1 Overview and Research Impact

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. The availability of an enormous amount of data has led 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, Giraph, Hive, Hyracks, Naiad, Spark, and Storm. 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: 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 overhead, prolonged execution time, and inability to process datasets of even moderate sizes.

While there exists a large body of techniques that can improve Big Data performance, they focus on horizontal scaling, i.e., scaling data processing to a large cluster of machines with an assumption that the processing task on each machine yields satisfactory performance. However, our experience [8] shows that, in many 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), which can take 40-60% of a job’s end-to-end execution time. The problem becomes increasingly severe in latency-sensitive distributed cloud applications such as web servers or real-time analytical systems, where one low-performing node can hold up the entire cluster, delaying the processing of user requests for an unacceptably long time.

Unlike the mainstream approach which tackles the scalability problem by spending more resources (e.g., machines or memory), we approach this problem from a different perspective. My research focuses on systematically improving the performance of each data-processing program, i.e., enabling vertical scaling. By optimizing the program running on each worker node, large performance gains can be expected in a data center. With the reduction of data processing time on each node, the data center is expected to have increased throughput and reduced energy consumption, leading to economic benefits.

Throughout my Ph.D. studies, I have designed and implemented a series of techniques, spanning programming model, compiler, and runtime system, that can significantly improve the efficiency of various aspects of Big Data processing. My work has attracted much attention from both academia and industry. The Facade system [8] inspired a line of follow-up works — for instance, MSR and University of Cambridge used the same idea [2] to improve Naiad, Microsoft’s dataflow engine; Oracle and UC Berkeley developed a coordinated GC system for latency-sensitive workloads [4, 5]; and the idea was also adopted by researchers from China, Denmark, and Hongkong to improve the RDD management in Spark [3, 10]. As another example, the Yak GC [7] I developed was recently commercialized by Huawei for use in their telecommunication systems.

2 My Ph.D. Research

The research I conducted during my Ph.D. studies centers around four important observations we made on widely-deployed Big Data systems. These observations reveal different kinds of inefficiencies in an existing
managed runtime system for processing large amounts of data, and motivate our proposals of novel ideas that can dramatically improve the runtime system for modern Big Data workloads.

**Observation #1 — Memory bloat inherent in the managed runtime**

The term *memory bloat* refers to the inefficiency of using large amounts of memory to store information that is not strictly necessary for the execution. Bloat commonly exists in modern enterprise computing, significantly affecting applications’ scalability and performance. One common source of bloat is a fixed-size header added to each Java object, which stores information needed for locking and memory management. Another major source of bloat is the massive volumes of references (pointers) used in object-oriented data structures to implement multiple layers of delegation. The impact of this space overhead can be significantly magnified in a Big Data application that often creates and uses an extremely large number of small objects (e.g., Integers and Strings) where it is difficult for the space overhead to get amortized across the actual data payloads.

Moreover, a typical tracing garbage collector 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 input dataset. When the heap becomes large (e.g., dozens to hundreds of GBs) and most objects in the heap are live, a single GC invocation can become exceedingly long. Due to the memory-intensive nature of a Big Data application, the GC is often frequently triggered and eventually becomes a major bottleneck that prevents the main processing threads from making satisfactory progress.

**My solution** I published in ASPLOS’15 my first work in this area, Facade [8]. Facade targets the performance problem caused by excessive object creation in an object-oriented Big Data application. It has a compiler and a runtime system — the compiler transforms the application code in a semantics-preserving manner to allocate and manage objects in the off-heap, native memory. Data records are allocated and stored in native memory and “facade” objects are created in the managed heap as proxies to bridge the program and natively-stored data records. The compiler transformation is done in a special way such that the number of data objects created in the transformed code can be “statically” bounded. As a result, the number of heap objects representing data items does not grow proportionally with the cardinality of the dataset. Instead, it is only related to certain static properties of the program, regardless of how much data an application has to process. Facade breaks the long-held principle in object orientation that objects are used to both represent and store data. In the generated code, heap objects serve only as representations and their corresponding payloads are stored natively, and hence, each object can be reused multiple times to represent different data items. A thorough evaluation with several widely-deployed systems shows that the Facade-generated programs are much more (time and memory) efficient and can scale to $4 \times$ larger datasets, compared to their object-based counterparts.

