Yak: A High-Performance Big-Data-Friendly Garbage Collector

Khanh Nguyen, Lu Fang, Guoqing Xu, and Brian Demsky; ***University of California, Irvine;***
Shan Lu, ***University of Chicago;*** **Sanazsadat Alamian,** ***University of California, Irvine;***
Onur Mutlu, ***ETH Zurich***

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/nguyen

This paper is included in the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

Open access to the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Yak: A High-Performance Big-Data-Friendly Garbage Collector

Khanh Nguyen[†] Lu Fang[†] Guoqing Xu[†] Brian Demsky[†]
Shan Lu[‡] Sanazsadat Alamian[†] Onur Mutlu[§]

University of California, Irvine[†] *University of Chicago*[‡] *ETH Zürich*[§]

Abstract

Most "Big Data" systems are written in managed languages, such as Java, C#, or Scala. These systems suffer from severe memory problems due to the massive volume of objects created to process input data. Allocating and deallocating a sea of data objects puts a severe strain on existing garbage collectors (GC), leading to high memory management overheads and reduced performance.

This paper describes the design and implementation of Yak, a "Big Data" friendly garbage collector that provides high throughput and low latency for *all* JVM-based languages. Yak divides the managed heap into a control space (CS) and a data space (DS), based on the observation that a typical data-intensive system has a clear distinction between a control path and a data path. Objects created in the control path are allocated in the CS and subject to regular tracing GC. The lifetimes of objects in the data path often align with *epochs* creating them. They are thus allocated in the DS and subject to region-based memory management. Our evaluation with three large systems shows very positive results.

1 Introduction

It is clear that Big Data analytics has become a key component of modern computing. Popular data processing frameworks such as Hadoop [4], Spark [67], Naiad [48], or Hyracks [12] are all developed in managed languages, such as Java, C#, or Scala, primarily because these languages 1) enable fast development cycles and 2) provide abundant library suites and community support.

However, managed languages come at a cost [36, 37, 39, 47, 51, 59, 60, 61, 62, 63]: memory management in Big Data systems is often prohibitively expensive. For example, garbage collection (GC) can account for close to 50% of the execution time of these systems [15, 23, 49, 50], severely damaging system performance. The problem becomes increasingly painful in latency-sensitive distributed cloud applications where long GC pause times on one node can make many/all other nodes wait, potentially delaying the processing of user requests for an unacceptably long time [43, 44].

Multiple factors contribute to slow GC execution. An obvious one is the massive volume of objects created by Big Data systems at run time. Recent techniques propose to move a large portion of these objects outside the managed heap [28, 50]. Such techniques can significantly reduce GC overhead, but inevitably substantially increase the burden on developers by requiring them to manage the non-garbage-collected memory, which negates much of the benefit of using managed languages.

A critical reason for slow GC execution is that object characteristics in Big Data systems do *not* match the heuristics employed by state-of-the-art GC algorithms. This issue could potentially be alleviated if we can design a more suitable GC algorithm for Big Data systems. Intelligently adapting the heuristics of GC to object characteristics of Big Data systems can enable efficient handling of the large volume of objects in Big Data systems without relinquishing the benefits of managed languages. This is a promising yet challenging approach that has not been explored in the past, and we explore it in this work.

1.1 Challenges and Opportunities

Two Paths, Two Hypotheses The key characteristics of heap objects in Big Data systems can be summarized as *two paths, two hypotheses*.

Evidence [15, 28, 50] shows that a typical data processing framework often has a clear logical distinction between a *control path* and a *data path*. As exemplified by Figure 1, the control path performs cluster management and scheduling, establishes communication channels between nodes, and interacts with users to parse queries and return results. The data path primarily consists of data manipulation functions that can be connected to form a data processing pipeline. Examples include data partitioners, built-in operations such as Join or Aggregate, and user-defined data functions such as Map or Reduce.

These two paths follow different heap usage patterns. On the one hand, the behavior of the control path is similar to that of conventional programs: it has a complicated logic, but it does not create many objects. Those created objects usually follow the *generational hypothesis*: most recently allocated objects are also most likely to become unreachable quickly; most objects have short life spans.

On the other hand, the data path, while simple in code logic, is the main source of object creation. And, objects created by it do *not* follow the generational hypothesis. Previous work [15] reports that more than 95% of the objects in Giraph [3] are created in supersteps that represent graph data with `Edge` and `Vertex` objects. The

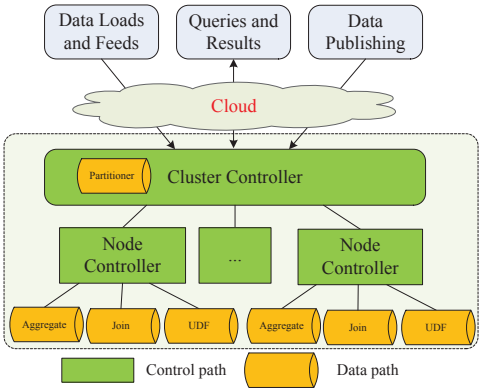


Figure 1: Graphical illustration of control and data paths.

execution of the data path often exhibits strong *epochal behavior* — each piece of data manipulation code is repeatedly executed. The execution of each epoch starts with allocating many objects to represent its input data and then manipulating them. These objects are often held in large arrays and stay alive throughout the epoch (*cf.* §3), which is often *not* a short period of time.

State-of-the-art GC State-of-the-art garbage collection algorithms, such as generational GC, collect the heap based on the *generational hypothesis*. The GC splits objects into a young and an old generation. Objects are initially allocated in the young generation. When a *nursery* GC runs, it identifies all young-generation objects that are reachable from the old generation, promotes them to the old generation, and then reclaims the entire young generation. Garbage collection for the old generation occurs infrequently. As long as the generational hypothesis holds, which is true for many large conventional applications that make heavy use of short-lived temporary data structures, generational GCs are efficient: a small number of objects *escape* to the old generation, and hence, most GC runs need to traverse only a small portion of the heap to identify and copy these escaping objects.

The Hypothesis Mismatch We find that, while the generational hypothesis holds for the control path of a data-intensive application, it does *not* match the epochal behavior of the data path, where *most* objects are created.

This mismatch leads to the fundamental challenge encountered by state-of-the-art GCs in data-intensive applications. Since newly-created objects often do *not* have short life spans, most GC runs spend significant time for identifying and moving young-generation objects into the old generation, while reclaiming little memory space. As an example, in GraphChi [41], a disk-based graph processing system, graph data in the *shard* defined by a vertex interval is first loaded into memory in each iteration, followed by the creation of many vertex objects to represent the data. These objects are *long-lived* and frequently visited to perform vertex updates. They cannot be reclaimed until the next vertex interval is processed. There can be dozens to hundreds of GC runs in each interval. Unfortunately, these runs end up moving *most* objects to the old generation and scanning almost the *entire* heap, while reclaiming *little* memory.

The epochal behavior of the data path also points to an opportunity not leveraged by existing GC algorithms – many data-path objects have the same life span and can be reclaimed together at the end of an epoch. We call this the *epochal hypothesis*. This hypothesis has been leveraged in *region-based memory management* [1, 8, 14, 25, 26, 28, 29, 30, 32, 40, 49, 50, 58], where objects created in an *epoch* are allocated in a memory region and efficiently deallocated as a whole when the epoch ends.

Unfortunately, existing region-based techniques need either sophisticated static analyses [1, 8, 14, 25, 26, 28, 29], which cannot scale to large systems, or heavy manual refactoring [28, 50], to guarantee that objects created in an epoch are indeed unreachable at the end of the epoch. Hence, such techniques have not been part of any garbage collector, to our knowledge.

1.2 Our Solution: The Yak GC

This paper presents Yak,[1] a high-throughput, low-latency GC tailored for managed Big Data systems. While GC has been extensively studied, existing research centers around the generational hypothesis, improving various aspects of the collection/application performance based on this hypothesis. Yak, in contrast, tailors the GC algorithm to the two very different types of object behavior (generational and epochal) observed in modern data-intensive workloads. Yak is the *first hybrid GC* that splits the heap into a control space (CS) and a data space (DS), which respectively employ generation-based and region-based algorithms to automatically manage memory.

Yak requires the developer to mark the beginning and end points of each epoch in the program. This is a simple task that even novices can do in minutes, and is already required by many Big Data infrastructures (*e.g.*, the `setup/cleanup` APIs in Hadoop [4]). Objects created inside each epoch are allocated in the DS, while those created outside are allocated in the CS. Since the number of objects to be traced in the CS is very small and only escaping objects in the DS need tracing, the memory management cost can be substantially reduced compared to a state-of-the-art generational GC.

While the idea appears simple, there are many challenges in developing a practical solution. First, we need to make the two styles of heap management for CS and DS smoothly co-exist inside one GC. For example, the generational collector that manages the CS in normal

[1] Yak is a wild ox that digests food with multiple stomachs.

ways should ignore some outgoing references to avoid getting in the way of DS management, and also keep track of incoming references to avoid deallocating CS objects referenced by DS objects (§5.4).

Second, we need to manage DS regions correctly. That is, we need to correctly handle the small number of objects that are allocated inside an epoch but escape to either other epochs or the control path. Naïvely deallocating the entire region for an epoch when the epoch ends can cause program failures. This is exactly the challenge encountered by past region-based memory management techniques.

Existing Big Data memory-management systems, such as Facade [50] and Broom [28], require developers to manually refactor both user and system programs to take control objects out of the data path, which, in turn, requires a deep understanding of the life spans of *all objects* created in the data path. This is a difficult task, which can take experienced developers weeks of effort or even longer. It essentially brings back the burden of manual memory management that managed languages freed developers from, imposing substantial practical limitations.

