions which are costly as they may require loading and storing values to main memory (if the values are not cached). In other words, ROLP introduces a not so expensive short branch to avoid an expensive memory access. These instructions need to read and write to the current execution stack state which is stored `context_offset` bytes away from the Thread Local Storage (TLS, which is kept in a special register). Other than these two instructions (`add` and `sub`), only the `mov` instruction in line 2 requires memory access (which is much slower compared to operations performed using only registers or cached values). However, even for this instruction, which is necessary to load into memory the value that is added to the thread stack state, we try to keep it in cache by storing it right next to the compiled code in memory. Thus, when the method's Assembly code is loaded before it is executed, the value of as_{mB+1} will most likely be cached in the CPU.

3.3 Updating the Object Lifetime Distribution Table

The information regarding the number of objects allocated per allocation context and age, is kept in the global Object Lifetime Distribution table (see Fig. 1). Besides being updated upon object allocation (to increment the number of objects with age zero), this table is also updated during GC cycles to update the number of objects that survived a GC cycle. In particular, let's assume an object allocated in the allocation context ac_{m+i+t} with age age_o that survives a GC cycle. The Object Lifetime Distribution table will be updated to: i) decrement the number in the cell corresponding to row ac_{m+i+t} and column age_o (one object less with age age_o); ii) increment the number in the cell corresponding to row ac_{m+i+t} and column age_{o+1} (one object more with age age_{o+1}).

This process is also depicted in Figure 1. In short, with ROLP, GC worker threads that are copying survivor objects will look into an object's header (see Fig. 2) and extract the allocation context and the age of the object (field maintained and updated by the collector). If the object is biased locked or if the allocation context is not present in the Object Lifetime Distribution table, the object is not considered for profiling purposes. Otherwise, the worker thread will update the table. By the end of each GC cycle, the global table presented in Figure 1 contains the number of objects organized by allocation context and age.

4 Inferring Object Lifetimes

In order to infer the lifetime of objects allocated through a particular allocation context, e.g., ac_x, ROLP periodically analyzes the number and age of the objects allocated through ac_x. This operation is performed for every allocation context once every 16 GC cycles. This value is used because it is the maximum age of objects in HotSpot (considering that

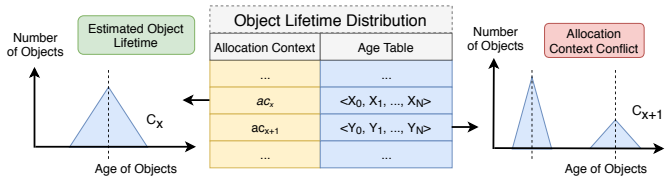


Figure 4. Curves from the Object Lifetime Distribution Table

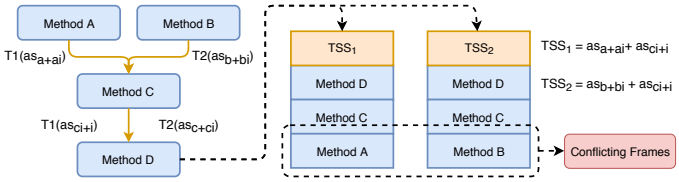


Figure 5. Thread Stack State on Context Conflicts

the age bits in an object's header is only 4 bits long), after which, the age of the object does not increase more. In order to ensure freshness, the Object Lifetime Distribution table is cleared after inferring the lifetime of all allocation contexts.

To estimate the lifetime of objects allocated through ac_x, a curve (C_x) plotting the number and age of (objects allocated through ac_x) is created (Fig. 4, left side). The resulting curve is most often very similar to a triangular shape (similar to the triangular distributions previously reported by Jones and Ryder [26]), whose maximum shows at which age most objects die. Hence, by determining the maximum of C_x, it is possible to infer with high confidence the estimated lifetime of objects allocated through ac_x.

It is possible, however, that a single curve (C_{x+1}, for example) shows not only one, but multiple triangular-like shapes (Fig. 4, right side). Such a curve shows that the objects allocated through the allocation context ac_{x+1} may live for different spans of time. In such a situation, we consider that we found a context conflict, which is possible if the same allocation site is being reached through multiple call paths. In the following section, we discuss how we deal with these allocation context conflicts.

5 Allocation Context Conflicts

Tracking the thread stack state is potentially harmful for the performance of an application as such tracking introduces a considerable amount of profiling effort. Therefore, a trade-off needs to be found. In one hand, not tracking the thread stack state means that ROLP would fail to solve allocation context conflicts (i.e., distinguish allocations that share the same allocation site but use different call paths); on the other hand, updating the thread stack state on every method call (and return) introduces undesired throughput overheads.

The sweet spot for this trade-off problem is achievable by identifying the minimum set of method calls that allows ROLP to distinguish different call paths leading to the same allocation site. With such a minimum set of method calls, called S, it is sufficient to profile only the method calls in the set to solve all allocation context conflicts. In other words, ROLP only has to update the execution stack state when the methods calls in S are executed, thus avoiding conflicts with the minimum amount of throughput overhead. Figure 5 presents an example of two different thread stacks (call paths) that lead to the same allocation site and also shows (in red) the conflicting frame. In this particular example, S must contain either the call from A to C or B to C.

Identifying such minimum set of methods calls (S) is not a trivial task mainly due to the extreme use of polymorphism in modern languages. Hence, it is not possible to have precise information regarding callers and callees, at runtime, without extreme performance overhead. Therefore, we propose a low-overhead algorithm that iteratively finds S. The algorithm works as follows:

1. at JVM startup, no method call is profiled (i.e., a thread's stack state is not updated when the thread enters or exits a method); only allocation sites are profiled to install allocation site identifier into object headers;

2. conflict checking is performed during object lifetime inference (as described in Section 4). Whenever a conflict is detected (multiple triangle shapes in the same curve), P method calls are randomly selected to start tracking the thread-local stack state. P stands for an implementation specific number of method calls to profile at a time (we recommend that P should not be higher than 20 % of the total number of jitted method calls to avoid too much throughput overhead);

3. upon the next object lifetime inference, if the conflict was resolved, S must be contained in P method calls. In this case, ROLP can start to turn off method calls tracking until S is found. If the conflict was not solved, then a new random set of P method calls must be selected (avoiding repeated method calls) and the process continues until all method calls are exhausted or until the conflict is resolved.

It is possible to have multiple sets of P methods being tracked at the same time, i.e., trying to solve multiple conflicts. Note, however, that P should be adjusted (reduced) as the number of parallel conflicts may increase so as to avoid high throughput overhead.