**Observation #2 — Two vastly different types of object lifetime characteristics**

The cost of memory management in Big Data systems can be exceedingly high. As discussed earlier, GC is frequently triggered and a single GC can take a long time to scan the heap. Our experience with dozens of data-intensive frameworks shows a critical reason for the poor GC performance — the characteristics of data objects in Big Data systems do not match the assumptions used by state-of-the-art GC algorithms. We observed 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. This path, although having complicated code logic, does not create many objects. Meanwhile, the data path consists of user-provided data manipulation functions such as Map, Reduce, and relational operations. Compared to the control path, this path has a much simpler code logic, but is the main source of object creation (e.g., 95% of all runtime objects are created there).

The object lifetime patterns in these two paths are completely different. The conventional generational hypothesis aligns well with the control path (i.e., most objects die young), but it does not hold for the data path where most objects are created. Instead, data objects 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 all upon the generational hypothesis. Since most data objects are long-lived, most GC runs spend a significant amount of time identifying and moving live objects, and yet each or them can reclaim only little memory. This explains the high cost of memory management in Big Data systems.

**My solution**  My OSDI’16 work [7] seeks to alleviate the hypothesis mismatch by intelligently adapting the heuristics of the GC to object characteristics in Big Data systems. Our solution is a new garbage collector, called Yak, tailored for data-intensive applications. It enables efficient handling of the large volume of objects in Big Data systems by marrying the state-of-the-art generational GC with region-based memory management techniques.

Yak is the first hybrid GC that combines two seemingly-different styles of memory management harmoniously into one runtime 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 lifetime patterns (generational and epochal) observed in modern data-intensive workloads. Yak treats the data space and control space differently. It allocates data objects in epoch-based regions that are deallocated as a whole at the end of an epoch, and efficiently tracks a small number of objects whose lifetimes cross region boundaries. Control 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 by up to $7 \times$ on three widely-deployed real Big Data systems: Hadoop, Hyracks, and GraphChi, while requiring minimal user effort — users are only required to provide lightweight code annotation, a task that can be easily done in minutes by novices.

**Observation #3 — Overly parallel execution**

Many algorithms used in Big Data applications are inherently data-parallel: the input dataset can be readily decomposed into many disjoint parts and processed in parallel. As multi-core machines are prevalently used, developers tend to create a large number of worker threads in order to fully utilize the underlying parallelism. However, threads do not come for free: the memory cost of data processing gets duplicated upon the creation of a new thread; when a program has too many threads running, it can easily run out of memory if any thread hits the memory wall and fails its processing. Using a small number of threads reduces the risk of running out of memory, but at the cost of sacrificed processing speed.

Practical solutions for performance center around making “best-practice” recommendations for manual tuning of framework parameters. However, the tuning process is time-consuming and labor-intensive as it is impossible to find a one-size-fits-all solution even for a single application on different datasets. How to striking a balance dynamically between the degree-of-parallelism and the application’s performance is a great challenge due to the self-centered characteristics of the worker threads — each thread operates in a very much isolated manner, without any global knowledge of the entire system. Hence, threads are often in a fierce competition for resources (e.g., memory or CPU time). With many aggressive, memory-hungry worker threads, the runtime system constantly struggles to find memory by frequently invoking GC, and may eventually crash when it cannot find sufficient memory to satisfy all running threads.