Yak offers an *automated and systematic solution*, requiring *zero* code refactoring. Yak allocates all objects created in an epoch in the DS, automatically tracks and identifies all escaping objects, and then uses a *promotion algorithm* to migrate escaping objects during region deallocation. This handling completely frees the developers from the stress of understanding object life spans, making Yak practical enough to be used in real settings (§5).

Third, we need to manage the DS region efficiently. This includes efficiently tracking escaping objects and migrating them. Naïvely monitoring every heap access to track escaping objects would lead to prohibitive overhead. Instead, we require light checking only before a heap write, but *not* on any heap read (§5.2). To guarantee memory correctness (*i.e.*, no live object deallocation), Yak also employs a lightweight "stop-the-world" treatment when a region is deallocated, without introducing significant stalls (§5.3).

1.3 Summary of Results

We implemented Yak inside Oracle's production JVM, OpenJDK 8. The JVM-based implementation enables Yak to work for all JVM-based languages, such as Java, Python, or Scala, while systems such as Facade [50] and Broom [28] work only for the specific languages they are designed for. We have evaluated Yak on three popular frameworks, *i.e.*, Hyracks [12], Hadoop [4], and GraphChi [41], with various types of applications and workloads. Our results show that Yak reduces GC latency by 1.4 – 44.3× and improves overall application performance by 12.5% – 7.2×, compared to the default Parallel Scavenge production GC in the JVM.

2 Related Work

Garbage Collection *Tracing garbage collectors* are the mainstream collectors in modern systems. A tracing GC performs allocation of new objects, identification of live objects, and reclamation of free memory. It traces live objects by following references, starting from a set of root objects that are directly reachable from live stack variables and global variables. It computes a *transitive closure* of live objects; objects that are unreachable during tracing are guaranteed to be dead and will be reclaimed.

There are four kinds of canonical tracing collectors: *mark-sweep*, *mark-region*, *semi-space*, and *mark-compact*. They all identify live objects the same way as discussed above. Their allocation and reclamation strategies differ significantly. Mark-sweep collectors allocate from a free list, mark live objects, and then put reclaimed memory back on the free list [24, 46]. Since a mark-sweep collector does not move live objects, it is time- and space-efficient, but it sacrifices locality for contemporaneously allocated objects. Mark-region collectors [7, 11, 13] reclaim contiguous free regions to provide contiguous allocation. Some mark-region collectors such as Immix [11] can also reduce fragmentation by mixing copying and marking. Semi-space [5, 6, 10, 17, 22, 34, 56] and mark-compact [19, 38, 55] collectors both move live objects. They put contemporaneously-allocated objects next to each other in a space, providing good locality.

These canonical algorithms serve as building blocks for more sophisticated algorithms such as the generational GC (*e.g.*, [56]), which divides the heap into a young and an old generation. Most GC runs are *nursery (minor) collections* that only scan references from the old to the young generation, move reachable objects into the old generation, and then free the entire young generation. When nursery GCs are not effective, a *full-heap (major) collection* scans both generations.

At first glance, Yak is similar to generational GC in that it promotes objects reachable after an epoch and then frees the entire epoch region. However, the regions in Yak have completely different and much richer semantics than the two generations in a generational GC. Consequently, Yak encounters completely different challenges and uses a design that is different from a generational GC. Specifically, in Yak, regions are thread-private; they reflect nested epochs; many regions could exist at any single moment. Therefore, to efficiently check which objects are escaping, we cannot rely on a traditional tracing algorithm; escaping objects may have multiple destination regions, instead of just the single old generation.

Connectivity-based garbage collection (CBGC) [33] is a family of algorithms that place objects into partitions by performing connectivity analyses on the object graph. A connectivity analysis can be based on types, allocations, or the partitioning introduced by Harris [31]. Garbage

First (G1) [22] is a generational algorithm that divides the heap into many small regions and gives higher collection priority to regions with more garbage. While CBGC, G1, and Yak each uses a notion of *region*, each has completely different semantics for the region and hence a different design. For example, objects inside a G1 region are *not* expected to have lifespans that are similar to each other.

Region-based Memory Management Region-based memory management was first used in the implementations of functional languages [1, 58] such as Standard ML [30], and then was extended to Prolog [45], C [25, 26, 29, 32], Java [18, 54], as well as real-time Java [8, 14, 40]. Existing region-based techniques rely heavily on *static analyses*. Unfortunately, these analyses either examine the whole program to identify region-allocable objects, which cannot scale to Big Data systems that all have large codebases, or require developers to use a brand new programming model, such as region types [8, 14]. In contrast, Yak is a pure dynamic technique that easily scales to large systems and requires only straightforward marking of epochs from users.

Big Data Memory Optimizations A variety of data computation models and processing systems have been developed in the past decade [4, 12, 16, 20, 21, 35, 52, 53, 57, 64, 65, 66, 67]. All of these frameworks were developed in managed languages and can benefit immediately from Yak, as demonstrated in our evaluation (*cf.* §6).

Bu et al. studied several data processing systems [15] and showed that a "bloat-free" design (*i.e.*, no objects allowed in data processing units), which is unfortunately impractical in modern Big Data systems, can make the system orders of magnitude more scalable.

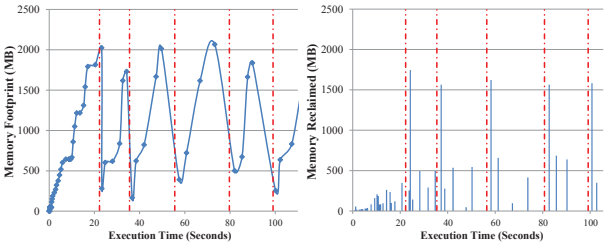
This insight has inspired recent work, like Facade [50], Broom [28], lifetime-based memory management [42], as well as Yak. Facade allocates data items into native memory pages that are deallocated in batch. Broom aims to replace the GC system by using regions with different scopes to manipulate objects with similar lifetimes. While promising, they both require extensive programmer intervention, as they move most objects out of the managed heap. For example, users must annotate the code and determine "data classes" and "boundary classes" to use Facade or explicitly use Broom APIs to allocate objects in regions. Yak is designed to free developers from the burden of understanding object lifetimes to use regions, making region-based memory management part of the managed runtime.

NumaGiC [27] is a new GC for "Big Data" on NUMA machines. It considers data location when performing (de-)allocation. However, as a generational GC, NumaGiC shares with modern GCs the problems discussed in §1.

Another orthogonal line of research on reducing GC pauses is building a holistic runtime for distributed Big Data systems [43, 44]. The runtime collectively manages the heap on different nodes, coordinating GC pauses to make them occur at times that are convenient for applications. Different from these techniques, Yak focuses on improving per-node memory management efficiency.

3 Motivation

We have conducted several experiments to validate our epochal hypothesis. Figure 2 depicts the memory footprint and its correlation with epochs when PageRank was executed on GraphChi to process a sample of the twitter-2010 graph (with 100M edges) on a server machine with 2 Intel(R) Xeon(R) CPU E5-2630 v2 processors running CentOS 6.6. We used the state-of-the-art Parallel Scavenge GC. In GraphChi, we defined an epoch as the processing of a sub-interval. While GraphChi uses multiple threads to perform vertex updates in each sub-interval, different sub-intervals are processed sequentially.



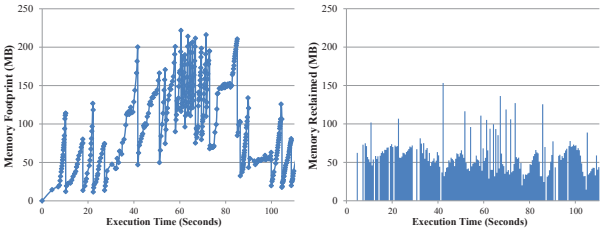
(a) Memory footprint (b) Memory reclaimed by each GC

Figure 2: Memory footprint for GraphChi [41] execution (GC consumes 73% of run time). Each dot in (a) represents the memory consumption measured right after a GC; each bar in (b) shows how much memory is reclaimed by a GC; dotted vertical lines show the epoch boundaries.

In the GraphChi experiment, GC takes 73% of run time. Each epoch lasts about 20 seconds, denoted by dotted lines in Figure 2. We can observe clear correlation between the end point of each epoch and each significant memory drop (Figure 2 (a)) as well as each large memory reclamation (Figure 2 (b)). During each epoch, many GC runs occur and each reclaims little memory (Figure 2 (b)).

For comparison, we also measured the memory usage of programs in the DaCapo benchmark suite [9], widely-used for evaluating JVM techniques. Figure 3 shows the memory footprint of Eclipse under large workloads provided by DaCapo. Eclipse is a popular development IDE and compiler frontend. It is an example of applications that have complex logic but process small amounts of data. GC performs well for Eclipse, taking only 2.4% of total execution time and reclaiming significant memory in each GC run. We do not observe epochal patterns in Figure 3. While other DaCapo benchmarks may exhibit some epochal behavior, epochs in these programs are often not clearly defined and finding them is not easy

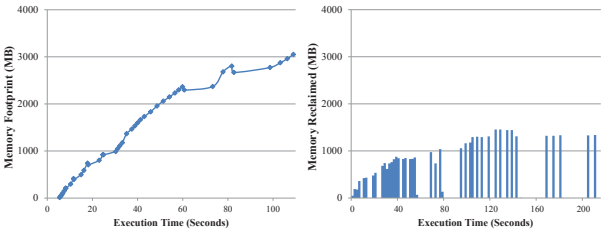
for *application developers* who are not familiar with the *system codebase*.



(a) Memory footprint (b) Memory reclaimed by each GC

Figure 3: Eclipse execution (GC takes 2.4% of time).

