UNIVERSITY OF CALIFORNIA

Los Angeles

Compiler and Runtime Support for Efficient and Scalable Big Data Processing

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Khanh Truong Duy Nguyen

2019

ABSTRACT OF THE DISSERTATION

Compiler and Runtime Support for Efficient and Scalable
Big Data Processing

by

Khanh Truong Duy Nguyen
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2019
Professor Harry Guoqing Xu, Chair

Large-scale data analytical 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. However, Big Data processing is fundamentally limited by 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 runtime system cost can be easily magnified and become the critical performance bottleneck. Our experience with dozens of real-world systems reveals the root cause is the mismatch between the fundamental assumptions based on which the current runtime is designed and the characteristics of modern data-intensive workloads.

This dissertation consists of a series of techniques, spanning programming model, compiler, and runtime system, that can efficiently mitigate the mismatches in real-world systems, and hence, significantly improve the efficiency of various aspects of Big Data processing. Specifically, this dissertation makes the following three contributions. The *first contribution* is the development of a framework named Facade aiming to reduce the cost of object-based representation without an intrusive modification of a JVM. Facade uses a compiler to generate highly efficient data manipulation code by automatically transforming classes in such a way that objects are created only as proxies. Facade advocates for the separation of data storage and data manipulation. The execution model enforces a statically-bounded total number of data objects in an application regardless of how much data it processes. The *second contribution* is the design and implementation of Yak, the *first hybrid* garbage collector tailored for Big Data systems. Yak provides high throughput and low latency for all JVM-based languages by adapting its algorithms to two vastly different types of object lifetime behaviors in Big Data applications. Finally, the *third contribution* is a JVM-based alternative to enable instantaneous data transfer across processing nodes in clusters called Skyway. Skyway optimizes away inefficiencies of relying on reflection — a heavyweight runtime operation, and handcrafted procedures in converting data format by transferring objects as-is, without changing much of their existing format.

We have extensively evaluated those compiler and runtime techniques in several real-world, widely-deployed systems. The results show significant improvement of the system over the baseline: faster execution, reduced memory management costs, and improved scalability. The techniques are also highly practical and easy to integrate without much user efforts, making the adoption in real setting possible. The work has inspired a line of several follow-up work from academia. Moreover, the Yak system has been adopted by a telecommunication company.

The dissertation of Khanh Truong Duy Nguyen is approved.

Jens Palsberg

Miryung Kim

Anthony John Nowatzki

Harry Guoqing Xu, Committee Chair

University of California, Los Angeles

2019

To my parents,

Nguyễn Thanh Xuân & Phan Thị Kim Thoa

TABLE OF CONTENTS

1 Introduction **1**

2 Background **8**

2.1 Memory Bloat 8

2.2 Garbage collection 9

2.3 Behavior of a Big Data application 11

2.4 Data shuffling 12

3 Mitigating Object-based Representation Mismatch **15**

3.1 An Empirical Study on Memory Bloat Caused by the Use of Objects 16

3.2 Facade overview 17

3.2.1 Technical challenges and solutions 17

3.3 The Facade Execution Model 20

3.3.1 Data Storage Based on Native Memory 20

3.3.2 Using Objects as Facades 22

3.3.3 Bounding the Number of Facades in Each Thread 27

3.3.4 Performance Benefits 28

3.4 Facade Design and Implementation 29

3.4.1 Our Assumptions 29

3.4.2 Data Class Transformation 30

3.4.3 Type specialization for `Object` and `Object[]` 33

3.4.4 Computing Bounds 34

3.4.5 Supporting Concurrency 35

3.4.6 Data Conversion Functions 37

3.4.7 Memory Allocation and Page Management 37

3.4.8 Correctness and Profitability Arguments 40

3.4.9 Modeling of Important Java Library Classes 42

3.4.10 Implementation and Optimizations 43

3.5 Evaluation . . . 44
3.5.1 GraphChi . . . 44
3.5.2 Hyracks . . . 50
3.5.3 GPS . . . 53
3.6 Summary . . . 56

4 Mitigating Garbage Collection Mismatch . . . 58
4.1 The Garbage Collection Mismatch . . . 59
4.2 Object Characteristics in Real Systems . . . 60
4.3 Design Overview . . . 63
4.4 Implementation . . . 67
4.4.1 Region & Object Allocation . . . 67
4.4.2 Tracking Inter-region References . . . 69
4.4.3 Region Deallocation . . . 73
4.4.4 Collecting the CS . . . 79
4.5 Evaluation . . . 79
4.5.1 Methodology and Benchmarks . . . 79
4.5.2 Epoch Specification . . . 81
4.5.3 Latency and Throughput . . . 82
4.5.4 Memory Usage . . . 85
4.5.5 Performance Breakdown . . . 86
4.5.6 Comparison with G1 . . . 88
4.6 Summary . . . 90

5 Mitigating Data Transfer Mismatch . . . 92
5.1 Serialization/Deserialization Costs in Real World . . . 92
5.2 Skyway Design Overview . . . 94
5.2.1 Correctness . . . 96
5.2.2 Efficiency . . . 96
5.2.3 Ease of Integration . . . 98

5.3 Implementation 99
5.3.1 Global Class Numbering 99
5.3.2 Sending Object Graph 103
5.3.3 Receiving Object Graph 108
5.4 Evaluation 110
5.4.1 Java Serializer Benchmark Set 110
5.4.2 Improving Apache Spark with Skyway 113
5.4.3 Improving Apache Flink with Skyway 118
5.5 Summary 119

6 Related Work 122
6.1 Software Bloat Analysis 122
6.2 Reducing the number of objects via program analysis 123
6.3 Region-based Memory Management 125
6.4 Garbage Collection 126
6.5 Effort in programming languages community 127
6.6 Memory Management Optimizations in Big Data Systems 128
6.7 Object Sharing in the OS 130

7 Conclusions and Future Directions 132

LIST OF FIGURES

2.1 Graphical illustration of control and data paths. 12

2.2 A graphical illustration of data transfer. 13

3.1 The ratios between the total bytes of data objects and the size of their actual payload 16

3.2 A data structure in regular Java and its corresponding data layout in a native page. 21

3.3 A transformation example; part (a). 22

3.4 A transformation example; part (b) 23

3.5 A graphical representation of threads and pools, where *AFacade*, *BFacade*, . . ., and *ZFacade* are facade types. 35

3.6 A program (a) and its corresponding page manager tree after Line 7 is executed (b), assuming 2 threads t_1 and t_2 are executing. 40

3.7 Performance comparisons between the original (P) and Facade-transformed programs (P') on GraphChi. 46

3.8 Performance improvements with and without data record inlining. 48

3.9 Computational throughput of GraphChi on various graphs; each trend-line is a least-squares fit to the average throughput of a program. 49

3.10 Performance comparisons between the original (P) and Facade-transformed (P') programs on Hyracks. 52

3.11 Performance comparisons between the original (P) and Facade-transformed (P') programs on GPS. 55

4.1 Memory footprint for the first 160 seconds of the executions when GraphChi runs PR (top) and CC (bottom). Each dot in (a) represents the memory consumption measured right after a GC; each bar in (b) shows how much memory is reclaimed by a GC; dotted vertical lines show the iteration boundaries. 61

4.2 Eclipse execution (GC takes 2.4% of time). 62

4.3 Hyracks WordCount (WC) execution (GC takes 33.6% of time). 63
4.4 An example of regions: (a) a simple program and (b) its region semilattice at some point of execution. 65
4.5 The heap layout in Yak. 68
4.6 (a) An object referenced by b escapes its epoch via the stack variable a; (b) An object O created by thread t and referenced by $a.g$ escapes to thread t' via the load statement $b = p.g$. 71
4.7 Examples showing potential problems with references on a remote stack: (a) moving object D is dangerous; and (b) object E, which is also live, is missed in the transitive closure. 72
4.8 An example heap snapshot (a) before and (b) after the deallocation of region $\langle r_{21}, t_1\rangle$. 77
4.9 Performance comparisons between PS and Yak on various programs. 83
4.10 Memory footprints collected from `pmap`. 84
4.11 Yak performance comparisons on GraphChi between different page sizes 88
4.12 Performance comparisons between Yak and PS when datasets of various sizes were sorted by Hyracks ES on a 24GB heap 88
4.13 Performance comparisons between G1 and Yak on Hyracks 89

5.1 Spark serialization/deserialization costs: (a) performance breakdown and (b) bytes shuffled under the Java and Kryo serializer. partitions. 93
5.2 Skyway's overview. 95
5.3 Skyway's system architecture. Purple and orange rectangles represent input (in-heap) buffers and output (native) buffers, respectively; objects flow along red arrows. 95
5.4 Type registries used for global class numbering. 101

5.5 Skyway object layout in the heap (a) and an output buffer (b). This is an `Integer` array of three elements on a 64-bit HotSpot JVM. `mark` contains object locks, hash code of the object, and GC bits. `klass` points to the meta object representing the object's class. What follows is the data payload – three references to `Integer` objects. `baddr` and `tID` are both added by Skyway. 106

5.6 Serialization, deserialization, and network performance of different S/D libraries. Although we have compared Skyway with 90 existing libraries, we include in this figure Skyway and 27 fastest-performing libraries. The last bar is simply a placeholder of the 63 libraries that perform slowly. 111

5.7 Size of serialized object by different libraries. 112

5.8 Spark performance with Java serializer, Kryo, and Skyway. 115

5.9 Ratio between the number of bytes generated by Skyway and the number of bytes generated by Java serializer and Kryo. 116

5.10 Flink performance with Flink's built-in serializer and Skyway. 118

LIST OF TABLES

3.1 A summary of code generation. Suppose variables a and b have types A and B, respectively. 32

3.2 Performance summary of GraphChi on Twitter-2010 graph: reported are execution time of original run (T) and Facade run (T'), the reduction of total execution times (%ET), engine update times (%UT), data load times (%LT), garbage collection times (%GC), and peak memory consumption (%PM) under three different memory budgets (e.g., 8GB, 6GB, and 4GB) 47

3.3 GraphChi performance comparisons on different datasets: reported are the total execution times of the original program (T), and its Facade-version (T'); %ET reports Facade's reduction in execution time and %TR reports the throughput improvement. 49

3.4 Hyracks performance comparisons on different datasets: reported are the total execution times of the original program (T), and its Facade-version (T'); OME(n) means the program runs out of memory in n seconds. %GC and %PM report Facade's reduction in GC time and peak memory consumption, respectively. 51

3.5 GPS performance comparisons on different datasets: reported are the Facade's reduction in execution time (%ET) and GC time (%GC); LJ-A. . .LJ-E are different synthetic supergraphs of the Live Journal (LJ) graph; TW is the Twitter-2010 graph. 54

4.1 Auxiliary functions used in Algorithms 1 – 4 69

4.2 Our benchmarks and their descriptions. 80

4.3 Datasets and heap configurations used to run our programs; for Hadoop, the configurations a&b GB are the max heap sizes for each map (a) and reduce task (b). 80

4.4 Summary of Yak performance normalized to baseline PS in terms of **Overall** run time, **GC** time, including Yak's region deallocation time, **App**lication (non-GC) time, and **Mem**ory consumption across all settings on each framework. The values shown depict Min ∼ Max and (Mean), and are normalized to PS. A lower value indicates better performance versus PS. 82

4.5 Statistics on Yak's heap: numbers of cross-space references (CSR), cross-region references (CRR), and total references generated by stores (TR); average percentage of objects escaping to the CS (CSO) among all objects in a region when the region retires; and total number of regions created during execution (R). . . 87

4.6 Summary of G1 and Yak performance normalized to baseline PS. Each cell shows Min ∼ Max and (Mean), and are normalized to PS. Lower is better. . . . 90

5.1 Graph inputs for Spark. 114

5.2 Performance summary of Skyway and Kryo on Spark: normalized to baseline (Java serializer) in terms of **Overall** running time, **Ser**ialization time, **Write** I/O time, and **Des**erialization time, **Read** I/O time (including the network cost), and the **Size** of byte sequence generated. A lower value indicates better performance. Each cell shows a percentage range and its geometric mean. . . . 116

5.3 Descriptions of the queries used in Flink. 118

5.4 Performance improvement summary of Skyway on Flink: normalized to Flink's built-in serializer. Each cell shows a percentage range and its geometric mean. A lower value indicates better performance. 119

ACKNOWLEDGMENTS

From the bottom of my heart, oceans of gratitude go to my advisor, Professor Harry Xu. I simply would not be where I am now without him. Without his encouragement, I would not have started graduate school. And without his tireless effort in training and mentoring me, I would have not finished this seven-year journey. Professor Xu has inspired me to become a passionate researcher, a dedicated professor, and a supportive mentor, just like him. When I self-doubted, he encouraged me. When I felt lost, he guided me back on track. When all I saw was darkness, he lighted a path. I have been very fortunate to have him as my advisor. I will forever reserve my deepest gratitude for his guidance, understanding, and support towards my professional success and personal growth.

At UCLA I would like to thank Professor Tony Nowatzki, Professor Miryung Kim, Professor Jens Palsberg; at UC Irvine I owe my gratitude to Professor Michael Carey, Professor Aparna Chandramowlishwaran, Professor Brian Demsky, Professor Alex Nicolau for serving on my doctoral committee. Their valuable feedback has been of tremendous help completing this dissertation.

I dedicate this dissertation to my parents, Xuan Nguyen and Thoa Phan, who made countless sacrifices raising me and four siblings. Even though we are now half-the-globe apart, I know they are always besides me. I also cannot thank my two uncles, Vu Phan and Huy Phan and their spouses enough. Thank you for many family meals that really warm my heart. And thank you for being there for me during my ups and downs in life.

I would like to thank Dr. Kathryn McKinley and Chris MacGregor for the summer internship at Google. The experience has enriched my life with real-world problems and industrial work environment. I very much look forward to a chance working together again in the future.

My special thanks to Dr. Shan Lu from University of Chicago and Dr. Rajiv Gupta from UC Riverside. The honor of winning the prestigious Google Ph.D. Fellowship would have not been possible without their strong support.

My former labmate, Dr. Lu Fang, deserves a special mention. I count myself lucky to have him as a bother-in-arms on many research projects, and as a close friend. With his strong technical skills and great professionalism, he is the best colleague one could ask for.

I would like to thank my academic family, the AnalySys lab — Dr. Zhiqiang Zuo, Dr. Keval Vora, Dr. Chenxi Wang, Dr. Jon Eyolfson, Dr. Kai Wang, Cheng Cai, Bojun Wang, John Thorpe, Tim Nguyen, Christian Navasca. Thanks for making my journey less stressful and fill it with friendship and teamwork.

I have the opportunity to mentor many high school students, undergraduates from UC Irvine and from Kookmin University whom regrettably, I cannot acknowledge all by name. Thank you for helping me become a better mentor.

It would be incomplete if I do not convey my gratitude to my teachers in early days. Mr. Son Tran and Mr. Tuan Nguyen from Thang Long High School in Vietnam patiently taught me how to program in Pascal, kindled my interest in Computer Science. Professor Scott Edwards and Professor Stephen Plett from Fullerton College tremendously helped me keep the fire burning.

Above all, I thank God for the chance to meet every single person, the luck I have, and all the much-needed strength during this journey. The end of Ph.D. is very exciting but I am more excited for the beginning of the next chapter of my life.

The work presented in this dissertation has been supported by the National Science Foundation under grants CNS-1321179, CCF-140982, CNS-1613023, and CNS-1703598, by the Office of Naval Research under grants N00014-14-1-0549 and N00014-16-1-2913, and a Google Ph.D. Fellowship.

