Broom: sweeping out Garbage Collection from Big Data systems

Ionel Gog* Jana Giceva†* Malte Schwarzkopf
Kapil Vaswani‡ Dimitrios Vytiniotis‡ Ganesan Ramalingan‡ Manuel Costa‡
Derek Murray∅◇ Steven Hand∅◇ Michael Isard∅
University of Cambridge † *ETH Zurich* ‡ *Microsoft Research* ∅ *Unaffiliated*
* parts of the work were done while at Microsoft Research Silicon Valley ◇ now at Google, Inc.

Abstract

Many popular systems for processing "big data" are implemented in high-level programming languages with automatic memory management via garbage collection (GC). However, high object churn and large heap sizes put severe strain on the garbage collector. As a result, applications underperform significantly: GC increases the runtime of typical data processing tasks by up to 40%.

We propose to use region-based memory management instead of GC in distributed data processing systems. In these systems, many objects have clearly defined lifetimes. Hence, it is natural to allocate these objects in fate-sharing regions, obviating the need to scan a large heap. Regions can be memory-safe and could be inferred automatically. Our initial results show that region-based memory management reduces emulated Naiad vertex runtime by 34% for typical data analytics jobs.

1 Introduction

Memory-managed languages dominate the landscape of systems for computing with "big data": Hadoop, Spark [22], DryadLINQ [21] and Naiad [15] are only some examples of systems running atop a Java Virtual Machine (JVM) or the .NET common language runtime (CLR). Managed languages are attractive as they offer strong typing, automated memory management and higher-order functions. These features improve the productivity of system developers and end-users.

However, these benefits do not come for free. Data processing tasks stress the runtime GC by allocating a large number of objects [4, 17]. This results in long GC pauses that reduce application throughput or cause "straggler" tasks [14, 18]. In §2, we show that the impact of GC on job runtime can range between 20 and 40%.

In this paper, we argue that a different approach can be used in distributed data processing systems. Their operation is highly structured: most such systems are based on an implicit or explicit graph of stateful data-flow operators executed by worker threads. These operators perform event-based processing of arriving input data, and therefore behave as independent actors. For example, at any one time, MapReduce runs a "map" or "reduce" function (i.e., operator) in each task [5], Dryad [11] and Naiad [15] execute a "vertex" per worker thread, and Spark runs an "action" per task [22]. Each data-flow operator's objects live at most as long as the operator itself. Moreover, they are often grouped in logical batches – e.g., according to keys or timestamps – that can be freed atomically. This architecture presents an opportunity to revisit standard memory-management, because:

1. Actors explicitly share state via message-passing.
2. The state held by actors consists of many fate-sharing objects with common lifetimes.
3. End-users only supply code fragments to system-defined operators, which makes automatic program transformations and region annotations practical.

In §3, we illustrate these points with reference to Naiad.

Region-based memory management [19] works well for sets of related objects in the absence of implicit sharing. While writing programs using regions is difficult in the general case, this old concept is a good fit for the restricted domain of distributed data processing systems (§4). In addition, region-based allocation can offer memory safety and may be as transparent to the user as GC-based memory management. We sketch how this can be achieved in a distributed data processing system in §5.

Using Broom, a proof-of-concept implementation of region-based memory allocation for Naiad vertices, we show that region-based memory management eliminates the overheads of GC and improves execution time by up to 34% in memory-intensive operators (§6).

2 Motivation

We illustrate the effect of GC on Naiad's performance using two simple experiments:

1. We measure the fraction of job runtime spent in GC for two data-intensive batch jobs: TPC-H Q17 and a join-heavy business analytics workflow (§2.1).

