Trash Day: Coordinating Garbage Collection in Distributed Systems

Martin Maas[⋆] [†] [*] Tim Harris[†] Krste Asanović[⋆] John Kubiatowicz[⋆]
[⋆] *University of California, Berkeley* [†] *Oracle Labs, Cambridge*

Abstract

Cloud systems such as Hadoop, Spark and Zookeeper are frequently written in Java or other garbage-collected languages. However, GC-induced pauses can have a significant impact on these workloads. Specifically, GC pauses can reduce throughput for batch workloads, and cause high tail-latencies for interactive applications.

In this paper, we show that distributed applications suffer from each node's language runtime system making GC-related decisions independently. We first demonstrate this problem on two widely-used systems (Apache Spark and Apache Cassandra). We then propose solving this problem using a *Holistic Runtime System*, a distributed language runtime that collectively manages runtime services across multiple nodes.

We present initial results to demonstrate that this *Holistic GC* approach is effective both in reducing the impact of GC pauses on a batch workload, and in improving GC-related tail-latencies in an interactive setting.

1 Introduction

Garbage-collected languages are the dominant choice for distributed applications in cloud data centers. Languages such as C#, Go, Java, JavaScript/node.js, PHP/Hack, Python, Ruby and Scala account for a large portion of code in this environment. Popular frameworks such as Hadoop, Spark and Zookeeper are written in these languages, cloud services such as Google AppEngine and Microsoft Azure target these languages directly, and companies such as Twitter [7] or Facebook [4, 5] build a significant portion of their software around them.

This reflects the general trend towards higher-level languages. The productivity and safety properties of these languages often outweigh performance disadvantages, particularly for young companies that cannot afford the large engineering workforce to write and maintain large native code bases. Furthermore, high-level languages often make it easier for developers to iterate more quickly from prototype to product.

Garbage collection (GC) underpins many of the advantages of high-level languages. GC helps productivity because it reduces the engineering effort of explicitly managing pointer ownership. It also eliminates a large class of bugs, increases safety and avoids many sources of memory leaks (the latter is very important in cloud settings where applications often run for a long time).

GC performs well for many single-machine workloads. However, as we show in Section 2, it is a double-edged sword in the cloud setting. In latency-critical applications such as web servers or databases, GC pauses can cause requests to take unacceptably long times (this is even true for minor GC pauses at the order of milliseconds, as sub-millisecond latency requirements are increasingly common). This is exacerbated in applications that are composed of hundreds of services, where the overall latency depends on the slowest component (as common in data center workloads [10]). GC also poses a problem for applications that distribute live data across nodes, since pauses can make a node's data unavailable.

In our work, we investigate the sources of GC-related problems in current data center applications. We first show how to alleviate these problems by coordinating GC pauses between different nodes, such that they occur at times that are convenient for the application. We then show how a Holistic Runtime System [15] can be used to achieve this in a general way using an approach we call *Holistic Garbage Collection* (Section 3). We finally present a work-in-progress Holistic Runtime System currently under development at UC Berkeley (Section 4).

2 GC in Distributed Applications

Data center applications written in high-level languages are typically deployed by running each process within its own, independent language runtime system (such as a

[*]Work was started while at Oracle Labs, Cambridge.

Java Virtual Machine or Common Language Runtime). Frameworks such as Hadoop or Spark hence run over multiple runtime systems on different nodes, communicating through libraries such as Apache Thrift [2].

A consequence of this approach is that each runtime system makes decisions independently, including over when to perform GC. In practice, this means that GC pauses occur on different nodes at different times based on when memory fills up and needs to be collected. Depending on the collector and workload, these pauses can range from milliseconds to multiple seconds.

In this section, we show how GC pauses cause problems in two representative real-world systems, Apache Spark [20] and Apache Cassandra [14]. We next demonstrate how even simple strategies can alleviate these problems. In Section 3, we then show how these strategies can be generalized to fit a wider range of systems.