VITA

Khanh Truong Duy Nguyen

EDUCATION

University of California, Los Angeles	2018 - 2019
– Ph.D. in Computer Science	*(expected)*
University of California, Irvine	
– Ph.D. in Computer Science	2012 - 2018
Advisor: Prof. Harry Xu	*(moved to UCLA)*
2017 - 2019 Google Ph.D. Fellow	
2017 Facebook Ph.D. Fellowship Finalist	
University of California, Irvine	
– M.S. in Computer Science	2012 - 2015
– B.S. in Computer Science	2010 - 2012
Cum Laude, Donald Bren School of ICS Honor Program Graduate	
Fullerton College	
– A.S. in Computer Science	2006 - 2010
– A.A. in Mathematics	2006 - 2010
High Honors	

EXPERIENCE

Research Assistant University of California, Los Angeles, CA	September 2018 - June 2019
Software Engineer Intern Google Inc, Seattle, WA	June 2018 - September 2018
Research Assistant University of California, Irvine, CA	June 2012 - June 2018
Teaching Assistant University of California, Irvine, CA	Fall 2013, Summer 2017
Proctor University of California, Irvine, CA	July 2011 - September 2011
Web Assistant Webstorm Internet Media, Newport Beach, CA	August 2010 - December 2010

PUBLICATIONS

P8. Cheng Cai, Qirun Zhang, Zhiqiang Zuo, **Khanh Nguyen**, Harry Xu, and Zhendong Su.
Calling-to-Reference Context Translation via Constraint-Guided CFL-Reachability
ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)

P7. **Khanh Nguyen**, Lu Fang, Christian Navasca, Harry Xu, Brian Demsky, and Shan Lu.
Skyway: Connecting Managed Heaps in Distributed Big Data Systems
The 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)

P6. **Khanh Nguyen**, Kai Wang, Yingyi Bu, Lu Fang, and Harry Xu.
Understanding and Combating Memory Bloat in Managed Data-Intensive Systems
ACM Transactions on Software Engineering and Methodology (TOSEM)

P5. **Khanh Nguyen**, Lu Fang, Harry Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu.
Yak: A High-Performance Big-Data-Friendly Garbage Collector
The 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)

P4. **Khanh Nguyen**, Lu Fang, Harry Xu, and Brian Demsky.
Speculative Region-based Memory Management for Big Data Systems
The 8th Workshop on Programming Languages and Operating Systems (PLOS)

P3. Lu Fang, **Khanh Nguyen**, Harry Xu, Brian Demsky, and Shan Lu.
Interruptible Tasks: Treating Memory Pressure As Interrupts for Highly Scalable Data-Parallel Programs
The 25th ACM Symposium on Operating Systems Principles (SOSP)

P2. **Khanh Nguyen**, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Harry Xu.
FACADE*: A Compiler and Runtime Support for (Almost) Object-Bounded Big Data Applications*
The 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)

P1. **Khanh Nguyen** and Harry Xu.
Cachetor: Detecting Cacheable Data to Remove Bloat
The 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/ FSE)

CHAPTER 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. As another example, the seventh Digital Universe study jointly conducted by Dell EMC and International Data Corporation [75] — the only study to quantify the amount of global data produced annually — reveals that the digital universe to grow 10-fold by 2020 — from 4.4 zettabytes (i.e., trillion gigabytes) in 2013 to 44 zettabytes. To put that in perspective, the amount of information is equivalent to 6.6 stacks of 128GB iPad Air tablets, each stack is enough to cover the distance from the earth to the moon.

The availability of an enormous amount of data has led to the proliferation of large-scale, data-intensive applications. Big Data analytics quickly become a key component of modern enterprise computing, transforming science, healthcare, finance, business, and ultimately, the whole society. As the digital universe expands, companies, government agencies, 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. For instance, retailers and media companies leverage users history to offer personalized product recommendations. Government agencies such as the SEC and the SSA as well as commercial banks fight frauds daily using Big Data technologies to monitor huge amounts of financial transactions.

The mainstream approach to tackle the scalability challenge 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 [35], Giraph [34], Hive [36], Hyracks [51], Storm [38], Naiad [130], Spark [182], and TensorFlow [28]. 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 [77, 127, 132–135, 137, 169, 171–173, 175]: memory management in Big Data systems is prohibitively expensive. These systems commonly suffer from severe memory problems, causing poor performance and low scalability. That is due to a mismatch of memory usage in this emerging environment. Under the object-oriented modeling of data, a concrete object is created for each data item, thus, the heap of a Big Data application contains *several orders of magnitude* more objects than a general, non-data-intensive application. Allocating and deallocating a sea of data objects puts a severe strain on the runtime system, leading to high memory management overhead — objects compete for memory storage as well as CPU and networking time, causing prolonged execution time, and inability to process datasets of even moderate sizes. For example, Bu et al. [55] reports an average-sized application on Giraph [34], 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 [29, 30, 51, 56, 57, 65, 89, 100, 116, 140, 144, 160, 161, 177, 179, 184], 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, in practice, and our experience [135] confirms that, in many cases, the data-processing program running on a worker machine suffers from extensive memory pressure — the execution pushes the memory 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 [55, 77, 134, 135]. The problem becomes increasingly severe in latency-sensitive distributed cloud applications such as web servers or real-time analytical systems, where one slow-performing node can hold up the entire cluster, delaying the processing of user requests for an unacceptably long time [118, 119].

Going back to unmanaged language? Unsatisfactory performance and low scalabilty in contemporary Big Data systems is a real problem that has seriously concerned developers of data-intensive systems. Debates on whether systems development should forsake managed languages to avoid their inherent inefficiencies and switch to unmanaged languages such as C and C++ can be found almost everywhere [7, 8, 25]. While that appears to be a reasonable choice, unmanaged languages are more error-prone; debugging memory bugs in an unmanaged language is known to be a notoriously painful task, which can be further exacerbated by the many Big Data effects, such as distributed execution environment, huge volumes of heap objects, and extremely long running time. Furthermore, since most existing data processing frameworks are already developed in a managed language, it is unrealistic to re-implement them from scratch.

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. This dissertation focuses on enabling *vertical scaling*, i.e., systematically improving the performance of each data-processing program. 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 benefit.

Our extensive experience with dozens real-world, data-itensive systems reveals that the core reason for the low performance of existing Big Data engines is the mismatch between ancient assumptions and abstractions, based on which runtime systems are developed, and behavior of modern Big Data workloads. Specifically, the mismatches are:

1. **Object-based representation mismatch.** In a Big Data application, comparing to a standard application, the heap contains several orders of magnitude more of objects and references since they are created proportionally with the cardinality of the input dataset. Each object has a memory overhead in term of object header space and pointer reference. While the memory cost of maintaining millions of objects in non-Big-Data applications can be acceptable, the existence of billions of objects in a heap of a Big Data application exacerbates the runtime cost, limiting scalability.

2. **Garbage collection mismatch.** State-of-the-art garbage collectors are built upon *generational hypothesis* which assumes objects are short-lived. The hypothesis holds true for many applications, and dictates the design of the managed heap. However, the behavior of the majority of runtime objects in Big Data applications do not match such assumption. Instead, they often exhibit strong epochal behavior: their lifetime often spans the execution of a data manipulation task, which is often long. Much of the GC effort is wasted by repeatedly scanning a huge number of objects while they are not reclaimable. The challenge encountered by state-of-the-art GCs in data-intensive applications is fundamental, causing unsatisfactory performance.

3. **Serialization mismatch.** Data transferring process is designed very slow and inefficient: an object must be serialized (i.e., turning an object from heap format to byte format) by the sender; and then deserialized (i.e., a complete opposite process, turning a byte sequence into a heap object) by the receiver. It is understandable that the serialization interface was developed in such a way since only a few objects needed to be serialized in standard applications. However, in Big Data systems, data transferring is a common task in the processing pipeline that needs to be invoked on a sea of objects to be transferred via network. Slow serialization is an unacceptable hurdle in such a performance-sensitive task.

The mismatches in existing managed runtime system motivate the development our techniques that can dramatically improve the runtime system efficiency for modern Big Data workloads. This dissertation consists of a series of techniques, spanning programming model, compiler, and runtime system, aiming to mitigate the mismatches, optimizing away runtime system inefficiencies, and thus, significantly improve the efficiency of various aspects of Big Data processing. In particular, the contributions of this dissertation are:

- **Contribution 1.** Facade [135], a non-intrusive approach targets the performance problem caused by excessive object creation in an object-oriented Big Data application. We propose a facade-based programming model in which data are stored in raw memory buffers and objects are created as proxies only to satisfy control purposes and to manipulate buffered data. Facade guarantees that the number of runtime objects for each data type is statically bounded. The compiler performs a semantics-preserving transformation on data classes to store and access data in native memory pages.

  We have implemented Facade and used it to transform 7 common applications on 3 real-world data processing frameworks: GraphChi, Hyracks, and GPS. Our experimental results show the The execution of an object-bounded program is much more memory-efficient. The generated programs have (1) achieved a 3%–48% execution time reduction and and up to 88× GC time reduction; (2) consumed up to 50% less memory, and (3) scaled to much larger datasets.

- **Contribution 2.** Yak [134], a new hybrid collector that can adapt intelligently to two different types of object lifetime characteristics in Big Data workloads. In Yak, data objects are speculatively allocated into lattice-based data regions while the generational GC only scans and collects the control space, which is much smaller. By moving all data objects into regions and deallocating them as a whole at the end of the lifetime of each region, significant reduction in GC overheads can be achieved.

  We implemented Yak inside Oracle's production JVM, OpenJDK 8. The JVM-based implementation enables Yak to work for all JVM-based languages such as Java, Python, and Scala. Our experiments on several real-world systems such as Hadoop, Hyracks and GraphChi with various types of applications and workloads demonstrate that Yak outperforms the default Parallel Scavenge production GC in OpenJDK on real Big Data systems. Specifically, Yak reduces GC latency by 1.4 – 44.3× and improves overall application performance by 12.5% – 7.2× while requiring almost zero user effort.

- **Contribution 3.** Skyway [132], a more efficient, tailored-for-Big-Data-systems serializer to reduce the data transfer cost among managed heaps. Skyway enhances the JVM, directly connects managed heaps on different machines. Under Skyway, objects are transferred as-is from the source to the destination without changing much of the format, enabling them to be used immediately after the transfer. Unlike all existing approaches, Skyway provides performance benefits to any JVM-based system by completely eliminating the need of invoking reflection functions to convert data formats, thus saving computation power. It also requires less developer intervention in registering classes to be serialized as well as in hand-writing serialization functions.

  We have implemented Skyway also in Oracle's production JVM, OpenJDK 8. Our evaluation on a Java serializer benchmark set JSBS, Apache Spark, and Apache Flink shows that (1) Skyway outperforms all the 90 existing serializers in JSBS. It even outperforms popular industrial effort: 2.2× faster than Kryo and 1.9× faster than the Google‘s protobuf; (2) compared with Kryo and the Java serializer, Skyway improves the overall Spark performance by 16% and 36% for four representative analytical tasks over four real-world datasets; (3) for another real-world system Flink, Skyway improves its overall performance by 19% compared against Flink's highly-optimized built-in serializers.

Impact. Asides from improving dramatically efficiency of several popular Big Data systems as demonstrated in our evaluations, this line of work has attracted much attention from both academia and industry. The Facade system [135] inspired a line of several follow-up work — for instance, researchers from Microsoft Research and University of Cambridge used the same idea [86] to improve Naiad, Microsoft's dataflow egnine; Oracle and UC Berkeley researchers developed a coordinated GC system for latency-sensitive workloads [118, 119]; and the idea was also adopted by a group of scholars from China, Denmark, and Hongkong to improve the memory management in Apache Spark [117, 151]. As another example, the Yak GC [134] was of interest of Oracle and recently commercialized by Huawei for use in their telecommunication systems.

Organization. The remainder of this dissertation is organized as follows. Chapter 2 introduces several basic concepts. Chapter 3 discusses Facade, a framework to statically bound total number of objects created regardless of the size of the workload at the compiler level. Yak, a new hybrid garbage collector for Big Data systems is introduced in Chapter 4. Chapter 5 describes the system Skyway built to directly connect managed heaps, reducing data transfer cost in clusters. Related work is discussed extensively in Chapter 6. Finally, we conclude in Chapter 7 with some exciting future directions.

CHAPTER 2

Background

In this chapter, we introduce some basic concepts to facilitate the discussion of our techniques in subsequent chapters.

2.1 Memory Bloat

The term memory bloat refers to excessive memory space required for processing. Memory bloat commonly exists in enterprise computing [169, 171, 172, 175]. In object-oriented programming, for example, each Java object has a fixed-size header space to store its type and the information necessary for garbage collection (e.g., in the Oracle 64-bit HotSpot JVM, the header takes 8 bytes for regular objects and 12 bytes for arrays). In an application that contains a huge number of objects to represent and manipulate data items of small size, since the data contents in each such object do not take much space, the space overhead cannot be amortized. That is not all, a significant source of this space overhead is object references due to the pervasive use of pointer to form data structures. With multiple layers of indirection, these pointers consume a significant portion of the heap. Based on a study reported in [127], the fraction of the useful payload data in an IBM application is only 13% of the total space. The space impact can be significantly magnified in a data-intensive application when massive amounts of input data need to be represented and processed in memory. Studies in [55, 136] found that packing factor, a metric which is defined as the maximal amount of actual data that be accommodated into a fixed amount of memory, in various system is below satisfactory. Moreover, in these applications, because the input dataset is too large to fit totally into memory, data processing pipeline often must be divided to work on a small partition of the input resulting in significantly increased round-trip I/O costs.

As a result, data processing systems written in managed languages frequently face severe memory problems that cause them to fail with out-of-memory errors, reducing its scalability although the size of the processed dataset is much smaller than the heap size. For example, in Spark [181, 182], a leading system in Big Data processing, even machines with reasonably large memory resources cannot satisfy its need and out-of-memory errors have been constantly reported on StackOverflow [24] and the Apache mailing list [1]. Similarly, numerous examples of Hadoop users facing out-of-memory errors can also be found on StackOverflow (e.g., [9–21, 23]). Graph processing systems are not excluded; systems like Giraph [34] is reported not able to process moderate-sized graphs on large clusters although the amount of data for each machine is well below its memory capacity [55].

2.2 Garbage collection

Garbage collection (GC) is extremely useful, as it simplifies the programming model, therefore freeing valuable programmer time, while avoiding bugs and memory leaks that are notoriously hard to track and fix. As any C programmer can witness, the manual management of a dynamic heap is extremely complex and highly error-prone. If the heap is shared by many applications written by different programmers, accessed in parallel, or includes disk storage or distributed access, then manual resource management is simply out of question.

The mainstream collectors in modern systems are *tracing garbage collectors*. A tracing GC performs allocation of new objects, identification of live objects, and reclamation of free memory. It identifies live objects by following references (thus called tracing), starting from a set of root objects that are directly reachable from live stack variables and global variables. It computes a transitive closure of live objects; objects that are unreachable during tracing are guaranteed to be dead and will be reclaimed.

There are four kinds of canonical tracing collectors: *mark-sweep*, *mark-region*, *semi-space*, and *mark-compact*. They all identify live objects in the same way as discussed above. Their allocation and reclamation strategies differ significantly:

- Mark-sweep collectors [78, 124] allocate from a free list, mark live objects, and then put reclaimed memory back on the free list. Since a mark-sweep collector does not move live objects, it is time- and space-efficient, but it sacrifices locality for contemporaneously allocated objects.

- Mark-region collectors [44, 48, 52] reclaim contiguous free regions to provide contiguous allocation. Some mark-region collectors such as Immix [48] can also reduce fragmentation by mixing copying and marking.

- Semi-space [40, 43, 47, 59, 67, 98, 156] and mark-compact [62, 104, 148] collectors both move live objects. They put contemporaneously-allocated objects next to each other in a space, providing good locality.

These canonical algorithms serve as building blocks for more sophisticated algorithms such as the generational GC [156], which built upon the popular, conventional *generational hypothesis*. The hypothesis states objects are short-lived: most recently allocated objects are more likely to become garbage and won't survive a GC run. This dictates the design of the heap, which divides the heap into a young and an old generation. Allocation happens in young generation. Most GC runs are *nursery (minor) collections* that only scan references from the old to the young generation, promotes (i.e., copies) reachable objects to the old generation, and then frees the entire young generation. Garbage collection for the old generation occurs infrequently. Only when nursery GCs are not effective, not yielding enough memory for the applications, a *full (major) collection* scans both generations to free more memory. As long as the generational hypothesis holds, which is true for many large conventional applications that make heavy use of short-lived temporary data structures, generational GCs are efficient: a small number of objects escape to the old generation, and hence, most GC runs need to traverse only a small portion of the heap to identify and copy these escaping objects.

More formally, if the number of live objects is n and the total number of edges in the object graph (i.e., number of references) is e, the asymptotic computational complexity of a tracing garbage collection algorithm is $O(n + e)$. For a typical data-intensive application, its object graph often consists of a great number of isolated object subgraphs, each of which represents either a data item or a data structure created for processing data items. As such, there often exists an extremely large number of in-memory data objects, and both n and e can be orders of magnitude larger than those of a regular Java application. In fact, commonly GC is the performance bottleneck that significantly hurts system performance. Evidence shows that GC accounts for 25-50% [55, 77, 132, 135], and could go up to as much as 73% of the total execution time [134].

2.3 Behavior of a Big Data application

It is important to first understand the behavior of a typical Big Data application. Unlike traditional, object-oriented programs, evidence [55, 86, 135] shows that a Big Data program often has a very clear logical distinction between a *control path* and a *data path* as exemplified in Figure 2.1. The control path performs cluster management and scheduling, organizes tasks into pipelines, establishes communication channels between nodes, and interacts with users to parse queries and return results. Meanwhile, the data path primarily consists of data manipulation functions such as `Aggregate`, `Join` or user-defined functions (e.g., `Map`, `Reduce` in Hadoop) that can be connected to form a data processing pipeline.