This algorithm presents two interesting properties. First, it is possible to dynamically control the number of method calls that are being tracked (or profiled) at a time while trying to resolve conflicts. Second, the algorithm converges in linear time to the number of jitted method calls divided by P and multiplied by the number of GC cycles between each conflict checking operation (16 GC cycles); this means that it is possible to predict, on the worst-case, how long it will take to finish (we show this experiment in Section 8).

In short, as demonstrated in the evaluation (Section 8), conflicts are rare. Therefore, in order to solve conflicts, ROLP uses a low-overhead technique, as described above, opposed to using more performance intrusive techniques.

6 Updating Profiling Decisions

The lifetime of objects allocated through a particular allocation context can change over time if, for example, the application workload changes. To cope with these changes, ROLP needs to continuously update its profiling decisions. Two types of situations are specially important for object lifetime profiling: i) if the lifetime of objects allocated through an allocation context increases, or ii) decreases.

On one hand, if the lifetime of objects allocated through allocation context ac_x increases, it means that objects allocated through ac_x are surviving more collections than before (and the Object Lifetime Distribution table will evidence that). This allows a pretenuring collector to take action and pretenure objects allocated by ac_x to an older space which is collected less often.

On the other hand, if the lifetime of objects allocated through ac_y decreases, the only visible effect is the increase in memory fragmentation (this information is updated by the collector at the end of each memory tracing cycle). When fragmentation is detected, ROLP identifies which allocation contexts are allocating objects in the fragmented memory regions and decrements their estimated object lifetime.

7 Implementation and Optimizations

ROLP is implemented for the OpenJDK 8 HotSpot JVM (build 25-b70). Since HotSpot is a highly optimized production JVM, new algorithms/techniques must be implemented carefully to prevent breaking JVM's performance. This section describes some of ROLP's implementation details, in particular, the ones we believe to be important for realistically implementing ROLP in a production JVM.

7.1 Integration with Pretenuring GC

ROLP is integrated with NG2C [11], a freely available recently proposed pretenuring collector that allows the heap to be divided into an arbitrary number of generations. The motivation behind NG2C is to allocate objects with similar lifetimes in the same generation to reduce fragmentation.

In order to integrate ROLP with NG2C, we pre-configured NG2C to have 16 generations (the young generation, the old generation, and other 14 generations used to hold pretenured objects, separated by estimated lifetime). In practice, what NG2C does is to sub-divide G1's old generation into multiple allocation spaces, and allow the collector to allocate application objects into each of these allocation spaces (which are called dynamic generations). Sixteen generations are used as it the maximum age of an object in HotSpot.

With ROLP, we modified NG2C to use ROLP profiling results to select a generation for allocation. Upon object allocation, we instruct NG2C to look into the table that contains lifetime estimations (which results from the analysis done in Section 4) and to use the estimated age of an object (a number between 0 and 15) as the number of the generation to use (i.e., where that object will be allocated). If the estimated age is zero, NG2C allocates the object in the young generation; if the estimated age is "G" (0<G<15), NG2C allocates the object in one of the dynamic generations (generation "G" in this case). The fifteenth corresponds to the old generation.

With regards to collecting garbage, ROLP does not bring any modification besides updating the Object Lifetime Distribution table whenever an object survives a collection. For more details on how collection is done using multiple generations, please refer to Bruno et al. [11].

7.2 Inlining, Exceptions, and OSR

The HotSpot JVM is one of the most optimized runtimes. In this section, we analyze some techniques used by the JVM and how ROLP handles them.

7.2.1 Method Inlining

Method inlining is an important performance optimization for JVM applications. It allows a call to method A to be replaced with its code. This can lead to significant performance improvements as the cost of the method call is completely avoided. There are a number of factors that control how the JIT compiler in HotSpot deals with inline methods such as the size of the method, and if the call is polymorphic or not (i.e., if it can result in an invocation to different methods).

After studying this problem and analyzing both real application code and execution logs from JIT compilation, we realized that most methods being inlined contain very little control flow, and are mostly simple operations that, because of being done many times, are abstracted into a separate method. With this observation in mind, and trying to reduce the number of profiled method calls (to reduce the throughput impact of ROLP), we decided not to profile inlined method calls, i.e., whenever the JIT is inlining a method call (i.e., replacing the call with the actual method implementation), we do not include any profiling code to track the thread stack state around the method that is being inlined. In addition, we conducted several experiments with an without this optimization (using the benchmarks described in Section 8) and we noticed that no conflict was left unresolved after using this optimization.

7.2.2 Exception Handling

Exception handling is another important topic as it breaks the assumption that after returning from a method, the thread stack state of the executing thread will be updated (remember that we update the thread stack state before and after each method call). However, exceptions can break this technique as an unhandled exception will climb the stack until: i) there is a suitable exception handler, or ii) the application terminates with the exception.

In practice, when an exception is thrown, the JVM will look for a suitable exception handler to handle it. If there is no suitable handler in the current method, the exception is

automatically re-thrown, and is going to be catched by the JVM stubs in the caller method. Note that when the JVM re-throws an exception, the execution goes directly to the JVM stub in the caller method, i.e., the profiling code installed right after the call is not executed.

In order to fix this problem, and to avoid the thread-local stack state being inconsistent with the execution stack, ROLP hooks code to update the stack state whenever the JVM decides to re-throw an unhandled exception. This way, even if exceptions are not handled in the current method, exiting a method through an unhandled exception will not lead to a corruption of the stack state.

7.2.3 On Stack Replacement

On Stack Replacement (OSR) is yet another important technique used by the HotSpot JVM. This technique allows the JVM to change how a particular method is executed (either through interpretation or through JIT compiled code) while the method is being executed. It is particularly useful for JIT compiling long-running methods and for method de-optimization. OSR can be harmful for ROLP's thread stack state updates because any method in the stack can go from an interpreted method into a compiled method. Given that ROLP only installs profiling code in compiled/jitted code, switching implementations after entering a particular method would corrupt the thread stack state.

To solve this problem, ROLP periodically verifies the correctness of an application threads stack state by traversing the thread stack and computing the expected thread stack state. This is done at the end of each GC cycle, while all application threads are still stopped. If ROLP finds an incorrect context state, it will correct its value, making it consistent with the real execution stack. After testing the performance of applications with and without this technique, we concluded that its cost is negligible, and is absorbed by the cost of the other collection tasks. ROLP could also have tackled this problem by patching all code locations where OSR is triggered. However, this would require a large engineering effort and would probably lead to throughput overhead. The proposed solution trades short-term imprecision for low throughput overhead.