Strawman Can we solve the problem by forcing GC runs to happen *only* at the end of epochs? This simple approach would not work due to the multi-threaded nature of real systems. In systems like GraphChi, each epoch spawns many threads that collectively consume a huge amount of memory. Waiting until the end of an epoch to conduct GC could easily cause out-of-memory crashes. In systems like Hyracks [12], a distributed dataflow engine, different threads have various processing speeds and reach epoch ends at different times. Invoking the GC when one thread finishes an epoch would still make the GC traverse many live objects created by other threads, leading to wasted effort. This problem is illustrated in Figure 4, which shows memory footprint of one slave node when Hyracks performs word counting over a 14GB text dataset on an 11-node cluster. Each node was configured to run multiple Map and Reduce workers and have a 12GB heap. There are no epochal patterns in the figure, exactly because many worker threads execute in parallel and reach the end of an epoch at different times.



(a) Memory footprint (b) Memory reclaimed by each GC

Figure 4: Hyracks WordCount (GC takes 33.6% of time).

4 Design Overview

The overall idea of Yak is to split the heap into a conventional CS and a region-based DS, and use different mechanisms to manage them.

When to Create & Deallocate DS Regions? A region is created (deallocated) in the DS whenever an epoch starts (ends). This region holds *all* objects created inside the epoch. An epoch is the execution of a block of data transformation code. Note that the notion of an epoch is well-defined in Big Data systems. For example, in Hyracks [12], the body of a dataflow operator is enclosed by calls to `open` and `close`. Similarly, a user-defined (Map/Reduce) task in Hadoop [4] is enclosed by calls to `setup` and `cleanup`.

To enable a unified treatment across different Big Data systems, Yak expects a pair of user annotations, *epoch_start* and *epoch_end*. These annotations are translated into two native function calls at run time to inform the JVM of the start/end of an epoch. Placing these annotations requires negligible manual effort. Even a novice, without much knowledge about the system, can easily find and annotate epochs in a few minutes. Yak guarantees execution correctness regardless of where epoch annotations are placed. Of course, the locations of epoch boundaries do affect performance: if objects in a designated epoch have very different life spans, many of them need to be copied when the epoch ends, creating overhead.

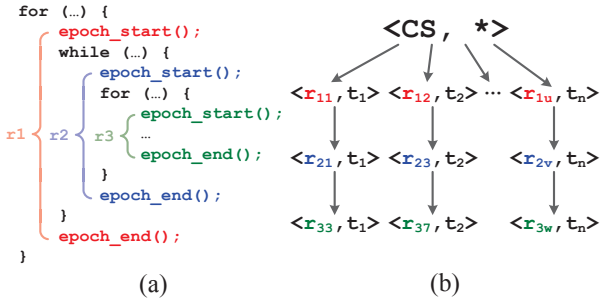
In practice, we need to consider a few more issues about the epoch concept. One is the *nested relationships* exhibited by epochs in real systems. A typical example is GraphChi [41], where a computational iteration naturally represents an epoch. Each iteration iteratively loads and processes all shards, and hence, the loading and processing of each memory shard (called *interval* in GraphChi) forms a *sub-epoch* inside the computational iteration. Since a shard is often too large to be loaded entirely into memory, GraphChi further breaks it into several *sub-intervals*, each of which forms a *sub-sub-epoch*.

Yak supports *nested regions* for performance benefits – unreachable objects inside an inner epoch can be reclaimed long before an outer epoch ends, preventing the memory footprint from aggressively growing. Specifically, if an *epoch_start* is encountered in the middle of an already-running epoch, a sub-epoch starts; subsequently a new region is created, and considered a child of the existing region. All subsequent object allocations take place in the child region until an *epoch_end* is seen. We do not place any restrictions on regions; objects in arbitrary regions are allowed to mutually reference one another.

The other issue is how to create regions when multiple threads execute the same piece of data-processing code concurrently. We could allow those threads to share one region. However, this would introduce complicated thread-synchronization problems; and might also delay memory recycling when multiple threads exit the epoch at different times, causing memory pressure. Yak creates one region for each dynamic instance of an epoch. When two threads execute the same piece of epoch code, they each get their own regions without having to worry about synchronization.

Overall, at any moment of execution, multiple epochs and hence regions could exist. They can be partially ordered based on their nesting relationships, forming a

semilattice structure. As shown in Figure 5, each node on the semilattice is a region of form $\langle r_{ij}, t_k \rangle$, where r_{ij} denotes the j-th execution of epoch r_i and t_k denotes the thread executing the epoch. For example, region $\langle r_{21}, t_1 \rangle$ is a child of $\langle r_{11}, t_1 \rangle$, because epoch r_2 is nested in epoch r_1 in the program and they are executed by the same thread t_1. Two regions (*e.g.*, $\langle r_{11}, t_1 \rangle$ and $\langle r_{12}, t_2 \rangle$) are *concurrent* if their epochs are executed by different threads.



(a) (b)

Figure 5: An example of regions: (a) a simple program and (b) its region semilattice at some point of execution.

How to Deallocate Regions Correctly and Efficiently? As discussed in §1, a small number of objects may outlive their epochs, and have to be identified and carefully handled during region deallocation. As also discussed in §1, we do not want to solve this problem by an iterative manual process of code refactoring and testing, which is labor-intensive as was done in Facade [50] or Broom [28]. Yak has to automatically accomplish two key tasks: (1) identifying escaping objects and (2) deciding the relocation destination for these objects.

For the first task, Yak uses an efficient algorithm to track cross-region/space references and records all *incoming references* at run time for each region. Right before a region is deallocated, Yak uses these references as the *root set* to compute a transitive closure of objects that can escape the region (details in §5.2).

For the second task, for each escaping object O, Yak tries to relocate O to a live region that will not be deallocated before the last (valid) reference to O. To achieve this goal, Yak identifies the source regions for each incoming cross-region/space reference to O, and *joins* them to find their *least upperbound* on the region semilattice. For example, in Figure 5, *joining* $\langle r_{21}, t_1 \rangle$ and $\langle r_{11}, t_1 \rangle$ returns $\langle r_{11}, t_1 \rangle$, while joining any two concurrent regions returns the CS. Intuitively, if O has references from its parent and grand-parent regions, O should be moved up to its grand-parent. If O has two references coming from regions created by different threads, it has to be moved to the CS.

Upon deallocation, computing a transitive closure of escaping objects while other threads are accessing them may result in an incomplete closure. In addition, moving objects concurrently with other running threads is dangerous and may give rise to data races. Yak employs a lightweight "stop-the-world" treatment to guarantee memory safety in deallocation. When a thread reaches an *epoch_end*, Yak pauses all running threads, scans their stacks, and computes a closure that includes all potential live objects in the deallocating region. These objects are moved to their respective target regions before all mutator threads are resumed.

5 Yak Design and Implementation

We have implemented Yak in Oracle's production JVM OpenJDK 8 (build 25.0-b70). In addition to implementing our own region-based technique, we have modified the two JIT compilers (C1 and Opto), the interpreter, the object/heap layout, and the Parallel Scavenge collector (to manage the CS). Below, we discuss how to split the heap and create regions (§5.1); how to track inter-region/space references, how to identify escaping objects, and how to determine where to move them (§5.2); how to deallocate regions correctly and efficiently (§5.3); and how to modify the Parallel Scavenge GC to collect the CS (§5.4).

5.1 Region & Object Allocation

Region Allocation When the JVM is launched, it asks the OS to reserve a block of virtual addresses based on the maximum heap size specified by the user (*i.e.*, -Xmx). Yak divides this address space into the CS and the DS, with the ratio between them specified by the user via JVM parameters. Yak initially asks the OS to commit a small amount of memory, which will grow if the initial space runs out. Once an *epoch_start* is encountered, Yak creates a region in the DS. A region contains a list of pages whose size can be specified by a JVM parameter.

Heap Layout Figure 6 illustrates the heap layout maintained by Yak. The CS is the same as the old Java heap maintained by a generational GC, except for the newly added *remember set*. The DS is much bigger, containing multiple regions, with each region holding a list of pages.

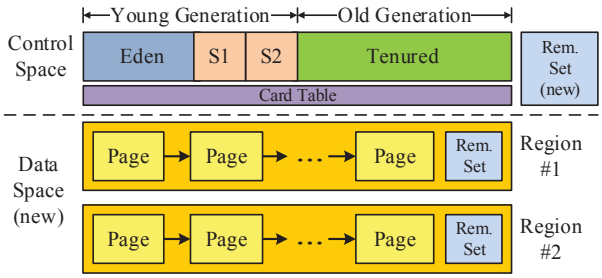


Figure 6: The heap layout in Yak.

The *remember set* is a bookkeeping data structure maintained by Yak for every region and the CS space. It is used to determine what objects escape a region r and where to relocate them. The remember set of CS helps identify live objects in the CS. The remember set of a region/s-

pace r is implemented as a hash table that maps an object O in r to all references to O that come from a different region/space.

Note that a remember set is one of the many possible data structures to record such references. For example, the generational GC uses a *card table* that groups objects into fixed-sized buckets and tracks which buckets contain objects with pointers that point to the young generation. Yak uses remember sets, because each region has only a few incoming references; using a card table instead would require us to scan *all objects* from the CS and other regions to find these references.

Allocating Objects in the DS When the execution is in an epoch, we redirect all allocation requests made to the Eden space (*e.g.*, young generation) to our new `Region_Alloc` function. Yak filters out JVM meta-data objects, such as class loader and class objects, from getting allocated in the region. Using a quick *bump pointer* algorithm (which uses a pointer that points to the starting address of free space and bumps it up upon each allocation), the region's manager attempts to allocate the object on the last page of its page list. If this page does not have enough space, the manager creates a new page and appends it to the list. For a large object that cannot fit into one page, we request a special page that can fit the object. For performance, large objects are never moved.

