Gerenuk: Thin Computation over Big Native Data Using Speculative Program Transformation

Christian Navasca, Cheng Cai, Khanh Nguyen
UCLA

Brian Demsky
UC Irvine

Shan Lu
University of Chicago

Miryung Kim, Guoqing Harry Xu
UCLA

Abstract

Big Data systems are typically implemented in object-oriented languages such as Java and Scala due to the quick development cycle they provide. These systems are executed on top of a managed runtime such as the Java Virtual Machine (JVM), which requires each data item to be represented as an *object* before it can be processed. This representation is the direct cause of many kinds of severe inefficiencies.

We developed Gerenuk, a compiler and runtime that aims to enable a JVM-based data-parallel system to achieve near-native efficiency by transforming a set of statements in the system for direct execution over *inlined native bytes*. The key insight leading to Gerenuk's success is two-fold: (1) analytics workloads often use *immutable and confined* data types. If we *speculatively* optimize the system and user code with this assumption, the transformation can be made tractable. (2) The flow of data starts at a *deserialization* point where objects are created from a sequence of native bytes and ends at a *serialization* point where they are turned back into a byte sequence to be sent to the disk or network. This flow naturally defines a *speculative execution region* (SER) to be transformed. Gerenuk compiles a SER speculatively into a version that can operate directly over native bytes that come from the disk or network. The Gerenuk runtime aborts the SER execution upon violations of the *immutability* and *confinement* assumption and switches to the *slow path* by deserializing the bytes and re-executing the original SER. Our evaluation on Spark and Hadoop demonstrates promising results.

CCS Concepts • **Information systems → Data management systems**; • **Software and its engineering → Compilers**.

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359643

Keywords Dataflow system, speculative transformation, static analysis, runtime system

ACM Reference Format:
Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, and Miryung Kim, Guoqing Harry Xu. 2019. Gerenuk: Thin Computation over Big Native Data Using Speculative Program Transformation. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3341301.3359643

1 Introduction

Modern Big Data systems, such as Hadoop [12], Spark [73], Flink [14], or Hive [13], were all implemented in object-oriented languages such as Scala and Java due to the high productivity enabled by these languages. However, managed runtime systems incur a heavy runtime cost [19, 30, 56, 58], leading to reduced efficiency and processing capabilities. A major source of this cost comes from the fundamental abstraction of object-orientation—everything is an *object*.

For data-intensive systems, each computational iteration needs to process billions of data items, which are represented as billions of objects. Previous work shows that this object-based representation can inflate the memory usage by almost four times [19] due to the extra meta-information required by the managed runtime such as object headers, padding, pointers, etc. Such a huge memory inflation incurs exceedingly high garbage collection (GC) overhead, up to 70% of the total execution time [57], and results in either increased computation cost to satisfy the memory need or increased memory-disk round trips.

Big Data systems often need to shuffle billions of objects, and representing each data item as an object dictates that each shuffle needs to *serialize* many objects into native bytes, transfer them over the network, and *deserialize* the bytes back into objects before they can be processed at a remote machine. Evidence shows that serialization and deserialization can take up to 30% of the execution [56]. This cost will only increase as systems become more heterogeneous by incorporating heterogeneous software/hardware components, such as UDFs in different languages or accelerators. All represent data in their unique formats, dictating even more frequent serialization and deserialization.

State of the Art. The cost of representing data as objects has been a known problem, with much recent effort attempting to reduce this cost. Some work tackles one aspect of the problem. Skyway [56] lowers the (de)serialization cost by directly transferring objects through the network, and hence, cannot help with memory inflation and GC problems (it actually brings some data inflation into network packages). Yak [57] lowers the GC cost by adapting the GC algorithm to fit lifetime patterns in data-intensive systems, and does not solve memory inflation or (de)serialization problems. Broom [34] provides APIs for Naiad [54] developers to manually allocate/deallocate data objects in regions. It works only for Naiad and does not provide automated support.

Some work provides a partial solution by focusing only on certain data types. For example, Tungsten, a Spark project, enables abstractions such as `DataFrame` and `DataSet` to be stored in native memory on which hashing and sorting can be directly performed. However, such abstractions work only for simple primitive or sequence types, but not for user-defined types that involve structures and pointers, such as various sparse/dense vectors used in machine learning algorithms.

One approach that may fundamentally address this problem is to automatically transform the (system and user) code so that data items are represented as native bytes and data processing is performed in native memory. The feasibility of this approach depends on how data are processed.

The good news is that prior work [19] observed that the majority, more than 95%, of runtime objects are created and used by a rather small and simple codebase that primarily conducts data manipulation like map, reduce, and relational operations, which are amenable to and can benefit greatly from such a transformation. Only less than 5% of runtime objects are created by a large and complex codebase for cluster management, scheduling, communication, and others, which are extremely difficult to, and fortunately need not to, be transformed. The bad news, however, is that there does not exist a *clear separation* between the former, referred to as *data path*, and the latter, referred to as *control path*—they are often heavily mixed inside one class and even one method.

A recent technique Facade [58] attempts to provide such a transformation as described above. However, since it aims to transform one whole class at a time, and turns every field in a class into a native representation and every statement in every method of the class into a native-byte operation, using Facade requires a huge amount of code refactoring to split many classes and methods to make sure that data and control code modules do not interfere, making it extremely difficult to use in practice (more details in §2).

Our Key Insight. To make transformation more effective, *our first insight* is that we should perform *fine-grained* transformation on individual statements rather than a class/method as a whole—if we identify and transform only the statements to which *data objects* can flow, the analysis and transformation effort is much more focused and the need for manual refactoring is much less. Specifically, even if a class contains both control-item and data-item fields, we do not need to split the class; even if a method contains some statements processing control items and some processing data items, we can leave the former *as is* and only transform the latter. Hence, no refactoring is needed at the class level to achieve a clean separation between control and data paths.

Of course, fine-grained transformation does not solve all the problems. Our goal is to represent all data objects by *inlining only their payloads* in native bytes. This cannot be done under certain circumstances. First, although extremely rare, some data objects may escape to and get referenced by classes in the control path, and hence cannot be turned into a native representation. Second, although extremely rare (again), some data items may be used on the data path in a way that is not amenable to the use of a native representation. For example, if field f of object o is to be updated during execution, o cannot be turned into native bytes, which would inline the object referenced by $o.f$ and leave no reference inside o for update.

We solve these remaining problems and hence eliminate the need for manual refactoring through our *second insight*: since data objects created by most applications are *indeed immutable* and *confined* (i.e., they never escape to external objects), we can develop an *optimistic* technique to *speculatively* transform programs assuming *immutability and confinement* of data objects. Such a transformation can easily succeed with little user involvement and apply to a broad range of applications, as long as the transformed program can notice and respond appropriately to rare *mis-predictions* about data-object properties at run time.

Gerenuk. Based on these insights, we developed Gerenuk, a Java-based compiler and runtime. The Gerenuk runtime contains a serializer that represents all data objects in an *inlined, serialized* form with all headers and pointers eliminated and stores them in native-memory buffers. The Gerenuk compiler automatically transforms a program, converting its data-manipulating statements and making them operate directly on these inlined native bytes. Achieving this ambitious goal has three major challenges.

The *first challenge* is how to identify which code statements to transform. Our approach is based on a key observation that data-processing logic is *task-based.* As illustrated in Figure 1, each task (e.g., a stage in Spark or a Map/Reduce execution in Hadoop) starts at a shuffling phase where each compute node reads in a new set of data items; these data items get deserialized into objects, which then flow through a series of system-level and user-defined processing functions; at the end of the task is another shuffling phase that serializes each object into a sequence of bytes and then sends them to files or the network. In other words, data flows from the deserialization point at which heap objects are created

from native bytes (e.g., a call to method `readObject`) to the serialization point at which heap objects are converted back to bytes (e.g., a call to method `writeObject`).

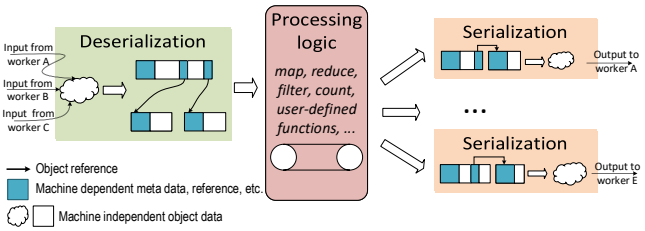

Figure 1. The flow of data objects.

This data flow naturally defines a *speculative execution region* (SER) for transformation and execution. Using these start and end points, Gerenuk automatically identifies the set of statements involved in the flow, transforming them with the goal to skip the entire deserialization and serialization process and let the processing logic operate directly over the *native, serialized form* of data objects.

Note that this is a *one-stone-multiple-bird* approach. First, no objects including their headers and pointers are created for data items, leading to great reduction in memory consumption and GC overhead. Second, Gerenuk eliminates almost all serialization/deserialization effort, which has been shown to be expensive [55, 56]. Third, since a native buffer stores only data items for one particular task, the buffer is naturally amenable to *region-based memory management* — we can safely release the buffer as a whole at the end of the task without even needing to scan the items. Essentially, Gerenuk removes *all* three types of overhead—memory usage inefficiency, serialization, and garbage collection—from the managed runtime for data processing whereas existing techniques could eliminate only one or a subset of them.