7.3 Reduce Profiler Overhead on Large Applications

Profiling large scale applications can be challenging from the performance point of view. As shown in Section 8, even for DaCapo benchmarks with no context conflicts, some benchmarks experienced more than 10 % throughput overhead. In other words, even with highly optimized produced JIT code for profiling the application, it is not possible to reduce the throughput to negligible values for some applications.

To further reduce the throughput overhead, ROLP allows the definition of package-based filters to either profile or not a package (and all its sub-packages). We found this extremely useful and effective to bound the throughput overhead. In practice, we used this technique in the large scale workloads (described in the Section 8) to focus the profiling effort on packages that manage application data structures. In addition, identifying these packages is effortless for most programmers as even the name of the packages is, most of the time, indicative of the purpose of the code in it (as it happens in the platforms used to evaluate ROLP).

7.4 Shutting Down Survivor Tracking to Reduce Application Pause Times

During ROLP's development we realized that ROLP, after reducing the number of objects being copied during a collection, the profiling code was the new bottleneck during a collection. After analyzing this effect, we found out that this was due to the profiling code that extracts the allocation context from an object's header, and looks it up in the Object Lifetime Distribution table. This operation is performed for every object that survives a collection. Thus, we noticed that, after starting to pretenure objects (using NG2C), the dominating phase of a GC cycle was the survivor processing phase.

Therefore, to further reduce the application pause times, ROLP can dynamically turn off the survivor tracking code. By doing this, it is possible to reduce even further GC pause times. Note that ROLP only performs this optimization (i.e., turning off the survivor tracking code) if the workload is stable (i.e., the profiling decisions regarding the estimated lifetime of objects did not change in the last iteration). Obviously, it is also possible to turn on the survivor tracking code again. Currently, this code is only turned back on if the average pause time increases over 10% (this is a configurable value) compared to the last recorded value when the survivor tracking code was active.

7.5 Object Lifetime Distribution Table Scalability

ROLP uses a global table (Object Lifetime Distribution) which is accessed very frequently. In order to provide average constant time for insertion and search, this data structure is implemented as a hashtable.

Another important concern is how large is the memory budget to hold this table in memory. In the worst-case scenario, and since the allocation context is a 32 bit value, one could end up with a table with 2^{32} entries which would take 4 bytes * 16 columns * 2^{32} entries (approximately 256 GB). However, in practice, we are able to keep the size of this table to a much lower value (as can be see in the Section 8).

The table is initialized with 2^{16} entries, one for each possible allocation site identifier. At this point, the table occupies approximately 4 MB of memory. Whenever a conflict is detected, the table size is increased by 2^{16} to be able to accommodate all possible thread stack state values for the specific allocation site where the conflict was found. Hence,

the size of the table is 2^{16} * (1 +N) entries, which is equivalent to 4 * (1 +N) MB, where N is the number of conflicts.

7.6 Updating the Object Lifetime Distribution Table

The Object Lifetime Distribution table is updated by application threads during object allocation, and by GC threads during object promotion/compaction. By design, application threads and GC threads do not update the table at the same time. However, concurrent accesses still exist between each type of threads (application or GC).

In order to allow fast updates by application threads, two options were analyzed: i) have a thread-local table, which periodically is used to update the OLD table; ii) use the OLD table with no access synchronization (risking some increment misses). ROLP uses the latter approach for three reasons: i) it is possible to write more efficient native code (jitted code) because the address where the counter that needs to be incremented resides is already known at JIT time; ii) it requires less memory to hold profiling information; iii) the probability of loosing counter increments is small as two threads would have to match the same exact allocation context at the same time. In other words, ROLP trades performance for slight imprecision. According to our experience while developing ROLP, this loss of precision is not enough to change profiling decisions, i.e., the profiler takes the same decisions with and without synchronized counters.

GC worker threads must also update the OLD table to account for objects that survive collections. However, opposed to application threads, the contingency to access the global table is higher since all worker threads may be updating the table at the same time during a garbage collection. This would lead to significant loss of precision if no synchronization takes place. In order to avoid that, private tables (one for each GC worker thread) containing only information regarding the objects promoted/compacted by a particular worker thread are used. All these private tables are used to update the OLD table right after the current garbage collection finishes.

8 Evaluation

The goal of this evaluation section is twofold. First, we analyze the performance overhead introduced by profiling code. Second, we measure the pause time reductions resulting from ROLP's profiling information and compare it to previous works.

Five systems/collectors available for the OpenJDK HotSpot JVM are compared in this evaluation: i) **CMS**, the throughput oriented collector; ii) **G1**, the current default collector; iii) **ZGC**, a newly proposed fully concurrent collector; iv) **NG2C** the pretenuring collector (based on G1) which uses hand-placed code annotations to indicate estimated object lifetimes; and v) **ROLP**, the runtime object lifetime profiler, integrated with NG2C. Note that we do not show pause times for ZGC as it is fully concurrent and we did not observe pauses superior to 10 ms.

The evaluation was performed in a server equipped with an Intel Xeon E5505, with 16 GB of RAM. The server runs Linux 4.13. Each experiment runs in complete isolation for 5 times (enough to be able to detect outliers). All workloads run for 30 minutes each. When running each experiment, the first five minutes of execution are discarded to ensure minimal interference from JVM loading, JIT compilation, etc. We also ran experiments such as Cassandra (described below) in a cluster environment but, for the purposes of this evaluation, there is no difference between exercising a single Cassandra instance or to use a cluster of Cassandra instances and then look at the GC behavior in each one.

8.1 Workload Description

This section presents the workloads used to evaluate ROLP. We prepared two groups of benchmarks: i) a set of benchmarks from **DaCapo 9.12 bach-MR1** benchmark suite, and ii) a set of common large-scale production platforms to mimic real Big Data workloads.

The DaCapo benchmark suite is a well known and widely studied set of benchmarks to study the performance of JVM implementations. Table 2 presents the used benchmarks and their heap sizing configuration; these heap sizes were determined as the necessary amount of memory to run with the best possible throughput, i.e., the less amount of memory to run with the highest possible throughput.

To evaluate our solution with platforms and workloads similar to real-world scenarios, we use the following three platforms (see Table 1). First, we use **Apache Cassandra 2.1.8** [30], a large-scale Key-Value store. Cassandra is used with three different workloads with different read and write request percentages. Second, we use **Apache Lucene 6.1.0** [32], a high performance text search engine which we use to index a Wikipedia dump. Third, we use **GraphChi 0.2.2** [29], a large-scale graph computation engine, which we use to run Connected Components and Page Rank on top of a Twitter graph dump [28]. We also present the packages that we filter for profiling. These specific packages were selected because they are the ones that handle most data in each platform. All platforms run with a memory budget of 6 GB. According to our experience, this memory budget is high enough to avoid memory pressure, allowing both good latencies and throughput. Increasing the memory budget would lead to similar results (i.e., the conclusions take from this experimental evaluation hold with higher memory budgets). Reducing the memory budget would lead to higher GC overhead as there is not enough memory to keep application objects (working set) in memory.