We use the *commodity* OpenJDK Hotspot JVM (using the GC settings provided by each application). There are specialized systems – such as those used in real-time scenarios – that limit or even eliminate GC pauses by running GC concurrently with the application [18]. However, this usually comes at the cost of reduced overall performance or increased resource utilization (e.g., from barrier or trap handling). Furthermore, these specialized runtime systems still incur pauses if memory fills up faster than it can be collected. To our knowledge, none of these systems are widely used in cloud settings.

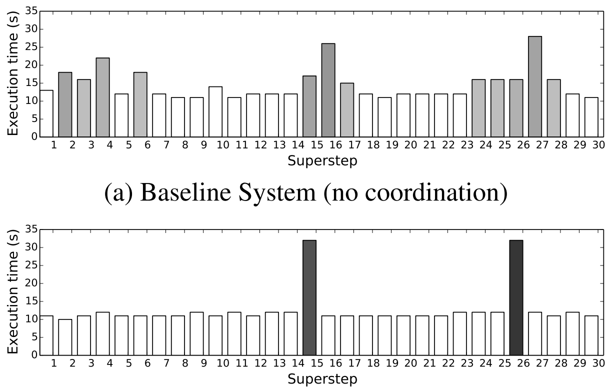
We perform all experiments on a cluster of 2-socket Xeon E5-2660 machines with 256GB RAM, connected through Infiniband. All our workloads run on dedicated nodes, but the network is shared with other jobs.

2.1 Case Study I: Apache Spark

Apache Spark is a distributed computation framework representative for a class of applications often called *batch workloads*. Spark jobs perform large-scale, long-running computations on distributed data sets and performance is measured in overall job execution time.

Problem. When running a job on a cluster, Spark spawns a worker process on each node. Each worker then performs a series of tasks on its local data. Occasionally, workers communicate data between nodes (for example, during *shuffle* operations). In those cases, no node can continue until data has been exchanged with every other node (equivalent to a cluster-wide barrier).

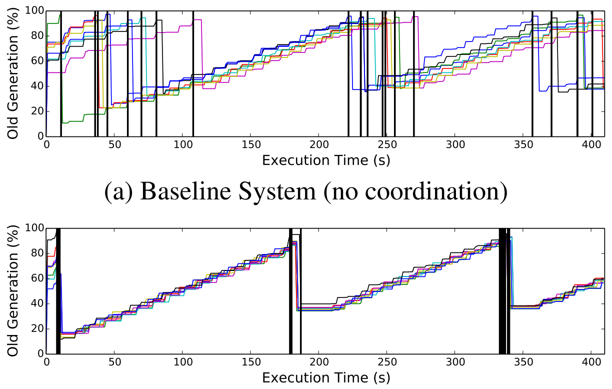
This synchronization causes problems in the presence of GC pauses: if even a single node is stuck in GC during such an operation, no other node can make progress, and therefore all nodes stall for a significant amount of time. Worse, once the stalled node finishes its GC, execution will continue and may quickly trigger a GC pause on a different node, as nodes are likely to not allocate any memory while stalling idly, and therefore only trigger GC while actually performing productive work.



(a) Baseline System (no coordination)

(b) Coordinating GC (stop-the-world everywhere)

Figure 1: *Impact of GC on the superstep durations of Spark PageRank (darker = more nodes performing GC during a superstep; white = no GC). This does not count minor collections, which occur much more frequently but have negligible impact.*



(a) Baseline System (no coordination)

(b) Coordinating GC (stop-the-world everywhere)

Figure 2: *Memory (old generation) occupancy and GC pauses of Spark PageRank. Each of the colors represents a node, and vertical lines indicate the start of a GC.*

To illustrate this problem, Figure 1a shows a PageRank computation using Spark on an 8-node cluster (the same workload as in the original Spark paper [20]). We show the time that each PageRank superstep takes, as well as how many nodes incur a GC pause during that superstep. We clearly see that while superstep times are homogeneous in the absence of GC, they increase significantly as soon as at least one node is performing a collection. Figure 2a shows the reason for this – memory occupancy increases independently on the different nodes, and once it reaches a threshold, a collection is performed, independently from the state of the other nodes.

