Compiler and Runtime Supports
for Efficient and Scalable Big Data Systems

Research Statement

Khanh Nguyen
University of California, Los Angeles
khanhtn1@ucla.edu

Abstract

Big Data analytics applications such as social network analysis and web analysis have revolutionized modern computing. The processing demand posed by an unprecedented amount of data challenges both industrial practitioners and academia researchers to design and implement highly efficient and scalable system infrastructures. Unfortunately, Big Data processing is fundamentally limited by memory inefficiencies inherent with the underlying programming languages. While offering several invaluable benefits, a managed runtime comes with time and space overheads. In large-scale systems, the memory management cost can be easily magnified and become the critical performance bottleneck. Throughout my Ph.D., I have designed and developed a series of system optimizations to enable scalable Big Data processing including a new programming model, and several novel compiler and runtime supports. In the future, I plan to continue addressing the low-performance issue in data-intensive systems by developing practical solutions across the whole software stack, spanning from the processing model to language extensions.

1 Introduction

The past decade has witnessed the explosion of “Big Data”, a phenomenon where massive amounts of information are created at an unprecedented scale and speed. As a reference data point, former Google CEO Eric Schmidt, at the Techonomy’10 conference, shared that the accumulation of data generated in just two days could dwarf the total information that had been created in all human history up until 2003.

The availability of an enormous amount of data has lead to the proliferation of large-scale, data-intensive applications. Companies, government organizations, and academic institutions have increasing demands for scalable software systems that can quickly analyze data of massive scale at petabyte or higher to discover patterns, trends, and associations, especially those relating to human behaviors and interactions.

The mainstream approach to scalability is to enable distributed processing using a large number of machines in clusters or in the cloud. An input dataset is split among machines so that many processors can work simultaneously on a computing task. Popular Big Data systems include, to name a few, Hadoop [4], Spark [40], Naiad [22], Hyracks [7], Giraph [3], Hive [5], and Storm [6]. Often, these Big Data systems are developed in managed languages, such as Java, C#, or Scala. This is primarily because these languages 1) enable fast development cycles due to simple usage and automatic memory management and 2) provide abundant library suites and community support.
However, a managed runtime comes at a cost [13, 21, 24, 25, 26, 27, 32, 33, 34, 35, 36]: memory management in Big Data systems is often prohibitively expensive. These systems commonly suffer from severe memory problems, causing low performance and scalability. Allocating and deallocating a sea of data objects puts a severe strain on the runtime system, leading to high memory management overheads, prolonged execution time, and an inability to process even datasets of moderate sizes. For example, Bu et al. [8] reports an average-size application on Giraph [3], an Apache open-source graph analytics framework initiated by Yahoo!, cannot successfully process 1GB of input data despite having 12GB of memory available.

While there exists a large body of techniques that can improve Big Data performance [1, 2, 7, 9, 10, 11, 14, 15, 18, 28, 29, 30, 31, 38, 39, 41], they focus on horizontal scale-up, i.e., scaling Big Data to data-center-level of machines, assuming data processing on each machine yields satisfactory performance. However, in [26] we show this simple assumption does not hold in the real world. In almost all cases, the data-processing program running on a worker machine suffers from extensive memory pressure – the execution pushes the heap’s limit soon after it starts and the system struggles with finding allocation space for new objects throughout the execution. Consequently, data-intensive applications crash due to out-of-memory errors in a surprisingly early stage. Even if a program runs successfully to the end, its execution is usually dominated by garbage collection (GC) effort – e.g., 40-60% of the end-to-end time [8, 13, 25, 26]. The problem becomes increasingly severe in latency-sensitive distributed cloud applications such as web servers or real-time analytics systems, where even one low-performance node can hold up the whole cluster, delaying the processing of user requests for an unacceptable long time [19, 20].

Unlike the mainstream approach which solves scalability issues by spending more resources (e.g., machines or memory), my research tackles this problem from a different angle. My work, up to now, focuses on systematically improving performance of each individual processing unit, i.e., enabling scaling-up vertically. By optimizing each worker node, large performance gains can be expected in clusters at sizes such as Google’s data centers. Moreover, with shorter execution time, more efficient data-processing, data centers will also have significant energy savings, leading to economic (e.g., reduced costs) as well as social benefits (e.g., decreased CO₂ emission).