8.2 Profiling Performance Overhead

This section presents ROLP's overhead in the DaCapo benchmark suite. To do so, we devised two experiments: i) run each

Platform	Workload	Data	Packages	PAS	PMC	#CFs	NG2C	OLD
Cassandra	WI - 10k ops/s, 75% writes	YCSB	cassandra.db,	0.023 %	0.020 %	2	22	12MB
Cassandra	RW - 10k ops/s, 50% writes	YCSB	cassandra.utils,	0.030 %	0.023 %	2	22	12MB
Cassandra	RI - 10k ops/s, 25% writes	YCSB	cassandra.memory	0.029 %	0.025 %	2	22	12MB
Lucene	25k ops/s, 80% writes	Wikipedia	lucene.store	0.014 %	0.014 %	0	8	4MB
GraphChi	CC - 42M vert. 1.5B edges	Twitter	graphchi.datablocks,	0.023 %	0.001 %	3	9	16MB
GraphChi	PR - 42M vert. 1.5B edges	Twitter	graphchi.engine	0.021 %	0.001 %	3	9	16MB

Table 1. ROLP Big Data Benchmark Description (left) and Profiling Summary (right)

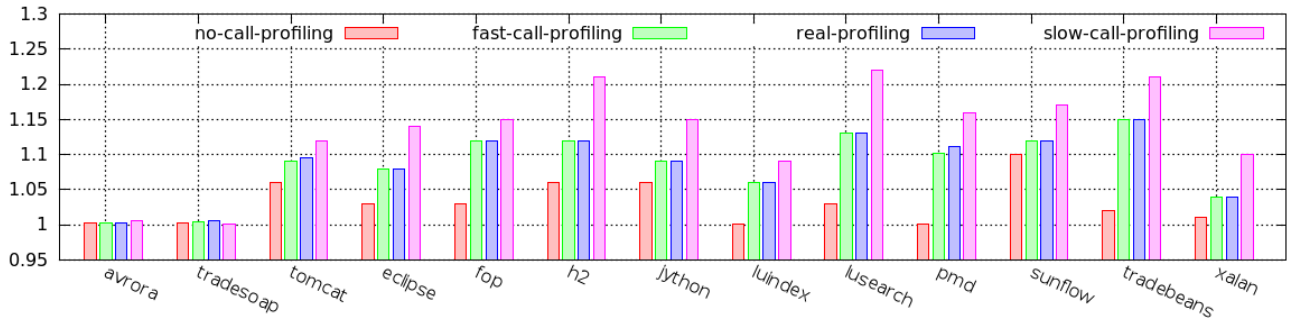


Figure 6. DaCapo Benchmark Execution Time Normalized to G1

Workload	HS	PMC	PAS	CF (# - %)
avrora	32 MB	374	69	0 - 0.04
eclipse	1 GB	1378	329	0 - 1.20
fop	512 MB	3102	829	0 - 0.02
h2	1 GB	1416	116	0 - 1.80
jython	128 MB	11801	741	0 - 1.20
luindex	256 MB	464	89	0 - 0.60
lusearch	256 MB	558	127	0 - 1.80
pmd	256 MB	3157	369	6 - 1.20
sunflow	128 MB	346	225	0 - 1.00
tomcat	512 MB	2891	436	4 - 0.60
tradebeans	512 MB	2145	227	0 - 1.20
tradesoap	512 MB	5815	254	3 - 0.60
xalan	64 MB	2037	406	0 - 1.80

Table 2. DaCapo Profiling (left) and Conflicts (right)

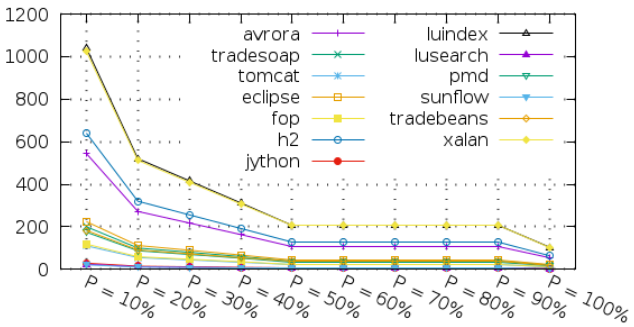


Figure 7. Worst-Case Conflict Resolution Time (ms)

benchmark with different levels of profiling to measure the impact of each type of profiling code in the benchmark's performance; and ii) simulate what would be the overhead of the conflict resolution algorithm (presented in Section 5) and how long it would take in the worst-case scenario.

Figure 6 presents the average execution time of each benchmark normalized to G1 (our baseline). Values above one means it took longer than G1 took to execute. For each benchmark, there are four columns (from left to right): i) **no-call-profiling** represents the execution time with no call profiling, i.e., only object allocation is profiled in this experiment and therefore, the execution overhead comes only from the profiling code inserted for allocation tracking. In other words, no profiling code in inserted for method calls; ii) **fast-call-profiling** represents the execution with all the profiling code enabled except thread stack state, i.e., no method call actually triggers the update of the thread stack state (as described in Section 3.2.4). In other words, all method calls fall through the fast profiling branch, which does not update the thread stack state; iii) **real-profiling** represents real benchmark execution, with all the profiling code; iiii) **slow-call-profiling** represents the worst-case possible execution, with all profiling code, forcing all method calls to update the thread stack state (as described in Section 3.2.4), i.e., all method calls fall through the slow profiling branch, which updates the thread stack state.

We found the results in Figure 6 very interesting as different applications exercise the profiling code in different ways

resulting in different overheads for the same profiling code across different benchmarks. For example, for benchmarks such as `fop`, allocation profiling (the first bar from the left) leads to around 3% overhead while method call profiling leads to almost 10% overhead (difference between the first and second bars from the left). Other benchmarks reveal very different behavior, e.g., the `sunflow` benchmark, with high overhead for allocation profiling and almost zero overhead for method call profiling. It is also interesting to note that the **real-profiling** overhead is very close to the **fast-call-profiling** meaning that very few method calls were profiled in order to solve allocation context conflicts. The left side of Table 2 presents the memory budget (Heap Size) used to run each benchmark, number of profiled method calls (**PMC**), the number of profiled allocation sites (**PAS**), and the number of conflicts found while executing each benchmark. From these results, we confirm that conflicts are not frequent.