Strategy. Instead of nodes collecting independently once their memory fills up, we want all nodes to perform GC at the same time – this way, none of the nodes waste time waiting for another node to finish collecting. This is reminsiscent of the use of gang-scheduling in multi-threaded synchronization-heady workloads. Figure 1b and Figure 2b show the effect of this strategy: we instrumented all the JVMs in our Spark cluster to track their occupancy, and as soon as any one node reaches an occupancy threshold (80% in this case), we triggered a GC on all nodes in the Spark cluster. Even on our small cluster, the PageRank computation completed 15% faster overall, without tuning or modifications to the JVM. This effect will become significantly more pronounced on larger cluster sizes, since this increases the likelihood of some node incurring a GC during each superstep. Typical Spark deployments can contain 100 nodes or more [20].

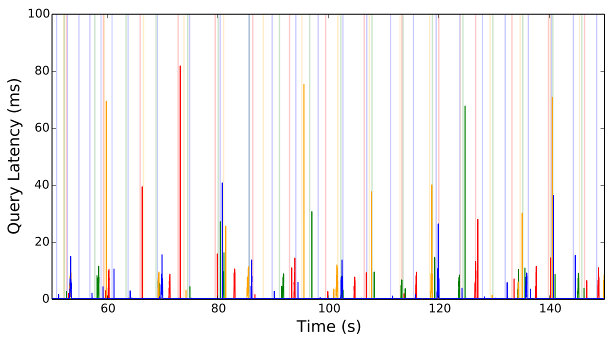
2.2 Case Study II: Apache Cassandra

Apache Cassandra is a distributed key-value store. It uses consistent hashing to replicate key-value pairs across multiple nodes. Requests can be sent to any node, which then acts as a coordinator and contacts all replicas of a particular key-value pair to assemble a read or write quorum. Cassandra is an example of an *interactive* workload: The most important metric is the latency of each query. Systems usually require that queries take a somewhat predictable time. Requests that take longer than, say, the 99.9 percentile of requests are called "stragglers" and cause problems with other services relying on data from the Cassandra cluster, e.g., to serve web requests.

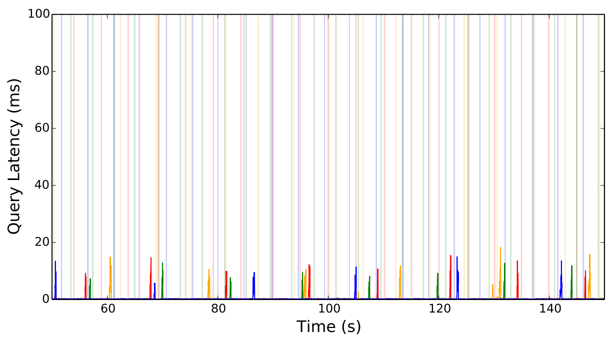
Cassandra uses a different collector than the Spark example from Section 2.1: pauses are only incurred for collecting the young generation that stores freshly allocated objects. A concurrent collector is used for the old generation (running alongside the application workload on other threads). While Spark incurs minor collections as well, they have little impact there – however, they do affect Cassandra due to its low-latency requirements.

Problem. GC is a key contributor to stragglers in many interactive systems [6, 8]. Figure 3a shows Cassandra update request latencies for a YCSB [9] workload averaged over 10ms intervals. While 99.9% of queries take less than 3.3ms, the remainder takes up to 25× longer. While GC is not the only cause, most of these long query latencies coincide with GC on at least one of the nodes.

The impact of GC is two-fold. While many stragglers stem from a GC pause in the coordinator responsible for a request, others result from GC pauses in nodes holding accessed replicas, making it impossible for the coordintor to assemble a quorum. Other (non-GC) sources of stalls include periodic tasks such as anti-entropy [1].



(a) Baseline System (no coordination)



(b) Steering requests away from nodes in GC

Figure 3: *Latencies of Cassandra update queries over time. Colors represent the nodes that act as the coordinator for each query, and the faded vertical lines in the background indicate GC pauses on those nodes.*

Strategy. Stragglers due to stalled coordinators can be avoided by anticipating GCs and redirecting requests to nodes that are not about to incur a GC pause. Even though minor GC pauses may only be a few ms in duration, modern data center networks enable such coordination between machines over much finer timescales.