These two paths follow different heap usage patterns. The control path behaves similar to a regular application. It could have a complicated logic to drive the program flow but create a small number of objects. More importantly, control objects follow the conventional wisdom: recently allocated objects are also most likely to become unreachable quickly; most objects have short lifespans. Meanwhile, the data path, not only is the main source where most objects are created (e.g., this path can create up to 95% of the total number of objects [55]), its execution of exhibits strong *epochal behavior*—each piece of data manipulation code is repeatedly executed. Objects created in this path is several order of magnitudes more than

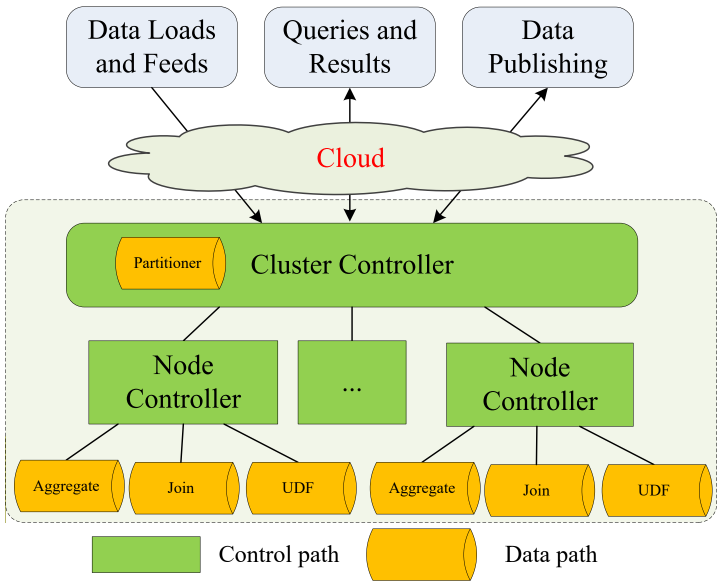


Figure 2.1: Graphical illustration of control and data paths.

control objects—they represent the input data set as well as intermediate and final results of processing. The execution of each epoch starts with allocating many objects and then manipulating them. These objects are often held in large arrays and stay alive throughout the epoch, which is often not a short period of time.

2.4 Data shuffling

Modern Big Data systems need to frequently shuffle data in the cluster – a map/reduce framework such as Hadoop [35] shuffles the results of each map worker before performing reduction on them; a dataflow system such as Apache Spark [182] supports many RDD transformations that need to shuffle data across nodes. As most of these systems are written in managed languages such as Java and Scala, data is represented as objects in a managed heap. This task utilizes serialization/deserialization procedures.

Transferring an object O across nodes is complicated, involving three procedures shown in Figure 2.2. First, a *serialization* procedure turns the whole object graph reachable from O 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. Then, this byte sequence is transferred to a receiver machine via the network. Finally, a *deserialization* procedure reads out the byte sequence, creates objects accordingly, and eventually rebuilds the object graph in the heap of the receiver machine.

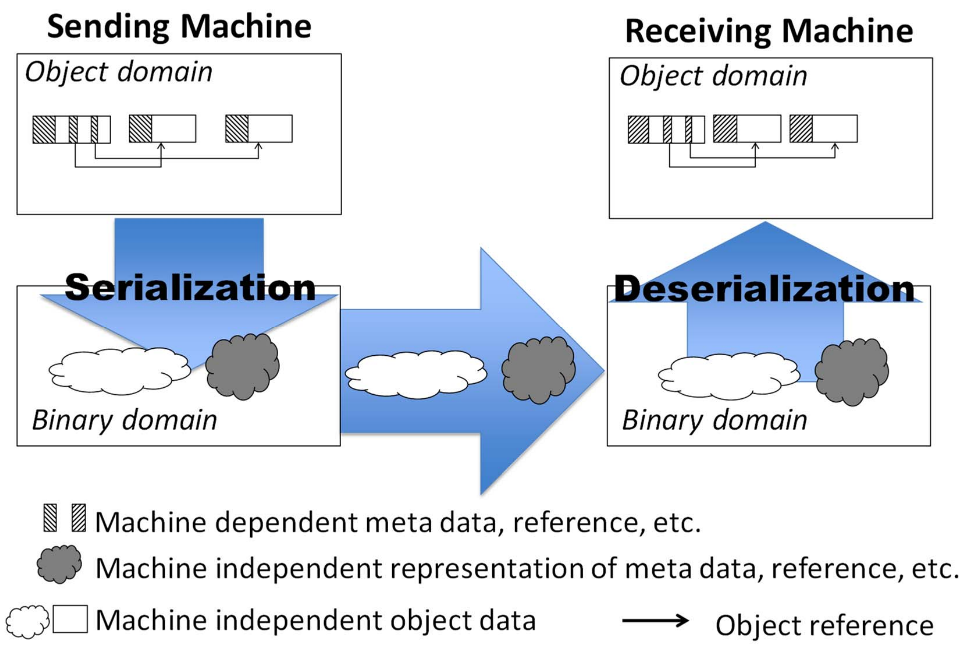

Sending Machine
Object domain
Serialization
Binary domain
Receiving Machine
Object domain
Deserialization
Binary domain
Machine dependent meta data, reference, etc.
Machine independent representation of meta data, reference, etc.
Machine independent object data
Object reference

Figure 2.2: A graphical illustration of data transfer.

The serialization interface, unfortunately, was designed very compute-inefficient. To illustrate, we discuss the handling of three key pieces of information these procedures have to extract, transfer, and reconstruct for every object reachable from O: (1) object data (i.e., primitive-type fields), (2) object references (i.e., reference-type fields), and (3) object type.

1. Object data access: A serializer needs to invoke *reflective functions* such as `Reflection.getField` and `Reflection.setField` to enumerate and access every field to extract, on the sender side, and then write-back, on the receiver side, each primitive object field individually. In a Big Data system, each data transfer involves many millions of objects, which would invoke these functions for millions of times or more. It is worth noting that reflection is a very expensive runtime operation. It allows the program to dynamically inspect or invoke classes, methods, fields, or properties without

type information available statically at the cost of time-consuming string lookups, and is undesirable in performance-critical tasks.

2. Reference adjustment: References contained in reference-type fields of transferred objects need to be adjusted, since those objects will be placed in different addresses on the receiver node. The Java serializer uses reflection to obtain and inline the contents of referenced objects into the binary representation of the referencing object. It constructs all objects reachable from O on the receiver machine using reflection, and then sets reference fields with the addresses of the just created referenced objects through reflection.

3. Type representation: Each type is represented by a special meta object in a managed runtime, and is referenced by the headers of the objects of the type. However, type references cannot be used to represent types in a byte sequence, because the meta objects representing the same type may have different addresses in different runtimes. The Java serializer represents every type by a string that contains the name of a class and all its super classes. This design causes meta data (i.e., type strings) to consume a huge portion of the byte sequence transferred across the network. Furthermore, *reflection* must be used to resolve the type from each string during object re-creation on the receiver node.

While many serialization/deserialization libraries [39, 106, 145] have been developed, large inefficiencies exist in their implementations. Both our own experience and evidence from previous work [131] show that data shuffling has a non-trival cost: serialization/deserialization accounts for 30% of the execution time in Spark. Microsoft researchers reported that 5% of all Google datacenter traffic are from Protobuffers that serialize and deserialize data between components [102].

CHAPTER 3

Mitigating Object-based Representation Mismatch

One major performance issue that the managed runtime suffers from is the memory bloat due to the wasted space for objects' header and excessive amount of pointers and references, leading to high space overhead and low memory packing factors. Comprehensive studies across many contemporary Big Data systems confirm that these overheads lead to significantly reduced scalability—e.g., applications crash with out-of-memory errors, although the size of the processed dataset is much smaller than the heap size [55, 136].

This chapter presents our effort in mitigating the object-based representation mismatch is Big Data systems. We develop a system named Facade that can optimize the memory overheads caused by the use of objects, and thus, provide bare-metal performance for all JVM-based Big Data applications. The key to success is reducing the ratio between auxiliary data and actual payload by bounding the number of heap objects and references at compiler level.

We begin with a study to validate the memory bloat issue in Section 3.1. Section 3.2 gives an overview of our approach, including several challenges and how we overcome them. Then, the following 2 sections discuss Facade execution model (3.3), and implementation details (3.4). After experimental results are presented in Section 3.5, we conclude the chapter in Section 3.6.

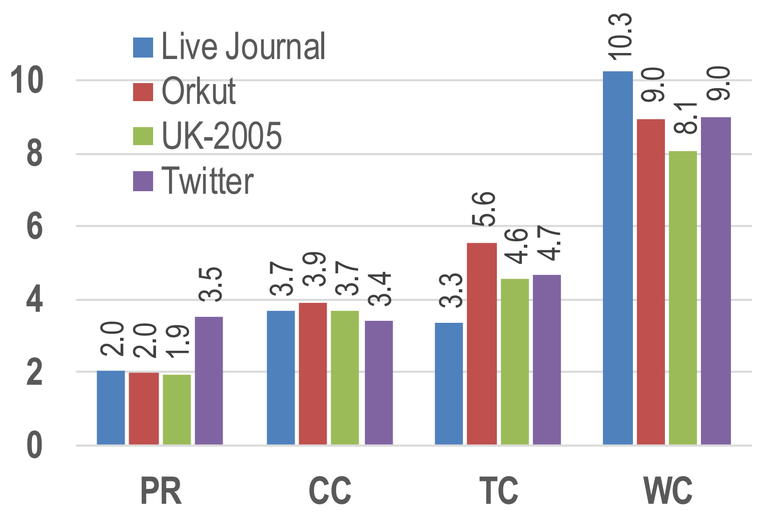


Figure 3.1: The ratios between the total bytes of data objects and the size of their actual payload

3.1 An Empirical Study on Memory Bloat Caused by the Use of Objects

To empirically quantify object headers and references space overhead , we ran three graph analytics programs and one data analytic program on Apache Spark [182], the de-facto leader in Big Data processing. 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 Page Rank (PR), Connected Components (CC) and Triangle Counting (TC); the other program was Word Count (WC) over four real-world graphs, Live Journal [42], Orkut [26], UK-2005 [50], and Twitter-2010 [107]. Kryo [106], a high-performance serializer, recommended for Spark, was used.

To understand the size difference, we modified Kryo to report the numbers of bytes occupied by data objects before and after they were serialized by Kryo. 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. Finally, these numbers were aggregated across all machines.

Figure 3.1 reports the ratios between these two numbers. Across all programs, on average, the space overheads of the object-based representation is 3.2×. The main reason for this large overhead is that these applications create billions of small data objects (e.g., `java.lang.Integer`, `java.lang.Long`, or `java.lang.Double`), whose corresponding header/-pointer overhead cannot be amortized by actual payloads. This experiment validates existing studies in [55, 136] and motivates our work.

3.2 Facade overview

The key observation made in the study is that to develop a scalable system, the number of *data objects and their references in the heap* must *not* grow proportionally with the cardinality of the dataset. To achieve this goal, we develop Facade, a compiler and runtime system, that aims to *statically* bound the number of heap objects during the execution of a data-intensive application, thereby significantly reducing the space and temporal overhead incurred by the object-based data representation.

3.2.1 Technical challenges and solutions

There are three major challenges in Facade's design. We briefly discuss these challenges and how Facade overcomes them.

Challenge 1: How to bound the size of the managed heap? Despite a rich body of work that attempts to reduce the number of objects [49, 61, 70, 80, 114, 152, 170], none of them can work with modern data processing frameworks whose codebases have millions lines of codes or they can work with large heaps containing billions of objects. This is because they all rely on expensive inter-procedural analyses which are known not able to scale to large systems.

In order to overcome this challenge, Facade does not use such heavy-weight analyses. Instead, we propose a novel execution model. Facade advocates to separate data storage from data manipulation: data are stored in the off-heap memory(i.e., not recognized as heap objects) while heap objects are created as *facades* only for control purposes such as function calls (i.e., bounded). As the program executes, a *many-to-one mapping* is maintained between arbitrarily many data items in the native memory and a statically-bounded set of facade objects in the heap. In other words, each facade keeps getting reused to represent data items.

To enforce this model, our Facade compiler transforms an existing data processing program into an (almost) object-bounded program: the number of heap objects created for a data class in one thread is bounded by certain source code properties (i.e., a compile-time constant).

More formally, Facade reduces the number of data objects from $O(s)$ to $O(t * n * m + p)$, where s represents the cardinality of the dataset, t is the number of threads, n is the number of data classes, m is the maximum number of facades in a data class's pool, and p is the number of page objects used to store data.

Challenge 2: What to transform? Real-world data-intensive systems have very large codebases and relied heavily on (third-party) libraries. Furthermore, features such as reflection and dynamic class loading are prevalently used to instantiate types in third-party libraries and frameworks that cannot be resolved statically, there is little hope that a whole-program analysis/transformation can be done for any real system. No real-world programs can be guaranteed to be safely transformed without using prohibitively expensive, sophisticated program analyses.

Luckily, as discussed in Section 2.3, there often exists a clear boundary between a *control path* and a *data path* in a Big Data processing system. Although the data path creates most of the runtime objects to represent and process data items, its implementation is rather simple and its code size is often small (e.g., 35% of lines of code [55]) . This property enables a systematic solution for data-intensive programs. Since a data path often contains simple data manipulation functions, developing a compiler to transform these functions is much more feasible than transforming the entire application.

Facade requires developers to provide a list of Java classes that form the data path. Each class in this list is transformed by our compiler into a *facade-based* class. Many performance problems result from the extensive use of large collections. For each collection class C (e.g., HashMap or ArrayList) in the standard Java library, we also transform it into a facade-based class C'. The original class C is used in normal ways in the control path while type C' will be used in the data path to substitute C.

Challenge 3: How to reclaim data objects? As data objects are no longer subject to garbage collection, an important question is how and when to reclaim them from native memory. A great deal of evidence shows that a data path is *iteration-based* [55, 77, 135]. In this chapter, "iteration" refers to a piece of data processing code that is repeatedly executed. Its definition includes but is more general than that of "computational iteration" performed in graph algorithms. For example, an iteration can also be a MapReduce task or a dataflow operator in data-parallel frameworks. Iterations are very well defined in data processing frameworks and can be easily identified by even novices. For example, in GraphChi [108], a computational iteration that loads shards into memory, processes vertices, and writes updates back to disk is explicitly defined as a pair of callbacks (iteration_start() and iteration_end()). It took us only a few minutes to find these iterations although we had never studied GraphChi before. In a data-parallel system such as Hadoop, the code for a Map or Reduce task can be considered as an iteration because it is repeatedly executed to process data partitions.

There is a strong correlation between the lifetime of an object and the lifetime of the iteration in which it is created: such objects often stay alive until the end of the iteration but rarely cross multiple iterations. Hence, we develop an *iteration-based memory management* that allocates data objects created in one iteration together in a native region and deallocates the region as a whole when the iteration finishes. There may be a small number of control objects that are also created in the iteration, and naively reclaiming the whole region may cause failures. We rely on developers to refactor the program code to move the creation of control objects out of the data path. In reality, this effort is very little because a data path rarely creates control objects (but it does use control objects passed from the control path).

3.3 The Facade Execution Model

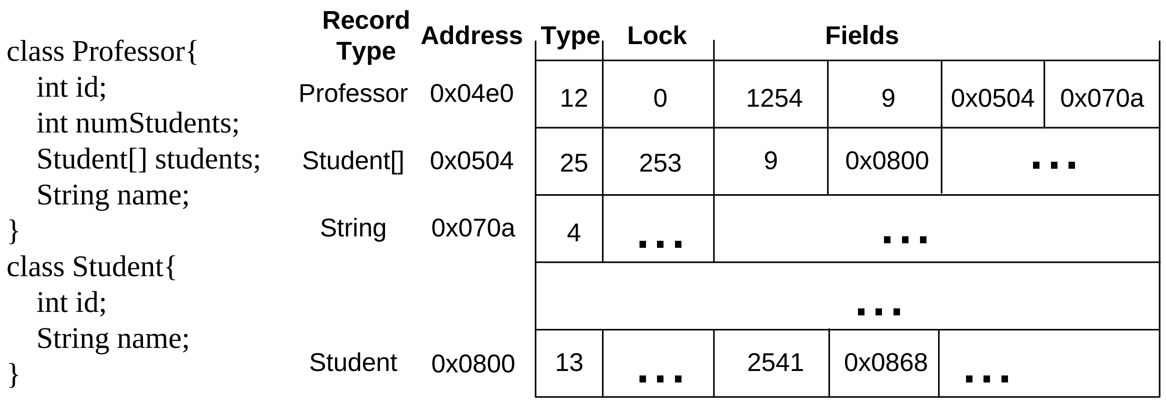
To overcome the fundamental problem of memory bloat caused by the object-based representation mismatch, we design the Facade framework that exploits compiler and runtime system support to separate data storage from data manipulation in a data-intensive program. The key idea is simple: data contents are allocated separately in native memory; heap objects no longer contain data, and they only provide data-manipulating methods. Objects now only represent data processors, not data contents, and hence, the number of heap objects is no longer proportional to the cardinality of the input dataset.

3.3.1 Data Storage Based on Native Memory

We propose to store data records in native, non-GCed memory. Similarly to regular memory allocation, our data allocation operates at the page granularity. A memory page is a fixed-length contiguous block of memory in the off-heap memory, obtained through a JVM's native support.

To provide a better memory management interface, each native page is wrapped into a Java object, with functions that can be inserted by the compiler to manipulate the page. Note that the number of page objects (i.e., p in $\mathrm{O}(t * n * m + p)$) cannot be statically bounded in our system, as it depends on the amount of data to be processed. However, by controlling the size of each page and recycling pages, we often need only a small number of pages to process a large dataset. The scalability bottleneck of an object-oriented data-intensive application lies in the creation of small data objects and data structures containing them; our system aims to bound their numbers.