5.2 Tracking Inter-region References

Overview As discussed in §4, Yak needs to efficiently track all inter-region/space references. At a high level, Yak achieves this in three steps. First, Yak adds a 4-byte field *re* into the header space of each object to record the region information of the object. Upon an object allocation, its *re* field is updated to the corresponding region ID. A special ID is used for the CS.

Second, we modify the write barrier (*i.e.*, a piece of code executed with each heap write instruction $a.f = b$) to detect and record heap-based inter-region/space references. Note that, in OpenJDK, a barrier is already required by a generational GC to track inter-generation references. We modify the existing write barrier as shown in Algorithm 1.

Algorithm 1: The write barrier $a.f = b$.

Input: Expression $a.f$, Variable b

1 **if** ADDR$(O_a) \notin$ SPACE(CS) ***OR*** ADDR$(O_b) \notin$ SPACE(CS) **then**
2 **if** REGION$(O_a) \neq$ REGION(O_b) **then**
3 Record the reference ADDR(O_a) + OFFSET(f) $\xrightarrow{\text{REGION}(O_a)}$ ADDR(O_b) in the remember set *rs* of O_b's region
4 ... // Normal OpenJDK logic (for marking the card table)

Finally, Yak detects and records local-stack-based inter-region references as well as remote-stack-based references when *epoch_end* is triggered. These algorithms are shown in Lines 1 – 4 and Lines 5 – 10 in Algorithm 2.

Details We describe in detail how Yak can track all inter-region references, following the three places where the reference to an escaping object can reside in – the heap, the local stack, and a remote stack. The semantics of writes to static fields (*i.e.*, globals) as well as array stores are similar to that of instance field accesses; we omit the details of their handling. Copies of large memory regions (*e.g.*, `System.arraycopy`) are also tracked in Yak.

(1) In the heap. An object O_b can outlive its region r if its reference is written into an object O_a allocated in another (live) region r'. Algorithm 1 shows the write barrier to identify such escaping objects O_b. The algorithm checks whether the reference is an inter-region/space reference (Line 2). If it is, the pointee's region (*i.e.*, REGION(O_b)) needs to update its remember set (Line 3).

Each entry in the remember set is a reference which has a form $a \xrightarrow{r} b$ where a and b are the addresses of the pointer and pointee, respectively, and r represents the region the reference comes from. In most cases (such as those represented by Algorithm 1), r is the region in which a resides and it will be used to compute the target region to which b will be moved. However, if a is a stack variable, we need to create a placeholder reference with a special r, determined based on which stack a comes from. We will shortly discuss such cases in Algorithm 2.

To reduce overhead, we have a check that quickly filters out references that do not need to be remembered. As shown in Algorithm 1, if both O_a and O_b are in the same region, including the CS (Lines 1 – 2), we do not need to track that reference, and thus, the barrier proceeds to the normal OpenJDK logic.

(2) On the local stack. An object can escape by being referenced by a stack variable declared beyond the scope of the running epoch. Figure 7 (a) shows a simple example. The reference of the object allocated on Line 3 is assigned to the stack variable a. Because a is still alive after *epoch_end*, it is unsafe to deallocate the object.

Yak identifies this type of escaping objects through an analysis at each *epoch_end* mark. Specifically, Yak scans the local stack of the deallocating thread for the set of live variables at *epoch_end* and checks if an object in r can be referenced by a live variable (Lines 1 – 4 in Algorithm 2). For each such escaping object O_{var}, Yak adds a placeholder incoming reference, whose source is from r's parent region (say p), into the remember set *rs* of r (Line 4). This will cause O_{var} to be relocated to p. If the variable is still live when p is about to be deallocated, this would be detected by the same algorithm and O_{var} would be further relocated to p's parent.

```
1  a = ...;
2  //epoch_start
3  b = new B();
4  if (/*condition*/) {
5    a = b;
6  }
7  //epoch_end
8  c = a;
```

(a)

```
1  Thread t:
2  //epoch_start
3  a = A.f;
4  a.g = new O();
5  //epoch_end
6
7  Thread t':
8  //epoch_start
9  p = A.f;
10 b = p.g;
11 p.g = c;
12 //epoch_end
```

(b)

Figure 7: (a) An object referenced by b escapes its epoch via the stack variable a; (b) An object O created by thread t and referenced by $a.g$ escapes to thread t' via the load statement $b = p.g$.

(3) On the remote stack. A reference to an object O created by thread t could end up in a stack variable in thread t'. For example, in Figure 7 (b), object O created on Line 4 escapes t through the store at the same line and is loaded to the stack of another thread t' on Line 10. A naïve way to track these references is to monitor every read (*i.e.*, a *read barrier*), such as the load on Line 10 in Figure 7 (b).

Yak avoids the need for a read barrier, whose large overhead could affect practicality and performance. Before proceeding to discuss the solution, let us first examine the potential problems of missing a read barrier. The purpose of the read barrier is for us to understand whether a region object is loaded on a remote stack so that the object will not be mistakenly reclaimed when its containing region is deallocated. Without it, a remote thread which references an object O in region r, may cause two potential issues when r is deallocated (Figure 8).

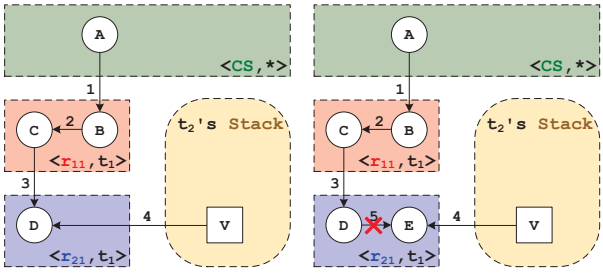


Figure 8: Examples showing potential problems with references on a remote stack: (a) moving object D is dangerous; and (b) object E, which is also live, is missed in the transitive closure.

Problem 1: Dangerous object moving. Figure 8 (a) illustrates this problem. Variable v on the stack of thread t_2 contains a reference to object D in region $\langle r_{21}, t_1 \rangle$ (by following the chain of references starting at object A in the CS). When this region is deallocated, D is in the escaping transitive closure; its target region, as determined by the semilattice, is its parent region $\langle r_{11}, t_1 \rangle$. Obviously, moving D at the deallocation of $\langle r_{21}, t_1 \rangle$ is dangerous, because we are not aware that v references it and thus cannot update v with D's new address after the move.

Problem 2: Dangerous object deallocation. Figure 8 (b) shows this problem. Object E is first referenced by D in the same region $\langle r_{21}, t_1 \rangle$. Hence, the remote thread t_2 can reach E by following the reference chain starting at A. Suppose t_2 loads E into a stack variable v and then deletes the reference from D to E. When region $\langle r_{21}, t_1 \rangle$ is deallocated, E cannot be included in the escaping transitive closure while it is being accessed by a remote stack. E thus becomes a "dangling" object that would be mistakenly treated as a dead object and reclaimed immediately.

Solution Summary Yak's solution to these problems is to pause all other threads and scan their stacks when thread t deallocates a region r. Objects in r that are also on a remote stack need to be explicitly marked as *escaping roots* before the escaping closure computation because they may be dangling objects (such as E in Figure 8 (b)) that are already disconnected from other objects in the region. §5.3 provides the detailed algorithms for region deallocation and thread stack scanning.

5.3 Region Deallocation

Algorithm 2 shows our region deallocation algorithm that is triggered at each *epoch_end*. This algorithm computes the closure of escaping objects, moves escaping objects to their target regions, and then recycles the whole region.

Algorithm 2: Region deallocation.

Input: Region r, Thread t

```
1  Map⟨Var, Object⟩ stackObjs ← ScanStack(t, r)
2  foreach ⟨var, O_var⟩ ∈ stackObjs do
3      if Region(O_var) = r then
4          Record a placeholder reference Addr(var)
               --r.parent--> Addr(O_var) in r's remember set rs
5  PauseOtherThreads()
6  foreach Thread t' ∈ Threads() : t' ≠ t do
7      Map⟨Var, Object⟩ remoteObjs ← ScanStack(t', r)
8      foreach ⟨var, O_var⟩ ∈ remoteObjs do
9          if Region(O_var) = r then
10             Record a placeholder reference Addr(var)
                   --CS--> Addr(O_var) in r's remember set rs
11 ClosureComputation()
12 ResumePausedThreads()
13 Put all pages of r back onto the available page list
```

Finding Escaping Roots There are three kinds of escaping roots for a region r. First, pointees of inter-

region/space references recorded in the remember set of r. Second, objects referenced by the local stack of the deallocating thread t. Third, objects referenced by the remote stacks of other threads.

Since inter-region/space references have already been captured by the write barrier (§5.2), here we first identify objects that escape the epoch via t's local stack, as shown in Lines 1 – 4 of Algorithm 2.

Next, Yak identifies objects that escape via remote stacks. To do this, Yak needs to synchronize threads (Line 5). When a remote thread t' is paused, Yak scans its stack variables and returns a set of objects that are referenced by these variables and located in region r. Each such (remotely referenced) object needs to be explicitly marked as an escaping root to be moved to the CS (Line 10) before the transitive closure is computed (Line 11).

No threads are resumed until t completes its closure computation and moves all escaping objects in r to their target regions. Note that it is unsafe to let a remote thread t' proceed even if the stack of t' does not reference any object in r. To illustrate, consider the following scenario. Suppose object A is in the CS and object B is in region r, and there is a reference from A to B. Only A but not B is on the stack of thread t' when r is deallocated. Scanning the stack of t' would not find any new escaping root for r. However, if t' is allowed to proceed immediately, t' could load B onto its stack through A and then delete the reference between A and B. If this occurs before t completes its closure computation, B would not be included in the closure although it is still live.