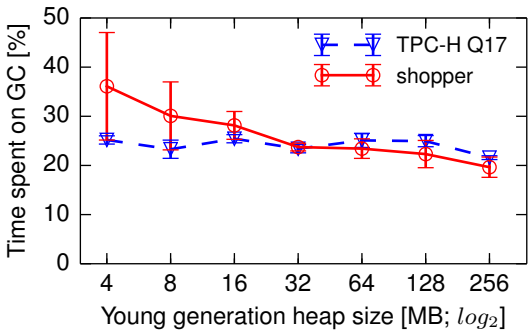


Figure 1: *The Naiad TPC-H Q17 and "shopper" workflows spend 20–40% of their total runtime on GC, independent of the young generation heap size.*

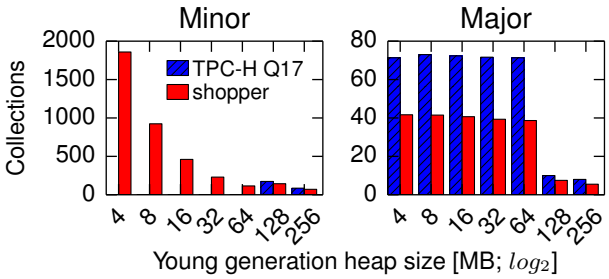


Figure 2: *Increasing the young generation heap trades more minor collections for fewer major collections in TPC-H, and reduces total collections for "shopper".*

2. We measure the effects of GC stalls when computing strongly connected components, an iterative workflow with frequent synchronization (§2.2).

We run Naiad v0.4 on Linux using Mono v2.10.8.1 with the generational GC (`sgen`) enabled.

2.1 Batch processing workflows

We run two typical batch processing workflows with high object churn on a single machine.[1] The first is query 17 from the TPC-H benchmark and the second is "shopper", a business intelligence workflow that selects users from the same country who bought a specific product (a `JOIN`–`SELECT`–`JOIN` workflow). In these experiments, Naiad uses eleven worker threads on the 12-core machine.

With the default GC configuration, we found that the TPC-H workflow spends around 25% of its runtime in GC, while the "shopper" workflow reaches about 37% (Figure 1). This makes sense: "shopper" generates many small objects that are subsequently freed in minor collections of the 4 MB young generation heap. Increasing the size of the young generation heap reduces the number of objects promoted to the next generation in "shopper", and thus the overall number of collections (Figure 2). This reduces the time spent on GC for "shopper"

[1] AMD Opteron 4243 (12× 3.1 GHz) with 64 GB of DDR3-1600.

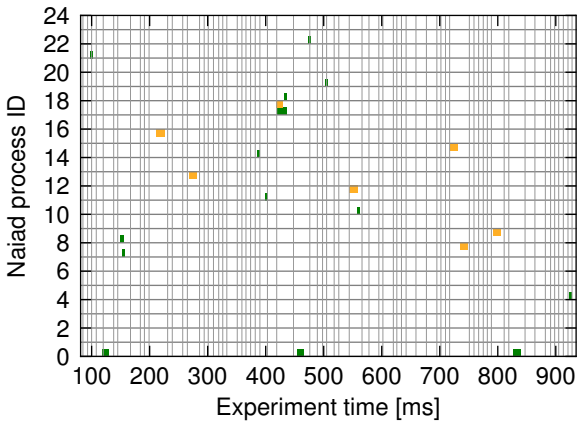


Figure 3: *Trace of incremental strongly connected components in 24 parallel Naiad processes: uncoordinated GC pauses (orange and green bars) delay synchronization barriers (gray vertical lines).*

(where many objects die young), but the increased young generation heap size does not help TPC-H Q17 (which uses stateful `JOIN`s). In fact, the number of minor collections in TPC-H Q17 *increases* with young generation heap size as they are traded for major ones (Figure 2).

This experiment looked at the GC behavior of a single data-intensive process that does not communicate. In the next experiment, we show that the problem is exacerbated when dependencies exist between processes.

2.2 Synchronized iterative workflow

We run an incremental strongly connected components workflow on a graph of 15M vertices and 80M edges. Each incremental step changes one random edge in the graph. The computation synchronizes all nodes after every graph change. We collect a trace of 24 Naiad processes on four machines (six per machine) by logging the timestamps of the synchronization steps and the times at which the processes run their GC.

Figure 3 shows a subset of the trace, with each gray vertical bar corresponding to a synchronization step at the end of an iteration. GC pauses are shown as horizontal bars for major (orange) and minor (green) collections. It is evident that GC invocations sometimes delay a synchronization by tens of milliseconds.