From a regular Java program P, Facade generates a new program P', in which the data contents of each instantiation of a data class are stored in a native memory page rather than in a heap object. To facilitate transformation, the way of a data record is stored in a page is exactly the same as the way it was stored in an object except that the native-memory-based record does not contain a heap object's header and padding.

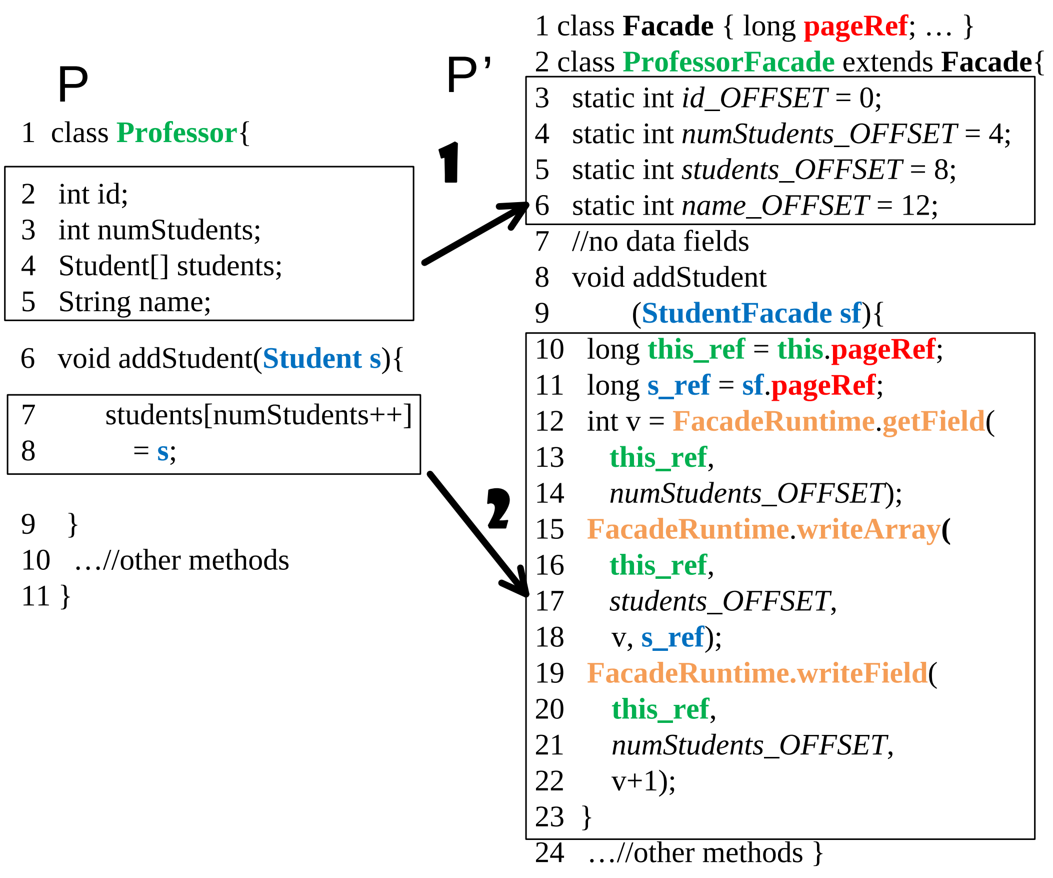


```
class Professor{
    int id;
    int numStudents;
    Student[] students;
    String name;
}
class Student{
    int id;
    String name;
}
```

Figure 3.2: A data structure in regular Java and its corresponding data layout in a native page.

Figure 3.2 shows the data layout for an example data structure in our page-based storage system. Each data record (which used to be represented by an object in P) starts with a 2-byte type ID, representing the type of the record. For example, the IDs for `Professor`, `Student[]`, `String`, and `Student` are 12, 25, 4, and 13, respectively. These types will be used to implement virtual method dispatch during the execution of P'. Type ID is followed by a 2-byte lock field, which stores the ID of a lock when the data record is used to synchronize a block of code. We find it sufficient to use 2 bytes to represent class IDs and lock IDs — in our experiments with large systems, the number of data classes is often much smaller than 2^{15}; so is the number of distinct locks needed. Details of the lock implementation and the concurrency support can be found in §3.4.5.

For an array record, the length of the array (4 bytes) is stored immediately after the lock ID. In the example, the number of student records in the array is 9. The actual data contents (originally stored in object fields) are stored subsequently. For instance, field `id` of the professor record contains an integer 1254; `numStudents` stores an integer 9; the fields `students` and `name` contain memory addresses `0x0504` and `0x070a`, respectively. These references are referred to as *page references*, as opposed to *heap references* in a normal Java program. Note that for efficiency, page references are not offsets into the native page where the memory is held, they are the absolute memory address so that the Facade runtime can operate directly on them without further calculation.



```
class Professor{
  int id;
  int numStudents;
  Student[] students;
  String name;
  void addStudent(Student s){
        students[numStudents++]
            = s;
  }
  …//other methods
}
```

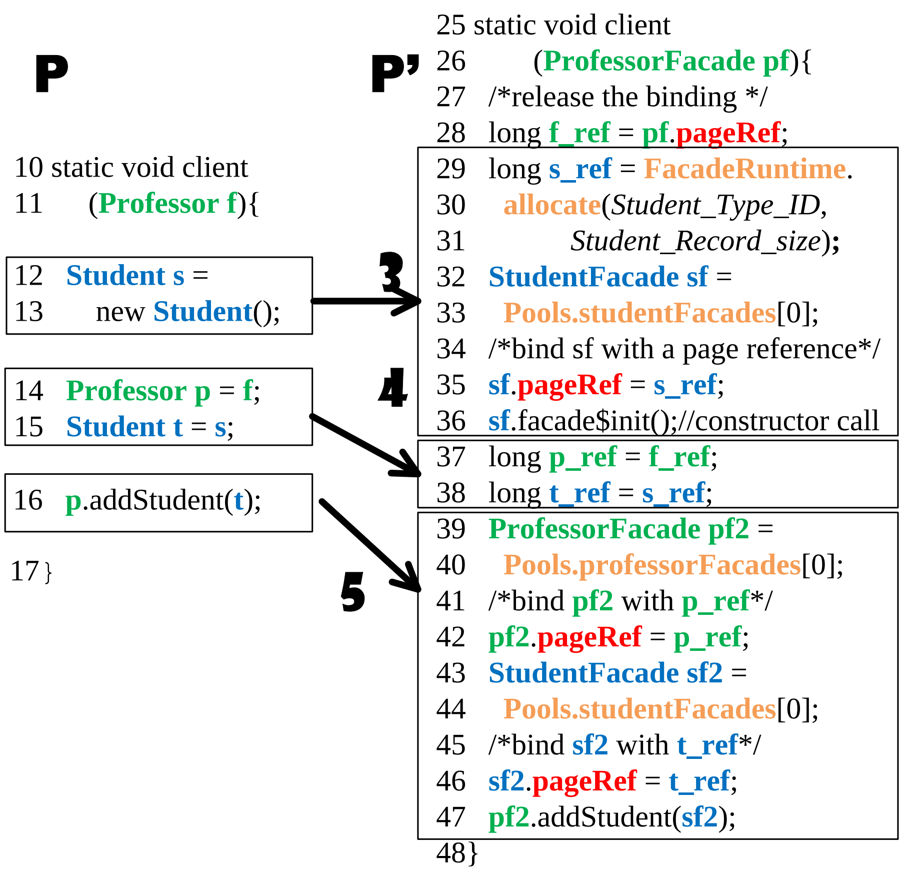
```
class Facade { long pageRef; … }
class ProfessorFacade extends Facade{
  static int id_OFFSET = 0;
  static int numStudents_OFFSET = 4;
  static int students_OFFSET = 8;
  static int name_OFFSET = 12;
  //no data fields
  void addStudent
        (StudentFacade sf){
   long this_ref = this.pageRef;
   long s_ref = sf.pageRef;
   int v = FacadeRuntime.getField(
     this_ref,
     numStudents_OFFSET);
   FacadeRuntime.writeArray(
     this_ref,
     students_OFFSET,
     v, s_ref);
   FacadeRuntime.writeField(
     this_ref,
     numStudents_OFFSET,
     v+1);
  }
  …//other methods }
```

Figure 3.3: A transformation example; part (a).

3.3.2 Using Objects as Facades

We propose to create heap objects as *facades* for a data class, that is, they are used only for control purposes such as method calls, parameter passing, or dynamic type checks, but do *not* contain actual data. Figure 3.3 and Figure 3.4 depict an example with five transformations using the same `Professor` class in Figure 3.2. For simplicity of illustration, we show the unoptimized version of the generated program, under the assumption that the program is single-threaded and free of virtual calls. We will discuss the support of these features later.

Class transformation The transformations #1 and #2 from Figure 3.3 show an example of class transformation. For illustration, let us assume both `Professor` and `Student` are data classes. For `Professor`, Facade generates a facade class `ProfessorFacade`, containing all methods defined in `Professor`. `ProfessorFacade` extends class `Facade`, which has a field



```
static void client
    (Professor f){
  Student s =
      new Student();
  Professor p = f;
  Student t = s;
  p.addStudent(t);
}
```

```
static void client
      (ProfessorFacade pf){
/*release the binding */
long f_ref = pf.pageRef;
long s_ref = FacadeRuntime.
  allocate(Student_Type_ID,
          Student_Record_size);
StudentFacade sf =
  Pools.studentFacades[0];
/*bind sf with a page reference*/
sf.pageRef = s_ref;
sf.facade$init();//constructor call
long p_ref = f_ref;
long t_ref = s_ref;
ProfessorFacade pf2 =
  Pools.professorFacades[0];
/*bind pf2 with p_ref*/
pf2.pageRef = p_ref;
StudentFacade sf2 =
  Pools.studentFacades[0];
/*bind sf2 with t_ref*/
sf2.pageRef = t_ref;
pf2.addStudent(sf2);
}
```

Figure 3.4: A transformation example; part (b)

`pageRef` that records the page reference of a data record (such as `0x0504` in Figure 3.2). Setting a page reference to the field `pageRef` of a facade binds the data record with the facade, so that methods defined in the corresponding facade class can be invoked on the facade object to process the record. Readers can think of this field as the `this` reference in a regular Java program.

`ProfessorFacade` does not contain any instance fields; for each instance field f in `Professor`, `ProfessorFacade` has a static field f_Offset, specifying the offset (in number of bytes) of f to the starting address of the data record. These offsets will be used to transform field accesses.

Method transformation For method `addStudent` in `Professor`, Facade generates a new method with the same name in `ProfessorFacade`. Because we no longer have any data objects, for each reference of a data object in the original program, we substitute it with either a page reference or a reference of a corresponding facade object using the following criteria:

- For any assignment, load, or store that involves a data object, the reference of the data object is substituted with its page reference. For example, in Figure 3.4, the transformation #4 replaces the variable assignments (Lines 14 – 15) in P with page reference assignments (Lines 37 – 38) in P'. In these cases, the generated statements do not have any heap objects involved.
- For parameter passing and value return in a call site, it is difficult to substitute object references completely with page references, because if an object is used as a receiver object to call a method, replacing it with a page reference would make it impossible to make the call. In this case, we replace references to data objects with references to their corresponding facade objects. For example, in Figure 3.3, the signature of method `addStudent` is changed in a way so that the `Student` type parameter is replaced with a new parameter of type `StudentFacade`.

In the generated `addStudent` method (Lines 8 – 23 in Figure 3.3), the new facade parameter sf is used only to pass the page reference of the data record that corresponds to the original parameter in P. The first task inside the generated method is to retrieve the page references (Lines 10 and 11 in P') from the receiver (i.e., `this`) and sf, and store them in two local variables $this\_ref$ and s_ref. Any subsequent statement that uses `this` and s in P will be transformed to use the page references $this\_ref$ and s_ref in P', respectively. The field accesses in Lines 5 and 6 in P are transformed to three separate calls to our library methods that read values from and write values to a native page. Note that what is written into the array is the page reference s_ref pointing to a student record — all references to regular data objects in P are substituted by page references in P'.

Allocation transformation In Figure 3.4, the allocation in Lines 12 – 13 in P is transformed to Lines 29 – 36 in P' (Transformation #3). Facade allocates space based on the size of type `Student` by calling a library method `allocate`, which performs native-memory-page-based allocation and returns a page reference *s_ref* which is a native memory address. Details of the allocation algorithm and memory management are discussed in §3.4.7.

Call site transformation Since statements in P' all use page references, a challenge in transforming a call site is how to generate the receiver object on which the call can be made. Our idea is to use facade objects (i.e., objects are created only as proxies). If a call is made on a data object in P, we can obtain a facade object to call the same method in P', because a data class and its corresponding facade class have the same methods. However, doing so naively would generate a large number of facade objects, which would still cause space and memory management overhead. Hence, special care needs to be taken to minimize the number of facade objects used.

We solve the problem by pooling facade objects. For each facade class, we maintain a pool that contains a small number of objects of the class. This number can be statically bounded as discussed shortly. For instance, before generating an allocation site, as shown in Transformation #3 in Figure 3.4, the Facade compiler first generates code to retrieve an available facade object from the pool (Lines 32 – 33 in P') and bind it with the page reference *s_ref* (Line 35). In this example, the first facade in the pool is available; the reason will be explained shortly. The constructor of class `Student` in P is converted to a regular method `facade$init` in P'. Facade then generates a call to `facade$init` on the retrieved facade object (Line 36).

Similarly, Transformation #5 shows how a call to method `addStudent` on the `Professor` object in P (Line 16) is transformed to a call to the same method on the `ProfessorFacade` object in P' (Line 47). This new call site needs (1) a receiver object and (2) a parameter object. To prepare for these objects, we generate the statements from Line 39 to Line 44, which retrieve a `ProfessorFacade` object *pf2* for the receiver and a `StudentFacade` object *sf2* for the parameter from their respective pools, and then bind them with their corresponding page references (Line 46). Finally, the call site in Line 47 is generated.

Note that the `ProfessorFacade` object *pf2* and the `StudentFacade` object *sf2* are needed because the object references p and s in P have been replaced with page references *p_ref* and *s_ref* in P', both of which have a `long` type. It would not be possible to call `addStudent` with these (long) page references. Hence, facade objects are retrieved to (1) enable the method call and (2) take the page references into the callee.

Code generation invariants Our code transformation algorithm maintains the following three major invariants, guaranteeing transformation correctness. First, for each reference r of a data object in P, P' must contain a page reference pr pointing to the native memory location at which the same object is stored. Since there is a one-to-one mapping between r and pr, any non-call statement that reads/writes the object referenced by r in P must have a corresponding statement reads/writes the native memory based object referenced by pr in P'.

Second, for two different references r and r' in P, the two corresponding page references pr and pr' in P' must be different as well. While facade objects are reused, page references are never shared among variables. This is straightforward to see given that Facade performs *literal translation* for allocation sites, loads, stores, and assignments.

Third, for each parameter (including receiver) of a data class D at each call site in P, a facade object of type *DFacade* is retrieved in P' from the *DFacade* object pool. The only purpose of the facade object is to pass a page reference between a caller and a callee. For example, for parameter passing, the page reference is written into a facade object right before the call, and then released from the facade and written into a local variable in the very beginning of the callee. For value returning, the page reference is written into a facade right before the return statement, and released and written into a stack variable immediately after the call site.

More formally, the invariant regarding the facade usage is that for a pair of instructions (e.g., s and t) that bind a facade with a page reference and release the binding, t is the immediate successor of s on the data dependence graph. In other words, no instructions between s and

t can read or write the facade object accessed by s or t. We refer to the period between the executions of s and t as a *use span* of the facade accessed by s and t. Examples of such instruction pairs include Lines 42 and 10, and Lines 46 and 11 in P' of Figure 3.4. This invariant guarantees that the page reference read from the facade object by t is exactly the one written into the same facade object by s, and thus, page references are appropriately propagated between methods.

3.3.3 Bounding the Number of Facades in Each Thread

Our facade pooling is different from traditional object pooling where the objects requested cannot be reused until they are explicitly returned to the pool. A facade object does not need to be explicitly returned because its goal is only to carry a page reference across the method boundary. Its use span automatically ends when the callee returns to the caller (if used for value return) or the callee is about to execute (if used for parameter passing). In other words, the way facades are used dictates that the use spans of the facade objects requested at different statements are completely *disjoint*. Hence, in most cases, upon a request for a facade (e.g., at a call site), all facades in the pool are available to use. This explains why it is always safe to use the first facade of the pool in Lines 33, 40, and 44 in Figure 3.4.