After all escaping objects are relocated, the entire region is deallocated with all its pages put back onto the free page list (Line 13).

Closure Computation Algorithm 3 shows the details of our closure computation from the set of escaping roots detected above. Since all other threads are paused, closure computation is done together with object moving. The closure is computed based on the remember set *rs* of the current deallocating region r. We first check the remember set *rs* (Line 1): if *rs* is empty, this region contains no escaping objects and hence is safe to be reclaimed. Otherwise, we need to identify all reachable objects and relocate them.

We start off by computing the target region to which each *escaping root* O_b needs to be promoted (Lines 2 – 4). We check each reference $addr \xrightarrow{r'} O_b$ in the remember set and then *join* all the regions r' based on the region semilattice. The results are saved in a map *promote*.

We then iterate through all escaping roots in topological order of their target regions (the loop at Line 5).[2] For each

[2]The order is based on the region semilattice. For example, CS is ordered before any DS region.

Algorithm 3: Closure computation.

Input: Remember Set *rs* of Region r

```
1  if The remember set rs of r is NOT empty then
2      foreach Escaping root O_b ∈ rs do
3          foreach Reference addr --r'--> ADDR(O_b) in rs do
4              promote[O_b] ← JOIN (r', promote[O_b])
5      foreach Escaping root O_b in topological order of
       promote[O_b] do
6          Region tgt ← promote[O_b]
7          Initialize queue gray with {O_b}
8          while gray is NOT empty do
9              Object O ← DEQUEUE(gray)
10             Write tgt into the region field of O
11             Object O* ←MOVE(O, tgt)
12             Put a forward reference at ADDR(O)
13             foreach Reference addr --x--> ADDR(O) in r's rs
               do
14                 Write ADDR(O*) into addr
15                 if x ≠ tgt then
16                     Add reference addr --x--> ADDR(O*)
                       into the remember set of region tgt
17             foreach Outgoing reference e of O* do
18                 Object O' ← TARGET(e)
19                 if O' is a forward reference then
20                     Write the new address into O*
21                 Region r' ← REGION(O')
22                 if r' = r then
23                     ENQUEUE(O', gray)
24                 else if r' ≠ tgt then
25                     Add reference ADDR(O*) --tgt-->
                       ADDR(O') into the remember set of
                       region r'
26 Clear the remember set rs of r
```

escaping root O_b, we perform a breadth-first traversal inside the current region to identify a closure of *transitively escaping* objects reachable from O_b and put all of them into a queue *gray*. During this traversal (Lines 8 – 23), we compute the regions to which each (transitively) escaping object should be moved and conduct the move. We will shortly discuss the details.

Identifying Target Regions When a transitively escaping object O' is reachable from only one escaping root O_b, we simply use the target region of O_b as the target of O'. When O' is reachable from multiple escaping roots, which may correspond to different target regions, we use the "highest-ranked" one among them as the target region of O'.

The topological order of our escaping root traversal is key to our implementation of the above idea. By com-

puting closure for a root with a "higher-ranked" region earlier, objects reachable from multiple roots need to be traversed only once – the check at Line 22 filters out those that already have a region r' ($\neq r$) assigned in a previous iteration of the loop because the region to be assigned in the current iteration is guaranteed to be "lower-ranked" than r'. When this case happens, the traversal stops further tracing the outgoing references from O'.

Figure 9 (a) shows a simple heap snapshot when region $\langle r_{21}, t_1 \rangle$ is about to be deallocated. There are two references in its remember set, one from region $\langle r_{11}, t_1 \rangle$ and a second from $\langle r_{12}, t_2 \rangle$. The objects C and D are the escaping roots. Initially, our algorithm determines that C will be moved to $\langle r_{11}, t_1 \rangle$ and D to the CS (because it is reachable from a concurrent region $\langle r_{12}, t_2 \rangle$). Since the CS is higher-ranked than $\langle r_{11}, t_1 \rangle$ in the semilattice, the transitive closure computation for D occurs before C, which sets E's target to the CS. Later, when the transitive closure for C is computed, E will be ignored (since it has been visited).

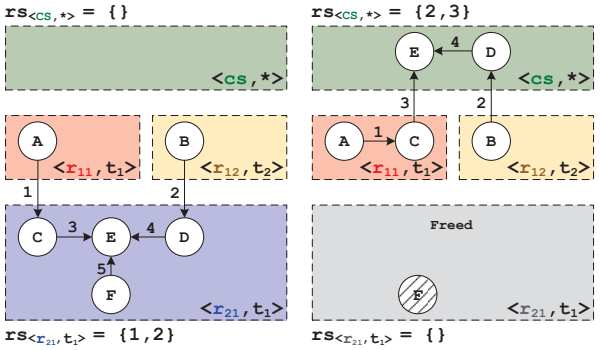


Figure 9: An example heap snapshot (a) before and (b) after the deallocation of region $\langle r_{21}, t_1 \rangle$.

Updating Remember Sets and Moving Objects Because we have pause all threads, object moving is safe (Line 11). When an object O is moved, we need to update *all* (stack and heap) locations that store its references. There can be three kinds of locations from which it is referenced: (1) intra-region locations (*i.e.*, referenced from another object in r); (2) objects from other regions or the CS; and (3) stack locations. We discuss how each of these types is handled by Algorithm 3.

(1) Intra-region locations. To handle intra-region references, we follow the standard GC treatment by putting a special *forward reference* at O's original location (Line 12). This will notify intra-region incoming references of the location change – when this old location of O is reached from another reference, the forward reference there will be used to update the source of that reference (Line 20).

(2) Objects from another region. References from these objects must have been recorded in r's remember set. Hence, we find all inter-region/space references of O in the remember set rs and update the source of each such reference with the new address O^* (Line 14). Since O^* now belongs to a new region tgt, the inter-region/space references that originally went into region r now go into region tgt. If the regions contain such a reference are not tgt, such references need to be explicitly added into the remember set of tgt (Line 16).

When O's outgoing edges are examined, moving O to region tgt may result in new inter-region/space references (Lines 24 – 25). For example, if the target region r' of a pointee object O' is not tgt (*i.e.*, O' has been visited from another escaping root), we need to add a new entry $\text{ADDR}(O^*) \xrightarrow{tgt} \text{ADDR}(O')$ into the remember set of r'.

(3) Stack locations. Since stack locations are also recorded as entries of the remember set, updating them is performed in the same way as updating heap locations. For example, when O is moved, Line 14 would update each reference going to O in the remember set. If O has (local or remote) stack references, they must be in the remember set and updated as well.

After the transitive closure computation and object promotion, the remember set rs of region r is cleared (Line 26).

Figure 9 (b) shows the heap after region $\langle r_{21}, t_1 \rangle$ is deallocated. The objects C, D, and E are escaping objects and will be moved to the target region computed. Since D and E belong to the CS, we add their incoming references 2 and 3 into the remember set of the CS. Object F does not escape the region, and hence, is automatically freed.

5.4 Collecting the CS

We implement two modifications to the Parallel Scavenge GC to collect the CS. First, we make the GC run locally in the CS. If the GC tracing reaches a reference to a region object, we simply ignore the reference.

Second, we include references in the CS' remember set into the tracing roots, so that corresponding CS objects would not be mistakenly reclaimed. Before tracing each such reference, we validate it by comparing the address of its target CS object with the current content in its source location. If they are different, this reference has become invalid and is discarded. Since the Parallel Scavenge GC moves objects (away from the young generation), Yak also needs to update references in the remember set of each region when their source in the CS is moved.

Yak also implements a number of optimizations on the remember set layout, large object allocation, as well as region/thread ID lookup. We omit the details of these optimizations for brevity.

6 Evaluation

This section presents an evaluation of Yak on widely-deployed real-world systems.

6.1 Methodology and Benchmarks

We have evaluated Yak on Hyracks [12], a parallel dataflow engine powering the Apache AsterixDB [2] stack, Hadoop [4], a popular distributed MapReduce [21] implementation, and GraphChi [41], a disk-based graph processing system. These three frameworks were selected due to their popularity and diverse characteristics. For example, Hyracks and Hadoop are distributed frameworks while GraphChi is a single-PC disk-based system. Hyracks runs one JVM on each node with many threads to process data while Hadoop runs multiple JVMs on each node, with each JVM using a small number of threads.

For each framework, we selected a few representative programs, forming a benchmark set with nine programs – external sort (ES), word count (WC), and distributed grep (DG) for Hyracks; in-map combiner (IC), top-word selector (TS), and distributed word filter (DF) for Hadoop; connected components (CC), community detection (CD), and page rank (PR) for GraphChi. Table 1 provides the descriptions of these programs.

FW	P	Description
Hyracks	ES	Sort a large array of data that cannot fit in main memory
	WC	Count word occurrences in a large document
	DG	Find matches based on user-defined regular expressions
Hadoop	IC	Count word frequencies in a corpus using local aggregation
	TS	Select a number of words with most frequent occurrences
	DF	Return text with user-defined words filtered out
GraphChi	PR	Compute page ranks (SpMV kernel)
	CC	Identify strongly connected components (label propagation)
	CD	Detect communities (label propagation)

Table 1: Our benchmarks and their descriptions.

Table 2 shows the datasets and heap configurations in our experiments. For Yak, the heap size is the sum of the sizes of both CS and DS. Since we fed different datasets to various frameworks, their memory requirements were also different. Evidence [11] shows that in general the heap size needs to be at least twice as large as the minimum memory size for the GC to perform well. We selected the heap configurations shown in Table 2 based on this observation – they are roughly $2\times$ – $3\times$ of the minimum heap size needed to run the original JVM.