On the right side of Table 2 we present the number of context conflicts and simulation results on the expected percentage of throughput overhead for having 20% of all method calls being tracked (P from Section 5 is 20%). This throughput overhead is directly proportional to P and thus, the higher P is, the higher is the throughput overhead. P also impacts the time to resolve conflicts. Figure 7 shows how long conflict resolution would take in the worst-case scenario (we estimate this by taking the average time between two GC cycles) for different values of P.

The low number of conflicts that most benchmarks evidence (right side of Table 2) suggests that thread stack state tracking can be used to solve allocation context conflicts which, however, are rare. For P equals 20%, it is possible to observe that: i) conflict resolution overhead is never above 1.8% of additional throughput overhead, and ii) conflict resolution can take up to 520 seconds but for most benchmarks it does not take more than 2 minutes. It is still possible to reduce the duration by increasing P to higher percentages. However, note that: i) ROLP is targeted to long running applications and this setup time is negligible taking into consideration the overall runtime; ii) during the setup time, the JVM is performing (w.r.t pause times) exactly like G1 (i.e., with no profiling information, memory management resorts to G1 with no modification).

8.3 Large-Scale Application Profiling

This section summarizes the profiling used when evaluating ROLP with the large-scale workloads, and also compares it to the amount of human-made code modifications necessary for NG2C[11]. Table 1 presents a number of metrics for each workload: **PAS**, percentage of allocation sites where profiling code was actually inserted; **PMC**, percentage of method calls where profiling code was actually inserted; **CFs**, number of allocation context conflicts; **NG2C**, number of code locations that were changed to evaluate NG2C (as previously reported [11]); **OLD**, approximate memory overhead of the Object Lifetime Distribution table (see Figure 1);

From Table 1, three important points must be retained. Fist, looking at PAS and PMC, the percentage of profiled allocation sites and method calls is small. This demonstrates that the profiling effort is greatly reduced by only profiling hot code locations, and by using optimizations such as avoiding inlined methods calls. Second, looking at OLD size, the memory overhead introduced to support profiling information does not exceed 16MB, a reasonable memory overhead considering the performance advantages that can be achieved by leveraging the information in it. Finally, the number of allocation context conflicts does not exceed 3, showing that, despite using a weak hash construction (based on addition and subtraction of hashes), it is possible to achieve a low number of conflicts.

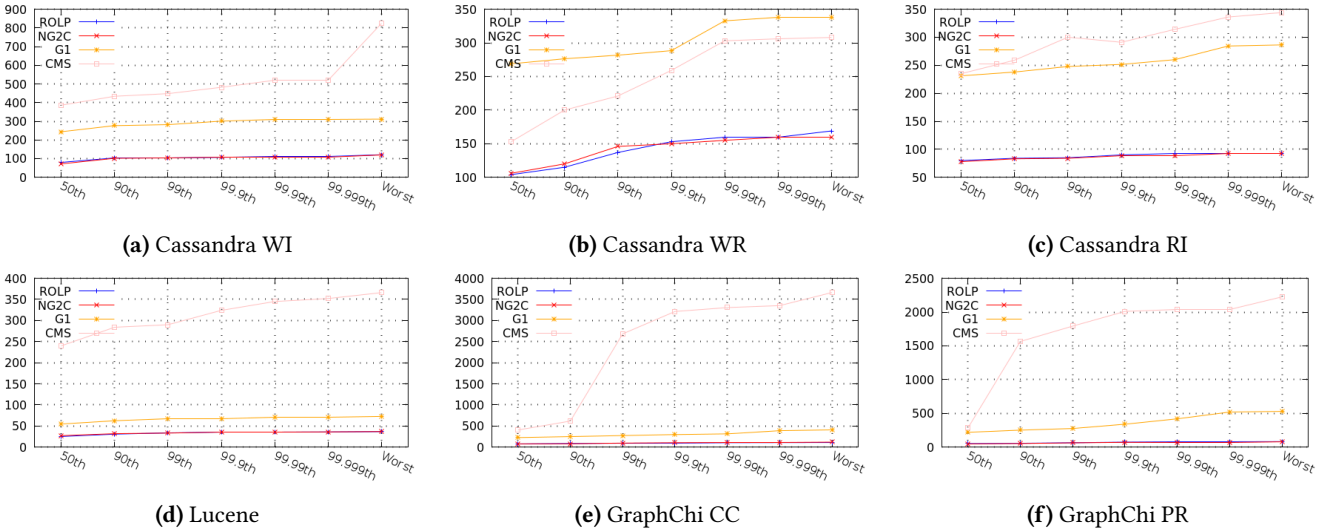
It is worthy to note that all the code changes done on the applications, which are needed to use NG2C, require either human knowledge (i.e., the programmer), or the use of ROLP. When using ROLP, such changes are done automatically, i.e., the code is profiled and changes are done with no human intervention. ROLP additionally profiles other code locations (which are not used for NG2C), leading to additional improvements.

8.4 Pause Time Percentiles and Distribution

Figure 8 presents the results for application pauses across all workloads. Pauses are presented in milliseconds and are organized by percentiles. Note that these are pauses triggered by GC only. Other pauses coming from I/O, OS syscalls, among others, are not considered in this experiments so that we can concentrate our analysis on GC-induces pauses. In the remaining text, we name the results obtained for NG2C with ROLP simply as ROLP.

Compared to G1 and CMS, ROLP significantly improves application pauses for all percentiles across all workloads. Regarding NG2C (which requires developer knowledge), ROLP approaches the numbers provided by NG2C in all workloads. From these results, the main conclusion to take is that ROLP can significantly reduce long tail latencies when compared to G1 and CMS; in addition, it can also keep up with NG2C, but without requiring any programming effort and knowledge. Both ROLP and NG2C produce very stable pause times for most benchmarks, i.e., which do not increase significantly across percentiles, presenting a close to horizontal plotted line. Finally, reducing application pause time does not mean reducing the GC throughput overhread (as presented in the next section). This mostly comes from the fact that the total application pause time (i.e., sum of all application pauses) is very low when compared to the total execution time of the application.