**My solution** I co-authored the interruptible task (ITask) paper that appeared in SOSP’15 [1]. ITask is a systematic approach that makes data-parallel tasks work cooperatively in Big Data systems. Inspired by how processors handle hardware interrupts, we developed ITask, a new type of data-parallel tasks that can be interrupted when the system is in memory shortage and reactivated later when more memory becomes available. When a task is interrupted, its consumed memory is reclaimed, making more resources (e.g., memory, CPU) available 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 a minimal restructuring effort. This effort is mainly on specifying how to treat different parts of the memory consumed by the task upon its

---

1This paper was the highest ranked paper in OSDI’16.
interruption. ITask also has a runtime system that proactively monitors system states, automatically performs task interruption, as well as auto-tunes the degree of parallelism to balance between the memory usage and the processing speed. With ITask, Big Data systems, such as Hadoop and Hyracks, have enjoyed improvements both in both scalability (3-24× larger dataset due to more intelligent use of memory) and performance (1.5-3× faster due to dynamically tuned thread numbers and significantly reduced GC costs).

**Observation #4 — Costly data transfer in a cluster**

A Big Data system often needs to frequently shuffle data in a cluster. The process is complicated and inefficient — the system needs to *serialize* a sea of heap objects into a byte sequence before sending them over the network. This procedure re-formats each object: it extracts the object payload, strips the object’s header, removes all references stored, and changes the metadata representation. The remote node receiving the bytes then *deserializes* them back into objects, relying heavily on reflection, which is an expensive runtime operation. Due to the extremely large number of invocations of serialization and deserialization functions during sending and receiving, data transfer can take a large portion of a task’s execution, e.g., up to 30% of the end-to-end running time.

**My solution** The key problem with the existing approaches is that 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. My ASPLOS’18 work, Skyway [6], provides a much more effective alternative. Skyway provides runtime system support specialized for Big Data processing. It supports transferring data objects *as is* without changing their format. To send an object o from one node to another, the Skyway-enhanced JVM performs a GC-like heap traversal, copying all objects reachable from o into an output buffer and conducting a lightweight adjustment to the machine-dependent meta data stored in each object’s header. This output buffer then is transferred as a whole directly into the remote heap and can be used immediately after only one linear scan at the receiver side. Skyway completely eliminates the need of changing objects’ format and developing serialization and deserialization functions, leading to reduced computation costs and manual development efforts.

Skyway was implemented in Oracle’s commercial JVM OpenJDK. A thorough evaluation shows that Skyway outperforms *all the 90 existing S/D libraries* in Java Serializer Benchmark Suite — for example, it is 2.2× faster than Kryo, a high-performance serializer recommended for use in Apache Spark, and 67.3× faster than the standard Java serializer. Skyway also improves widely-deployed systems such as Spark and Flink: with Skyway, Spark’s overall execution time was reduced by 16% - 36% and Flink’s time was reduced by 19%, when they were both run with a set of real-world programs and datasets.

**Runtime Bloat Detection with Cachetor**

Modern object-oriented software commonly suffers from runtime bloat that significantly affects its performance and scalability. Studies have shown that one important pattern of bloat is the work repeatedly done to compute the same data values. Very often the cost of computation is very high and it is thus beneficial to memoize the invariant data values for later use. While this is a common practice in real-world development, manually finding invariant data values is a daunting task during development and tuning.

To help developers find caching opportunities for improved performance, I developed a novel run-time profiling tool, called Cachetor [9], which I presented in FSE’13. Cachetor uses a combination of dynamic dependence profiling and value profiling in a novel way so that distinct values are used to abstract instruction instances to identify and report operations that keep generating identical data values. Based on the abstracted dependence graph, we developed three different cacheable data detectors: I-Cachetor, D-Cachetor, and M-Cachetor, that find cacheable data at the instruction-, data-structure-, and call-site-level, respectively. We have implemented Cachetor in Jikes Research Virtual Machine and evaluated it on a set of 14 large Java applications. Our experimental results show that Cachetor is effective in exposing caching opportunities and substantial performance gains can be achieved by modifying a program to cache the reported data.
3 Future Work