In the remainder of this statement, I will first briefly introduce my existing work addressing this pressing scalability problem in Section 2. I then discuss potential future research projects in Section 3. Finally, Section 4 concludes this statement.

## 2 Existing Work

In this Section, I will present several runtime causes that prevent scalability in state-of-the-art Big Data systems, and my solution to address each of them, respectively.

### Issue 1 — Memory bloat inherent in managed languages

The term *memory bloat* refers to the inefficiency of using large amounts of memory to store information which is not strictly necessary for the execution. Bloat commonly exists in modern enterprise computing, significantly affects applications’ scalability and performance [32, 33, 34, 36]. Specifically, bloat comes in the form of fixed-size header which each Java object must have to store information needed for memory management. Another major component of memory bloat is the massive amount of references generated in object-oriented data structures with multiple layers of delegations. This space overhead impact is significantly magnified in a Big Data application which often contains a great number of objects. The space overhead cannot be amortized by the actual data content.

Moreover, a typical tracing GC periodically traverses the live object graph; its cost grows with the number of objects and references in the heap. If an object is created for each data item in a Big Data application, the number of objects and references grow proportionally with the cardinality of the massive-
scale input dataset. When the heap becomes large (e.g., a few dozens of GBs) and most objects in the heap are live, a single GC invocation can become exceedingly long. In addition, because the size of input data set of a Big Data application is often orders of magnitude more than the heap size, the heap is constantly being used and exhausted. Therefore, the GC is frequently triggered and eventually becomes a major bottleneck that prevents mutator threads from making satisfactory progress.

My solution I published in ASPLOS’15 my first work in this area, FACA DEC [26]. It is a non-intrusive approach – no modification of the Java Virtual Machine (JVM) is required. FACA DEC targets the performance problem caused by excessive object creation in an object-oriented Big Data application. FACA DEC has a compiler and a runtime system that perform semantics-preserving code transformation and native memory management, respectively, to statically bound the number of data objects created in Big Data applications. Developers are asked to annotate classes for the runtime to determine whether an object should be allocated in the heap or in FACA DEC-managed native memory pages. The key insight contributing to the success is that in the generated code, the number of heap objects representing data items does not grow proportionally with the cardinality of the dataset. Instead, it is statically bounded regardless of how much data an application has to process. FACA DEC is novel by breaking the long-held programming principle that advocates the use of object to both represent and manipulate data. FACA DEC-generated programs are shown to be more (time and memory) efficient (up to 50% reduction) and can scale to 4 \times \text{larger datasets comparing to their object-based counterparts.}

Issue 2 — Two vastly different types of object lifetime behavior

The memory management cost in Big Data systems is expensive. As discussed earlier, not only GC is triggered excessively, a single GC run can take a long time. Based on our experience with dozens of data-intensive frameworks, we discover a critical reason for slow GC execution: object characteristics in Big Data systems do not match the heuristics employed by state-of-the-art GC algorithms. The key observation is that, a typical data-processing framework has a clear logical distinction between a control path and a data path. The control path performs cluster management and job scheduling as well as interacts with the outside world. This path, despite having complicated logic, does not create many objects. Meanwhile, the data path consists of user-provided data manipulation functions such as Map or Reduce. Comparing to the control path, this path is order of magnitude simpler, but is the main source of object creation (e.g., 95% of all runtime objects [8]).

These two paths follow different heap usage patterns. While the conventional generational hypothesis holds for the control path (i.e., most recently allocated objects are also most likely to become dead), it does not hold for the data path, where most objects are created. In fact, objects allocated in the data path often exhibit strong epochal behavior: they are created en masse at the beginning of an epoch – a long computational event, and stay alive throughout the epoch.

This mismatch leads to a fundamental challenge encountered by state-of-the-art GCs, which are built based on the generational hypothesis in Big Data applications. Since newly-created objects often do not have short lifetime, most GC runs spend significant amount of time to identify and move live objects, yet reclaim very little memory space. This also explains why the overall management cost is prohibitively high in Big Data systems.

My solution My OSDI’16 work [25] seeks to alleviate the hypothesis mismatch by intelligently adapting the heuristics of GC to object characteristics in Big Data systems. The solution is a system with tailored GC algorithms for data-intensive applications called Yak. It enables efficient handling of the large volume of objects in Big Data systems by combining the state-of-the-art generational GC with region-based memory management technique.