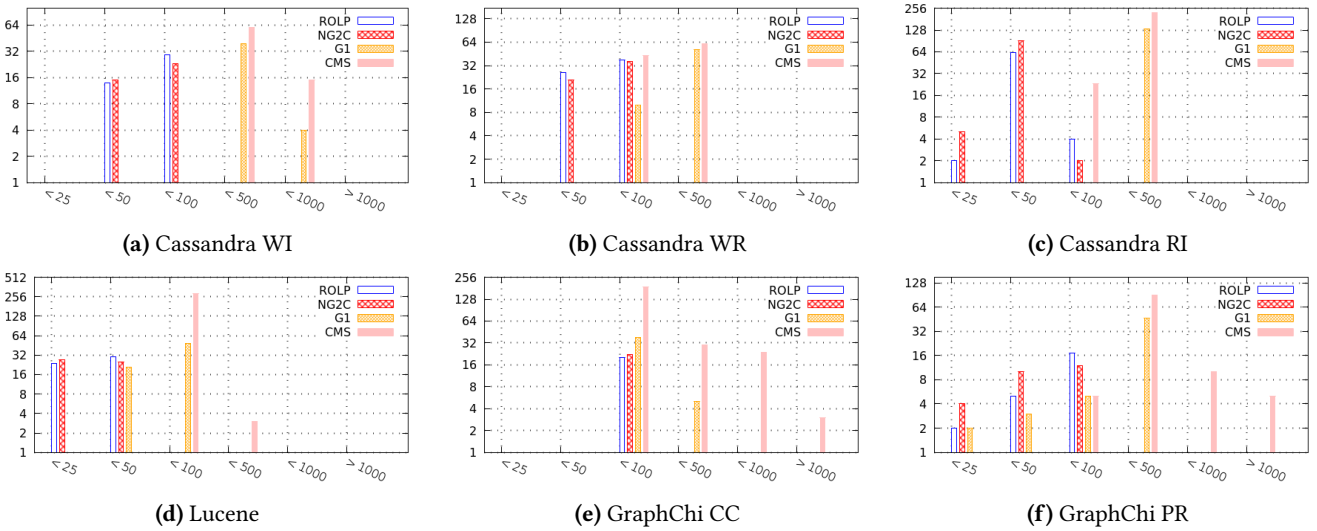
So far, the presented application pause times were organized by percentiles. Figure 9 presents the number of application pauses that occur in each pause time interval. Pauses



(a) Cassandra WI **(b)** Cassandra WR **(c)** Cassandra RI

(d) Lucene **(e)** GraphChi CC **(f)** GraphChi PR

Figure 8. Pause Time Percentiles (ms)



(a) Cassandra WI **(b)** Cassandra WR **(c)** Cassandra RI

(d) Lucene **(e)** GraphChi CC **(f)** GraphChi PR

Figure 9. Number of Application Pauses Per Duration Interval (ms)

with shorter durations appear in intervals to the left while longer pauses appear in intervals to the right. In other words, the less pauses to the right, the better. ROLP presents significant improvements regarding G1 and CMS, i.e., it results in less application pauses in longer intervals, across all workloads. When comparing ROLP with NG2C, both solutions present very similar pause time distribution.

In sum, ROLP allows NG2C to reduce application pauses by automatically pretenuring objects from allocation contexts that tend to allocate objects with longer lifetimes. When compared to G1 and CMS, ROLP can greatly reduce application pauses and object copying within the heap. Once again, we can say that when compared to NG2C, ROLP presents equivalent performance without requiring programmer effort and knowledge.

8.5 Warmup Pauses, Throughput, and Memory

This section shows results on application warmup pause times, throughput, and max memory usage. Note that application warmup happens when its workload changes and ROLP is still detecting (i.e., learning) the lifetime of objects. Clearly, such time interval should be the minimum possible. Thus, the goal of this section is to show: i) how the learning curve of ROLP affects pause times during warmup and how long does it take; ii) that ROLP does not inflict a significant throughput overhead due to its profiling code; and iii) that ROLP does not negatively impact the max memory usage.

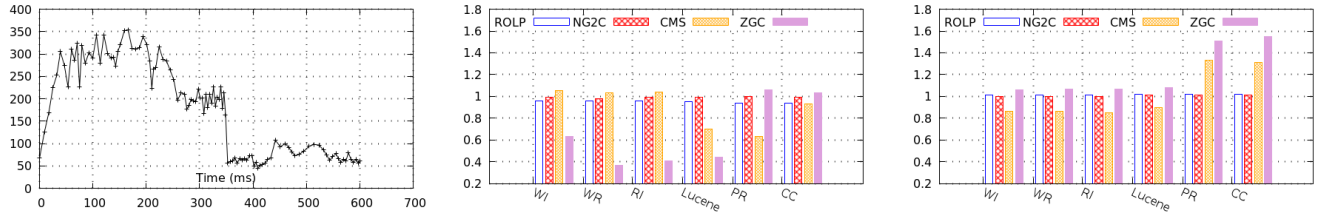


Figure 10. Cassandra WI Warmup Pause Times in ms (left) and Throughput (middle) and Max Mem. Usage norm. to G1 (right)

Figure 10, plot on the left, shows the Cassandra WI warmup pause times for the first 600 seconds of the workload. Pause times during the warmup phase can be divided into three parts. The first part spans from the beginning to around 250 seconds. During this part of the execution, no information is given to NG2C since ROLP is still gathering information regarding objects' lifetimes. Around second 250 (until second 350), ROLP already performed some lifetime estimations, and NG2C starts pretenuring application objects resulting in reduced pause times. Finally, the third part of the warmup starts around the second 350 when NG2C receives more information profiling information which is used to pretenure more objects. In short, ROLP takes about 350 seconds to stabilize the profiling information in Cassandra. In a real production environment, in which such workloads can run for days, 350 seconds represents a very small time to stabilize the system given its performance benefits.

With regards to throughput and max memory usage, Figure 10 (plots in the center and in the right) shows results normalized to G1 (i.e., G1 results represent 1 for all columns). ROLP presents a negligible throughput decrease, less than 5% (on average) for most workloads, compared to G1. Only for GraphChi workloads, ROLP presents an average throughput overhead of 6% for both PR and CC). We consider this a negligible throughput overhead considering the great reduction in application long tail latencies. Memory usage also shows a negligible overhead of ROLP compared to both G1 and NG2C. Nevertheless, we are currently working on integrating previously proposed sampling techniques [27] to further reduce the throughput overhead introduced by ROLP. As discussed in Section 2.1, ZGC (concurrent collector) trades fully concurrent collection by extreme throughput overhead and higher memory usage.

9 Related Work

Profiling plays a key role in managed runtimes, either for code optimization or memory management decisions [1, 2, 23, 24, 35, 40, 41]. We focus on getting quality profiling information to drive object pretenuring. ROLP is, to the best of our knowledge, the first online profiler targeting the dynamic pretenuring of objects in Big Data applications running on HotSpot JVM, a highly optimized production JVM. This section compares our work with state-of-art systems for object lifetime profiling (both offline and online) and Big Data-friendly memory management techniques.