Speculative optimization for native data based computation  My experience with various platforms shows that the object-based representation of data causes a great deal of inefficiencies in Big Data systems. Developing compiler and runtime support that can transform existing data-processing programs (e.g., inlining data structures into arrays) to make it operate directly on the native data items would effectively eliminate such overhead. To precisely transform existing program, however, is a daunting task due to the large code size and the pervasive use of complex object-oriented data structures. An important observation is that although transforming the entire program into an object-free version is nearly impossible, most parts of the program — especially those dealing with data processing — can be easily transformed to operate over an array of native bytes. Practical compiler support can be developed if we can focus on the common data-processing parts and speculatively optimize these parts. Another observation made in our Skyway project is that, the life cycle of each data object often starts at a deserialization point at which the object is created from a sequence of native bytes and ends at a serialization point at which the object is transformed back into a byte sequence to be written to disk or sent to another machine. Based on this observation, we can formulate this life cycle as a transaction and speculatively optimize the common parts. The system can detect and abort a transaction if a corner case is encountered and an assumption under which our optimization is made is violated. Upon an abort, the program is reverted to the original object-based execution path.

Non-volatile memory (NVM)  The embrace of NVM in Big Data systems is inevitable — NVM offers larger capacity compared to DRAM and consumes less energy compared to SSD. It is possible to satisfy the high memory requirement of Big Data systems with a small number of computer nodes while simultaneously reducing energy costs. However, na"ively using NVM will result in performance degradation because NVM’s access latency is higher (e.g., 2-4×) than that of DRAM. The key task is to develop intelligent data placement and data migration policies that can minimize performance overhead. Using hybrid memories in Big Data is further complicated because Big Data systems often run on top of a managed runtime with built-in memory management components such as the garbage collector (GC), which is not aware of the physical memories. To efficiently support Big Data processing over hybrid memories, I plan to develop techniques that can make holistic allocation/migration decisions across multiple layers of the compute stack.

Leveraging machine learning in memory management  A managed runtime was designed with many built-in assumptions. For instance, the design of modern garbage collectors is based entirely on the generational hypothesis — the heap is divided into an old and a young generation and the GC runs frequently in young space to move long-lived objects to old regions and infrequently performs full-heap collection. However, as new workloads (e.g., data analytics or machine learning) emerge, the generational hypothesis does not hold any more, leading to poor JVM performance. Machine learning has been shown to be able to effectively react to unseen data with no explicit assumptions about the data. The use of supervised or unsupervised learning can open up opportunities to learn models that reflects the object characteristics of different (seen and unseen) workloads. The learned models can be used to predict object lifetimes to significant reduce the GC effort. As an example, an object-lifetime model can be used to guide the object allocation so that objects can be segregated based on their expected lifetimes upon their creation without the need to perform expensive runtime migration. A model learned over the object access patterns can help with data placement — infrequently used objects will be placed in NVM to minimize energy costs while data that are frequently accessed stay in DRAM to utilize its fast access speed. These idea of embedding learned models in a system may open up massive opportunities for developing compilers and runtime systems for emerging workloads.

Automated data persistence in Big Data systems  In-memory dataflow engines such as Apache Spark often cache data (e.g., RDDs) in memory to speed up iteration algorithms and recovery upon faults. However, users are burdened with the task of explicitly invoking APIs (such as persist in Spark) on the RDDs of their choice to cache or uncache them. On the one hand, holding everything in memory accelerates computation at
the cost of memory blowup. On the other hand, recomputing a dataset instead of caching it can save storage space at the cost of performance sacrifice. Striking a balance is a difficult task, especially for long-running analytical applications with complicated workflow. Currently, the decisions of whether or not to cache a dataset is made entirely at compile time, and thus cannot adapt effectively to the runtime environment and the data characteristics. Develop support at the system level that can automatically identify caching opportunities would be extremely useful, as this would remove burden from the developer’s shoulders and enable quick development. A preliminary experiment in Spark I conducted shows that a performance gain of up to 44% can be achieved if the system can automatically cache and uncache data at run time. I plan to develop such support and thoroughly evaluate its performance in a large number of real-world systems.

References