Yak is the first hybrid GC that combines two seemingly-different styles of memory management into one cohesive system. It provides high throughput and low latency for all JVM-based languages such as
Java, C#, and Scala. Yak divides the managed heap into a control space and a data space, based on the two very distinct object behaviors (generational and epochal) observed in modern data-intensive workloads. Yak treats the data space and control space differently. Yak allocates data-space objects in epoch-based regions which are deallocated as a whole at the end of an epoch, while efficiently tracks a small number of objects whose lifetimes span across region boundaries. Control-space objects, meanwhile, are managed by the conventional generational GC. Doing so greatly reduces the memory management cost in Big Data systems. Yak outperforms the default production GC (i.e., the Parallel Scavenge) in Oracle’s JVM OpenJDK on three widely-used real Big Data systems: Hadoop [4], Hyracks [7] and GraphChi [16], while requiring minimal user effort — users are required to provide lightweight code annotation, a task that can be easily done in minutes by novices. Yak can deliver up to $7 \times$ reduction in overall execution time for Big Data applications.

**Issue 3 — Overly parallel execution**

Many algorithms used in Big Data applications are inherently data-parallel: the dataset can be readily decomposed into many disjoint parts and processed in parallel. As multi-core machines are increasingly being used, developers tend to statically create a large number of worker threads to fully utilize the underlying parallelism. However, threads do not come for free: cost from object-oriented data structures easily gets duplicated upon the creation of a new thread, making many programs crash with an out-of-memory error. Using too few threads reduces the risk of running out of memory, but at the cost of sacrificing processing speed.

Practical solutions for performance center around making “best-practice” recommendations for manual tuning of framework parameters. However, the tuning process is highly time-consuming and labor-intensive as it is impossible to find a one-size-fit-all solution even for a single application on various datasets. Striking a balance of the degree-of-parallelism and the application’s performance dynamically is a great challenge due to the self-centered characteristic of the worker threads — each thread operates in isolation, being oblivious to the runtime system’s state. Hence, threads are in a fierce competition for resources (e.g., memory, CPU time) to complete the assigned computation. With many aggressive, heap-space-hungry worker threads, the runtime system constantly struggles to find memory by invoking GC excessively, and eventually crashes due to insufficient memory to satisfy all running threads.

**My solution** I co-authored the ITask work which was published in SOSP’15 [13]. ITask is a unique, systematic approach to make data-parallel tasks work cooperatively in Big Data systems. Inspired by how processors handle hardware interrupts, we advocate interruptible task, a new type of data-parallel tasks that can be interrupted when the system is in memory shortage and reactivated later. When a task is interrupted, its consumed memory is reclaimed, giving more resources (e.g., memory, CPU time) for other tasks to proceed to completion. The work consists of a novel programming model that can be used by developers to turn an existing task into an interruptible task with minimal restructuring effort. It also has a runtime system that proactively monitors system states, automatically performs task interruption as well as auto-tunes the degree of parallelism to reduce memory usage. Using ITask, Big Data systems have improvement both in scalability (3-24 $\times$ larger dataset) because out-of-memory crashes are avoided and performance (1.5-3 $\times$ faster) due to significantly reduced GC time.

**Issue 4 — Inefficient data transfer mechanism**

Modern Big Data systems need to frequently shuffle data in the cluster. Under a managed runtime, transferring an object across nodes is complicated and very inefficient, consisting of two procedures as follows: (1) A serialization procedure turns the whole object graph reachable from the root object into a binary sequence. This procedure reformats each object – among other things, it extracts the object data, strips the object header, removes all references stored in an object, and changes the representation of certain meta data. (2) A deserialization procedure reads out the byte sequence, creates objects accordingly, and eventually rebuilds
the object graph in the managed heap of the receiver machine. While many serialization/deserialization (S/D) libraries have been developed, large inefficiencies exist in their implementations. Both our own experience and evidence from previous work show that S/D accounts for 30% of the execution time.