The *second challenge* is how to transform the program to conduct processing over native bytes, which represent *inlined* data structures rooted at a set of *top-level* objects. For example, in Gerenuk, our serializer represents each data object o as a byte sequence by *inlining* the data payloads from o as well as other objects reachable from o on the object graph, with all pointers eliminated. The question here is, thus, without deserialization, can the transformed program directly process each inlined data structure containing only payloads, rather than individual objects connected by pointers?

As discussed earlier, we adopt an *optimistic approach* — we transform the program *speculatively* assuming that all data objects are *immutable and confined* in their data structures.

Our compiler identifies a set of program locations at which this assumption could be potentially violated. The compiler instruments, at each violation point, code that aborts the SER. Once a SER is aborted, the Gerenuk runtime discards the current task execution and re-executes the original, unmodified task with the same input data. This task deserializes bytes back into heap objects and uses these objects for processing.

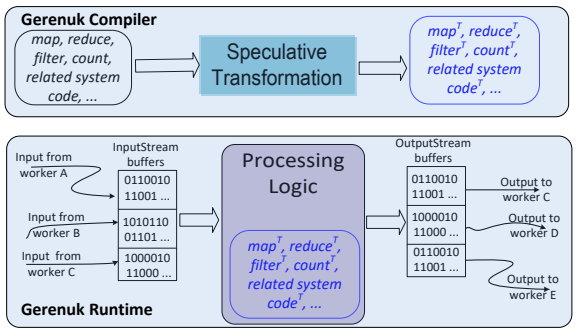

Figure 2. Gerenuk's overview.

```
1 class DenseVector[V] (
2   val data: Array[V],
3   val stride: Int)
4   ...
5 }
```

```
1 class LabeledPoint (
2   val value: Double,
3   val features:
4   DenseVector[Double])
5   ...
6 }
```

Figure 3. A user-defined data structure in Spark for Logistic Regression program.

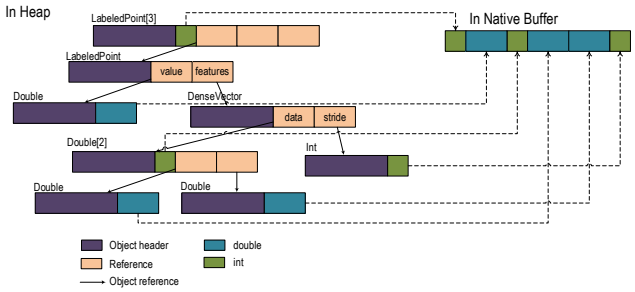

Figure 4. The difference in memory layout between the heap and native-buffer (inlined) representation of an array of three `LabeledPoint` objects; the space overhead incurred by the JVM is nearly 2× larger than the actual data size.

The *third challenge* is how to safely re-execute upon a violation. Our solution is based on a simple observation that data objects are immutable and hence the execution of any task would never modify the input buffers (i.e., the execution gets aborted before modifying). Upon the abortion of a SER, the current executor (e.g., a thread in Spark or a JVM instance in Hadoop) is terminated with all intermediate buffers discarded. The failed SER may have modified some control objects, but once the executor terminates, all of the control information of the executor is removed as well. Gerenuk then launches a new executor to execute the "slow path", which is the unmodified SER, with the original input buffers.

Note that this is possible only for data-parallel systems — there is no global state shared between multiple tasks (e.g., a similar observation has also been used in [30]) — a SER either succeeds and produces results to be fed to another SER, or aborts, causing the system to instead execute the unmodified version of the same SER on the same input. The implementation of *abort* only needs to call a few methods to launch a new executor and terminate the current executor.

An overview of Gerenuk is illustrated in Figure 2. We have implemented the Gerenuk runtime in OpenJDK 8 and the Gerenuk compiler on the Soot Java compiler framework [1]. Using Gerenuk, we automatically transformed a set of applications on Apache Spark [73] and Apache Hadoop [12]. Gerenuk's transformation has improved the end-to-end performance for these two systems by an overall factor of **2**× and **1.4**×, respectively.

2 Extended Motivation

How Data Path and Control Path Mix. Consider the class `StreamingContext`, which is the main entry point for streaming-related functionality in Apache Spark. Many methods defined in this class are data-processing methods (e.g., `textFileStream` which creates an RDD from a text file) while other methods contain control code that does not manipulate user data. All of these methods need to share certain global control state, which is maintained in instance fields of the class. For example, one of these fields stores the current file system directory information, which is used by both data manipulation methods and control methods.

If we follow the whole-class transformation philosophy in Facade [58], developers must manually refactor and split `StreamingContext` into a data component (e.g., a new `DataContext` class) and a control component (e.g., a new `ControlContext` class) that contain methods that process vs. do not process user data, respectively. Since the shared instance fields that used to be in the same class now get split into two copies (one in the heap and a second in native memory), the user must also guarantee consistency between these copies. For any real-world system, such manual refactoring effort is huge (e.g., several weeks to months), creating significant obstacles for practical adoption.

Analytical Motivation. To understand the space overhead incurred by the object-based data representation, we studied a Spark application that computes logistic regressions and analyzed the number of bytes consumed by its data objects. In this application, the user defines a `LabeledPoint` class as the data type (i.e., the type of the RDD elements). This definition is shown in Figure 3. Each `LabeledPoint` object references a variable-length vector of `double` values.

Figure 4 compares the object-based representation of an array of three `LabeledPoint` objects and its corresponding native, inlined representation. Under the native representation, each inlined `LabeledPoint` contains 3 `int` and 3 `double` values, taking 36 bytes. An array with three `LabeledPoint` records all inlined only needs 4+36×3 = 112 bytes, where 4 is the number of bytes needed to record the array size. However, the object-based representation, which connects these objects with pointers, requires, in addition to the 112-byte data payload, an overhead of 8 object headers and 9 object references. These headers and references take 8×16+9×8 = 200 bytes, bringing the total bytes needed to 312 bytes.

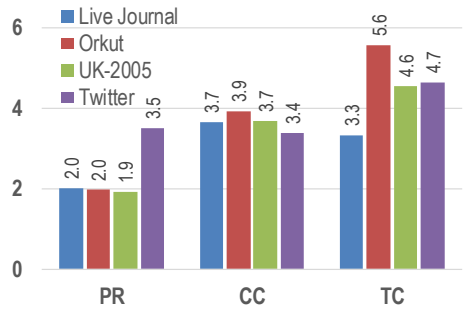

Figure 5. The ratios between the total bytes of data objects and the size of their actual payload.

Hence, the object representation overhead is nearly twice the size of the actual data payload.

Empirical Motivation. To empirically quantify this space overhead and its related runtime overhead, we ran three graph analytics programs on Apache Spark. We used Spark version 2.1.0 on a cluster of 11 nodes, each with 2 Xeon(R) CPU E5-2640 v3 processors, 32GB memory, 1 SSD, running CentOS 6.9 and connected by a 1000Mb/s Ethernet. The three graph programs were PageRank (PR), ConnectedComponents (CC) and TriangleCounting (TC) over four real-world graphs, LiveJournal [15], Orkut [36], UK-2005 [17], and Twitter-2010 [42]. Kryo [41], a high-performance serializer, was used.

To understand the size difference between data objects and their serialized bytes, we modified Kryo to report the numbers of bytes occupied by data objects before and after they were serialized by Kryo for each shuffle. The first number includes the size of the actual data as well as the space overhead incurred by the JVM, and the second number represents the size of the actual data in the *inlined form*. Finally, these numbers were aggregated across the machines.

Figure 5 reports the ratios between these two numbers. Across the programs, the overall ratio is 3.5×. In other words, the extra space incurred by the object-based representation is 2.5× as large as the actual payload size. The main reason for this large overhead is that these applications create billions of small objects (e.g., `java.lang.Integer`, `java.lang.Long` or `java.lang.Double`), whose corresponding header/pointer overhead cannot be amortized by actual payloads.

Anticipated Benefits. These results highlight several potential benefits that can be achieved if the processing engine can work directly over native bytes. First, we expect that much of the 2.3× memory overhead can be eliminated as the program directly operates over the serialized bytes. Memory savings in Big Data systems can often translate to increased degrees of parallelism due to more available memory and thus the ability to run more executors. Second, we expect significant reductions in computation time due to reduced pointer chasing effort as well as eliminated runtime checks (e.g., array bound checks and write barriers performed by

the managed runtime). Third, we expect that the serialization/deserialization time, which could take up to a third of the execution time [55, 56], will be significantly reduced because the same (native) format is used when data is transferred over the network. Finally, we expect large savings in garbage collection since data items are no longer represented as objects, and hence, the GC only needs to scan a much smaller number of objects. In the initial experiment described earlier in this section, we did not see much GC overhead (< 10%) from Spark's executions because the size of data per machine was small relative to the size of the heap (i.e., 30GB) used. However, the GC overhead can grow to dominate the execution for large inputs, as shown in prior works [19, 34, 51, 57] as well as our experimental results (§4).