It is also worth observing that a GC in one process is occasionally immediately followed by another GC in a different process (e.g. at times 170–180, 430–460, 530–560 and 720–750). This occurs because some changes to the graph affect state in several processes. Downstream processes may trigger a GC as soon as they receive messages from upstream ones that have just finished their GC. In other words, the GCs are effectively serialized when a parallel execution would offer better performance.

3 Case study: Naiad

We illustrate the memory management patterns common in distributed data processing by example of Naiad. Listing 1 shows the code of the Naiad `Aggregate` vertex that groups and aggregates input data. The inputs to the vertex are batched in `Message` objects, each associated with a *logical timestamp*. The results of the aggregation are stored in a dictionary keyed by the logical timestamp (line 2). Upon receiving a message, Naiad calls the user-provided `OnReceive` method (ln. 4). The method processes the input and applies the `Aggregate` function to each entry (ln. 12), storing the results. Finally, the `OnNotify` method is called once the actor is guaranteed to receive no more messages with a logical timestamp less than or equal to the one passed (ln. 15). When `OnNotify` is called, all data held in the dictionary for the timestamp received by `OnNotify` can be freed.

Generational garbage collectors make the assumption that "most objects die young" [13]. This is true in general applications, but in Naiad most objects can only be collected after their logical timestamp is notified. Consequently, there is no point in a GC traversing these objects. Moreover, objects' lifetime depends on the events triggered by other processes (e.g., `OnReceive` calls).

While the details of this example are specific to Naiad, other systems exhibit similar behavior. For example, MapReduce and Spark, retain intermediate data for a key until all the records with this key have been seen. Likewise, stream processing systems (e.g., Storm, S4 [16], Spark Streaming [23]) have "windowed" operators that accumulate state for a fixed amount of time.

4 Broom: out with the GC!

General purpose garbage collectors are not tuned to specific application patterns. While specialized concurrent real-time garbage collectors eliminate GC pauses [1, 3], these collectors still have to scan a large heap. Instead, we propose a radically different approach.

Region-based memory management works well when similar objects with known lifetimes are handled. This information is available in distributed data processing systems. Many such systems are based on a model of actors communicating via message-passing. This significantly simplifies the use of regions: data sharing is explicit, and an object's lifetime does not exceed the allocating actor's unless the object is sent in a message. Moreover, only system developers write object management code: users merely supply limited user-defined functions that access objects with defined lifetimes.

Hence, only three types of regions are required for distributed data processing using communicating actors:

1. **Transferable regions** are used for messages. Their lifetime can extend over the lifetimes of multiple actors. However, only one actor (the current owner) can access the objects in a transferable region.
2. **Actor-scoped regions** are private to the owning actor. Their lifetime is equal to the actor's lifetime. They are used to store long-lived state.
3. **Temporary regions** are short-lived scratchpad memory blocks that are lexically scoped and cannot persist across method boundaries. They are useful when temporary data are generated and only a small subset of them is passed in messages or retained.

```
1  public class AggregateActor {
2    private Dictionary<Time, Dictionary<K, V>> state;
3
4    public void OnReceive(Time time, Message msg) {
5     if (state[time] == null) {
6       state[time] = new Dictionary<K,V>();
7       NotifyAt(time);
8     }
9     foreach (var entry in msg) {
10      var key = SelectKey(entry);
11      state[time][key] =
12        Aggregate(state[time][key], entry);
13    }
14   }
15   public void OnNotify(Time time) {
16     // Remove state for timestamp time
17     state.remove(time);
18     Send(outgoingMsg);
19   }
20 }
```

Listing 1: *Example of a Naiad aggregate actor.*

We have implemented these three types of regions in Broom, which extends the Bartok compiler [12, p. 75] with region-based memory management. In Listing 2, we demonstrate how Naiad's `Aggregate` actor implementation from Listing 1 is extended with regions.

In order to allocate within a region, a user must first get a handle to the region (e.g., ln. 7). Using the handle, an allocation context can be opened by calling `OpenAlloc()` (e.g., ln. 8). All objects created in an allocation context are stored in the corresponding region.[2]

The aggregate actor demonstrates the use of two region types: transferable and actor-scoped regions. In the actor scoped region (`actorRegion`, ln. 4), we keep two dictionaries (`regions` and `state`, ln. 2–3) that are alive for the entire duration of the computation. These dictionaries are indexed by logical timestamps and store region handles or references to data stored in those regions.