One exception is that if a call site has multiple parameters of the same data class, multiple objects of the corresponding facade class are needed simultaneously to pass page references. Hence, the number of facades needed for a data class depends on the number of parameters of this class needed in a call site. For example, if a call site in P requires n parameters of type `Student`, we need at least n `StudentFacade` objects in P' for parameter passing (e.g., `Pools.studentFacades[0]`, …, `Pools.studentFacades[`n` - 1]`). The number of facades for type `StudentFacade` in P' is thus bounded by the maximal number of `Student`-type parameters needed by a method call in P. Based on this observation, we can inspect all call sites in P in a pre-transformation pass and compute a bound statically for each data class. The bound will be used to determine the size of the facade pool for that type (e.g., `Pools.studentFacades`) *at compile time.*

Note that at different statements, different facades may be retrieved from the pool to carry the same page reference. For instance, in Figure 3.4, although variable p (Line 16) and parameter f (Line 11) refer to the same object in P, their corresponding facades *pf* and *pf2* in P' may not be the same. In a single-threaded execution, this would not cause any inconsistency because page references determine data records and facades are used only to execute control flow. Naively doing so in a multi-threaded environment may lead to concurrency bugs. Hence, for a multi-threaded program, a facade pool has to be maintained for each thread to ensure thread safety. Since a program can create an arbitrary number of threads, the number of facades will no longer be statically bounded. However, the number of threads in data path is often very small, and hence, the total number of facade objects is still much smaller than the dataset cardinality.

3.3.4 Performance Benefits

P' has the following three clear performance advantages over P. First, all data records are stored in native pages and no longer subject to garbage collection. This can lead to an orders-of-magnitude reduction in the number of nodes and edges traversed by the GC.

Second, native-memory-based data storage reduces the memory access cost. Since all (billions of) data objects are represented as native bytes, Facade eliminates many sources of runtime overhead, including pointer chasing, write barriers (i.e., a piece of code executed per object write for GC purposes), memory dereferences, and array bound checks. The saving will reduce total execution time.

Third, significant reduction in memory consumption can be achieved for the following two reasons. (1) Each data record has only a 4-byte "header" space (8 bytes for an array) in P' while the size of an object header is 12 bytes (16 bytes for an array) in P. This is due to the reduction of the lock space as well as the complete elimination of space used for GC. (2) As discussed shortly in §3.4.10, Facade inlines all data records whose size can be statically determined, which reduces memory consumption for storing object headers and improves data locality. The savings in memory usage may potentially lead to reduced computing resource (e.g., number of CPUs or compute nodes) needed to process a dataset.

3.4 Facade Design and Implementation

To use Facade, a user needs to provide a list of data classes that form the data path of an application. Our compiler transforms the data path to page-allocate objects representing data items without touching the control path. This handling enables the design of a simple intra-procedural analysis and transformation as well as aggressive optimizations (such as type specialization), making it possible for Facade to scale to large-scale systems.

3.4.1 Our Assumptions

Based on the user-provided list of data classes, Facade makes two important "closed-world" assumptions based on our experience with dozens of real-world, data-intensive systems.

- ***Reference-closed world*** Facade requires all reference-typed fields declared in a data class to have data types. This is a valid assumption — there are two major kinds of data classes in a data-intensive application: classes representing data tuples (e.g., graph nodes and edges) and those representing data manipulation functions, such as sorter, grouper, etc. Both kinds of classes rarely contain fields of non-data types. Java supports a collections framework and data structures in this framework can store both data objects and non-data objects. In Facade, a collection (e.g., HashMap) is treated as a data class; a new class (e.g., HashMapFacade) is thus generated in the data path. The original class is still used in the control path. If Facade detects a data object flows from the control path to the data path or a paged data record flows the other way around, it automatically synthesizes a *data conversion function* to convert data formats. Detailed discussion can be found in §3.4.6.

- ***Type-closed world*** This assumption requires that for a data class c, c's superclasses (except `java.lang.Object`, which is the root of the class hierarchy in Java) and subclasses must be data classes. This is also a valid assumption because a data class usually does not inherit a non-data class (and vice versa). The assumption makes it possible for us to determine the field layout of a data record in a page — fields declared in a superclass are stored before fields in a subclass and their offsets can all be statically computed. The Facade compiler computes a closure of classes to be transformed from an initial list of user-specified data classes based on the inheritance relationships.

We allow both a data class and a non-data class to implement the same Java interface (such as `java.lang.Comparable`). Doing this will not create any page layout issue because an interface does not contain instance fields. Facade checks these two assumptions before transformation and reports compilation errors upon violations. The developer needs to refactor the program to fix the violations.

Boundary Classes. Based on the user-defined list of data classes, our compiler automatically identifies a set of *boundary classes*, via which the control and data paths interact. For example, a class A is a boundary class if A is not on the list itself, but has an instance field whose type is on the list. Boundary classes feed data to the data path before processing and later send the results back to the control path. They often have a mix of data-type and non-data-type fields. While Facade can transform all of them, page allocation of non-data objects would not bring much benefit. We design a `@DataObject` pragma which can be used by the developer to annotate data fields of a boundary class. Facade only page-allocates annotated objects.

3.4.2 Data Class Transformation

Class hierarchy transformation For each method m in a data class D, Facade generates a new method m' in a facade class *DFacade* such that m and m' have the same name; for each parameter of a data class type T in m, m' has a corresponding parameter of a facade type *TFacade*. No special treatment is required for overloaded methods since each method m has its distinct Facade method m'. If D extends another data class E, this relationship is preserved by having *DFacade* extend *EFacade*. All static fields declared in D are also in *DFacade*; however, *DFacade* does not contain any instance fields.

One challenge here is how to appropriately handle Java interfaces. If an interface I is implemented by both a data class C and a non-data class D, and the interface has a method that has a data-class type parameter, changing the signature of the method will create inconsistencies. In this case, we create a new interface *IFacade* with the modified method and make all facades *DFacade* implement *IFacade*. While traversing the class hierarchy to transform classes, Facade generates a type ID for each transformed class. This type ID is actually used as a pointer that points to a facade pool corresponding to the type — upon a virtual dispatch, the type ID will be used to retrieve a facade of the appropriate type at run time.

Instruction transformation Instruction transformation is performed on the control flow graph (CFG) of a single static assignment (SSA)-based intermediate representation (IR). The output of the transformation is a new CFG containing the same basic block structures but different instructions in each block. The transformations for different kinds of instructions are summarized in Table 3.1. Here we discuss only a few interesting cases. For a field write in (i.e., $a.f = b$ in case 3), if b has a data type but a does not (case 3.3), Facade considers this write as an *interaction point* (IP), an operation at which data flows across the control-data boundary. Facade synthesizes a data conversion function `long convertToB(B)` that converts data format from a paged data record back to a heap object (see §3.4.6). If a has a data type but b does not (case 3.4), Facade generates a compilation error as our first assumption (that data types cannot reference non-data types) is violated. The developer needs to refactor the program to make it Facade-transformable.

An IP may also be a load that reads a data object from a non-data object (case 4.3) or a method call that passes a data object into a method in the control path (case 6.3). At each IP, data conversion functions will be synthesized and invoked to convert data formats. Note that data conversion often occurs before the execution of the data path or after it is done.

Instructions in P	***Conditions***	***Code generation in*** P'
(1) Method prologue	(1.1) s is a parameter of data type in P	Create a variable s_ref for each facade parameter sf; emit instruction $s\_ref = sf.pageRef$; add $\langle s, s\_ref\rangle$ into the variable-reference table v
(2) $a = b$	(2.1) a has a data type	Look up table v to find the reference variable b_ref for b; emit instruction $a\_ref = b\_ref$; add $\langle a, a\_ref\rangle$ into v
	(2.2) Otherwise	Generate $a = b$
(3) $a.f = b$	(3.1) Both a and b have data types	Retrieve a_ref and b_ref from table v; emit a call `setField` (a_ref, f_Offset, b_ref)
	(3.2) Neither of them have a data type	Emit $a.f = b$
	(3.3) b has a data type, a doesn't (Interaction Point)	Synthesize a data conversion function B $covertToB(long)$; emit a call $a.f = convertToB(b\_ref)$
	(3.4) a has a data type, b doesn't	Assumption violation; generate a compilation error
(4) $b = a.f$	(4.1) Both a and b have data types	Retrieve a_ref from table v; emit a call $b\_ref =$ `getField` (a_ref, f_Offset); add $\langle b, b\_ref\rangle$ into v
	(4.2) Neither of them have a data type	Emit instruction $b = a.f$
	(4.3) a has a data type, b doesn't (Interaction Point)	Synthesize a data conversion function $long$ $covertFromB(B)$; emit a call $b\_ref = convertFromB(a.f)$; add $\langle b, b\_ref\rangle$ into v
	(4.4) b has a data type but a doesn't	Assumption violation; generate a compilation error
(5) $return\ a$	(5.1) a has a data type	Retrieve a_ref from v; emit three instructions: $AFacade\ af = Pools.aFacades[0]$; $af.pageRef = a\_ref$; $return\ af$
	(5.2) Otherwise	Emit instruction $return\ a$
(6) $a.m(\ldots, b, \ldots)$	(6.1) Both a and b have data types; b is the i-th parameter that has type B	Retrieve a_ref and b_ref from table v; emit five instructions: $AFacade\ af = resolve(a\_ref)$; $BFacade\ bf = Pools.bFacades[i]$; $af.pageRef = a\_ref$; $bf.pageRef = b\_ref$; $af.m(\ldots, bf, \ldots)$
	(6.2) a has a data type, b doesn't	Emit the same instructions as (6.1), except the last call is $af.m(\ldots, b, \ldots)$
	(6.3) b has a data type, a doesn't (Interaction Point)	Synthesize function B $covertToB(long)$; emit a call $a.m(\ldots, covertToB(b\_ref), \ldots)$
	(6.4) Neither of them have a data type	Emit a call $a.m(\ldots, b, \ldots)$
(7) $boolean\ t = a$ $instanceof\ B$	(7.1) a has a data type and B is a data type	Retrieve a_ref from v; emit two instructions: $AFacade\ af = resolve(a\_ref)$; $t = af\ instanceof\ BFacade$
	(7.2) B is an array type	Emit $t = arrayTypeID(a) == ID(B)$
	(7.3) Neither of them have a data type	Emit $t = a\ instanceof\ B$
(8) $a == b$	(8.1) a and b have a data type	Retrieve a_ref and b_ref from table v; emit a call $a\_ref == b\_ref$
	(8.2) They neither have a data type	Emit $a == b$

Table 3.1: A summary of code generation. Suppose variables a and b have types A and B, respectively.

Hence, these conversion functions would often not be executed many times and cause much overhead.

Resolving types In two cases, we need to emit a call to a method named `resolve` to resolve the runtime type corresponding to a page reference. First, when a virtual call $a.m(b, \ldots)$ is encountered (case 6.1), the type of the receiver variable a often cannot be statically determined. Hence, we generate a call `resolve`(a_ref), which uses the type ID of the record pointed to by a_ref to find a facade of the appropriate type. However, since this information can be obtained only at run time, it creates difficulties for the compiler to select a facade object as the receiver from the pool (i.e., what index i should be used to access `Pools.aFacades`$[i]$).

To solve the problem, we maintain a separate *receiver facade pool* for each data class. The pool contains only a single facade object; the `resolve` method always returns the facade from this pool, which is distinct from the parameter pool. Note that we do not need to resolve the type of a parameter (say b), because b is not used as a receiver to call a method. We can simply obtain a facade from the parameter pool based on b's declared (static) type, and use it to carry b's page reference.

The second case in which we need a `resolve` is the handling of an `instanceof` type check, which is shown in case 7 of Table 3.1.

3.4.3 Type specialization for `Object` and `Object[]`

Variables and parameters whose static types are `Object` and `Object[]` introduce additional challenges, because whether they reference data objects or not cannot be determined statically. Instead of using a complicated case analysis that generates different handling for different runtime types, Facade speculatively treats these variables as data-typed variables and generates code to validate this assumption at run time. Upon a violation (e.g., a variable/-parameter is not an instance of `Facade`), the generated program P' will throw an exception, and the developer can “blacklist” these variables/parameters to disable the speculation and recompile the program.

The usage of these general types depends heavily on applications. In fact, in the three frameworks we have experimented with, they rarely declare variables with `Object` and `Object[]`. We have only encountered six methods (in the application code) with parameters of the `Object` or `Object[]` type and these parameters were indeed used to pass data objects. However, for other applications such has Hive, methods with general-type parameters are extensively used. After a detailed inspection of Hive, we found almost all of these parameters represent data objects — since Hive is a data warehouse, it is designed to process queries in a way that is very similar to a database. Many methods simply perform filtering or aggregation on generic data records regardless of their types. Hence, we expect our speculative handling to be still effective for those applications.

3.4.4 Computing Bounds

Before the transformation, Facade inspects the parameters of each method in the data path to compute a bound for each data class. This bound will be used as the length of the facade array (i.e., the parameter pool) for the type. Note that the bound computation is based merely on the static types of parameters. Although a parameter with a general type may receive an object of a specific type at run time, a facade of the general type will be sufficient to carry the page reference of the data record (as discussed above) from a caller to a callee. Since we use a separate pool for receivers, the target method will always be executed appropriately. If the declared type of a parameter is an abstract type (such as interface) that cannot have concrete instances, we find an arbitrary (concrete) subtype c of this abstract type, and attribute the parameter to c when computing bounds. Facade generates code to retrieve a facade from c's pool to pass the parameter.

Once the bound for each data class is calculated, Facade generates the class `Pools` by allocating, for each type, an array as a field whose length is the bound of the type. The array will be used as the parameter pool for the type. Facade generates an additional field in `Pools` that

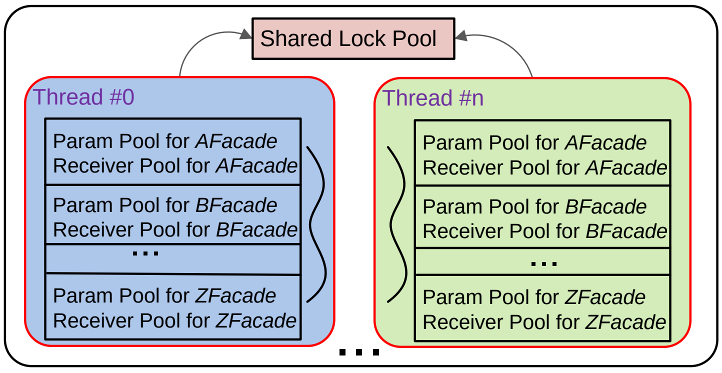


Figure 3.5: A graphical representation of threads and pools, where *AFacade*, *BFacade*, ..., and *ZFacade* are facade types.

references its receiver pool (i.e., one single facade) for the type. Eventually, Facade emits an `init` method in `Pools`, which will be invoked by our library to create facade instances and populate parameter pools.

3.4.5 Supporting Concurrency

Naively transforming a multi-threaded program may introduce concurrency bugs. For example, in P', two concurrent threads may simultaneously write different page references into the same facade object, leading to a data race. The problem can be easily solved by performing thread-local facade pooling: for each data class, the receiver pool and the regular pool are maintained for each thread. We implement this by associating one instance of class `Pools` with each thread; the `init` method (discussed earlier in §3.4.4) is invoked upon the creation of the thread.

Both implicit and explicit locks are supported in Java. Explicit locking is automatically supported by Facade: all `Lock` and `Thread` related classes are in the control path and not modified by Facade. For implicit locking (i.e., the intrinsic lock in an object is used), we need to add additional support to guarantee the freedom of race conditions. One possible solution is as follows: for each object o that is used as a lock in a `synchronized` $(o)\{\ldots\}$ construct (i.e., which is translated to an `enterMonitor`(o) and an `exitMonitor`(o) instruction to protect the code in between), Facade emits code to obtain a facade o' corresponding to o (if o has a data type) and then generates a new construct `synchronized` $(o')\{\ldots\}$. However, this handling may introduce data races—for two code regions protected by the same object in P, two different facades (and thus distinct locks) may be obtained in P' to protect them.

We solve the problem by implementing a special lock class and creating a new *lock pool* (shown in Figure 3.5) that is shared among threads; each object in the pool is an instance of the lock class. The lock pool maintains an atomic bit vector, each set bit of which indicates a lock being used. For each `enterMonitor`(o) instruction in P, Facade generates code that first checks whether the lock field of the data record corresponding to o already contains a lock ID. If it does, we retrieve the lock from the pool using the ID; otherwise, our runtime consults the bit vector to find the first available lock (say l) in the pool, writes its index into the record, and flips the corresponding bit. We replace o with l in `enterMonitor` and `exitMonitor`, so that l will be used to protect the critical section instead.

Each lock has a counter that keeps track of the number of threads currently blocking on the lock; it is incremented upon an `enterMonitor` and decremented upon an `exitMonitor`. If the number becomes zero at an `exitMonitor`, we return the lock to the pool, flip its corresponding bit, and zero out the lock space of the data record. Operations such as `wait` and `notify` will be performed on the lock object inside the block.

Worst-case object numbers in P and P' In P, each data item needs an object representation, and thus, the number of heap objects needed is $O(s)$, where s is the cardinality of the input dataset. In P', each thread has a facade pool for a data class. Suppose the maximum number of facades needed for a data class is m, a compile-time constant. The total number of facades in the system is thus $O(t*n*m)$, where t and n are the numbers of threads and data classes, respectively. Considering the additional objects created to represent native pages, the number of heap objects needed in P' is $O(t*n*m+p)$, where p is the number of native pages.