Applicability. Gerenuk is designed specifically for dataflow systems. There are two conditions under which a system is amenable to our transformations. First, data objects exhibit the data flow shown in Figure 1 (e.g., there are sources and sinks). Second, data objects do not carry state and are immutable. While our current implementation does not work directly for other dataflow systems such as Scope [20], Hyracks [39], or Naiad [54], the aforementioned two conditions hold for these systems, and hence they can all benefit from the proposed transformations. Gerenuk cannot optimize systems for which the conditions do not hold. For example, if a system does not clearly define sources and sinks for data objects, we cannot identify the dataflow. As another example, if data objects are not immutable, the transformed program would always abort, resulting in large performance penalties.

3 Gerenuk Design and Implementation

Gerenuk contains a compiler that performs speculative transformations and a runtime that implements speculative execution logic. During the execution of the transformed program, the input of each computational iteration is a sequence of bytes representing a set of inlined data structures each rooted at a top-level data item (i.e., an RDD element). The output of the iteration is another sequence of bytes, representing the generated data structures to be shuffled.

3.1 User Effort

Gerenuk's compiler takes as input a user program (e.g., a Spark dataflow program or a Hadoop map/reduce program) and three types of annotations described below. It then transforms both the user application and data-processing system code, and finally outputs a new version of the user application and system code that operates over native data. The old version of the code is kept, since it will be executed when a SER is aborted. Gerenuk requires the user to provide three pieces of information.

First, since a speculative execution region starts at a deserialization point and ends at a serialization point, Gerenuk relies on the user to identify these two points. For Hadoop, a call to `WritableDeserializer.deserialize()` in method `nextKeyValue` of class `ReduceContextImpl` is a deserialization point, while a call to `WritableSerializer.serialize()` in method `append` of `IFile` is a serialization point. There can be multiple deserialization and serialization points for a system. Gerenuk uses these (de)serialization points to automatically identify the statements through which data flows for transformation (see §3.5). Although specifying serialization APIs requires system-level knowledge, this task can be done by a Spark or Hadoop system developer once and for all and has only one-time cost, since Spark and Hadoop all have clear serialization modules and their APIs rarely change.

Second, the user needs to specify the types T of *top-level* data items processed in the program (e.g., via annotations). For example, for a Spark program, these data types include the element types of the RDDs in the program (e.g., `DenseVector`). For a Hadoop program, they include the types of keys and values that are read from or written into HDFS. The Gerenuk compiler will identify the data structure rooted at each such type to establish a mapping between its native (inlined) format and object-based format (see §3.3).

Third, the user specifies the type of data collections. In the case of Spark, the collection type is any subtype of `org.apache.spark.rdd.RDD`; for Hadoop, each input or output buffer is a data collection and the types of these buffers need to be specified. Based on these specified collections, Gerenuk will generate new collection types that use *long* as their elements instead of objects. For example, an RDD type `ResultRDD⟨DenseVector⟩` will be transformed into a new type `ResultRDD⟨long⟩` that represents `DenseVector` objects natively using byte buffers and each `long` value indicates the starting address of a data record.

3.2 SER Code Analyzer

The first component of the Gerenuk compiler is a *context-sensitive*, *path-sensitive* static analysis that traces the flow of data objects in a SER to identify all statements involved in the SER. These statements need to transformed, in the next step, to operate over the native bytes. Since we analyze Java bytecode rather than source code, Gerenuk works for all programming languages executed on JVMs, including Java, Scala, etc. Treating each pair of (user-specified) deserialization and serialization points as a *source* and a *sink*, Gerenuk analyzes both system and user application code to identify all statements through which data objects can flow from the source to the sink. These statements form the *data path* we want to automatically transform.

Our analysis is similar to a *static taint analysis*—it starts by tagging the variable defined at a deserialization point (e.g., `v` in `v = readObject()`) and then propagates the taint mark from one variable to another by tracking the data flow from the source to the sink. In general, for each assignment `a = b`, we taint `a` if `b` is tainted. A static taint analysis is known to be imprecise due to its reliance on static modeling

of heap accesses and static resolution of virtual method calls. To make the analysis report less irrelevant information, we adopt the following three approaches.

First, for an object field read or an array read, our analysis tracks the flow from both the pointer and the value dereferenced from the object field. For example, for a read statement `a = b.f`, suppose o is an allocation site in the *points-to set* of b. We taint a if (1) either b is tainted, or (2) the field $o.f$ is tainted where $o.f$ statically models the heap location referenced by $b.f$. The reason here is that if the object o referenced by b is a data object that comes from deserialization, then the object referenced by $b.f$ should also be a data object and hence be tracked by our analysis.

For an object write such as `b.f = a`, we do not taint the left hand side $o.f$ because if a references a data object, this write would indicate that the object escapes the current data structure to a different one. In this case, our compiler will insert an *abort* instruction at the write to abort the SER instead of tracking the object further. How *abort* instructions are inserted is discussed in detail in §3.4.

For an array write such as `b[...] = a`, we only need to taint o.*ELE* (where *ELE* is a placeholder representing array elements) if the object (say o_1) referenced by a is a *top-level* data record (of a user-defined data type T). In this case, this statement represents writing of a data record into a data collection (e.g., the backbone array of an RDD) and hence we must track the flow of the object. If o_1 is a *lower-level* object that belongs to the data structure rooted at type T, this write represents an *escape* operation and our compiler would insert an *abort*.

Second, unlike a traditional taint analysis that propagates the taint mark in all directions, the sink information in our analysis provides tremendous help in pruning away flows—flows that do not eventually lead to the sink are often due to the conservative handling of static analysis; they are eliminated and the statements not flowing to the sink are not considered for transformation. The underlying assumption here is that data objects in a SER will eventually all go to some kind of serialization (e.g., to disk files or the network), which is the case for modern data-parallel systems. An avid reader might notice that the correctness hinges on whether all serialization points can be found. Therefore, a conservative approach is to insert an *abort* on the paths that do not lead to any sink —even with a missed serialization point, the program can still run correctly.

Third, control flow is not tracked. Since our goal is to find a set of methods to be transformed instead of tracking full-blown information flow, we do not taint a variable if it is only control-dependent on a predicate that involves a tainted variable. Furthermore, unlike a traditional taint analysis that needs to propagate taint marks via arbitrary arithmetic operations, we focus only on object references; primitive-type values are irrelevant in this context.

Eventually, our analysis returns a set of statements (and their containing methods) on the flows of data objects from the point at which they are created from deserialization to the point at which they are serialized to bytes.

3.3 Data Structure Analyzer

The second component of Gerenuk compiler is a data structure analyzer. Given a top-level data type T specified by users (like an element type of an RDD as discussed in §3.1), the analyzer explores all classes referenced directly or transitively by T, and outputs a map that maps each primitive- or array-type field in these classes to its corresponding offset inside the native buffer-based representation of T. This offset information will later be used by Gerenuk to replace object-based field accesses in the original program with native buffer-based accesses in the transformed program.

At a high level, our algorithm uses a DFS traversal, starting from T, to recursively explore a class hierarchy rooted at the top class T. During the traversal, it calculates offsets in a bottom-up manner—it calculates the size of each class at the bottom of the hierarchy and takes into account their sizes when computing offsets for those primitive- or array-type fields in top classes. If every class has a fixed size (i.e., does not directly or transitively reference an array), the algorithm is straightforward, as the class sizes and field offsets can all be statically calculated. If a class has a variable size, our algorithm uses symbolic expressions in the size/offset calculation, which we elaborate below.

Consider the following class that defines a data structure that does not have a statically decidable serialized size: `class C { int a; long[] b; double c; }`.
The offsets for field `a` and field `b` in a record of C are straightforward, 0 and 4, but the offset for `c` is non-trivial. The offset of field c consists of (1) the size of field a, which is 4, and the size of array-field b in its native representation, which further consists of (2) 4 bytes that stores the length of array b, denoted as $b.len$ and (3) the size of the content of array b ($8 \times b.len$). Since $b.len$ is stored in front of the data content of b and right after field a, it can be accessed at run time by an auxiliary function provided by our runtime `readNative`, which takes three parameters, the base address of the current record, the offset in the current record, and the number of bytes to read. Consequently, the length of b can be computed by `readNative` (BASE_C, 4, 4), with BASE_C representing the starting address of the current record of C in the inlined bytes; the offset of field c can be computed by $4 + 4 + 8\times$`readNative`(BASE_C, 4, 4); and the total size of class C is $16 + 8\times$`readNative`(BASE_C, 4, 4).

The above size expression of C will be used to compute offsets for the fields in classes that directly or transitively reference C. Specifically, when the DFS traversal finishes C and returns to an upper-level class C', BASE_C will be replaced with an expression containing a new symbolic value $\text{BASE}_{C'}$ representing the starting address of a record of C'. Eventually,

when the DFS exploration finishes back at the top-level type T, all the offset expressions are represented w.r.t. the starting address of a record of T. The concrete value of this address will be provided by the runtime during execution.

Special Cases. Special support is developed to handle strings—since strings are commonly used types, instead of fully analyzing the `String` class, our analysis treats a string as a character array; specialized string operations are provided to access and manipulate the array. Gerenuk supports generics in Java/Scala. The analysis tracks the type parameters that a class is instantiated on and uses this information to determine the types of the class's internal fields. Some of the classes in the Java Collection framework (e.g., Vector<E> has an internal array of type Object and not an array of type E) do not fully make use of generic types for their internal fields—Gerenuk has been extended with the knowledge of internal field types for commonly-used collections.