The data corresponding to each logical timestamp are stored in a transferable region (`regions[time]`, ln. 14). This region is created when a `Message` with a new timestamp is received. After we create it, we store the region handle in the `regions` dictionary for reuse upon receipt of another message with the same timestamp. The `OnNotify` method is called by Naiad when it can guar-

[2] We piggy-back the manipulation of runtime region stacks onto C#'s existing `using` mechanism for nested scopes.

```
1  public class RegionAggregateActor {
2    private Dictionary<Time, <Dictionary<K, V>> state;
3    private Dictionary<Time, Region> regions;
4    private Region actorRegion; // actor-scoped region
5
6    public RAggregateActor (...) {
7      actorRegion = RegAlloc.NewRegion(ACTOR);
8      using (RegContext.OpenAlloc(actorRegion))
9        state = new Dictionary();
10   }
11   public void OnReceive(Time time, Message msg) {
12     if (state[time] == null) {
13       using (RegContext.OpenAlloc(actorRegion)) {
14         regions[time] = RegAlloc.NewRegion(TRANS);
15         // open transferable region for allocation
16         using (RegContext.OpenAlloc(regions[time]))
17           state[time] = new Dictionary();
18       }
19       NotifyAt(time);
20     }
21     foreach (var entry in msg) {
22       // open state[time] region for object access
23       using (RegContext.OpenAlloc(regions[time]))
24         state[time][key] =
25           Aggregate(state[time][key], entry);
26     }
27   }
28   public void OnNotify(Time time) {
29     // send message, free state[time]
30     Send(state[time]);
31     RegAlloc.FreeRegion(regions[time]);
32   }
33 }
```

Listing 2: *Region-based Naiad aggregate actor.*

antee that no new messages with a timestamp smaller or equal to `time` are going to be received (ln. 28). In this method, we send out the aggregated data (ln. 30) and free the memory used by that particular timestamp (ln. 31).

Readers may notice that any allocations that `Aggregate()` (ln. 25) induces are now part of the transferable region `regions[time]`. This may be undesirable as it pollutes the transferable region. Alternatively, the Naiad developer could create a *statically scoped* temporary region around the call to `Aggregate`. In this case, she must *clone* the result of `Aggregate` from the temporary to the transferable region.

5 Discussion

Regions are non-trivial to use, and while they have attractive benefits, they also introduce some challenges. However, we believe that many of these challenges have simple solutions in the context of distributed data processing.

Memory safety. References across objects residing in the different regions make it challenging to maintain memory safety. Broom therefore restricts the allowable relationships between objects in different region types. Figure 4 shows the allowed points-to relationships:

(1) Objects in a temporary region can point to objects in the same region or any other temporary region that outlives it. They are also allowed to point to objects in the related actor-scoped region and to objects in a transferable region if both regions are owned by the same actor.

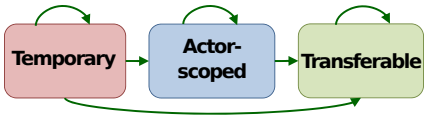


Figure 4: *Allowed points-to relationship between the different region types.*

(2) An actor-scoped region can include references to its own objects. It can also include references to transferable region *handles*, as long as they have the same owner. However, objects in actor-scoped regions must not reference objects *inside* a transferable region.

(3) Every transferable region needs to be self-contained and can only hold references to objects allocated in itself. Otherwise, transferring the ownership to another actor can lead to dangling pointers.

Our prototype does not yet enforce these restrictions on object references. We plan to enforce them through a combination of static and dynamic checks.