Note that the addition of the lock pool does not change this bound. The number of lock objects needed first depends on the number of synchronized blocks that can be concurrently executed (i.e., blocks protected by distinct locks), which is bounded by the number of threads t. Since intrinsic locks in Java are reentrant, the number of locks required in each thread also depends on the depth of nested synchronized blocks, which is bounded by the maximal depth of runtime call stack in a JVM, a compile-time constant. Hence, the number of lock objects is $O(t)$ and the total number of objects in the application is still $O(t * n * m + p)$. In our evaluation, we have observed that the number of locks needed is always less than 10.

3.4.6 Data Conversion Functions

For each IP that involves a data class D, Facade automatically synthesizes a conversion function for D; this function will be used to convert the format of the data before it crosses the boundary. An IP can be either an *entry* point at which data flows from the control path into the data path or an *exit* point at which data flows in a reverse direction. For an entry point, a `long convertFromA(A)` method is generated for each involved data class A; the method reads each field in an object of A (using reflection) and writes the value into a page. Exit points are handled in a similar manner. Our experiments on three systems show that the number of conversion functions needed is very small (≤ 4). This is expected and consistent with our observation that the control path and data path are well separated in data-intensive programs.

3.4.7 Memory Allocation and Page Management

The Facade runtime system maintains a list of pages, each of which has a size of 32KB (i.e., a common practice in the database design [87]). To improve allocation performance, we classify pages into size classes (similarly to what a high-performance allocator would do for a regular program), each used to allocate objects that fall into a different size range. When allocating a data record on a page, we apply the following two allocation policies whenever possible: (1) back-to-back allocation requests (of the same size class) get contiguous space to maximize locality; (2) large arrays (whose sizes are $\geq$ 32KB) are allocated on empty pages: allocating them on non-empty pages may cause them to span multiple pages, therefore increasing access costs. Otherwise, we request memory from the first page on the list that has enough space for the record. To allow fast allocation for multi-threading, we create a distinct page manager that maintains separate size classes and pages per thread so that different threads concurrently allocate data records on their thread-local pages. Having distinct page managers also eliminates potential fragmentation issue associated with size classes.

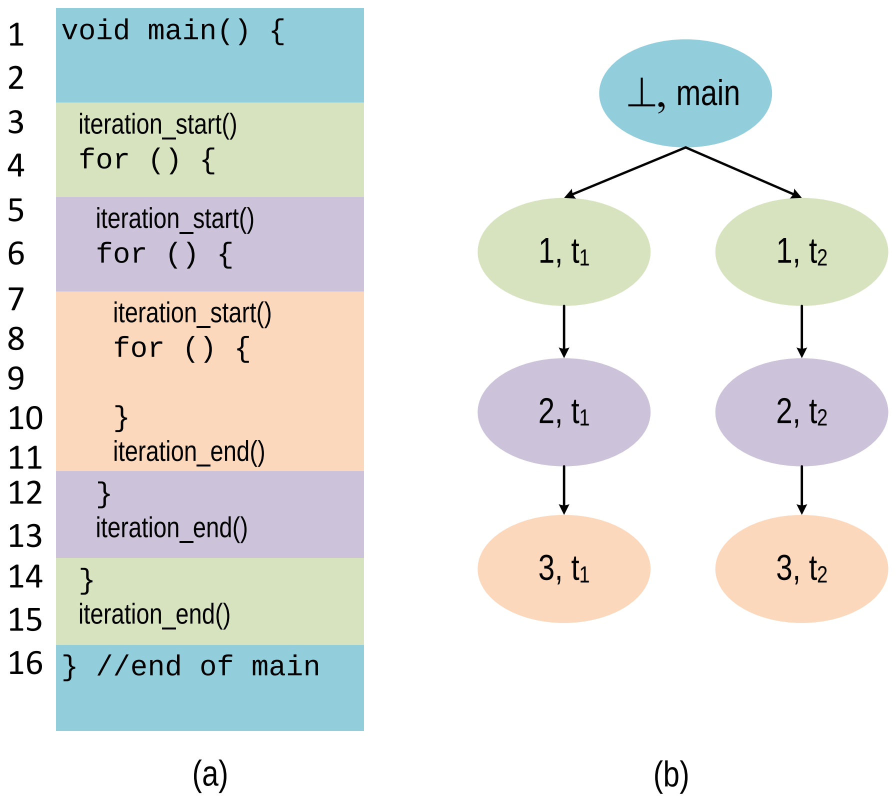
The data path is iteration-based. We define an iteration to be a repeatedly executed block of code such that the lifetimes of data objects created in different executions of this block are completely disjoint. In a typical data-intensive program, a dataset is often partitioned before being processed; different iterations of a data manipulation algorithm (e.g., sorting, hashing, or other computations) then process distinct partitions of the dataset. Hence, pages requested in one iteration of P' are released all at once when the iteration ends. Although different data processing frameworks have different ways of implementing the iteration logic, there often exists a clear mark between different iterations, e.g., a call to `start()` to begin an iteration and a call to `flush()` to end it.

We rely on a user-provided pair of `iteration_start()` and `iteration_end()` calls to manage our pages. Upon a call to `iteration_start()` that signals the beginning of an iteration, we create a page manager that will perform page allocation and memory allocation as discussed above. All pages are recycled immediately upon a call to `iteration_end()`. While this may sound non-trivial, our experience with a variety of applications shows that iterations are often very well-defined and program points to place these calls can be easily found even by novices without much understanding of the program logic. For example, in GraphChi [108], a single-machine graph processing framework, `iteration_start()` and `iteration_end()` are the callbacks explicitly defined by the framework. Although we had had zero knowledge about this framework, it took us only a few minutes to find these events. Note that iteration-based memory management is used only to deallocate data records and it is unsafe to use it to manage control objects. Those objects can cross multiple iterations and, hence, we leave them to the GC for memory reclamation.

In order to quickly recycle memory, we allow the developer to register nested iterations. If a user-specified `iteration_start()` occurs in the middle of an already-running iteration, a sub-iteration starts; we create a new page manager, make it a child of the page manager for the current iteration, and start using it to allocate memory. The page manager for a thread is made a child of the manager for the iteration where the thread is created. Hence, each page manager has a pair $\langle iterationID, thread \rangle$ identifier and they form a tree structure at run time. When a (sub-)iteration finishes, we simply find its page manager m and recursively release pages controlled by the managers in the subtree rooted at m. Recycling can be done efficiently by creating a thread for each page manager and letting them reclaim memory concurrently.

Since each thread t is assigned a page manager upon its creation, the pair identifier for its default page manager is $\langle \perp, t \rangle$; $\perp$ represents the fact that no iteration has started yet. Data records that need to be created before any iteration starts (e.g., usually large arrays) are allocated by this default page manager and will not be deallocated until thread t terminates.

To illustrate, Figure 3.6(a) and (b) show, respectively, a simple program and the corresponding page managers created in Facade. In the beginning, no iteration has started yet, all allocation requests are handled by the default page manager $\langle \perp, main \rangle$. Assuming there are 2 worker threads t_1 and t_2 executing the program. Upon the call to iteration_start in Line 3, 2 page managers $\langle 1, t_1 \rangle$ and $\langle 1, t_2 \rangle$ are created and placed as the children of the default manager to handle allocations in thread t_1 and t_2, respectively, for iteration #1. The call to `iteration_start()` in Line 5 signals the start of the sub-iteration #2. Consequently, the page managers $\langle 2, t_1 \rangle$ and $\langle 2, t_2 \rangle$ are created as the children of the managers $\langle 1, t_1 \rangle$ and $\langle 1, t_2 \rangle$, respectively. Facade allows arbitrarily nested iterations. When the execution encounters another `iteration_start()` in Line 7, Facade creates two page managers $\langle 3, t_1 \rangle$ and $\langle 3, t_2 \rangle$. The calls to `iteration_end()` in Lines 11, 13, and 15 triggers page reclamation in the reverse order of page allocation.



```
1   void main() {
2
3     iteration_start()
4     for () {
5       iteration_start()
6       for () {
7         iteration_start()
8         for () {
9
10        }
11        iteration_end()
12      }
13      iteration_end()
14    }
15    iteration_end()
16  } //end of main
```

Figure 3.6: A program (a) and its corresponding page manager tree after Line 7 is executed (b), assuming 2 threads t_1 and t_2 are executing.

3.4.8 Correctness and Profitability Arguments

It is easy to see the correctness of the class transformation and the generation of data accessing instructions as shown in Table 3.1, because the data layout in a native memory page is the same as in a heap object. This subsection focuses on the following aspects of correctness:

Facade usage correctness If a page reference is assigned to a facade that has not released another page reference, a problem would result. However, it is guaranteed that this situation will not occur because (1) a thread will never use a facade from another thread's pool and (2) for any index i in a facade pool p, the page reference field of $p[i]$ will never be written twice without a read of the field in between. The read will load the page reference onto the thread's stack and use it for the subsequent data accesses.

Memory management correctness Iteration-based memory management coverts dynamic memory reclamation to static reclamation and it is very difficult to make it correct for general objects in a scalable way. Facade performs iteration-based deallocation only for data items in native memory. Data items allocated in one iteration represent the data partition processed in the iteration. These items will often not be needed when a different data partition is processed (in a different iteration). Since practicality is our central design goal, we choose not to perform any conservative static analysis (e.g., escape analysis [61]) to verify whether data items can escape. A real-world data-intensive application often relies heavily on (third-party) libraries and the heavy use of interfaces in the program code makes it extremely difficult for any interprocedural analysis to produce precise results. Instead, we simply assume that instances of the user-specified data classes can never escape the iteration boundary. However, if a data object escapes an iteration through a control object, the synthesized conversion function will convert the paged record to an object.

Transformation safety Our transformation is mostly local and does not change the control flow of the original program. This feature makes it easier for Facade to preserve semantics in the presence of complicated language constructs such as exception handling (i.e., exception throwing logic is not changed by Facade: checked exception will be thrown in regular ways while unchecked exception still crashes the program). In other words, our transformation is safe. The memory management correctness thus relies solely on the user's correct specification of data classes. Section 3.5 reports our own experiences with finding data classes for real-world programs that we have never studied.

Transformation profitability Not all Java programs are suitable for Facade transformations. It is profitable using Facade when (1) the number of runtime objects of a class is exceptionally large and (2) the lifetimes of these objects follow epochal patterns, that is, they align with computational iterations in which they are created. While Facade works well for data-intensive systems, they may not be as effective to transform regular Java applications for performance improvement.

3.4.9 Modeling of Important Java Library Classes

All the primitive type wrappers in Java (e.g., `java.lang.Integer`, `java.lang.Float`, etc.) are considered as data classes. We manually implement the `StringFacade` class to support all string-related operations instead of generating it automatically from the `java.lang.String` class. This is, first, because we want to inline characters, rather than creating a two-layer structure (as done in the Java String implementation). In addition, the Java String implementation has many references and dependencies, which would make Facade transform classes we believe to be in control path (e.g., classes representing formats and calendars). Records for all primitive type wrappers and strings are inlined.

Commonly-used native methods such as `System.arraycopy` and `Unsafe.compareAndSwap` are manually modeled to operate on Facade-page-based data because it is not possible to transform native library methods. In generated programs, the original native methods are replaced with our own version. In addition, we provide implementations of the methods `hashCode`, `equals`, and `clone` in class `Facade`. For example, `equals` are implemented by using page reference as a substitution of object identity, while `hashCode` is implemented by computing a hash code based on the page reference contained in the facade.

A real-world program makes heavy use of collection classes in the `java.util` framework. Their implementations in the Oracle JDK often reference many other classes; for example, more than 100 classes all over the JDK can be transitively reached from class `HashMap`, and many of these classes have nothing to do with data processing. Instead of transforming all these classes, we create our own (page-based) implementations for all important collection classes, including various types of maps, sets, and lists. For example, since `HashMap` is in the data path, all its superclasses except `java.lang.Object` are included in the data path and they are all transformed manually into facade classes.

It took us about two weeks of programming to develop our own versions of library classes. The major part of the development is easy — we simply follow the logic from the original collection classes in the JDK. However, one challenge is how to break dependencies to classes that are in the control path. If a data class has a field of a non-collection class type, we carefully inspected the class to understand whether removing the field would cause any semantic inconsistencies. If it would, we include it in the data path and transform it into a facade class; otherwise, we remove the field and replace the code that uses the field with our own version that implements the same logic in different ways. Rigorous testing was performed to ensure that our class collections have the same semantics as their JDK counterparts.

3.4.10 Implementation and Optimizations

We have implemented Facade based on the Soot Java compiler infrastructure [3]. Facade consists of approximately 40K lines of Java code. Our transformation works on Jimple, a SSA-based three-address IR; the transformation occurs after all traditional dataflow optimizations are performed to eliminate redundancies in P, such as dead code and unnecessary loads. We develop a few additional optimizations that target common operations we have observed in data-intensive applications.

- **Array inlining** For a data array whose element size can be statically determined, we inline its elements to improve data locality and reduce pointer dereference costs. Upon the creation of an array, we allocate $l \times s$ bytes of memory where l and s are the array length and its element size. We perform a static analysis that identifies objects that *must* be written into the array and remove their allocation sites. Their indices in the array will be used as their page references. Although array inlining may lead to wasted space for general programs, it is very effective for data-intensive applications in which large arrays are often created and their elements are often “owned” by the arrays (i.e., accessed only through the arrays).

- **Special handling of oversized objects** Keeping large arrays that are no longer used in a page can lead to excessive memory usage. To solve the problem, we create a special allocate function in the page manager that allows us to allocate pages bigger than 32KB for large arrays. Each large array will take one single page, instead of spanning multiple (32KB) pages. These pages are located in an “oversize” class and can be freed manually (after they are no longer used) without waiting until the end of an iteration. Right now this optimization is enabled only in the implementation of the `resize` function in various collection classes, and it has been shown to be effective in improving memory efficiency of applications that make heavy use of maps and lists. We are in the process of developing an automated analysis that can detect such early deallocation opportunities for large objects in the generated code.

- **Static resolution of virtual calls** We use Spark [113], a simple and inexpensive context-insensitive points-to analysis to statically resolve virtual calls. For the resolved calls, their receiver facades can be statically determined.

3.5 Evaluation

We selected 3 different data processing frameworks and used Facade to transform their data paths. Our evaluation on 7 common data analytical applications on both single machines and clusters shows that, even for already well-optimized systems, Facade can still improve their performance and scalability considerably.

3.5.1 GraphChi

Transformation GraphChi [108] is a high-performance graph analytical framework that has been well optimized for efficient processing of large graphs on a single machine. Since we had not had any previous experience with GraphChi, we started out by profiling instances of data classes to understand the control and data path of the system. The profiling results show that `ChiVertex`, `ChiPointer`, and `VertexDegree` are the only three classes whose instances grow proportionally with the input data size. From these 3 classes, Facade detected 18 *boundary classes* that interact with data classes but do not have many instances themselves. Boundary classes have both data and non-data fields. We allow the user to annotate data fields with Java pragmas so that Facade can transform these classes and only page allocate their data fields.

With about 40 person-hours work (to understand data classes, profile their numbers, and annotate boundary classes for a system we had never studied before), Facade transformed all of these classes (7753 Jimple instructions) in 10.3 seconds, at a speed of 752.7 instructions per second. Iterations and intervals are explicitly defined in GraphChi—it took us only a few minutes to add callbacks to define iterations and sub-iterations.

Test setup We tested the generated code and compared its performance with that of the original GraphChi code. The experiments were performed on a 4-core server with 4 Intel Xeon E5620 (2.40GHz) processors and 50GB of RAM, running Linux 2.6.32. We experimented extensively with two representative applications, Page Rank (PR) and Connected Components (CC). The graph used was the Twitter-2010 graph [107], consisting of 42M vertices and 1.5B edges.

We used the Java HotSpot(TM) 64-bit Server VM (build 20.2-b06, mixed mode) to run all experiments. The state-of-the-art parallel generational garbage collector was used for memory reclamation. This GC combines parallel Scavenge (i.e., copying) for the young generation and parallel Mark-Sweep-Compact for the old generation to quickly reclaim unreachable objects. GraphChi uses a parallel sliding windows algorithm that partitions data into shards. Since the number of shards has only little impact on performance (as reported in Figure 8(c) in [108] and also confirmed in our experiments), we fixed the number of shards to 20 in our experiments.

Performance GraphChi determines the amount of data to load and process (i.e., memory budget) in each iteration dynamically based on the maximum heap size. This is a very effective approach to reduce memory pressure and has been shown to be much more efficient than loading a fixed amount data per iteration. We ran P and P' with the same maximal heap size so that the same amount of data is loaded in each iteration (i.e., guaranteeing the same I/O time in both executions). Note that P' actually does not need a large heap because of the use of native memory. We tried various heap sizes and found that the smallest heap size for running P' was 2.5GB while P could not execute when the heap was smaller than 4GB.