Our analyzer stops upon seeing a data structure whose shape is not a tree. Such data structures cannot be represented without pointers. In practice, however, real-world data types are often simple (e.g., at most two or three layers) and we have never seen such structures in our evaluation. Note that we built a customized serializer that inlines all data records. Hence, we do not need to compute offsets for pointers as they do not exist in the serialized bytes.

3.4 Computing Violation Conditions

Taking as input the set of methods returned by the SER code analyzer, Gerenuk transforms each method so that the transformed method can process the inlined bytes. The central idea of the transformation is to rewrite each field access that reads/writes a data object o as an access to o's corresponding inlined bytes stored in a native JVM buffer (that comes from disk files or the network). However, not all field accesses in the original program can be transformed. Before describing our transformation algorithm, we first present a list of *violation conditions*—those under which memory accesses *cannot* be performed on inlined data. Gerenuk transforms the program *optimistically* while instrumenting *abort* instructions at each violation point.

The fundamental assumption under which the transformed program can successfully process the inlined data is that objects in each data structure rooted at each user-defined data type T are *reference-immutable* and *confined in the data structure*. However, due to the conservative nature of static analysis, the Gerenuk compiler often sees cases where these conditions *may* be violated (although they may not actually be violated at run time). To guarantee that our transformation is safe, we build a "fence" around the violation points by inserting *abort* instructions.

- **Violation #1: Load-And-Escape.** Consider the following code snippet: `v = n.f; o = new O(); o.g = v;` Suppose our analysis finds that variable `n` refers to an object in an inlined data record (rooted at a user-defined type T). This case violates our *confinement* assumption—a reference read from `n.f` *escapes* the data structure and gets assigned into another object `o`. In the inlined bytes, such a reference would not exist, and hence, this piece of code would cause an execution failure.
- **Violation #2: Disrupt-the-Native-Space.** If there exists a statement `n.f = o` that writes a reference of a regular heap object `o` into an object `n` that is part of an inlined data record T, the execution would fail because the inlined bytes cannot hold Java references. This is a violation of our *immutability* assumption.
- **Violation #3: Invoke-Native-Method.** A violation occurs if we encounter a call site `p = n.m()` where `n` is an object in a data structure T and `m` is a native method, because a native method may create external side effects. However, certain native methods are frequently invoked. While calls to native methods are generally considered as violations, Gerenuk provides customized implementations (that can work with the inlined bytes) for a set of frequently-used native methods, such as `clone`, `hashcode`, `toString`, and `arrayCopy`, to improve usefulness.
- **Violation #4: Use-Object-Metainfo.** A violation occurs if the metadata (such as the lock) of an object in an inlined data record T is explicitly used. An example code snippet is `v = n.f; synchronize(v){...}`. Here variable `n` refers to an object in an inlined data record and hence v also points to an object in the data record. Use of v as a lock would lead to a violation detected by our compiler because in the transformed program, v and n are no longer objects and hence no lock can be obtained from them.

Note that these four conditions are *complete* in describing immutability and confinement violations—no violations can possibly occur without encountering an *abort* first. This can be easily seen by reasoning about the cause of violations—reference passing between native data and the Java heap. In particular, there are two possible scenarios in which reference passing can occur: (1) the program reads an object reference from native data and writes it into the Java heap and (2) the program reads a reference from the heap and writes it into a native buffer. These two cases are covered, respectively, by the first and second violation conditions.

Attempting to read a reference from native data and use it for any non-payload-access purposes (e.g., invoke a method or obtain the object metadata) should also be forbidden—this is not possible to do in the transformed execution because no object would exist in native buffers. Invocations of regular methods will be inlined (see Case 9 in Algorithm 1) and not exist after transformation. However, inlining cannot be done for calls to native methods, and hence, the third condition inserts an *abort* when encountering a native method call. The fourth condition protects the execution from running into any metadata-obtaining statement.

Since we focus on data-parallel systems where data objects are immutable and confined for most workloads, many of these statically-detected violations are false positives due to the conservative nature of a static analysis, and hence, they do not occur during execution and also do not lead to abort-and-retry under the Gerenuk runtime.

In most cases, a violation is generated when a lower-level type (e.g., `Vector`) in the class hierarchy of a user-defined data type T is instantiated in other locations; objects created in these other locations can be mutated but those created under T cannot. A conservative technique, such as Facade, has to rely on human developers to manually refactor violating statements to make sure the static analysis would not see these statements during transformation. On the contrary, Gerenuk does not attempt to successfully transform an entire method/class; it simply inserts *abort* instructions, making it significantly easier for our transformation to succeed.

All of the *abort*s that are inserted due to overly-conservative static analysis will not be triggered at run time, thus not degrading performance. Aborts may also be inserted when a data type cannot be represented as inlined bytes (e.g., objects of the type can be updated) —these *aborts* will be triggered, although such cases are extremely rare.

3.5 Speculative Transformation

Gerenuk first transforms each user-specified collection class (like RDD) by replacing each occurrence of each data type T with a long value that indicates the starting address of a record. In Spark, for example, all RDDs, after transformation, use `long` as their element types and all their iterator implementations also return a *long* value. This transformation, which is straightforward, is done before Algorithm 1 starts.

Given a set of statements returned by the SER code analyzer, Gerenuk transforms each statement s into another statement s' in which all accesses to data objects are replaced with accesses to the inlined bytes in native buffers. Specifically, the transformation is applied to 8 different types of statements that access objects whose classes are in the class hierarchy discovered by the data structure analyzer, as shown in Algorithm 1.

Replace and Emit in Algorithm 1 are two auxiliary functions that, respectively, replace the current instruction with a new instruction and emit a new instruction into the generated program.

We replace each reference-type variable with a `long`-type variable representing the address of the data item in the inlined bytes. These addresses are originated from the deserialization point (Case 1 in Algorithm 1)—we replace the deserialization method call such as `readObject` with a call to method `getAddress` provided by our runtime to obtain the address of the top-level record. The address gets assigned to a `long`-type variable and propagated in the program.

In Case 2, each variable assignment is replaced with an address assignment. Case 3 deals with the transformation of parameter passing statements. Case 4 and Case 5 handle heap stores and loads, respectively. These heap accesses are replaced with calls to our methods `writeNative` and

Algorithm 1: Our code transformation algorithm.

Input: (1) A set of statements S returned by the SER code analyzer;
(2) a set of classes C forming the class hierarchy of each inlined data structure;
(3) a map *Sizes* between each class in C and its (inlined) size;
(4) a map *Offsets* between fields and their offsets in the class hierarchy;
(5) a set V of violation points.
Output: A set of transformed statements S'.

```
1  foreach Statement s ∈ S do
2      /* Case 1: deserialization point */
3      if s is a deserialization "a = readObject()" then
4          Replace("long addr_a = getAddress()")
5      /* Case 2: regular assignment */
6      if s is "a = b" and Type(b) ∈ C then
7          Replace("long addr_a = addr_b")
8      /* Case 3: parameter-passing */
9      if s is "a = p" and p is a formal param then
10         if Type(p)∈ C then
11             Replace("long addr_a = addr_p")
12     /* Case 4: primitive-type field store on a data object*/
13     if s is "a.f = b" and Type(a) ∈ C and a.f is of a primitive type then
14         off ← 0
15         if Offsets[Type(a), f] is a static constant then
16             /* Offset is statically known*/
17             off ← Offsets[Type(a), f]
18             Replace("writeNative(addr_a," + off + "," + SizeOf(f) + ", b)")
19         else
20             /* Offset is an expression*/
21             Emit("t = resolveOffset(" + Offsets[Type(a), f] + ")")
22             Replace("writeNative(addr_a, t, b)")
23     /* Case 5: field load on a data object */
24     if s is "b = a.f" and Type(a) ∈ C then
25         off ← 0
26         if Offsets[Type(a), f] is a static constant then
27             /* Offset is statically known*/
28             off ← Offsets[Type(a), f]
29             Replace("b = readNative(addr_a," + off + ", " + SizeOf(f)
               +")")
30         else
31             /* Offset is an expression*/
32             Emit("t = resolveOffset(" + Offsets[Type(a), f] + ")")
33             Replace("b = readNative(addr_a, t, "+ SizeOf(f)) + ")")
34     /* Case 6: allocation site */
35     if s is "a = newA()" and A ∈ C then
36         size ← 0
37         /* The size of the structure is a constant*/
38         if Sizes["A"] is a static constant then
39             size ← Sizes["A"]
40             Replace("appendToBuffer(" + size + ")")
41         else
42             /* The size is an expression */
43             Emit("t = " + Sizes["A"])
44             Replace("appendToBuffer(t)")
45     /* Case 7: violation handling */
46     if s ∈ V then
47         Emit("Abort()")
48     /* Case 8: serialization point */
49     if s is a serialization "writeObject(a)" then
50         Replace("gWriteObject(addr_a)")
51     /* Case 9: method call */
52     if s is a call "n.m(a)" and Type(n) ∈ C and m is not native then
53         InlineAndTransform(m)
```

`readNative`, respectively, that access the inlined bytes. Note that for heap stores, we only allow writing into primitive-type fields because writing into reference-type fields are prohibited by the second violation condition.