9.1 Object Lifetime Profiling

Hertz et al. [24] introduced an algorithm where an object lifetime is tracked based on timestamps which introduces a 300 times slowdown compared to a non-profiled run. Ricci et al. [35] uses the same algorithm but adds new functionalities in terms of precision and comprehensiveness (weak references). Another system, Resurrector [40], relaxes precision to provide faster profiling but still introduces 3 to 40 times slowdown depending on the workload.

Blackburn et al. [6] extends the profile-based pretenuring of Cheng's solution [12] for Jikes RVM [1]. Blackburn et al. reports that it is particularly useful for tight heaps (at most 15 0MB) and not suitable for heaps with Gigabytes of objects.

Harris [23] proposes a dynamic profiling technique whose objective is to decide, at allocation time, to either pretenure the object being allocated or not. Compared to ROLP, this approach has one main limitation which is the fact that it targets heaps with only two generations. As shown in the previous section, more generations are necessary to effectively split objects by estimated lifetime.

Sewe et al. [36] presents a headroom schema which drives pretenuring based on the space left on the heap before garbage collection is necessary. Although their solution brings advantages to collection times, they push much of the overhead to the mutator and also to the off-line process, which is not always possible or accurate. Finally, Sewe et al.[36] do not target large heaps or a modern garbage collector like G1. Compared to previous offline solutions, ROLP does not require any source code modifications, no previous knowledge on the target workload, and it targets a widely employed industrial JVM.

However, in general, input influences the choices made during memory management [31] motivating the need for online profiling, which uncovers a number of new problems such as context tracking.

Ball and Laurus [4] compute a unique number for each possible path of a control flow graph inside a procedure. The computation is done offline and added to the source code. This is not suited for ROLP because modern workloads have many possible paths inside each routine, and the technique

can not capture the inter-procedure path needed for ROLP to distinguish allocation sites. Bond and McKinley [8] also compute a single value, but at runtime, to determine the sequence of active stack frames in a inter-procedural way. However, they need to maintain non-commutativity to differentiate call sequences. This is not a requirement for ROLP and so we can have a smaller impact on code instrumentation.

NightWatch [22] is an allocation context-aware memory allocator that tries to maximize cache locality. NightWatch and ROLP share the same idea of trying to capture the allocation context for profiling purposes (to detect cache locality in the case of NightWatch, and to measure lifetime in the case of ROLP). NightWatch is, however, optimized for large allocation chunks as it iterates through the call stack to generate an allocation context. In the case of ROLP, which targets object oriented applications, iterating the call stack for every object allocation is not feasible. Furthermore, runtimes (such as the OpenJDK HotSpot JVM) tend to manage their own heaps, meaning that there will be few but very large allocation calls to the OS with the resulting memory being used to contain objects with potentially very different lifetimes.

Memento [14] gathers online feedback regarding object lifetime by instrumenting allocation and installing *mementos* (allocation feedback) alongside objects. By using this information Memento starts pretenuring objects from particular allocation sites. Compared to ROLP, it has several drawbacks. First, it is only able to manage one tenured space, therefore applying a binary decision that will still potentially co-locate objects with possibly very different lifetimes, incurring in additional compaction effort. Second, Memento instruments all application code while it is still being interpreted. This has two disadvantages compared to ROLP: i) all the application code is being profiled, leading to a huge profiling overhead (in ROLP, we only profile hot code locations); ii) profiling stops when the code is JIT compiled, meaning that application behavior is only tracked while the application is starting and the code is not jitted. Third, Memento does not track allocation contexts (i.e., call graph), which we found to be important to properly profile complex platforms such as Cassandra.

9.2 Big Data-friendly Memory Management

We now describe systems that employ a less transparent approach by requiring modifications to the heap organization and/or the collaboration of the programmer to instrument the code.

The work by Nguyen et al. [33, 34] reduces the number of objects in the heap by separating data from control paths and putting data objects in an off-heap structure. This technique reduces the number of objects managed by the collector improving the application throughput and reducing latency. However, the programmer is responsible for identifying and instrumenting the data path in the application code. A similar approach is followed by Broom [21] where the heap is split into regions [38] explicitly created by the programmer (assumed to know which codebase creates related objects).

NG2C [11] extends G1 to support object pretenuring. However, it also needs programmer's help to identify the generation where a new object should be allocated. Cohen et al. [15] extends the operation of the Immix garbage collector in Jikes RVM [7] with a new programming interface between the application and the GC, in order to manage dominant data structures (i.e. a data structure holding most of the objects during the lifetime of the program) more efficiently.

10 Discussion and Conclusions

This work proposed ROLP, a runtime object lifetime profiler which tells the collector where to allocate objects in order to minimize fragmentation. ROLP is implemented[5] for the OpenJDK 8 HotSpot JVM and integrated with NG2C, a pretenuring collector based on G1 (the current default collector in HotSpot). Results show that ROLP can significantly reduce pause times inflicted by the GC with very low throughput overhead. These results confirm the hypothesis that object lifetimes can be inferred from allocation contexts for Big Data applications running on the JVM.

ROLP is provided as a launch time flag for the JVM and no user effort is required. ROLP also supports package-level filters to either profile or not profile parts of the application code. These filters can be used to reduce the profiling overhead on large applications. Compared to the code annotations required by NG2C, which requires programmers to guess the lifetime of objects and to understand the internals of the JVM, these package filters are a simpler alternative, requiring close to no knowledge about the application. Nevertheless, it is possible to combine NG2C (hand-placed code annotations), POLM2 (offline profiling), and ROLP (online profiling) as the three techniques use the same JVM and underlying collector.

ROLP can be easily ported to other runtimes. To do so, however, we envision two main challenges: i) if the object header is small (as it happens in many runtimes), profiling information might be stored elsewhere; ii) the collector might have to be modified to support pretenuring as it is not a common feature in most collectors. Future work directions include refining the profiling heuristics and continue research on low-overhead context hashing techniques.

Acknowledgments

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2019 (INESC-ID), FCT scholarships SFRH/BD/103745/2014, SFRH/BSAB/135197/2017, and project PTDC/EEI-COM/30644/2017.

[5]The source code publicly is available at github.com/rodrigo-bruno/rolp.

References

[1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2000. Adaptive Optimization in the JalapeÑO JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM, New York, NY, USA, 47–65. https://doi.org/10.1145/353171.353175

[2] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. 2005. A Survey of Adaptive Optimization in Virtual Machines. *Proc. IEEE* 93, 2 (Feb 2005), 449–466.