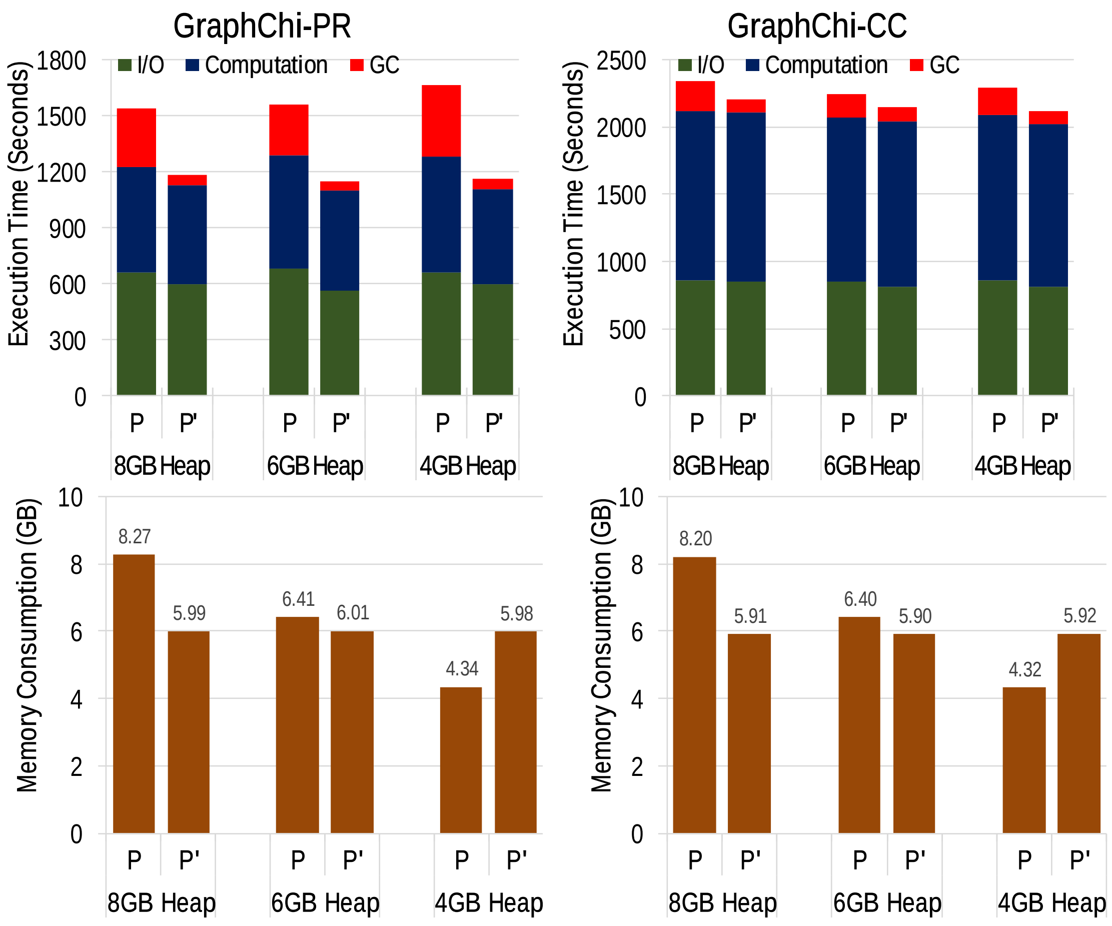


Figure 3.7: Performance comparisons between the original (P) and Facade-transformed programs (P') on GraphChi.

Figure 3.7 illustrates the comparisons between the original run and the Facade run in terms of running time broken down into various components (top row) and peak memory consumption (bottom row). Peak memory is computed by calculating the maximum from a set of samples of JVM memory consumptions collected periodically from `pmap`. We also exclude graph preprocessing time in the figure. Table 3.2 shows a more detailed performance comparisons. Note that our performance numbers may look different from those reported in [108], because their experiments used SSD and a C++ version of GraphChi.

On GraphChi, P' outperforms P for all configurations. The performance improvements Facade has achieved for PR and CC over the Twitter-2010 graph on average are, respec-

App	T(s)	T'(s)	%ET	%UT	%LT	%GC	%PM
PR-8G	1540.8	1180.7	23.4%	23.7%	25.7%	84.2%	27.6%
PR-6G	1561.2	1146.2	26.6%	25.2%	30.5%	81.7%	6.3%
PR-4G	1663.7	1159.2	30.3%	34.5%	28.5%	86.7%	-37.7%
Mean			*26.8%*	*27.8%*	*28.2%*	*84.2%*	
CC-8G	2338.1	2207.8	5.6%	6.4%	8.5%	77.0%	27.9%
CC-6G	2245.8	2143.4	4.6%	5.4%	10.0%	72.5%	7.8%
CC-4G	2288.5	2120.9	7.3%	9.4%	11.7%	74.4%	-36.8%
Mean			*5.8%*	*7.0%*	*10.1%*	*74.6%*	

Table 3.2: Performance summary of GraphChi on Twitter-2010 graph: reported are execution time of original run (T) and Facade run (T'), the reduction of total execution times (%ET), engine update times (%UT), data load times (%LT), garbage collection times (%GC), and peak memory consumption (%PM) under three different memory budgets (e.g., 8GB, 6GB, and 4GB)

tively, 26.8% and 5.8%; larger gains were seen when we experimented with smaller graphs (discussed shortly). The generated program P' not only has much less GC time (i.e., an average 5.1× reduction); data load and engine update time has also been reduced primarily due to inlining and direct memory accesses.

For PR, the number of objects for its data classes has been reduced from $14,257,280,923$ [1] to $1,363$, of which $1,000$ is the number of memory page objects and 363 is the number of total facade objects. Other than the main thread, GraphChi uses two thread pools, each containing 16 threads, and each thread has a pool of 11 facades. Hence, the total number of data objects equals $1000 + 11 * (16 * 2 + 1) = 363$, leading to dramatically decreased GC effort. The cost of page creation and recycling is negligible: the time it took to create and recycle pages was less than 5 seconds during the execution of PR' with 5 major iterations and 159 sub-iterations.

For P, its memory consumption is bounded by the maximum heap size, while the memory usage for P' is quite stable across different memory budget configurations. This is because our heap contains only objects in control path, whose numbers are very small; the off-heap data

[1]Obtained from our profiling that counts the number of runtime objects of data classes such as `ChiVertex, ChiPointer, and VertexDegree`.

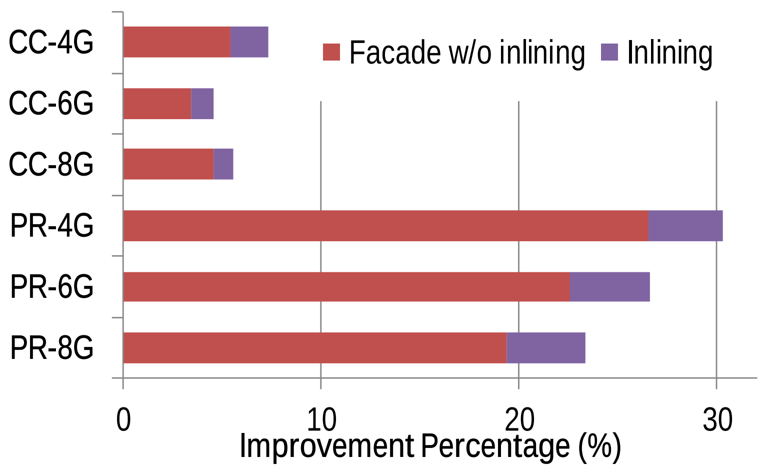


Figure 3.8: Performance improvements with and without data record inlining.

storage is not subject to the GC and is only determined by the amount of data processed. For both P and P', their running time does not vary much as the memory budget changes. This is primarily due to the adaptive data loading algorithm used by GraphChi. For systems that do not have this design, significant time increase and the GC efforts can often be seen when the heap becomes smaller, and thus, further performance improvement can be expected from Facade's optimization. Note that under a 4GB heap, P consumes less memory than P'. This is because the GC reclaims objects immediately after they become unreachable while Facade allows dead data records to accumulate until the end of a (sub-)iteration (i.e., trades off space for time).

To have a better understanding of how much inlining contributes to the performance improvements, we have compared the performance of the transformed GraphChi with and without inlining enabled. The results are shown in Figure 3.8. Inlining contributes to 4% and 1.5% of the improvements for PR and CC, respectively. The majority of the improvement comes from the reduction in GC efforts.

Scalability We measured scalability by computing throughput, which is defined as the number of edges processed per second. From the Twitter-2010 graph, we generated four smaller graphs with different sizes. We fed these graphs to PR and CC to obtain the scalability trends, which are shown in Figure 3.9. An 8GB heap was used to run P and P'. While 8GB

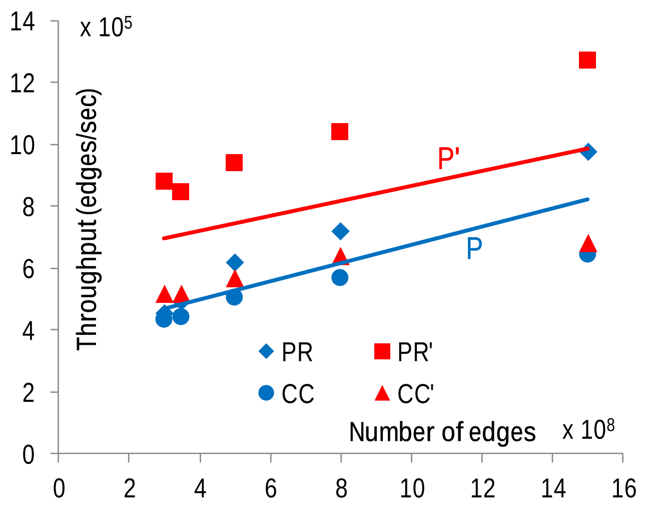


Figure 3.9: Computational throughput of GraphChi on various graphs; each trend-line is a least-squares fit to the average throughput of a program.

Input size			***Page Rank (PR)***				***Connected Component (CC)***			
$\|S\|$	$\|V\|$	$\|E\|$	T(s)	T'(s)	%ET	%TR	T(s)	T'(s)	%ET	%TR
5.1GB	33.9M	293.7M	661.3	**341.1**	48.4%	93.9%	696.3	**582.0**	16.4%	19.6%
6.4GB	35.6M	367.1M	711.4	**414.6**	41.7%	71.6%	796.8	**680.1**	14.6%	17.2%
8.5GB	37.2M	489.5M	808.5	**533.2**	34.1%	51.6%	991.6	**880.1**	11.3%	12.7%
12.8GB	38.7M	734.2M	1113.0	**770.9**	30.7%	44.4%	1411.0	**1250.8**	11.4%	12.8%
26.2GB	41.0M	1.5B	1540.8	**1180.7**	23.4%	30.5%	2338.1	**2207.8**	5.6%	5.9%

Table 3.3: GraphChi performance comparisons on different datasets: reported are the total execution times of the original program (T), and its Facade-version (T'); %ET reports Facade's reduction in execution time and %TR reports the throughput improvement.

appears to be a large heap for these relatively smaller graphs, our goal is to demonstrate, even with large memory resources and thus less GC effort, Facade can still improve the application performance substantially. For both versions, they scale very well with the increase of the data size. The generated program P' has higher throughput than P for all the graphs. In fact, for some of the smaller graphs, the performance difference, shown in Table 3.3, between P and P' is even larger than what is reported in Table 3.2. For example, on a graph with 300M edges, PR' and CC' are 48.4% and 16.4% faster than PR and CC, respectively. Since the cost of single-PC-based graph processing is dominated by disk accesses, the fraction of the I/O cost is smaller when processing smaller graphs and thus the effectiveness of the Facade optimizations becomes more obvious. This explains the reason why the performance difference is larger on smaller graphs.

3.5.2 Hyracks

Transformation Hyracks [51, 99] is a data parallel platform that runs data-intensive jobs on a cluster of stand-alone workers. It has been optimized manually to allow only byte buffers to store data and has been shown to have better scalability than object-based frameworks such as Hadoop. However, user functions can still (and mostly likely will) use object-based data structures for data manipulation.

After Facade transformed a significant portion of the high-level data manipulation functions in Hyracks, we evaluated performance and scalability with two commonly-used applications, Word Count (WC) and External Sort (ES). It took us 10 person-hours to find and annotate these user-defined operators; Facade transformed 8 classes in 15 seconds, resulting in a speed of 990 instructions per second.

Other than the 8 classes automatically transformed, the application code uses 15 collection classes from the Java library which we already modeled manually. Among these 8 classes, 2 are user-defined functions (one for WC and the other for ES), and the remaining 6 classes are Hyracks internal classes specifying properties and operator states. Three interaction points were detected: one for passing data from the control path into the data path and the other two passing processed data back into the control path (both via `ByteBuffer`). Data conversion functions were synthesized and calls to them were added at these points. Iterations are easy to identify: calls to `iteration_start()` and `iteration_end()` are placed at the beginning and the end of each Hyracks operator (i.e., one computation cycle), respectively.

Note that these 6 Hyracks internal classes do not include those that perform built-in data manipulation functions such as `join` and `filter`, because our search started from the user-defined application code and we were not familiar enough with the Hyracks framework to identify all built-in data processing functions. In general, whether a class is transformed or not relies on whether it is in the user-specified list, and Facade can transform different parts of the data path individually.

Test setup We ran Hyracks on a 10-slave-node (c3.2x large) Amazon EC2 cluster. Each machine has 2 quad-core Intel Xeon E5-2680 v2 processors (2.80GHz) and 15G RAM, running Linux 3.10.35, with enhanced networking performance. The same JVM and GC were used in this experiment. We converted a subset of Yahoo!'s publicly available AltaVista Web Page Hyperlink Connectivity Graph dataset [4] into a set of plain text files as input data. The dataset was partitioned among the slaves in a round-robin manner. The two applications were executed as follows: we created a total of 80 concurrent workers across the cluster, each of which reads a local partition of the data. Both WC and ES have a MapReduce-style computation model: each worker computes a local result from its own data partition and writes the result into the Hadoop Distributed File System (HDFS) running on the cluster; after hash-based shuffling, a reduce phase is then started to compute the final results.

Unlike GraphChi that adaptively loads data into memory, Hyracks loads all data upfront before the update starts. We ran both P and P' with an 8GB heap. When the heap is exhausted in P, the JVM terminates immediately with out-of-memory errors. Naïvely comparing scalability would create unfairness for P, because P' uses a lot of native memory. To enable a fair comparison, we disallowed the total memory consumption of P' (including both heap and native space) to go beyond 8GB. In other words, an execution of P' that consumes more than 8GB memory is considered as an "out-of-memory" failure.

Input size			***External Sort (ES)***				***Word Count (WC)***			
$\|S\|$	$\|V\|$	$\|E\|$	T(s)	T'(s)	%GC	%PM	T(s)	T'(s)	%GC	%PM
3GB	25.0M	313.8M	95.5	**89.3**	88.3%	-29.3%	**48.9**	57.4	98.2%	31.9%
5GB	33.2M	562.4M	178.2	**167.1**	96.9%	-3.3%	**72.5**	180.8	96.5%	31.5%
10GB	75.6M	1.1B	326.3	**302.5**	90.8%	4.4%	OME(683.1)	**1887.1**	*N/A*	*N/A*
14GB	143.1M	1.5B	459.0	**426.0**	93.4%	5.4%	OME(943.2)	**2693.0**	*N/A*	*N/A*
19GB	310.8M	1.8B	806.4	**607.5**	98.9%	8.5%	OME(772.4)	**3160.2**	*N/A*	*N/A*

Table 3.4: Hyracks performance comparisons on different datasets: reported are the total execution times of the original program (T), and its Facade-version (T'); OME(n) means the program runs out of memory in n seconds. %GC and %PM report Facade's reduction in GC time and peak memory consumption, respectively.