Depending on whether the offset of the field is a static constant or an expression that contains symbolic variables, code generation is done in different ways. For example, if the offset is an expression, the expression will be resolved by the runtime via an auxiliary function `resolveOffset` (see §3.6), assigned to a temporary variable t, which is then used to invoke `writeNative` and `readNative`.

The treatment of allocation sites (Case 6) is similar—we invoke our auxiliary function `appendToBuffer` with the size of the entire inlined data structure rooted at the type (e.g., A) as an argument. This size, computed by the data structure analyzer, may be a static constant or a symbolic expression.

Case 7 shows the handling of violations—our compiler simply inserts an `abort` instruction right before each violating statement. This guarantees that the execution will never reach the violating statement at run time.

Case 8 deals with the serialization—the call to `writeObject` is simply replaced with a call to our serializer `gWriteObject`, which takes a native address as input and writes the entire record into the output stream.

If a call is encountered (Case 9) and the call is made on an object whose type is in the class hierarchy C, we inline the method m into the caller and recursively transform m's body. We use a pointer analysis [48] to resolve virtual calls.

Discussion. We only represent data structures rooted at each user-defined type T as inlined bytes and these bytes are processed only by the transformed statements. If any type involved in the data structure is also used in other locations (e.g., control path), the original type will still be used there. Our data structure analyzer analyzes each T and records its structure into a *schema* file, which will be used to perform offset computation during transformation, serialization, and deserialization. Our underlying assumption here is that the creation and manipulation of data records are all performed over native buffers. During the execution of a SER, these data objects can only interact with the heap through reads/writes of primitive-type values; no references are allowed to be written into these buffers.

3.6 The Gerenuk Runtime

The Gerenuk Serializer. As the first component of Gerenuk's runtime, we implemented a new serializer/deserializer using a similar algorithm to existing serializers such as Kryo. Since our transformation is based on the statically computed offsets, we need to guarantee that the way our compiler computes these offsets is consistent with how data is actually serialized.

Our algorithm is standard —it recursively traverses the object graph starting from each given top-level object and inlines the structure by copying, recursively, the primitive-type contents for each object into a buffer. Each top-level object has a special field storing the size of the entire data structured rooted at the object after inlining. Each array has a field storing its length. Pointers are all eliminated.

Determining Offsets. Recall that each object in a data structure rooted at T has an offset computed statically by our data structure analyzer (§3.3). At run time, the content of the object will be written into a native buffer at the location specified by the offset. However, if an object (say o) follows an array in the data structure, o's offset is represented as a symbolic expression that contains the length of the array as a variable. This expression will be resolved at run time by the function `resolveOffset`, shown in Algorithm 1.

One challenge in implementing `resolveOffset` is that if o is created *earlier* than the array during the execution, o's location cannot be determined because the length of the array is unknown. We solve the problem by caching o's content in a temporary buffer and later copying the content to the actual native buffer when that array is created and its length becomes available. This can be done in an *event-driven* manner. For any object whose statically-computed offset depends on the array length, we define a *handler* and register it with a runtime service that monitors the array creation. Upon the creation of the array, the service generates an event and sends it to all the handlers, which respond by re-evaluating the offsets and copying the data from the temporary buffers into the actual buffers.

Re-execution. One major challenge of implementing the re-execution logic is how to restore a program state. A traditional approach is to perform record/replay during the execution, which incurs heavy runtime overhead. Fortunately, our work targets tasks in data-parallel systems, which do not share state with each other. The dependences between different tasks form a dataflow graph where the output of one task becomes the input of another. If one task instance fails (e.g., *abort*), we can simply terminate the executor executing the instance and launch a new executor to execute the same task with the same input.

The question is how to guarantee that we can still have access to the original input. Since data objects in any native buffers are already reference-immutable (otherwise the execution would abort), a stronger constraint we need to add here is that we do not allow any values (including primitive-type values) to be written into the original buffers (i.e., Case 4 in Algorithm 1 is entirely prohibited). Note that this constraint is checked only on the *original input buffers* of the SER by the runtime system through write-protecting the virtual pages constituting these buffers. Upon a fault, the runtime aborts the SER, ensuring that the input buffers remain clean throughout the execution. Once a SER aborts, Gerenuk launches a new executor that executes the original version of the same (failed) task with the same input buffers.

This logic of launching the new executor needs to be specified by the Gerenuk user in a method called `launch`. It simply makes a few calls to launch a new executor (thread or process) to work on the same data input. This method is easy to write and application-independent.

It is clear that the roll-back logic has a performance penalty, because upon a roll-back, all computations up to the violation point would be wasted. However, our empirical evaluation shows that aborts are rare for real-world programs and datasets, and we thus did not observe any violation triggered during our experiments.

Memory Management. Another clear advantage of Gerenuk's transformation is that all data objects are guaranteed to stay in native memory and never escape to the heap; likewise, temporary and control objects all stay in the heap and can never flow into native memory. This property of the memory leads to easy and straightforward implementations of *region-based memory management and garbage collection* for data objects —we can simply deallocate the input native buffers for each task once a SER ends successfully. This added benefit of improved garbage collection would not cause safety issues because Gerenuk guarantees no heap objects can flow into these buffers.

4 Evaluation

We wrote approximately 35K lines of code in Java, Scala, and C++. Our compiler infrastructure is based on the Soot compiler framework, with additional support for analyzing Scala programs (in particular, lambdas).

To evaluate Gerenuk, we transformed Apache Spark [73], and Apache Hadoop [12], two most popular, widely-deployed Big Data systems. All of our experiments were run on an 11-node cluster, each with 2 Xeon(R) CPU E5-2640 v3 processors, 32GB memory, 1 200GB SSD, running CentOS 6.9 and connected via InfiniBand. All benchmarks were run for three times and the median value is reported. We also verified that no incorrect results were produced by our transformation.

4.1 Improving Apache Spark with Gerenuk

We evaluated Gerenuk using Spark 2.4.0, under Hadoop 2.9.2 and Scala 2.11.8. Overall, Gerenuk has transformed statements in 55 classes in Spark. The static analysis reports 126 violation points, none of which were triggered at run time. We used a set of five programs: PageRank (PR) and KMeans (KM) from the GraphX [35] libraries, as well as Logistic Regression (LR), Chi Square Selector (CS) and Gradient Boosting Classification (GB) from the MLlib libraries. These programs were selected due to their diverse computation and data types and can represent a large class of applications.

Note that we focus on iterative programs for our benchmark selection because iterative processing is the domain for which Spark was designed. One observation here is that the top-level data types T used in four of the five programs (except PageRank) have complex data structures that have 3 or 4 levels of objects connected by pointers. These data types cannot be stored directly in native memory by `DataFrame` and Tungsten. For PageRank that uses simple types, we compared our performance with that of Tungsten and our results are reported in §4.3.

Table 1 shows the details of the input of those programs. Each Spark worker was given all available cores (i.e., 32) on each machine. The JVM on each node was evaluated under 3 different heap sizes: 10GB, 15GB, and 20GB. Kryo, the recommended high-performance serializer, was used.

Running Time. Figure 6(a) shows the runtime comparison between Spark and Gerenuk. A summary of different aspects of improvements is reported in Table 3. On average, Gerenuk makes Spark run 1.96× faster. The majority of the savings comes from the reduced computation time. Since all (billions of) data objects are represented as inlined native bytes, not only is data locality improved, Gerenuk eliminates many sources of runtime overhead, including pointer chasing, write barriers (i.e., a piece of code executed per object write for GC purposes), and array bound checks.

Note that serialization and deserialization is not completely eliminated by Gerenuk, as shown in the purple and orange bars in Figure 6(a). Most of these costs are associated with sending *closures* (i.e., lamdas) from the driver to worker nodes, *not* data objects.

Reduction in the GC time is moderate (i.e., 37%). That is mostly because GC did not take much time in our experiments with Spark — on average, only 12% of the execution was spent on GC. In an environment where GC dominates the execution (e.g., when the data size is much larger than the heap size), we expect more savings in the GC time from a Gerenuk-transformed program.

A comparison among the three heap configurations shows that, as the heap size grows, the performance gain becomes less although the difference is only marginal. For example, when the heap size is 10GB, the Gerenuk-transformed programs achieve a 2.02× speedup, which is reduced slightly to 1.93× when each worker was given twice as much memory. This is expected since with a smaller heap, Spark programs had increased GC effort, while for Gerenuk-transformed programs, their working sets were allocated primarily in native memory and not subject to the GC. Hence, the performance of the original Spark is much more sensitive to the heap size than that of the transformed programs.

Memory. Comparisons of peak memory usage are reported in Figure 7(a). Since Gerenuk uses native memory while the original Spark uses managed heap, to enable a fair comparison, we periodically ran `pmap` to measure the process-level memory consumption and reported the maximum consumption in Figure 7(b). Across all benchmarks under different heap configurations, Gerenuk saves up to 38% of memory, with an average of 18%. These savings are primarily from the elimination of object headers and references.

4.2 Improving Apache Hadoop with Gerenuk

We evaluated Gerenuk on Hadoop's latest version (2.9.2) using two real-world datasets: the full StackOverflow and Wikipedia data dumps. Table 2 shows the details of the seven programs we used. These programs are also real-world programs that were uploaded/discussed by developers on StackOverflow for technical inquiries. We converted them (e.g., by adding some code to make them compilable) for use as our benchmarks.