[3] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade. 2015. Performance Characterization of In-Memory Data Analytics on a Modern Cloud Server. In *2015 IEEE Fifth International Conference on Big Data and Cloud Computing*. 1–8. https://doi.org/10.1109/BDCloud.2015.37

[4] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29)*. IEEE Computer Society, Washington, DC, USA, 46–57. http://dl.acm.org/citation.cfm?id=243846.243857

[5] David A. Barrett and Benjamin G. Zorn. 1993. Using Lifetime Predictors to Improve Memory Allocation Performance. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/155090.155108

[6] Stephen M. Blackburn, Matthew Hertz, Kathryn S. Mckinley, J. Eliot B. Moss, and Ting Yang. 2007. Profile-based Pretenuring. *ACM Trans. Program. Lang. Syst.* 29, 1, Article 2 (Jan. 2007). http://doi.acm.org/10.1145/1180475.1180477

[7] Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: A Mark-region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, 22–32. http://doi.acm.org/10.1145/1375581.1375586

[8] Michael D. Bond and Kathryn S. McKinley. 2007. Probabilistic Calling Context. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 97–112. https://doi.org/10.1145/1297027.1297035

[9] Rodrigo Bruno and Paulo Ferreira. 2017. POLM2: Automatic Profiling for Object Lifetime-aware Memory Management for Hotspot Big Data Applications. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware '17)*. ACM, New York, NY, USA, 147–160. https://doi.org/10.1145/3135974.3135986

[10] Rodrigo Bruno and Paulo Ferreira. 2018. A Study on Garbage Collection Algorithms for Big Data Environments. *ACM Comput. Surv.* 51, 1, Article 20 (Jan. 2018), 35 pages. https://doi.org/10.1145/3156818

[11] Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. 2017. NG2C: Pretenuring Garbage Collection with Dynamic Generations for HotSpot Big Data Applications. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM 2017)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/3092255.3092272

[12] Perry Cheng, Robert Harper, and Peter Lee. 1998. Generational Stack Collection and Profile-driven Pretenuring. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. 162–173. http://doi.acm.org/10.1145/277650.277718

[13] R. Clapp, M. Dimitrov, K. Kumar, V. Viswanathan, and T. Willhalm. 2015. Quantifying the Performance Impact of Memory Latency and Bandwidth for Big Data Workloads. In *2015 IEEE International Symposium on Workload Characterization*. 213–224. https://doi.org/10.1109/IISWC.2015.32

[14] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM '15)*. ACM, New York, NY, USA, 105–117. https://doi.org/10.1145/2754169.2754181

[15] Nachshon Cohen and Erez Petrank. 2015. Data Structure Aware Garbage Collector. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM '15)*. ACM, New York, NY, USA, 28–40. https://doi.org/10.1145/2754169.2754176

[16] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1029873.1029879

[17] D. Dice, M.S. Moir, and W.N. Scherer. 2010. Quickly reacquirable locks. (Oct. 12 2010). https://www.google.ch/patents/US7814488 US Patent 7,814,488.

[18] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. 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)*. ACM, New York, NY, USA, Article 13, 9 pages. https://doi.org/10.1145/2972206.2972210

[19] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 2012. Assessing the Scalability of Garbage Collectors on Many Cores. *SIGOPS Oper. Syst. Rev.* 45, 3 (Jan. 2012), 15–19. https://doi.org/10.1145/2094091.2094096

[20] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 2013. A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 229–240. https://doi.org/10.1145/2451116.2451142

[21] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G. Murray, Steven Hand, and Michael Isard. 2015. Broom: Sweeping Out Garbage Collection from Big Data Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/gog

[22] Rentong Guo, Xiaofei Liao, Hai Jin, Jianhui Yue, and Guang Tan. 2015. NightWatch: Integrating Lightweight and Transparent Cache Pollution Control into Dynamic Memory Allocation Systems. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 307–318. http://dl.acm.org/citation.cfm?id=2813767.2813790

[23] Timothy L. Harris. 2000. Dynamic Adaptive Pre-tenuring. In *Proceedings of the 2nd International Symposium on Memory Management (ISMM '00)*. ACM, 127–136. http://doi.acm.org/10.1145/362422.362476

[24] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanović. 2006. Generating Object Lifetime Traces with Merlin. *ACM Trans. Program. Lang. Syst.* 28, 3 (May 2006), 476–516. http://doi.acm.org/10.1145/1133651.1133654

[25] Richard Jones, Antony Hosking, and Eliot Moss. 2016. *The garbage collection handbook: the art of automatic memory management*. CRC Press.

[26] Richard E. Jones and Chris Ryder. 2008. A Study of Java Object Demographics. In *Proceedings of the 7th International Symposium on Memory Management (ISMM '08)*. ACM, New York, NY, USA, 121–130. https://doi.org/10.1145/1375634.1375652

[27] Maria Jump, Stephen M. Blackburn, and Kathryn S. McKinley. 2004. Dynamic Object Sampling for Pretenuring. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. ACM, New York, NY, USA, 152–162. https://doi.org/10.1145/1029873.1029892

[28] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. ACM, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[29] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 31–46. http://dl.acm.org/citation.cfm?id=2387880.2387884

[30] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[31] Feng Mao, Eddy Z. Zhang, and Xipeng Shen. 2009. Influence of Program Inputs on the Selection of Garbage Collectors. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '09)*. ACM, 91–100. http://doi.acm.org/10.1145/1508293.1508307

[32] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. 2010. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0.* Manning Publications Co., Greenwich, CT, USA.

[33] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-performance Big-data-friendly Garbage Collector. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 349–365. http://dl.acm.org/citation.cfm?id=3026877.3026905

[34] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, 675–690. http://doi.acm.org/10.1145/2694344.2694345

[35] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2011. 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)*. 139–142. http://doi.acm.org/10.1145/2093157.2093178

[36] Andreas Sewe, Dingwen Yuan, Jan Sinschek, and Mira Mezini. 2010. Headroom-based Pretenuring: Dynamically Pretenuring Objects That Live "Long Enough". In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ '10)*. ACM, 29–38. http://doi.acm.org/10.1145/1852761.1852767

[37] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management (ISMM '11)*. ACM, New York, NY, USA, 79–88. https://doi.org/10.1145/1993478.1993491

[38] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Inf. Comput.* 132, 2 (Feb. 1997), 109–176. https://doi.org/10.1006/inco.1996.2613

[39] David Ungar. 1984. 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)*. ACM, New York, NY, USA, 157–167. https://doi.org/10.1145/800020.808261

[40] Guoqing Xu. 2013. Resurrector: A Tunable Object Lifetime Profiling Technique for Optimizing Real-world Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, 111–130. http://doi.acm.org/10.1145/2509136.2509512

[41] Yudi Zheng, Lubomír Bulej, and Walter Binder. 2015. Accurate Profiling in the Presence of Dynamic Compilation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 433–450. http://doi.acm.org/10.1145/2814270.2814281