We ran an experiment where we expose memory occupancy of all nodes in the cluster to the YCSB load generator and steer requests away from nodes that are about to collect, indicated by 80% occupancy of the young generation (Figure 4). This strategy is effective in eliminating many of the stragglers (Figure 3b). The 99.9 %ile update latency is improved from 3.3 ms to 1.6 ms, the worst case from 83 ms to 19 ms. A more general strategy could extend steering to periodic tasks, as well as ensuring that only one node within each quorum is collecting at a time (by staggering GCs, as we describe in Section 4).

3 A General Solution: Holistic GC

After identifying several strategies to coordinate GC in distributed systems (Section 2), our goal is to generalize these ideas to a wider range of applications. Today, applications implement ad-hoc solutions to this problem, such as reducing GC frequency using non-idiomatic Java

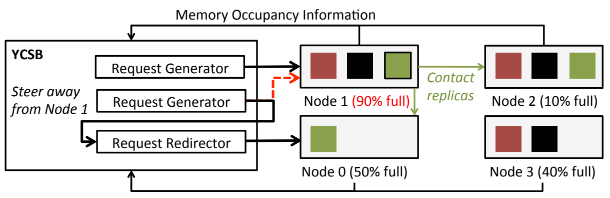


Figure 4: *Avoiding GC-induced stragglers in Cassandra by steering requests away from nodes during GC.*

(e.g., large `byte` arrays to store data structures), writing critical parts of Java applications in C (with explicit memory management), or treating old-generation GC as a failure mode and restarting the application. We believe that these strategies can be implemented in a better and more general way via a *Holistic Runtime System* [15].

A Holistic Runtime System is a language runtime system in which resource management policies span multiple nodes. Instead of making decisions about GC, JIT, etc. independently, the *per-node* runtime systems that underly a distributed application form a *distributed* system that allows the runtimes to make globally coordinated consensus decisions. We call the GC decisions made by such a system *Holistic GC*. Note that the heaps on the different nodes remain separate, and software still uses their same communication libraries for sharing data.

We briefly highlight the key components for enabling Holistic GC within a Holistic Runtime System:

(I) GC Policy. Single-node runtime systems provide different garbage collectors to match the requirements of different applications (e.g., throughput vs. pause times). The same can be true for Holistic GC. In the examples from Section 2, we saw two very different GC strategies: For Spark workloads, a good strategy is *Stop-the-world everywhere* (STWE), where all nodes stop for GC at the same time, while Cassandra requires the opposite, a *Staggered GC* (SGC) strategy that avoids bringing down too many replicas of an entry at the same time.

(II) Communication. It is often beneficial for the Holistic GC framework to communicate with the application-level framework. One example is exposing information about which nodes are about to perform GC, so that applications can select nodes that will remain available to fulfill a request (as with the Cassandra requests above). Similarly, we may want to handle maintenance tasks such as anti-entropy in addition to GC. The Holistic Runtime could call into the application-level framework to execute such tasks at suitable times.

Communication in the other direction is important as well. For example, an application can communicate how long it expects to be idle, so that the Holistic Runtime can make decisions to run an incremental GC pass [3].

(III) Reconfiguration. The correct strategy is not just dependent on the application, it can also depend on the application's program phase. For example, STWE may be correct for a shuffle phase in Spark, but not while Spark is accessing the file system (in which case it may be better to collect only on data nodes that are not currently accessed). Furthermore, the set of nodes that need to be coordinated may change depending on the operation (e.g., a distributed coordination operation may only involve a subset of the nodes). The system must be able to reconfigure itself to respond to these changes.

The above features should be implemented in a general and extensible way – there will be many application-specific special cases, and it must be possible to express them. At the same time, configuration in the general case should be easy – ideally, the developer should be able to choose from a set of prepared, configurable strategies, similar to how garbage collectors are configured today.

4 Towards a Prototype Holistic Runtime

We will now describe our vision for implementing these ideas in a Holistic Runtime System currently under development at UC Berkeley.