Overall, Gerenuk transformed statements across a total of 22 classes. Similarly, more than a hundred violation points were generated by the static analysis, but none of them were triggered during execution. The max heap size for each mapper and reducer is 15GB and 30GB, respectively.

Name	Dataset (Size)	Data Type *T*
PageRank (PR)	LiveJournal (1GB)	String, Double
KMeans (KM)	Synthetic 600M points (30GB)	DenseVector
Logistics Regression (LR)	Synthetic 10M points each with 10 features (2GB)	LabeledPoint, DenseVector
Chi Square Selector (CS)	Synthetic 55M points, each with 28 features (37GB)	LabeledPoint, SparseVector
Gradient Boosting Classification (GB)	Synthetic 55M points, each with 28 features (37GB)	LabeledPoint, DenseVector

Table 1. Description of Spark programs.

Name	Dataset (Size)	Description
IUF [3]	StackOverflow (37GB)	Inactive Users Filtering
UAH [8]		Active User Activity Histogram
SPF [6]		Spam Posts Filtering
UED [4]		User Engagement Distribution
CED [2]		Community Expert Detection
IMC [7]	Wikipedia Data (49GB)	In-Map Combiner
TFC [5]		Term Frequency Calculation

Table 2. Our Hadoop programs and their descriptions.

Running Time. Figure 6(b) shows the time comparison between the original Hadoop programs and the Gerenuk-transformed programs. Table 3 also shows the performance of Gerenuk normalized to that of the baseline of across all benchmarks and all configurations.

The end-to-end speedup for Hadoop is, on average, 1.4×. Similar to Spark, Gerenuk's benefit is achieved primarily from the reduced computation, which dominates the execution (e.g., on average 96.5% of the total time). The time spent on computation is reduced by 26% when using the inlined native bytes. The gain here is smaller, compared to what is observed in Spark, because of the pervasive use of primitive types such as Long, Double, Integer and String in Hadoop.

Since the data types are simple and do not contain complex pointer usage, they have already been well-optimized in the Hadoop. For example, in shuffling, key-value pairs are already organized in a byte array in each buffer to be sorted. Due to these optimizations, the cost of serialization and deserialization is small even for the original programs. Although the Gerenuk-transformed programs do not serialize and deserialize any data, the savings here are minor.

Memory. Because our Hadoop programs created billions of small data objects (e.g., of primitive and string types), inlining their data contents brings significant reductions in memory usage due to the elimination of *all* headers of these objects. As can be seen in Figure 7(b), the peak memory consumption on a worker node is reduced by up to 42% (with an average of 31%).

FW	Overall	GC	App	Mem
Spark	0.28 ~ 0.93 (0.51)	0.44 ~ 0.89 (0.63)	0.28 ~ 0.93 (0.50)	0.62 ~ 0.92 (0.82)
Hadoop	0.51 ~ 0.87 (0.72)	0.23 ~ 0.87 (0.54)	0.49 ~ 0.88 (0.74)	0.58 ~ 0.84 (0.69)

Table 3. Summary of Gerenuk performance normalized to baseline in terms of **Overall** run time, **GC** time, **App**lication (non-GC) time, and **Mem**ory consumption across all settings. A lower value indicates better performance. Each cell shows a percentage range and its geometric mean.

4.3 Comparing with Existing Systems

Spark Tungsten and `DataFrame`**.** Project Tungsten is a major effort of Spark aiming to bring the system performance close to "bare metal". Tungsten introduces the `DataFrame` API that automatically organizes data in native memory and perform operations over them directly. At the first glance, Tungsten and `DataFrame` appear to be similar to what Gerenuk aims to achieve. However, the `DataFrame` API has a significant limitation in that it can only support simple data types that do not involve structures and pointers (so that an iterator can be constructed appropriately to traverse data records). Complex user-defined types, such as `DenseVector` and `SparseVector` used extensively in machine learning algorithms, cannot benefit from Tungsten at this moment.

In our benchmark suite, PageRank is the only program whose data type can be optimized under Tungsten. We rewrote PageRank by using the `DataFrame` API and ran it with Tungsten enabled. Note that to use `DataFrame` and Tungsten, Spark needs to dynamically generate query plans. This does not work well for iterative algorithms such as PageRank because the query plan can keep growing. This is a known and yet unresolved issue since the beginning of `DataFrame`[1]. During our experiment, the `DataFrame`-based implementation of PageRank failed to converge even after 15 hours (while the RDD-based implementation reached convergence in less than 250 seconds). To reduce the execution time, we had to fix the number of iterations to 10. The performance comparison between the original PageRank, Tungsten-enabled PageRank, and Gerenuk-transformed PageRank is shown in Figure 8(a).

[1] https://issues.apache.org/jira/browse/SPARK-13346.

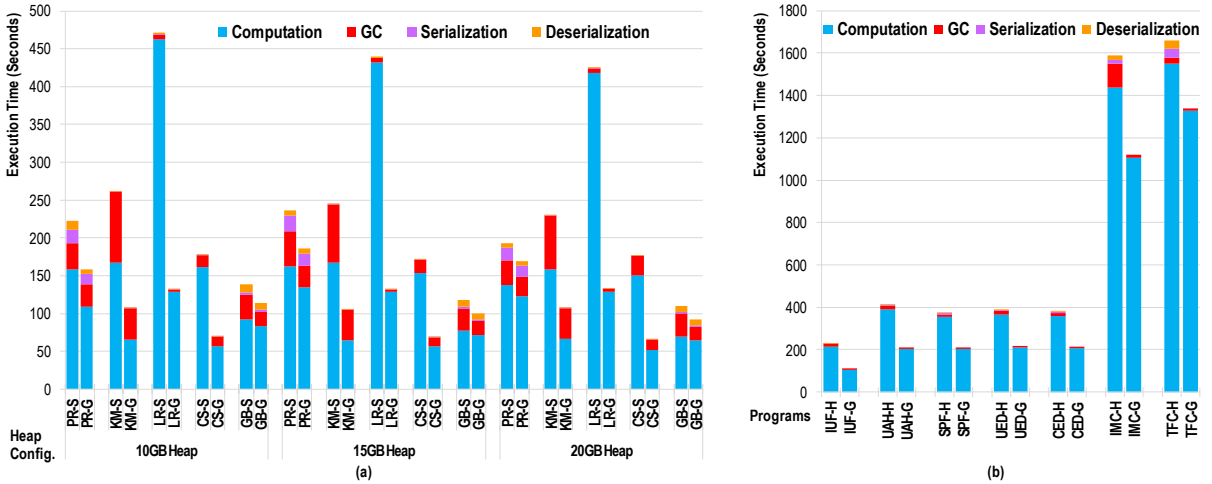

Figure 6. Runtime comparisons for Spark (a) and Hadoop (b); each group compares running time, for each program, between the baseline on the left and the **G**erenuk version on the right; each bar is further broken down into four components: computation (in blue), GC (in red), serialization (in purple), and deserialization (in orange).

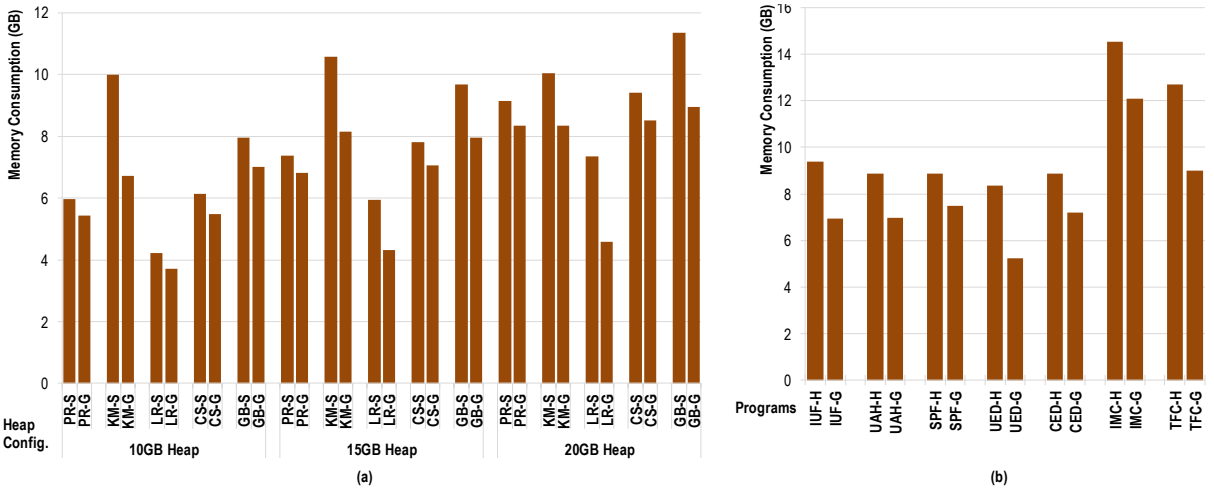

Figure 7. Peak memory comparisons between the baseline and the **G**erenuk version for Spark (a) and Hadoop (b); memory consumptions are collected by periodically running `pmap`.