**My solution** The key problem with existing S/D libraries is that, with an existing JVM, there are no alternative routes to transfer objects other than first disassembling and pushing them down to a different binary format, and then reassembling and pulling them back up into a remote heap. I advocate to build a connection between managed heaps so that data objects no longer need to be pushed down to a lower level for transfer. I developed Skyway (ASPLOS’18) [23], the first JVM-based technique that can directly connect managed heaps of different (local or remote) JVM processes. Under Skyway, objects in the source heap can be directly written into a remote heap without changing their formats. Skyway enhances the JVM, and enables object graphs to be moved as-is from heap to heap and used on a remote node right after the move. With very small manual user effort, Skyway provides performance benefits to any JVM-based system by completely eliminating the need of invoking expensive reflective serialization/deserialization functions, thus saving CPU time. Our evaluation shows that Skyway outperforms all existing S/D libraries and improves execution time of the widely-deployed systems Spark (by 16-36%) and Flink (by 19%).

## 3 Future Work

As shown in Section 2, my research is motivated by real problems, yields novel solutions, and has practical impact. Moving forward, I plan to keep exploring this high-performance, Big Data processing area. Some of my potential research directions are as follows:

**Statistically-sound reduction of datasets** Currently, when one needs to analyze an enormous dataset (e.g., Wikipedia data dump, the webgraph [37]), she would have to perform computation in a trial-and-error manner. That is, she would find a cluster that she thinks is capable of processing the dataset, and run the analytic tasks. She then waits to see whether the cluster can finish the computation. If the cluster is unfortunately incapable, she would waste hours or even days before knowing that fact. The situation would be different if at first, she performs a “scalability” test using the same cluster with a statistically-sound reduced set. This set must faithfully reflect important features of the original dataset to ensure similar computational behavior. Only when the test gives assurance of successful processing of the original dataset would the developer start the real computation. This approach will help to avoid the waste of developer’s time and computation power considerably.

**Auto-synthesis of out-of-core data structures** Many frameworks now support out-of-core computations (such as spilling in Hadoop and Spark) to reduce memory pressure. However, there is a lack of support for out-of-core data structures. Adding more memory into a machine to solve scalability issues is ineffective: while dataset size doubles every two years, memory capacity doubles every three years [12, 17]. Meanwhile, disks have become faster (e.g., 500MB/s or more per read/write for a consumer-grade solid state drive (SSD)). In addition, there already exists a huge library of in-memory data structures. Manually transforming each of these into a disk-based representation is not desired. The process is highly error-prone and labor-intensive to achieve good performance. Data-intensive applications can gain significant benefits from having the support of an auto-synthesizer that can generate customized out-of-core data structures.

**Big-Data-friendly programming model** Even though there is a clear logical separation of a control and a data path (Section 2–Issue 2) in Big Data applications, these two paths are usually tightly coupled physically in the codebase. While my OSDI’16 work [25] has been shown to be practical and successful in improving performance for Big Data applications by exploiting the distinct behaviors of these two paths, there is a drawback. That is, it is very intrusive: the JVM is modified significantly. This downside could prevent the adoption of the system in cases where the integrity of the JVM is security-sensitive. I plan to
develop a programming model and a library-based runtime support, i.e., a non-intrusive approach that can exploit the differences of the control path and the data path in Big Data systems.

**Unique optimizations for distributed Big Data applications** My work has largely focused on individual node performance. While it has been demonstrated that the solutions have been very effective, there is much room for exploration and improvement in the context of distributed environment. For example, stragglers – tasks that take unexpectedly long – are quite common in large-scale systems. How to detect and eliminate stragglers is still an open problem.

**Compiler-assisted optimizations using emerging hardware technologies** As new hardwares are emerging, e.g., *non-volatile memory, NUMA*; the exploitation of such innovations in software is usually delayed. Software developers might not understand fully the hardware advances to use, or be resistant to changes (due to the existence of legacy code or other constraints). However, if the hardware-based optimizations can be done *automatically and efficiently* by a compiler, this is a win-win situation for every entities involved. I plan to investigate and develop a line of compiler-assisted optimizations that can exploit new hardwares for performance.

## 4 Conclusion

Since its birth, Big Data has been evolving in all aspects: volume (data size), velocity (speed of change), variety (forms of data source), veracity (uncertainty of data), and value (ability to monetize). In order to process growing datasets, efficiency on all levels across the computing stack is required. As Big Data will continue to be a major challenge faced by both researchers and practitioners for the foreseeable future, I plan to continue my research in developing practical and impactful application and system solutions.
References


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