FW	Dataset	Size	Heap Configs
Hyracks	Yahoo Webmap	72GB	20GB, 24GB
Hadoop	StackOverflow	37GB	2&1GB, 3&2GB
GraphChi	Sample twitter-2010	E = 100M V = 62M	6GB, 8GB

Table 2: Datasets and heap configurations used to run our programs; for Hadoop, the configurations *a*&*b* GB are the max heap sizes for each map (*a*) and reduce task (*b*).

In a small number of cases, the JVM uses hand-crafted assembly code to allocate objects directly into the heap without calling any C/C++ function. While we have spent more than a year on development, we have not yet performed any assembly-based optimizations for Yak. Thus, this assembly-based allocation in the JVM would allow some objects in an epoch to bypass Yak's allocator. To solve the problem, we had to disable this option and force all allocation requests to go through the main allocation entrance in C++. For a fair comparison, we kept the assembly-level allocation option disabled for all experiments including both Yak and original GC runs. We saw a small performance degradation (2–6%) after disabling this option in the JVM.

We ran Hyracks and Hadoop on an 11-node cluster, each with 2 Xeon(R) CPU E5-2640 v3 processors, 32GB memory, 1 SSD, running CentOS 6.6. We ran GraphChi on one node of this cluster, since it is a single-PC system. For Yak, we let the ratio between the sizes of the CS and the DS be 1/10. We did not find this ratio to have much impact on performance as long as the DS is large enough to contain objects created in each epoch. The page size in DS is 32KB by default. We performed experiments with different DS-page sizes and report these results shortly. We focus our comparison between Yak and Parallel Scavenge (PS) – the Oracle JVM's default production GC.

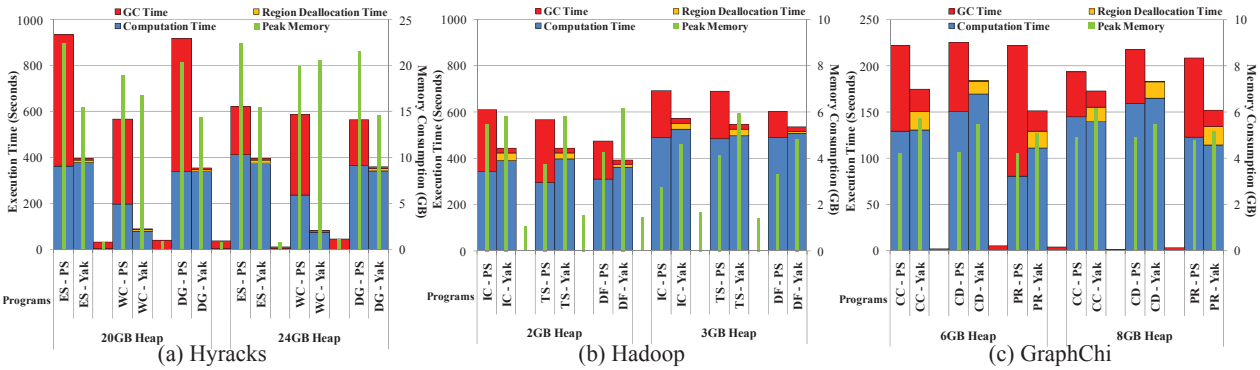
We ran each program for three iterations. The first iteration warmed up the JIT. The performance difference between the last two iterations were negligible (*e.g.*, less than 5%). This section reports the medians. We also confirmed that no incorrect results were produced by Yak.

6.2 Epoch Specification

We performed our annotation by strictly following *existing* framework APIs. For Hyracks, an epoch covers the lifetime of a (user-defined) dataflow operator (*i.e.*, via `open/close`); for Hadoop, it includes the body of a Map or Reduce task (*i.e.*, via `setup/cleanup`). For GraphChi, we let each epoch contain the body of a sub-interval specified by a `beginSubInterval` callback, since each sub-interval holds and processes many vertices and edges as illustrated in §3. A sub-interval creates many threads to load sliding shards and execute update functions. The body of each such thread is specified as a sub-epoch. It took us about ten minutes to annotate all three programs on each framework. Note that our optimization for these frameworks only scratches the surface; vast opportunities are possible if both user-defined operators and system's built-in operators are epoch-annotated.

6.3 Latency and Throughput

Figure 10 depicts the detailed performance comparisons between Yak and PS. Table 3 summarizes Yak's perfor-



(a) Hyracks (b) Hadoop (c) GraphChi

Figure 10: Performance comparisons on various programs; each group compares performance between Parallel Scavenge (PS) and Yak on a program with two "fat" and two "thin" bars. The left and right fat bars show the running times of PS and Yak, respectively, which is further broken down into three components: GC (in red), region deallocation (in orange), application computation (in blue) times. The left and right thin bars show maximum memory consumption of PS and Yak, collected by periodically running `pmap`.

FW	Overall	GC	App	Mem
Hyracks	0.14 ∼ 0.64 (0.40)	0.02 ∼ 0.11 (0.05)	0.31 ∼ 1.05 (0.77)	0.67 ∼ 1.03 (0.78)
Hadoop	0.73 ∼ 0.89 (0.81)	0.17 ∼ 0.26 (0.21)	1.03 ∼ 1.35 (1.13)	1.07 ∼ 1.67 (1.44)
GraphChi	0.70 ∼ 0.86 (0.77)	0.15 ∼ 0.56 (0.38)	0.91 ∼ 1.13 (1.01)	1.07 ∼ 1.34 (1.21)

Table 3: Summary of Yak performance normalized to baseline PS in terms of **Overall** run time, **GC** time, including Yak's region deallocation time, **App**lication (non-GC) time, and **Mem**ory consumption across all settings on each framework. The values shown depict Min ∼ Max and (Mean), and are normalized to PS. A lower value indicates better performance versus PS.

mance improvement by showing Overall run time, as well as GC and Application time and Memory consumption, all normalized to those of PS.

For Hyracks, Yak outperforms PS in all evaluated metrics. The GC time is collected by identifying the maximum GC time across runs on all slave nodes. Data-parallel tasks in Hyracks are isolated by design and they do not share any data structures across task instances. Hence, while Yak's write barrier incurs overhead, almost all references captured by the write barrier are intra-region references and thus they do not trigger the slow path of the barrier (*i.e.*, updating the remember set). Yak also improves the (non-GC) application performance — this is because PS only performs thread-local allocation for small objects and the allocation of large objects has to be in the shared heap, protected by locks. In Yak, however, all objects are allocated in thread-local regions and thus threads can allocate objects completely in parallel. Lock-free allocation is the major reason why Yak improves application performance because large objects (*e.g.*, arrays in HashMaps) are frequently allocated in such programs.

For Hadoop and GraphChi, while Yak substantially reduces the GC time and the overall execution time, it increases the application time and memory consumption. Longer application time is expected because (1) memory reclamation (*i.e.*, region deallocation) shifts from the GC to the application execution, with Yak, and (2) the write barrier is triggered to record a large number of references. For example, Hadoop has a state object (*i.e.*, `context`) in the control path that holds objects created in the data path, generating many *inter-space* references. In GraphChi, a number of large data structures are shared among different data-loading threads, leading to many *inter-region* references (*e.g.*, reported in Table 4). Recording all these references makes the barrier overhead stand out.

We envision two approaches that can effectively reduce the write barrier cost. First, existing GCs all have manually crafted/optimized assembly code to implement the write barrier. As mentioned earlier, we have not yet investigated assembly-based optimizations for Yak. We expect the barrier cost to be much lower when these optimizations are implemented. Second, adding extra annotations that define finer-grained epochs may provide further performance improvement. For example, if objects reachable from the state object can be created in the CS in Hadoop, the number of inter-space references can be significantly reduced. In this experiment, we did not perform any program restructuring, but we believe significant performance potential is possible with that: it is up to the developer to decide how much annotation effort she can afford to expend for how much extra performance gain she would like to achieve.

Yak greatly shortens the pauses caused by GC. When Yak is enabled, the maximum (deallocation or GC) pauses in Hyracks, Hadoop, and GraphChi are, respectively, 1.82, 0.55, and 0.72 second(s), while the longest GC pauses under PS are 35.74, 1.24, and 9.48 seconds, respectively.