Our system is based on the OpenJDK Hotspot JVM as the per-node runtime. It is a drop-in replacement for Java; all that is required is to change the `PATH`, and Java programs will run under the Holistic Runtime System, transparently to the application. This approach lets us run a large number of unmodified workloads (including the Apache Hadoop ecosystem). We augment each per-node runtime with a management process that is logically part of the runtime system and connects to Hotspot through its management interface. This process can interact with Hotspot by (for example) reading out memory diagnostics or triggering GC. We will extend Hotspot as necessary to expose additional control features not supported through the current management interface.

The per-node runtimes automatically connect to each other and implement a consensus protocol that executes the *GC Policy (I)*. The policy is written in a DSL and is a function that considers the current *state* of the entire system (such as the memory occupancy on each node) and produces a *plan* describing what language events each node should perform next (Figure 5).

We now divide execution into *epochs* of varying lengths, during which the runtimes exchange their state and execute the policy to generate a plan that defines 1) the length of the next epoch, 2) on which nodes to perform language operations such as GC during the next epoch, 3) any necessary *reconfiguration (III)* and 4) updates to a set of key-value pairs stored on each node. The latter are a general mechanism for *communication be-*

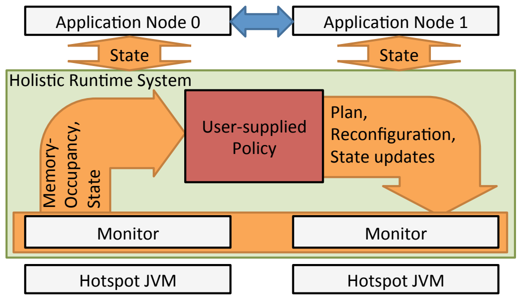


Figure 5: *Overview of the Holistic GC approach.*

tween Holistic Runtime and application (II), through an API to access the key-value pairs from user-level applications and frameworks.

Once the plan has been created, it is shared between all nodes and executed throughout the next epoch. The function describing the policy is user-defined (making it very flexible), but a library of standard strategies can be supplied for ease-of-use, and as a basis to compose them into more complex strategies. The system will allow users to write policies in a domain-specific language that can query state (such as memory occupancy or the key-value pairs), construct a plan (i.e., define language events to be triggered at a particular time on a specific set of nodes), and compose policies.

To make this possible, the framework abstracts away decentralized policy execution and time synchronization (we are planning to achieve this by using a failure-tolerant consensus protocol such as Raft [16]). We are also planning to integrate the Holistic Runtime with cluster managers such as Mesos [13], to feed information to the cluster manager that may help it to make scheduling decisions, and vice-versa. From a user perspective, all that needs to be implemented is a policy function that takes the states of all nodes and produces a plan.

Example Strategies. We anticipate that applications will require different strategies to coordinate GC between nodes. While these strategies will probably be based on small set of primitives (e.g., STWE), there will be application-specific aspects that require knowledge available to the programmer (e.g., about strategies that work best in particular phases of the application).

We therefore expect that applications will be shipped with a GC policy included. For example, the strategies identified for the two case studies from Section 2 could be described as follows (ignoring reconfiguration):

Algorithm 1 Spark (Section 2.1)

```
GC Policy (I)  STWE (in: nodes, out: plan)
  if nodes.filter(x: x.oldgen $\geq$ 0.8) $\neq$ {} then
    plan += MajorGC(nodes)
```

We assume a constant epoch for simplicity (e.g., 100ms), but both the epoch and the GC threshold (0.8) could be dynamically adapted to match the allocation rate.

Algorithm 2 Cassandra (Section 2.2)

```
GC Policy (I)  SGC (in: nodes, out: plan)
  static gc_group = 0
  for n $\in$ nodes do
    state[n_steeraway] = (n.newgen $>$ 0.8);
    if (n%Q == gc_group and n.newgen $>$ 0.8) then
      gc_nodes += n
  if len(gc_nodes) $\neq$ 0 then
    plan += MinorGC(gc_nodes)
  gc_group = (gc_group+1)%Q
Communication (II)  KEY-VALUE PAIRS
  bool n_steeraway: true tells application to avoid node
```

Here, the `gc_group` variable ensures that only one node in a quorum (where Q is the quorum size) can ever perform GC: Nodes in Cassandra are arranged in a ring, with replicas in successive nodes, so by ensuring that nodes that are Q apart from each other perform GC at the same time, we stagger GC in such a way that no two nodes in a quorum perform GC in the same time. Note that this assumes that memory does not fill up before a GC is triggered – this is ensured by steering requests away from nodes close to GC, reducing their allocation rates.