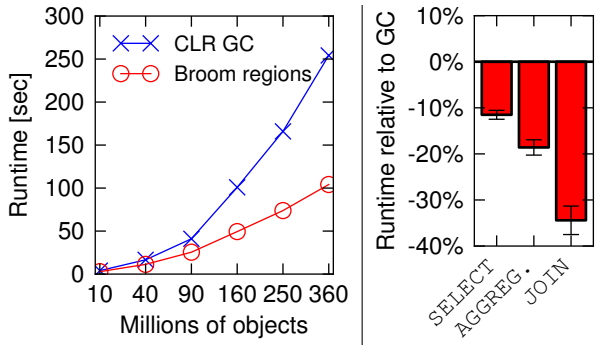
Programmability. The traditional downside of region-based memory management is the additional complexity of working with regions. However, end-users who write high-level data processing queries (e.g. LINQ queries) for the systems we are concerned with do not need to be aware of regions at all. Instead, region annotations occur only in the implementation of the system-provided Naiad vertices (actors). By contrast, data processing system developers must still explicitly use the Broom API for region-based memory management. To reduce the annotation burden on system developers, we are working on techniques that infer regions and their types automatically using static analysis. As distributed data processing systems already perform complex source-to-source transformations and just-in-time compilation, even otherwise expensive analyses can be amortized.

GC compatibility. Regions can co-exist with a GC'ed heap, as long as the garbage collector does not traverse objects allocated inside regions. Furthermore, actor-scoped regions may use a local GC *within* the region.

6 Preliminary results

As a proof of concept, we measure Broom's allocation performance and emulate several Naiad vertices after extending them with region support. All experiments were run on an AMD Opteron 2373 (4× 2.1 GHz) with 32GB RAM, running Windows Server 2012 R2.

Potential gain from using regions. Garbage collectors are most challenged by complicated structures of



(a) *Best-case approximation: an allocation-heavy micro-benchmark completes 59% faster with regions.*

(b) *Stateful in-memory Naiad vertices benefit most from regions.*

Figure 5: *Runtime reduction attained by Broom regions micro-benchmark (a) and emulated Naiad vertices (b).*

many tightly cross-referenced objects: they must traverse the structure on each collection and might need to copy objects between generations. By contrast, no traversal is required when using regions.

This allows us to empirically estimate an approximate upper bound on the performance benefit that regions can offer. To do so, we allocate lists of lists of basic objects, ensuring that the GC must visit every object. Each list contains n objects and there are n lists, with n ranging between 500 and 3000. Figure 5a shows the total time taken to allocate and free 40 such lists-of-lists as a function of the total number of objects allocated. Region-based memory management yields a 59% reduction in runtime for this micro-benchmark.

Naiad vertices with regions. To estimate the benefits that a real-world Naiad computation would experience from using regions instead of GC, we implemented region-based memory management for several widely-used Naiad vertices (`SELECT`, `AGGREGATE` and `JOIN`).

In the following experiments, we run a single-threaded Naiad vertex (actor) on two synthetic, incrementally generated inputs: the `documents` data set receives 500,000–600,000 new entries per time epoch, and the `authors` data set receives 10–20 new entries per time epoch.

For each vertex, Figure 5b shows the reduction in runtime after 40 epochs using regions compared to using GC. Using epochs is always beneficial, but the magnitude of the benefit varies:

- Mostly stateless vertices (e.g. `SELECT`) stream data through, so objects are short-lived. Regions do not help much in this case, although there is some benefit (13% runtime reduction).
- The `AGGREGATE` vertex stores, for each time epoch, a `Dictionary<Key, Values>`. This contains a set of partial aggregation results and can be freed in one go when the time epoch ends. Regions offer a 20% runtime reduction compared to GC, which must traverse all dictionaries on collection.
- Highly stateful vertices such as `JOIN` store their complete input data (two large dictionaries per time epoch for `JOIN`). This is where regions help the most: runtime is reduced by 36% compared to GC.

This confirms our expectations: computations with large collections of fate-sharing objects benefit the most from regions, coming close to the estimated upper bound.

Compared to the results of our batch processing experiments on Mono (§2.1), these results are plausible: the shopper workflow, which consists of two `JOIN`s and a `SELECT` spends 20–40% of its time in GC; regions would likely reduce this overhead significantly.

Yet, these experiments can only be indicative of real-world gains as they simplify matters somewhat. The single-threaded vertex case does not consider thread contention, network message delays or skew in data volume across workers. Analysing the effects of these on a real Naiad implementation is the subject of future work.