As the heap size increases, there is a small performance improvement for PS due to fewer GC runs. The heap

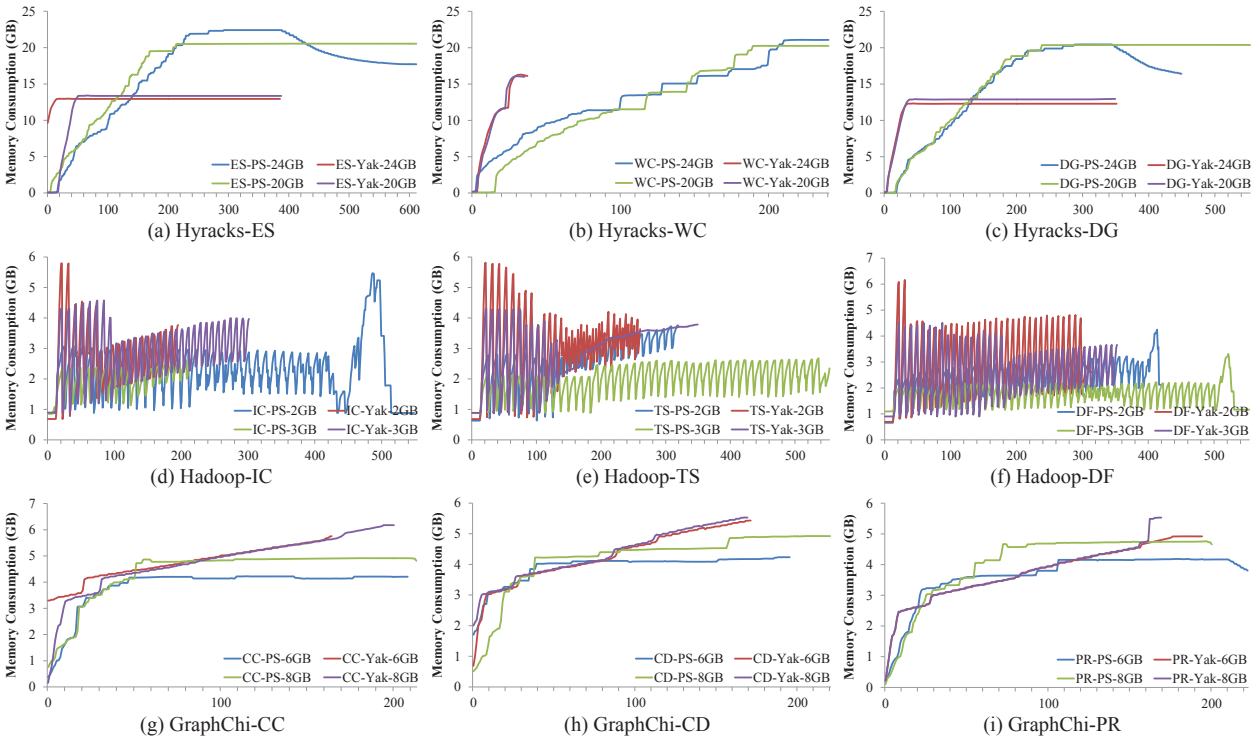


Figure 11: Memory footprints collected from `pmap`.

increase has little impact on Yak's overall performance, given that the CS is small anyways.

6.4 Memory Usage

We measured memory usage by periodically running `pmap` to understand the overall memory consumption of the Java process (for both the application and GC data). Figure 11 compares the memory footprints of Yak and PS under different heap configurations. For Hyracks and GraphChi, memory footprints are generally stable, while Hadoop's memory consumption fluctuates. This is because Hadoop runs multiple JVMs and different JVM instances are frequently created and destroyed. Since the JVM never returns claimed memory back to the OS until it terminates, the memory consumption always grows for Hyracks and GraphChi. The amount of memory consumed by Hadoop, however, drops frequently due to the frequent creation and termination of its JVM processes.

Note that the end times of Yak's memory traces on Hadoop in Figure 11 are earlier than the execution finish time reported in Figure 10. This is because Figure 11 shows the memory trace of the node that has the *highest memory consumption*; the computation on this node often finishes before the entire program finishes.

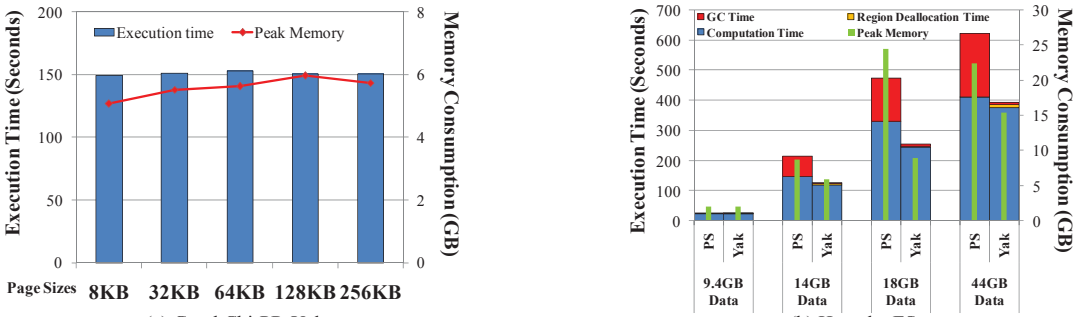
Yak constantly has lower memory consumption than PS for Hyracks. This is primarily because Yak can recycle memory *immediately* when a data processing thread finishes, while there is often a delay before the GC reclaims memory. For Hadoop and GraphChi, Yak has slightly higher memory consumption than PS. The main reason is that there are many control objects created in the data path and allocated in regions. Those objects often have shorter lifespans than their containing regions and, therefore, PS can reclaim them more efficiently than Yak.

Space Overhead To understand the overhead of the extra 4-byte field *re* in each object header, we ran the GraphChi programs with the unmodified HotSpot 1.8.0_74 and compared peak heap consumption with that of Yak (by periodically running `pmap`). We found that the difference (*i.e.*, the overhead) is relatively small. Across the three GraphChi benchmarks, this overhead varies from 1.1% to 20.8%, with an average of 12.2%.

6.5 Performance Breakdown

To provide a deeper understanding of Yak's performance, Table 4 reports various statistics on Yak's heap. Yak was built based on the assumption that in a typical Big Data system, only a small number of objects escape from the data path to the control path. This assumption has been validated by the fact that the ratios between numbers in **#CSR** and **#TR** are generally very small. As a result, each region has only very few objects (**%CSO**) that escape to the CS when the region is deallocated.

Figure 12 (a) depicts execution time and memory performance with Yak, when different page sizes are used. Execution time under different page sizes does not vary



(a) GraphChi-PR-Yak (b) Hyracks-ES

Figure 12: Performance comparisons between (a) different page sizes when Yak ran on GraphChi PR with a 6GB heap; (b) Yak and PS when datasets of various sizes were sorted by Hyracks ES on a 24GB heap.

Program	#CSR	#CRR	#TR	%CSO	#R
Hyracks-ES	2051	243	3B	0.0028%	103K
Hyracks-WC	2677	4221	213M	0.0043%	148K
Hyracks-DG	2013	16	2B	0.0034%	101K
Hadoop-IC	60K	0	2B	0%	598
Hadoop-TS	60K	0	2B	0%	598
Hadoop-DF	33K	0	1B	0%	598
GraphChi-CC	53K	25K	653M	0.044%	2699
GraphChi-CD	52K	14M	614M	1.3%	2699
GraphChi-PR	54K	24K	548M	0.060%	2699

Table 4: Statistics on Yak's heap: numbers of cross-space references (CSR), cross-region references (CRR), and total references generated by stores (TR); average percentage of objects escaping to the CS (CSO) among all objects in a region when the region retires; and total number of regions created during execution (R).

much (*e.g.*, all times are between 149 and 153 seconds), while the peak memory consumption generally goes up when page size increases (except for the 256KB case).

The write barrier and region deallocation are the two major sources of Yak's application overhead. As shown in Figure 10, region deallocation time accounts for 2.4%-13.1% of total execution time across the benchmarks. Since all of our programs are multi-threaded, it is difficult to pinpoint the exact contribution of the write barrier to execution time. To get an idea of the sensitivity to this barrier's cost, we manually modified GraphChi's execution engine to enforce a barrier between threads that load sliding shards and execute updates. This has the effect of serializing the threads and making the program sequential. For all three programs on GraphChi, we found that the mutator time (*i.e.*, non-pause time) increased by an overall of 24.5%. This shows that the write barrier is a major bottleneck, providing strong motivation for us to hand optimize it in assembly code in the near future.

Scalability To understand how Yak and PS perform when datasets of different sizes are processed, we ran Hyracks ES with four subsets of the Yahoo Webmap with sizes of 9.4GB, 14GB, 18GB, and 44GB respectively. Figure 12 (b) shows that Yak consistently outperforms PS and its performance improvement increases with the size of the dataset processed.

7 Conclusion

We present Yak, a new hybrid Garbage Collector (GC) that can efficiently manage memory in data-intensive applications. Yak treats the data space and control space differently for GC purposes since objects in modern data-processing frameworks follow two vastly-different types of lifetime behavior: data space shows epoch-based object lifetime patterns, whereas the much-smaller control space follows the classic generational lifetime behavior. Yak manages all data-space objects using epoch-based regions and deallocates each region as a whole at the end of an epoch, while efficiently tracking the small number of objects whose lifetimes span region boundaries. Doing so greatly reduces the overheads of traditional generational GC. Our experiments on several real-world applications demonstrate that Yak outperforms the default production GC in OpenJDK on three widely-used real Big Data systems, requiring almost zero user effort.

Acknowledgments

We would like to thank the many OSDI reviewers for their valuable and thorough comments. We are especially grateful to our shepherd Dushyanth Narayanan for his tireless effort to read many versions of the paper and provide suggestions, helping us improve the paper substantially. We thank Kathryn S. McKinley for her help with the preparation of the final version. We also appreciate the feedback from the MIT PDOS group (especially Tej Chajed for sending us the feedback).

This work is supported by NSF grants CCF-0846195, CCF-1217854, CNS-1228995, CCF-1319786, CNS-1321179, CCF-1409423, CCF-1409829, CCF-1439091, CCF-1514189, CNS-1514256, IIS-1546543, CNS-1613023, by ONR grants N00014-16-1-2149 and N00014-16-1-2913, and by a Sloan Fellowship.

References

[1] AIKEN, A., FÄHNDRICH, M., AND LEVIEN, R. Better static memory management: improving region-based analysis of higher-order languages. In *PLDI* (1995), pp. 174–185.

[2] ALSUBAIEE, S., ALTOWIM, Y., ALTWAIJRY, H., BEHM, A., BORKAR, V. R., BU, Y., CAREY, M. J., CETINDIL, I., CHEELANGI, M., FARAAZ, K., GABRIELOVA, E., GROVER, R., HEILBRON, Z., KIM, Y., LI, C., LI, G., OK, J. M., ONOSE, N., PIRZADEH, P., TSOTRAS, V. J., VERNICA, R., WEN, J., AND WESTMANN, T. AsterixDB: A scalable, open source BDMS. *Proc. VLDB Endow. 7*, 14 (2014), 1905–1916.