The application-level code that steers the load to the different nodes can access the "n_steeraway" key-value pairs to decide which nodes to send requests to.

Related Work. Several concurrent projects are also investigating the interaction between GC and distributed workloads. Broom [12] confirms the problems we showed in Spark for iterative Naiad workloads and proposes to eliminate GC altogether through region-based memory management. Other concurrent work confirms the impact of GC on interactive workloads [17], and proposes strategies [11, 19] similar to our steering approach. In contrast, we focus on supporting all these strategies in a general Holistic Runtime System that is simple to deploy and minimally invasive to existing applications.

5 Conclusion

We are currently working on a complete Holistic Runtime System, supporting a wide range of applications and allowing developers to easily implement whichever GC policies they like for their applications. In doing so, we want to provide fault tolerance and integrate into existing ecosystems such as Apache (e.g., supporting YARN). Our plan is to make this system openly available, for developers to implement and experiment with GC policies.

Acknowledgements

Research at UC Berkeley partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Huawei, LG, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

References

[1] "AntiEntropy (Apache Wiki Entry)." [Online]. Available: https://wiki.apache.org/cassandra/AntiEntropy

[2] "Apache Thrift." [Online]. Available: http://thrift.apache.org/

[3] "G1: One Garbage Collector To Rule Them All." [Online]. Available: http://www.infoq.com/articles/G1-One-Garbage-Collector-To-Rule-Them-All

[4] "Hack: A New Programming Language for HHVM." [Online]. Available: https://code.facebook.com/posts/264544830379293/hack-a-new-programming-language-for-hhvm/

[5] "On garbage collection." [Online]. Available: http://hhvm.com/blog/431/on-garbage-collection

[6] "Predictable Low Latency: "Cinnober on GC pause-free Java applications through orchestrated memory management"." [Online]. Available: http://www.cinnober.com/sites/cinnober.com/files/news/Cinnober%20on%20GC%20pause%20free%20Java%20applications.pdf

[7] "Twitter Shifting More Code to JVM, Citing Performance and Encapsulation As Primary Drivers." [Online]. Available: http://www.infoq.com/articles/twitter-java-use

[8] 29min, "Measuring SOLR Query Performance." [Online]. Available: https://29min.wordpress.com/2013/07/31/measuring-solr-query-performance/

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of 1st ACM Symposium on Cloud Computing*, 2010.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[11] H. Fan, A. Ramaraju, M. McKenzie, W. Golab, and B. Wong, "Understanding the Causes of Consistency Anomalies in Apache Cassandra," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, 2015.

[12] I. Gog, J. Giceva, M. Schwarzkopf, K. Viswani, D. Vytiniotis, G. Ramalingan, M. Costa, D. Murray, S. Hand, and M. Isard, "Broom: Sweeping out Garbage Collection from Big Data systems," in *Proceedings of the 15th USENIX/ACM Workshop on Hot Topics in Operating Systems*, 2015.

[13] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[14] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, Apr. 2010.

[15] M. Maas, K. Asanovic, T. Harris, and J. Kubiatowicz, "The Case for the Holistic Language Runtime System," in *First International Workshop on Rack-scale Computing*.

[16] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the 2014 USENIX Annual Technical Conference*, 2014.

[17] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making Sense of Performance in Data Analytics Frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation*, 2015.

[18] G. Tene, B. Iyengar, and M. Wolf, "C4: The Continuously Concurrent Compacting Collector," in *Proceedings of the International Symposium on Memory Management*, 2011.

[19] D. Terei and A. Levy, "Blade: A Data Center Garbage Collector," *ArXiv e-prints*, Apr. 2015.

[20] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.