The Gerenuk-transformed version is about 2.2× faster than the Tungsten-based version — the major savings come, again, from the reduced computation time. Since both Tungsten and Gerenuk use native memory, the amounts of GC efforts were comparable in these two versions.

To enable a fair comparison, we added a WordCount program that does not have iterative logic. A comparison between the original, the Gerenuk-transformed, and the Tungsten-enabled WordCount is shown in Figure 8(b). In this case, Tungsten outperforms Gerenuk by 20% primarily due to Tungsten's string optimizations, which are not performed in Gerenuk. However, these optimizations are designed for simple structured data while Gerenuk targets general user types and data structures.

Yak [57]. Yak is a Big-Data-friendly GC that is designed to enable region-based memory management for data objects. To understand Gerenuk's ability in reducing the GC cost,

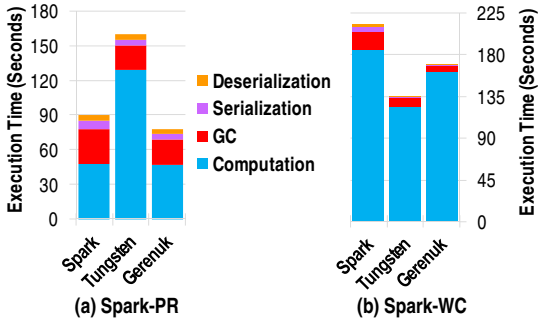

Figure 8. Comparison with Tungsten.

we have compared Gerenuk extensively with Yak. To ensure Yak is at its best performance, we obtained Yak's source code from the authors of [57] and replicated the environment described in [57]. We used the StackOverflow dataset as input and annotated the program to enable the Yak optimizations (i.e., by putting `epoch_start()` in the Hadoop method

`setup()` and `epoch_end()` in the Hadoop method `cleanup()`). We also followed the heap configurations described in [57]: for each map and reduce worker, we used two different heap configurations: 3GB (map) + 2GB (reduce) and 2GB (map) + 1GB (reduce). The experiment was done on the same cluster as described earlier in this section.

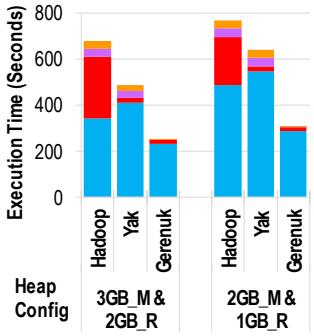

Figure 9. Comparison with Yak using Hadoop IMC.

Due to space constraints, we only report the result for the program IMC here. For the other programs, similar behaviors were observed. Figure 9 shows the execution time comparison for Hadoop between using the Parallel-Scavenge GC (i.e., the default GC in OpenJDK 8), Yak, and Gerenuk. Clearly, the GC cost for the Gerenuk-transformed IMC is lower than that of both Parallel-Scavenge and Yak. Compared to Parallel-Scavenge, the amount of GC effort during the execution of the transformed program was reduced by a factor of 13.7×. Compared with Yak, Gerenuk further reduces the GC time by 19%. This is because we do not need to scan objects when performing deallocation — our compiler guarantees that control objects can never flow into native buffers while Yak does not have such guarantees.

Furthermore, Gerenuk reduces computation significantly while Yak incurs additional overhead due to the cost of its write barrier that is executed per object write to record inter-region references. In particular, Gerenuk reduces the computation time by 45.6% compared to Yak and 35.6% compared to Parallel-Scavenge. The costs of serialization and deserialization, which take about 10-13% of the total time in Parallel-Scavenge and Yak, have been *completely* eliminated in Gerenuk. Overall, the Gerenuk-transformed program runs 2.4× faster than Parallel-Scavenge and 1.8× faster than Yak.

Facade [58] and Flare [29]. Facade transforms programs at the class granularity and relies on the human developer to refactor the program to create a clear boundary between the control and data path. We could not successfully compile Spark and Hadoop using Facade due to Facade's inability to support modern language features such as lambdas and due to its heavy requirement to manually refactor code.

Flare [29] is a compiler-based optimization technique for Apache Spark, which can generate code that processes data directly from optimized file formats. Although it works only for a single-machine environment, we also wanted to compare Gerenuk with it to understand performance differences. However, Flare's source code is not publicly available. Based on the results reported in [29] (e.g., from dozens to hundreds of times of speedup), we speculate that Flare would outperform Gerenuk on a single machine. However, Flare has limited generality because it does not work at all for a distributed environment.

4.4 Overhead of SER Aborts

With the programs we took directly from various libraries and online forums, we did not observe any SER aborts. To understand the impact of aborts, we found a Stack Overflow analytics application that uses more complex data types. This application powers a study on collaborative knowledge exchange over a variety of topics from socio-technical sites like StackOverflow [60]. The application has two phases: the first phase constructs a database of all posts grouped by user ID and the second phase uses NLP libraries for concept discovery. In terms of program transformation, we are interested only in the first phase. Internally, the application uses a complex user-defined type called `Account`, which represents a user and her posts. All of a user's posts are stored in a `Vector`. A violation is detected in its `resize` method — when its internal array overflows, a new array needs to be created to replace this array and this replacement involves a reference write, prohibited by our second violation condition. Hence, an *abort* instruction is inserted before writing the new array into the vector object.

Note that several other programs in our benchmark set also use vectors and hence aborts are inserted at similar resizing points. However, these programs perform machine learning tasks and their vectors contain features that do not grow during execution. Hence, none of the executions of these programs triggered these aborts. This new application, on the contrary, keeps collecting posts that belong to the same user and adding them into the vector. Hence, an abort is triggered upon each array resizing.

Figure 10(a) reports the results of running time. Overall, about 10% of all `Vector` instances required resizing, triggering aborts. With these violations and re-executions, the Gerenuk-transformed version is 7% slower than its original counterpart. The increase was largely due to wasted computations. We did not observe large increase in other runtime components such as GC, serialization, and deserialization.

To further investigate the cost of aborting a SER, we manually forced SERs to abort at arbitrary points during the execution of PageRank. We varied the number of aborts from 1 all the way to 20 (i.e., which is about 50% of the total number of iterations in PageRank) and measured various aspects of performance on the same cluster where each Spark worker was given a 20GB heap. Figure 10(b) shows the running time comparison. The leftmost bar represents the performance of the original program under vanilla Spark, followed by 8

bars showing the performance of the Gerenuk-transformed program with 0 – 20 re-executions.

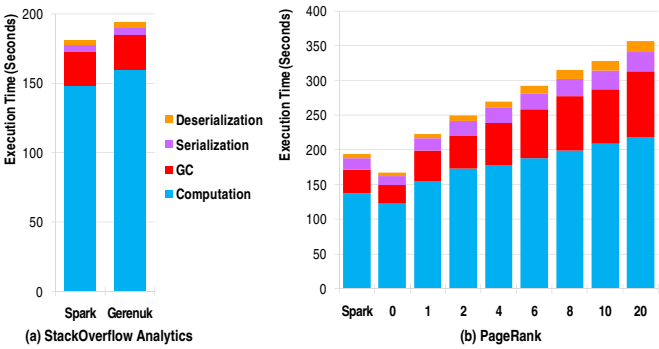

Figure 10. Overhead of re-executions in running time for StackOverflow Analytics (a) and PageRank (b).

Overall, each re-execution incurs a 9% overhead compared to a SER in the original Spark, and a 14% overhead compared to a SER in the Gerenuk-transformed Spark. This is primarily due to the round-trip format changing — each re-execution starts with a costly deserialization process that transforms all data in native memory into heap objects. As can be seen in Figure 10(b), the amounts of time spent on serialization and deserialization increase by 2× and 3×, respectively. We also observed significantly increased GC effort. In the original Spark, the GC takes about 19% of the execution time. However, with re-executions, the heap contains orders of magnitude more objects deserialized from the bytes, putting severe pressure on the GC. The GC time grows to 30.3% (with the average of 26.1%) of the execution time. The peak memory consumption also increases by 11%, compared to the original Spark run.

5 Related Work

Data-Parallel Systems. Numerous optimizations [9–11, 18, 21, 22, 24–27, 31, 37, 40, 43, 47, 53, 59, 61–64, 66, 67, 71, 72, 74] have been attempted from various research communities to improve the performance of Big Data systems. Since most Big Data systems were developed in managed languages, recent works [30, 34, 51, 52, 56–58] attempt to remove the penalty incurred by the managed runtime. Skyway [56] is a JVM-based technique that can improve the performance of serialization/deserialization by providing support to transfer objects *as is*. Closest to our work are the two compiler techniques Facade [58] and Deca [50]. As discussed earlier in §1, these they are both conservative transformation techniques that require heavyweight code refactoring from the user. Flare [29] also tries to transform Spark program into C program but it is limited to only the Spark system. Gerenuk, on the contrary, performs program transformation speculatively based on a transaction model, which significantly improves the transformation practicality.

Niijima [70] is an optimizing compiler that automatically consolidates C# computations on a SQL pipeline to reduce the cost of serialization and deserialization when data is passed from the native (.NET) to the .NET (native) runtime. While Gerenuk is also a compiler-based approach, we achieve soundness by inserting aborts at potential violation points instead of transforming the whole program. Since Gerenuk targets large Java systems, it is much more difficult to guarantee soundness for our transformations than for those performed by Niijima on SQL pipelines.