7 Related work

Facade [17] introduces a program transformation that splits applications' objects into (a) control objects stored on the GC'ed heap and (b) data objects stored in a per-iteration region. While Facade does not require changes to data processing systems themselves, end-users must identify "boundary classes" and annotate their code.

Berger *et al.* demonstrated that freeing individual objects in region-based memory management can be implemented efficiently [2]. Their work targets general programs, but similar techniques could be applied in Broom.

Like Broom, real-time Java [3] offers three types of memories: the GC heap, singleton immortal regions and statically scoped regions, but does not have *transferable* regions with dynamic lifetime.

ML Kit, based on the Tofte-Talpin static type system [20] introduces statically-scoped regions and static region inference. Later work extends it by combining GC and regions [9]. Cyclone [8] is a C variant with regions which relies on a strong type system for safety and region inference, and supports static and dynamic regions [10]. Dynamic enforcement of memory safety for regions is also covered by Gay and Aiken [6, 7].

8 Conclusions

It's time to revisit regions! They are an excellent match for big data runtimes in modern high-level languages. Domain-specific knowledge of object lifetimes and region-agnostic LINQ interface to end-users can address traditional usability difficulties. The performance benefits are worthwhile and motivate further work on memory safety alongside with programmability.

References

[1] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)* (2003), pp. 285–298.

[2] BERGER, E. D., ZORN, B. G., AND MCKINLEY, K. S. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Seattle, Washington, USA, Nov. 2002), pp. 1–12.

[3] BOLLELLA, G., BROSGOL, B., DIBBLE, P., FURR, S., GOSLING, J., HARDIN, D., TURNBULL, M., AND BELLIARDI, R. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[4] BU, Y., BORKAR, V., XU, G., AND CAREY, M. J. A bloat-aware design for big data applications. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM)* (2013), pp. 119–130.

[5] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM 51*, 1 (Jan. 2008), 107–113.

[6] GAY, D., AND AIKEN, A. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)* (1998), pp. 313–323.

[7] GAY, D., AND AIKEN, A. Language support for regions. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)* (2001), pp. 70–80.

[8] GROSSMAN, D., MORRISETT, G., JIM, T., HICKS, M., WANG, Y., AND CHENEY, J. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)* (2002), pp. 282–293.

[9] HALLENBERG, N., ELSMAN, M., AND TOFTE, M. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)* (2002), pp. 141–152.

[10] HICKS, M., MORRISETT, G., GROSSMAN, D., AND JIM, T. Experience with safe manual memory-management in cyclone. In *Proceedings of the 2004 International Symposium on Memory Management (ISMM)* (2004), pp. 73–84.

[11] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys)* (2007), pp. 59–72.

[12] LARUS, J., AND HUNT, G. The Singularity System. *Communications of the ACM 53*, 8 (Aug. 2010), 72–79.

[13] LIEBERMAN, H., AND HEWITT, C. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM 26*, 6 (June 1983), 419–429.

[14] MAAS, M., HARRIS, T., ASANOVIĆ, K., AND KUBIATOWICZ, J. Trash day: Coordinating garbage collection in distributed systems. In *Proceedings of the 15th USENIX/ACM Workshop on Hot Topics in Operating Systems (HotOS)* (Kartause Ittingen, Switzerland, May 2015).

[15] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: a timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (2013), pp. 439–455.

[16] NEUMEYER, L., ROBBINS, B., NAIR, A., AND KESARI, A. S4: Distributed stream computing platform. In *Proceedings of the 2010 International Conference on Data Mining (Workshops) (ICDM)* (2010), pp. 170–177.

[17] 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 *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015).

[18] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Oakland, CA, May 2015), pp. 293–307.

[19] TOFTE, M., AND TALPIN, J.-P. Region-based memory management. *Information and Computation 132*, 2 (1997), 109–176.

[20] TOFTE, M., AND TALPIN, J.-P. Region-based memory management. *Information and Computation 132*, 2 (Feb. 1997), 109–176.

[21] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., GUNDA, P. K., AND CURREY, J. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008), pp. 1–14.

[22] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. 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 (NSDI)* (2012).

[23] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (2013), pp. 423–438.