[3] Giraph: Open-source implementation of Pregel. http://incubator.apache.org/giraph/.

[4] Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org.

[5] APPEL, A. W. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper. 19*, 2 (1989), 171–183.

[6] BAKER, JR., H. G. List processing in real time on a serial computer. *Commun. ACM 21*, 4 (1978), 280–294.

[7] BEA SYSTEMS INC. Using the Jrockit runtime analyzer. http://edocs.bea.com/wljrockit/docs142/usingJRA/looking.html, 2007.

[8] BEEBEE, W. S., AND RINARD, M. C. An implementation of scoped memory for real-time Java. In *EMSOFT* (2001), pp. 289–305.

[9] BLACKBURN, S. M., GARNER, R., HOFFMAN, C., KHAN, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA* (2006), pp. 169–190.

[10] BLACKBURN, S. M., JONES, R., MCKINLEY, K. S., AND MOSS, J. E. B. Beltway: Getting around garbage collection gridlock. In *PLDI* (2002), pp. 153–164.

[11] BLACKBURN, S. M., AND MCKINLEY, K. S. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI* (2008), pp. 22–32.

[12] BORKAR, V. R., CAREY, M. J., GROVER, R., ONOSE, N., AND VERNICA, R. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE* (2011), pp. 1151–1162.

[13] BORMAN, S. Sensible sanitation understanding the IBM Java garbage collector. http://www.ibm.com/developerworks/ibm/library/i-garbage1/, 2002.

[14] BOYAPATI, C., SALCIANU, A., BEEBEE, JR., W., AND RINARD, M. Ownership types for safe region-based memory management in real-time Java. In *PLDI* (2003), pp. 324–337.

[15] BU, Y., BORKAR, V., XU, G., AND CAREY, M. J. A bloat-aware design for big data applications. In *ISMM* (2013), pp. 119–130.

[16] CHAIKEN, R., JENKINS, B., LARSON, P.-A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (2008), 1265–1276.

[17] CHENEY, C. J. A nonrecursive list compacting algorithm. *Commun. ACM 13*, 11 (1970), 677–678.

[18] CHEREM, S., AND RUGINA, R. Region analysis and transformation for Java programs. In *ISMM* (2004), pp. 85–96.

[19] COHEN, J., AND NICOLAU, A. Comparison of compacting algorithms for garbage collection. *ACM Trans. Program. Lang. Syst. 5*, 4 (1983), 532–553.

[20] CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., ELMELEEGY, K., AND SEARS, R. MapReduce online. In *NSDI* (2010), pp. 21–21.

[21] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004), pp. 137–150.

[22] DETLEFS, D., FLOOD, C., HELLER, S., AND PRINTEZIS, T. Garbage-first garbage collection. In *ISMM* (2004), pp. 37–48.

[23] FANG, L., NGUYEN, K., XU, G., DEMSKY, B., AND LU, S. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *SOSP* (2015), pp. 394–409.

[24] FENG, Y., AND BERGER, E. D. A locality-improving dynamic memory allocator. In *MSP* (2005), pp. 68–77.

[25] GAY, D., AND AIKEN, A. Memory management with explicit regions. In *PLDI* (1998), pp. 313–323.

[26] GAY, D., AND AIKEN, A. Language support for regions. In *PLDI* (2001), pp. 70–80.

[27] GIDRA, L., THOMAS, G., SOPENA, J., SHAPIRO, M., AND NGUYEN, N. NumaGiC: A garbage collector for big data on big NUMA machines. In *ASPLOS* (2015), pp. 661–673.

[28] GOG, I., GICEVA, J., SCHWARZKOPF, M., VASWANI, K., VYTINIOTIS, D., RAMALINGAM, G., COSTA, M., MURRAY, D. G., HAND, S., AND ISARD, M. Broom: Sweeping out garbage collection from big data systems. In *HotOS* (2015).

[29] GROSSMAN, D., MORRISETT, G., JIM, T., HICKS, M., WANG, Y., AND CHENEY, J. Region-based memory management in Cyclone. In *PLDI* (2002), pp. 282–293.

[30] HALLENBERG, N., ELSMAN, M., AND TOFTE, M. Combining region inference and garbage collection. In *PLDI* (2002), pp. 141–152.

[31] HARRIS, T. Early storage reclamation in a tracing garbage collector. *SIGPLAN Not. 34*, 4 (Apr. 1999), 46–53.

[32] HICKS, M., MORRISETT, G., GROSSMAN, D., AND JIM, T. Experience with safe manual memory-management in Cyclone. In *ISMM* (2004), pp. 73–84.

[33] HIRZEL, M., DIWAN, A., AND HERTZ, M. Connectivity-based garbage collection. In *OOPSLA* (2003), pp. 359–373.

[34] HUDSON, R. L., AND MOSS, J. E. B. Incremental collection of mature objects. In *IWMM* (1992), pp. 388–403.

[35] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007), pp. 59–72.

[36] JOAO, J. A., MUTLU, O., KIM, H., AGARWAL, R., AND PATT, Y. N. Improving the performance of object-oriented languages with dynamic predication of indirect jumps. In *ASPLOS* (2008), pp. 80–90.

[37] JOAO, J. A., MUTLU, O., AND PATT, Y. N. Flexible reference counting-based hardware acceleration for garbage collection. In *ISCA* (2009), pp. 418–428.

[38] KERMANY, H., AND PETRANK, E. The Compressor: Concurrent, incremental, and parallel compaction. In *PLDI* (2006), pp. 354–363.

[39] KIM, H., JOAO, J. A., MUTLU, O., LEE, C. J., PATT, Y. N., AND COHN, R. VPC prediction: Reducing the cost of indirect branches via hardware-based dynamic devirtualization. In *ISCA* (2007), pp. 424–435.

[40] KOWSHIK, S., DHURJATI, D., AND ADVE, V. Ensuring code safety without runtime checks for real-time control systems. In *CASES* (2002), pp. 288–297.

[41] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI* (2012), pp. 31–46.

[42] LU, L., SHI, X., ZHOU, Y., ZHANG, X., JIN, H., PEI, C., HE, L., AND GENG, Y. Lifetime-based memory management for distributed data processing systems. *Proc. VLDB Endow. 9*, 12 (2016), 936–947.

[43] MAAS, M., HARRIS, T., ASANOVIĆ, K., AND KUBIATOWICZ, J. Trash Day: Coordinating garbage collection in distributed systems. In *HotOS* (2015).

[44] MAAS, M., HARRIS, T., ASANOVIĆ, K., AND KUBIATOWICZ, J. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *ASPLOS* (2016), pp. 457–471.

[45] MAKHOLM, H. A region-based memory manager for Prolog. In *ISMM* (2000), pp. 25–34.

[46] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM 3*, 4 (Apr. 1960), 184–195.

[47] MITCHELL, N., AND SEVITSKY, G. The causes of bloat, the limits of health. In *OOPSLA* (2007), pp. 245–260.

[48] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *SOSP* (2013), pp. 439–455.

[49] NGUYEN, K., FANG, L., XU, G., AND DEMSKY, B. Speculative region-based memory management for big data systems. In *PLOS* (2015), pp. 27–32.

[50] NGUYEN, K., WANG, K., BU, Y., FANG, L., HU, J., AND XU, G. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS* (2015), pp. 675–690.

[51] NGUYEN, K., AND XU, G. Cachetor: detecting cacheable data to remove bloat. In *FSE* (2013), pp. 268–278.

[52] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD* (2008), pp. 1099–1110.

[53] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program. 13*, 4 (2005), 277–298.

[54] QIAN, F., AND HENDREN, L. An adaptive, region-based allocator for Java. In *ISMM* (2002), pp. 127–138.

[55] SACHINDRAN, N., MOSS, J. E. B., AND BERGER, E. D. Mc^2: High-performance garbage collection for memory-constrained environments. In *OOPSLA* (2004), pp. 81–98.

[56] STEFANOVIĆ, D., MCKINLEY, K. S., AND MOSS, J. E. B. Age-based garbage collection. In *OOPSLA* (1999), pp. 370–381.

[57] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow. 2*, 2 (2009), 1626–1629.

[58] TOFTE, M., AND TALPIN, J.-P. Implementation of the typed call-by-value lamda-calculus using a stack of regions. In *POPL* (1994), pp. 188–201.

[59] XU, G. Finding reusable data structures. In *OOPSLA* (2012), pp. 1017–1034.

[60] XU, G., ARNOLD, M., MITCHELL, N., ROUNTEV, A., SCHONBERG, E., AND SEVITSKY, G. Finding low-utility data structures. In *PLDI* (2010), pp. 174–186.

[61] XU, G., ARNOLD, M., MITCHELL, N., ROUNTEV, A., AND SEVITSKY, G. Go with the flow: Profiling copies to find runtime bloat. In *PLDI* (2009), pp. 419–430.

[62] XU, G., MITCHELL, N., ARNOLD, M., ROUNTEV, A., AND SEVITSKY, G. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *FoSER* (2010), pp. 421–426.

[63] XU, G., AND ROUNTEV, A. Detecting inefficiently-used containers to avoid bloat. In *PLDI* (2010), pp. 160–173.

[64] YANG, H.-C., DASDAN, A., HSIAO, R.-L., AND PARKER, D. S. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD* (2007), pp. 1029–1040.

[65] YU, Y., GUNDA, P. K., AND ISARD, M. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *SOSP* (2009), pp. 247–260.

[66] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., AND CURREY, J. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI* (2008), pp. 1–14.

[67] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. HotCloud, p. 10.