Compiler Optimizations and Static Analysis. Traditional optimization techniques [16, 23, 32, 33, 38, 68, 69] for object-oriented programs use various approaches to reduce the number of heap objects and their management costs. Free-Me [38] is a compiler-based technique that adds compiler-inserted frees to a GC-based system. Pool-based allocation proposed by Lattner et al. [44–46] uses a context-sensitive pointer analysis to identify objects that belong to the logical data structure and allocate them into the same pool to improve locality. Prolific types [65] is a static technique that splits objects into a prolific and a non-prolific region to reduce the GC cost. However, these compiler analyses are not designed for Big Data systems that exhibit strong iterational behaviors. Object inlining [28, 49] is a technique that statically inlines objects in a data structure into its root to reduce the number of pointers and headers. While object inlining offers significant performance benefit, existing inlining techniques are impractical as they give up on transforming a program upon finding a violation. By contrast, Gerenuk uses a speculative execution model that does optimistic transformation while aborting speculative executions when necessary for safety.

6 Conclusion

This paper presents Gerenuk, a compiler and runtime that enables data-processing programs to work with the native inlined representation of data items. The Gerenuk compiler transforms a program speculatively based on the assumption that user-defined types are immutable and confined. To guarantee safety, Gerenuk statically detects violations, at which *abort* instructions are inserted. An evaluation on Hadoop and Spark shows that our transformation can significantly improve various aspects of their performance.

Acknowledgements

We thank the SOSP reviewers for their thorough and insightful comments. We are especially grateful to our shepherd Angela Demke Brown for her feedback. This work is supported by National Science Foundation grants CCF-1460325, IIS-1546543, CNS-1514256, CCF-1527923, CNS-1613023, CNS-1703598, CCF-1723773, CCF-1764077, CNS-1763172, CNS-1764039, and OAC-1740210, and Office of Naval Research grants N00014-16-1-2913 and N00014-18-1-2037. Christian Navasca acknowledges a travel grant awarded by NSF. Khanh Nguyen is supported in part by a Google Ph.D. Fellowship.

References

[1] 2006. Soot Framework. http://www.sable.mcgill.ca/soot.

[2] 2012. Join operation with MapReduce. https://stackoverflow.com/questions/4053857.

[3] 2015. Active/Inactive Users. https://stackoverflow.com/questions/49442420.

[4] 2015. Count Number of Posts. https://stackoverflow.com/questions/39030644.

[5] 2015. Error in Computing Frequencies. http://stackoverflow.com/questions/23042829.

[6] 2015. Join to Filer Spams. https://stackoverflow.com/questions/29622750.

[7] 2015. The Performance Comparison between In-Mapper Combiner and Regular Combiner. http://stackoverflow.com/questions/10925840.

[8] 2015. User Activity. https://stackoverflow.com/questions/33411920.

[9] Foto N. Afrati and Jeffrey D. Ullman. 2010. Optimizing joins in a map-reduce environment. In *EDBT*. 99–110.

[10] Parag Agrawal, Daniel Kifer, and Christopher Olston. 2008. Scheduling shared scans of large data files. *Proceedings of VLDB Endow.* 1, 1 (2008), 958–969.

[11] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. 2014. ShuffleWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters. In *USENIX ATC*. USENIX Association, 1–13.

[12] Apache 2017. Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org.

[13] Apache 2017. The Hive Project. http://hive.apache.org.

[14] Apache Flink 2017. Apache Flink. http://flink.apache.org/.

[15] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *KDD*. 44–54.

[16] B. Blanchet. 1999. Escape Analysis for Object-Oriented Languages. Applications to Java. In *OOPSLA*. 20–34.

[17] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW*. 595–601.

[18] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*. 1151–1162.

[19] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J. Carey. 2013. A Bloat-Aware Design for Big Data Applications. In *ISMM*. 119–130.

[20] Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB* 1, 2 (2008), 1265–1276.

[21] Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.* 1, 2 (2008), 1265–1276.

[22] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI*. 363–375.

[23] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. 1999. Escape Analysis for Java. In *OOPSLA*. 1–19.

[24] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce online. In *NSDI*. 21–21.

[25] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2014. Exploiting Bounded Staleness to Speed Up Big Data Analytics. 37–48.

[26] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[27] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. 2010. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of VLDB Endow.* 3 (2010), 515–529.

[28] Julian Dolby and Andrew Chien. 2000. An automatic object inlining optimization and its evaluation. In *PLDI*. 345–357.

[29] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *OSDI*. 799–815.

[30] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. 2015. Interruptible Tasks: Treating Memory Pressure As Interrupts for Highly Scalable Data-Parallel Programs. In *SOSP*. 394–409.

[31] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2014. Making State Explicit for Imperative Big Data Processing. 49–60.

[32] D. Gay and B. Steensgaard. 2000. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In *CC*. 82–93.

[33] Ovidiu Gheorghioiu, Alexandru Salcianu, and Martin Rinard. 2003. Interprocedural compatibility analysis for static object preallocation. In *POPL*. 273–284.

[34] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G. Murray, Steven Hand, and Michael Isard. 2015. Broom: Sweeping Out Garbage Collection from Big Data Systems. In *HotOS*.

[35] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*. 599–613.

[36] Google. 2017. Orkut social network. http://snap.stanford.edu/data/com-Orkut.html.

[37] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. 2012. Spotting code optimizations in data-parallel pipelines through PeriSCOPE. In *OSDI*. 121–133.

[38] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. 2006. Free-Me: a static analysis for automatic individual object reclamation. In *PLDI*. 364–375.

[39] UC Irvine. 2014. Hyracks: A data parallel platform. http://code.google.com/p/hyracks/.

[40] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*. 59–72.

[41] Kryo 2017. The Kryo serializer. https://github.com/EsotericSoftware/kryo.

[42] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *WWW*. 591–600.

[43] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*. 31–46.

[44] Chris Lattner. 2005. *Macroscopic Data Structure Analysis and Optimization*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[45] Chris Lattner and Vikram Adve. 2005. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI*. 129–142.

[46] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *PLDI*. 278–289.

[47] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. 2011. YSmart: Yet Another SQL-to-MapReduce Translator. In *ICDCS*. 25–36.

[48] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *CC*. 153–169.

[49] Ondrej Lhotak and Laurie Hendren. 2005. Run-time evaluation of opportunities for object inlining in Java. *Concurrency and Computation: Practice and Experience* 17, 5-6 (2005), 515–537. https://doi.org/10.1002/cpe.848

[50] Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. 2016. Lifetime-based Memory Management for Distributed Data Processing Systems. *Proc. VLDB Endow.* 9, 12 (2016), 936–947.

[51] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. 2015. Trash Day: Coordinating Garbage Collection in Distributed Systems. In *HotOS*.

[52] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. 2016. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *ASPLOS*. 457–471.

[53] Derek Gordon Murray, Michael Isard, and Yuan Yu. 2011. Steno: automatic optimization of declarative queries. In *PLDI*. 121–131.

[54] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: A Timely Dataflow System. In *SOSP*. 439–455.

[55] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-tolerant Software Distributed Shared Memory. In *USENIX ATC*. 291–305.

[56] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: Connecting Managed Heaps in Distributed Big Data Systems. In *ASPLOS*. 56–69.

[57] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *OSDI*. 349–365.

[58] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS*. 675–690.

[59] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. 2010. MRShare: sharing across multiple queries in MapReduce. *Proceedings of VLDB Endow.* 3, 1-2 (2010), 494–505.

[60] Nigini Oliveira, Michael Muller, Nazareno Andrade, and Katharina Reinecke. 2018. The Exchange in StackExchange: Divergences Between StackOverflow and Its Culturally Diverse Participants. *Proc. ACM Hum.-Comput. Interact.* 2, CSCW, Article 130 (Nov. 2018), 22 pages.

[61] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. 2008. Automatic optimization of parallel dataflow programs. In *USENIX ATC*. 267–273.

[62] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*. 1099–1110.

[63] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. 2005. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.* 13, 4 (2005), 277–298.

[64] Douglas Santry and Kaladhar Voruganti. 2014. Violet: A Storage Stack for IOPS/Capacity Bifurcated Storage Environments. 13–24.

[65] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. 2002. Exploiting Prolific Types for Memory Management and Optimizations. In *POPL*. 295–306.

[66] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.* 2, 2 (2009), 1626–1629.

[67] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*. 996–1005.

[68] J. Whaley and M. Rinard. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *OOPSLA*. 187–206.

[69] Guoqing Xu, Dacong Yan, and Atanas Rountev. 2012. Static Detection of Loop-Invariant Data Structures. In *ECOOP*. 738–763.

[70] Guoqing Harry Xu, Margus Veanes, Michael Barnett, Madan Musuvathi, Todd Mytkowicz, Ben Zorn, Huan He, and Haibo Lin. 2019. Niijima: Sound and Automated Computation Consolidation for Efficient Multilingual Data-Parallel Pipelines. In *SOSP*.

[71] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. 2007. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*. 1029–1040.

[72] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*. 1–14.

[73] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *HotCloud*.

[74] Jingren Zhou, Per-Ake Larson, and Ronnie Chaiken. 2010. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *ICDE*. 1060–1071.