Performance and Scalability Performance comparison is shown in Figure 3.10. In the figure, if a program fails to process an input dataset (i.e., crash due to out-of-memory error),

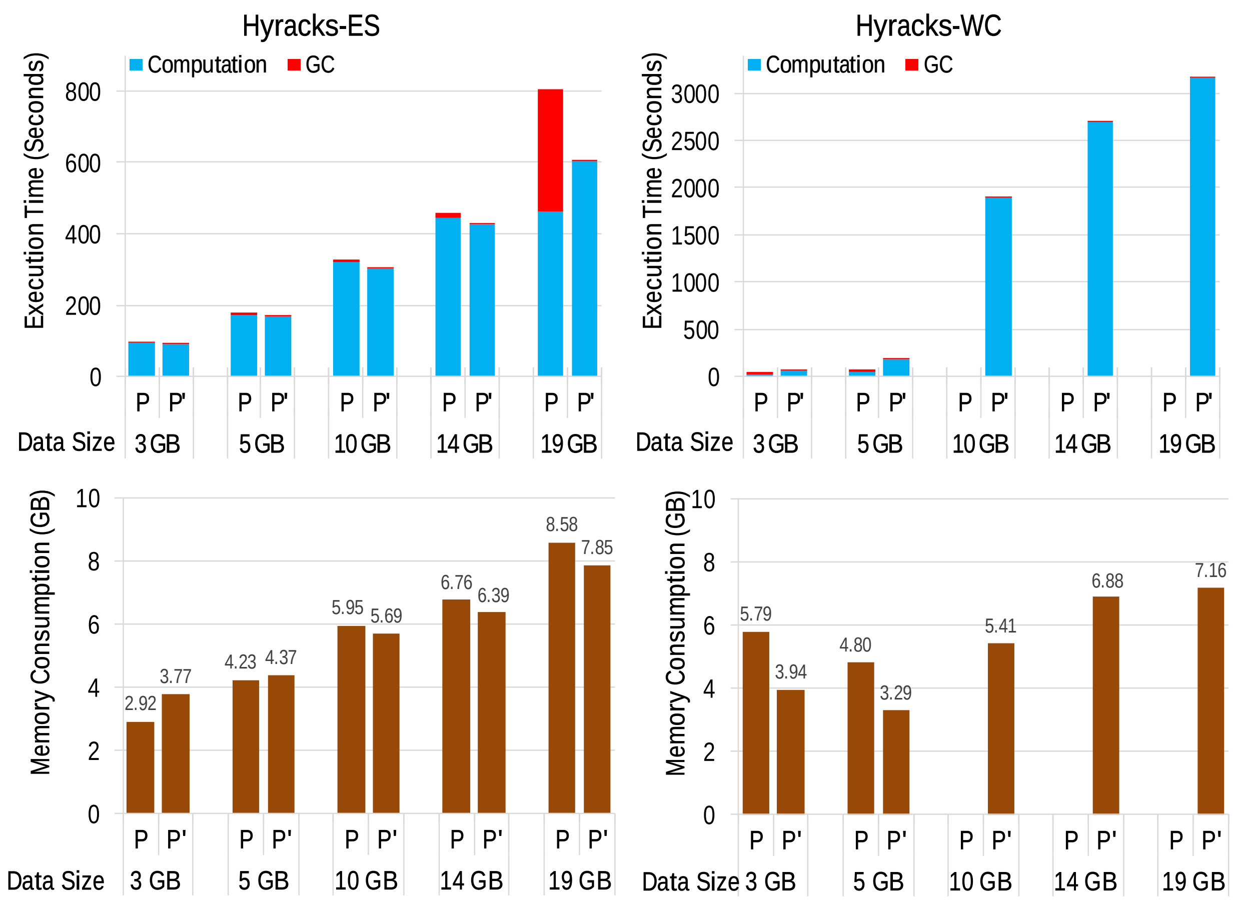


Figure 3.10: Performance comparisons between the original (P) and Facade-transformed (P') programs on Hyracks.

the bar is discarded. Table 3.4 shows a detailed running time comparison between P and P' on datasets of different sizes (which are all generated from the Yahoo! web graph data). P' outperforms P for all the inputs except the two smallest (3GB and 5GB) ones for WC. For these datasets, each machine processes a very small data partition (i.e., 300MB and 500MB). The GC effort for both P and P' is very small, and hence, the extra effort of pool accesses and page-based memory management performed in P' slows down the execution. However, as the size of the dataset increases, this effort can be easily offset from the large savings of GC costs.

We can also observe that P' scales to much larger datasets than P. For example, WC fails in 683.1 seconds when processing 10GB, while WC' successfully finishes in 3160.2 seconds for the 19GB dataset. Although both ES and ES' can scale to 19GB, ES' is about **24.7%**

faster than ES. In addition, P' has achieved an overall 25× reduction in the GC time, with a maximum 88× (from 346.2 seconds to 3.9 seconds).

The bottom row of Figure 3.10 shows the memory usage comparisons for ES and WC. It is clear to see that P' has smaller memory footprint than P in almost all the cases.

3.5.3 GPS

Transformation GPS [149] is a distributed graph processing system developed for scalable processing of large graphs. We profiled the execution and identified a total number of 4 (vertex- and graph-related) data classes whose instances grow proportionally with the data size. Starting from these classes, Facade further detected 44 data classes and 13 boundary classes. After an approximate 30-person-hour effort of understanding these classes, Facade transformed a total number of 61 classes (including 10691 Jimple instructions) in 9.7 seconds, yielding a 1102 instructions per second compilation speed.

Test setup We used three applications — Page Rank (PR), K-Means (KM), and Random Walk (RW) — to evaluate performance. The same (Amazon EC2) cluster environment was used to run the experiments. We created a total of 20 workers, each with a 4GB heap. The set of input graphs we used includes the Twitter-2010 graph [107], the Live Journal graph [42], and 5 synthetic supergraphs of Live Journal (e.g., the largest supergraph has 120M vertices and 1.7B edges).

GPS uses the distributed message-passing model of Pregel [121]. The behavior of each vertex is encapsulated in a function `compute`, which is executed exactly once in each super-step. GPS is overall less scalable than GraphChi and Hyracks due to its object-array-based representation of an input graph. GPS expects vertices to be labeled contiguously starting from 0, and therefore, vertices can be efficiently stored in an array. While the goal of this design is to improve performance by avoiding using Java data structures (e.g., `ArrayList`), it leads to memory inefficiencies in many cases. This is because each node processes an arbitrary set of vertices and thus the array becomes a sparse structure with a lot space wasted. To solve the problem, we replaced the array with an `ArrayList` before performing transformation — since Facade will allocate the `ArrayList` in the native memory, this replacement saves a lot of space without incurring extra GC overhead.

	Input size			***Page Rank (PR)***		***K-Means (KM)***		***Random Walk (RW)***	
	$\|S\|$	$\|V\|$	$\|E\|$	%ET	%GC	%ET	%GC	%ET	%GC
LJ	0.5GB	4.8M	68.0M	-0.3%	15.5%	0.1%	20.5%	0.2%	39.8%
LJ-A	2.5GB	24.0M	340.0M	4.7%	12.8%	2.9%	6.0%	9.4%	34.0%
LJ-B	5.0GB	48.0M	680.0M	1.6%	11.6%	6.4%	10.0%	7.2%	27.8%
LJ-C	7.5GB	72.0M	1.0B	2.4%	12.2%	8.2%	4.8%	6.5%	30.1%
LJ-D	10.0GB	96.0M	1.5B	5.5%	18.3%	7.7%	4.4%	7.6%	32.5%
LJ-E	12.5GB	120.0M	1.7B	17.3%	30.8%	13.5%	23.1%	10.9%	32.1%
TW	26.2GB	41.0M	1.5B	2.3%	14.4%	3.5%	7.19%	4.9%	19.9%

Table 3.5: GPS performance comparisons on different datasets: reported are the Facade's reduction in execution time (%ET) and GC time (%GC); LJ-A. . .LJ-E are different synthetic supergraphs of the Live Journal (LJ) graph; TW is the Twitter-2010 graph.

Performance Table 3.5 shows detailed performance comparisons on all datasets of different sizes. Compared to the original implementation P, the generated version P' has achieved a 3–15.4% running time reduction, a 10–39.8% GC time reduction, as well as an up to 14.4% space reduction. P and P' have about the same running time on the smallest graph (with 4.8M vertices and 68M edges). However, for all the other graphs in the input set, clear performance improvements can be observed on P', which is especially significant on the largest graph (`LJ-E`). The reason why Facade can improve GPS' performance is that, GPS still allows the creation of objects to represent vertex values (e.g., `DoubleWritable`, `TwoIntWritable`, `LongWritable`, etc.). These (small) objects in turns are stored in a huge array. Facade automatically inlines all of them into the array, therefore improves performance of GPS.

Unlike GraphChi where the majority of the improvement comes from the reduction in GC efforts, in GPS, data record inlining contributes the most to the improvement. Without inlining, Facade-generated programs run slightly faster (1-2%) than the original. This is because the design of GPS is very similar in spirit to what Facade intends to achieve in many ways. First, it has extensive use of raw (i.e., primitive) arrays to store data such as adjacent list or message content. Second, GPS reduces the memory cost of allocating many

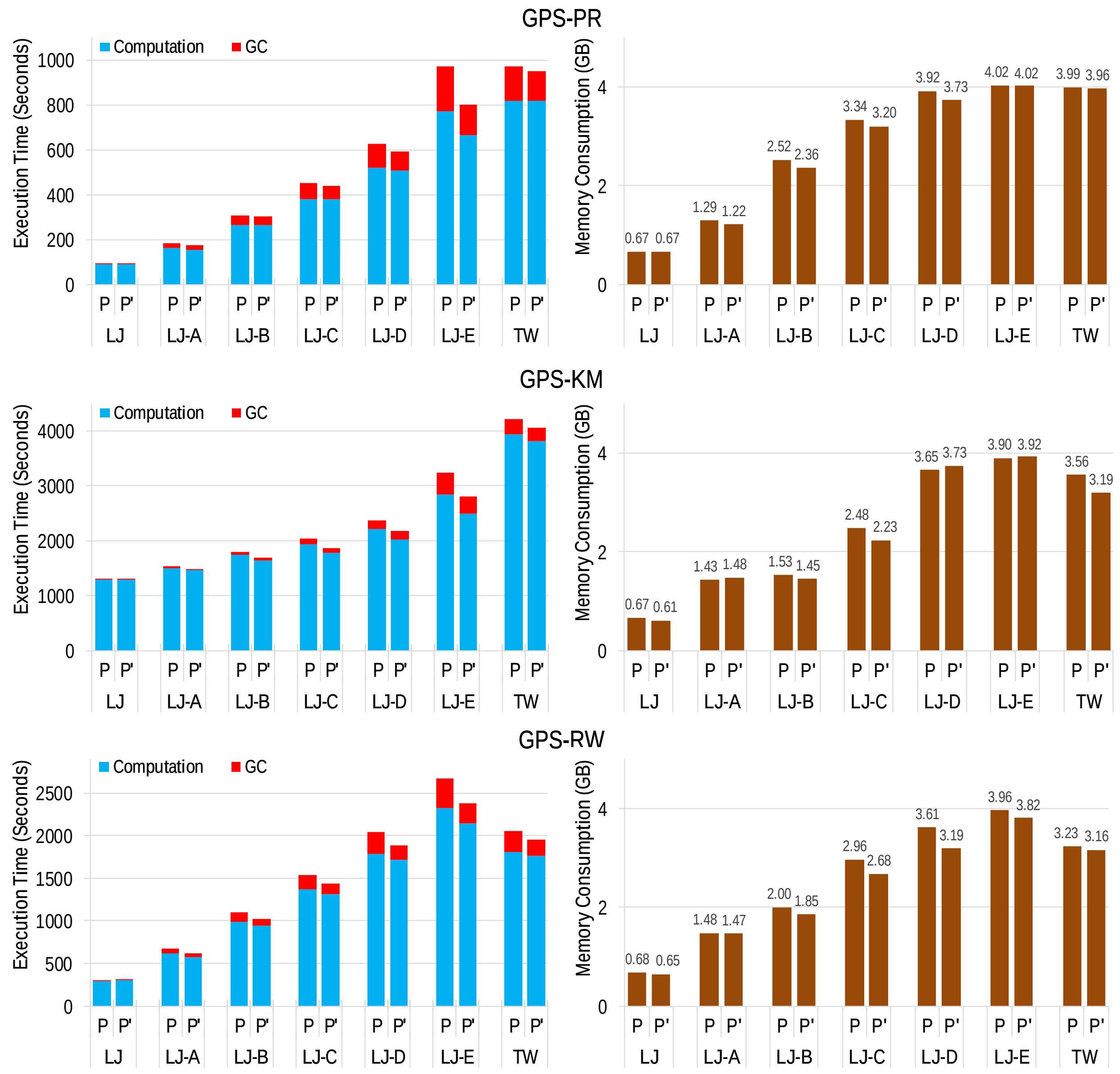


Figure 3.11: Performance comparisons between the original (P) and Facade-transformed (P') programs on GPS.

Java objects by using canonical objects. Instead of storing the value and the adjacency list of each vertex inside a separate `Vertex` object, and calling `compute` on each object as in Giraph, GPS workers use a single canonical `Vertex` object to perform program logic (i.e., invoke `compute`) with vertex values and adjacency list stored in seperate data structures. GPS also uses a single canoncial `Message` object for message passing. Incoming messages are stored as raw bytes in the message queues, and a message is deserialized into the canonical `Message` object only when the canonical `Vertex` object iterates over it.

3.6 Summary

Facade targets the performance problem caused by object-base representation mismatch in Big Data systems. Due to the excessive object creation in a managed Big Data system, the overhead of using objects to both represent and manipulate data becomes a performance bottleneck. Facade mitigates the mismatch using a compiler and runtime support. Facade achieves high efficiency by performing a semantics-preserving transformation of the data path of a Big Data program to statically bound the number of heap objects representing data items.

The design of Facade crosses the layers of compiler and runtime system, exploiting native memory to represent data objects instead of using static analysis to eliminate them. It is intentional with the goal to improve practicality. On one hand, the design enables our compiler to perform simple local (method-based) code transformation, making it possible for Facade to scale to a large codebase. On the other hand, the combination of code transformation and the leveraging of the native memory support from a commercial JVM eliminates the need to modify the JVM, enabling Facade to scale to a very large heap. The experimental results demonstrate that the generated programs are more (time and memory) efficient and scalable than their object-based counterparts.

Admittedly, to use Facade, a considerable amount of user effort is needed to understand the program and provide a correct list of data classes as well as iteration markers (i.e., `iteration_start()` and `iteration_end()` calls). While the markers are often well-defined in a Big Data platform, the user should understand such semantic information by profiling the application (as we did in our experiments) to distinguish data classes (e.g., classes that have a large number of runtime objects) from control classes. The Facade compiler will assist users in identifying the list of data classes by throwing compilation errors when assumptions (Section 3.4.1) are violated. Users are then to refactor the violating classes to make the application Facade-transformable. However, although we had never studied any of these frameworks before this work, we found that the majority of the manual effort was spent on profiling each system to understand the data path and setting up the execution environments (e.g., on average, it took us a day worth of work for each framework). Once we identified an initial set of data classes, the effort to specify iterations and annotate boundary classes was almost negligible. It would have taken much less time had the developers of these frameworks used Facade themselves.

One interesting direction of future work is to adapt Facade to Scala, which powers the entire Apache Spark [181, 182] framework. All techniques can be conceptually applied to optimize Scala programs seamlessly since they also rely on the JVM. One challenge is that there are many compiler-generated objects in Scala that do not exist in Java; leaving all of them in the heap would still cause large GC overhead. We need to modify the Scala compiler and develop techniques to identify those objects for native allocation.

Since Facade can also reduce the memory footprint of a program, the transformed program may likely need fewer machines to process a dataset compared to the original program. Hence, it is interesting to perform additional experiments to understand how much Facade can reduce resource usage while achieving the same analytical task (e.g., answering a query).

CHAPTER 4

Mitigating Garbage Collection Mismatch

Popular data processing frameworks such as Hadoop [35], Spark [182], Naiad [130], or Hyracks [51] are all developed in managed languages, such as Java, C#, or Scala, primarily due to the automatic memory management feature provided by the garbage collector (GC). However, the fact is garbage collection is prohibitively expensive. GC can account for close to 50% of the execution time of these systems, severely damaging system performance [55, 77, 118, 119, 133, 135].

A critical reason for slow GC execution is that object characteristics in Big Data systems do *not* match the heuristics employed by state-of-the-art GC algorithms. In this chapter, we present our solution for the object behavior mismatch by designing a more suitable GC, namely Yak, for Big Data systems. Yak was built to intelligently adapt to object characteristics of Big Data systems, and thus, can efficiently handle large volume of objects in these systems.

We first discuss the mismatch in more details (Section 4.1), followed by an extensive study of object characteristic in real systems (Section 4.2). Then we introduce the design and implementation of the system in full in Section 4.3 and Section 4.4. Experimental results on real systems are presented in Section 4.5 before we make some concluding remarks in Section 4.6

4.1 The Garbage Collection Mismatch

As discussed in Section 2.2, state-of-the-art GCs are all built upon the conventional generational hypothesis which simply states most objects die young — most recently allocated objects are also most likely to become unreachable. The hypothesis holds true for many large applications, which make heavy use of short-lived temporary data structures, and thus, generational GCs are efficient: most GC runs scan a small portion of the heap while being able to reclaim significant amount of memory. However, Big Data applications behave differently. The behavior can be summarized as *two paths, two hypotheses*.

Evidence [55, 86, 135] shows that a typical data processing framework often has a clear logical distinction between a *control path* and a *data path*, discussed earlier in Section 2.3. As a summary, the control path has a complex logic to drive the execution flow. (e.g., performs cluster management and scheduling, establishes communication between master and worker nodes); the data path primarily consists of data manipulation functions (e.g., data partitioning, built-in operations such as Join or Aggregate, and user-defined data functions such as Map or Reduce) that can be connected to form a data processing pipeline.

These two paths follow different heap usage patterns. While the generational hypothesis holds for the control path of a data-intensive application, it does *not* holds for the data path, where *most* objects are created. Specifically, on one hand, most control objects, since created only to drive the program logic, not only small in number, but also have short life spans On the other hand, data objects, exhibits strong *epochal behavior*. Their lifetime aligns well with each epoch — a piece of data manipulation code which is repeatedly executed. These objects are often created en masse at the beginning of an epoch, stored in large structures (to represent input data), and stay alive throughout the epoch (for processing), which is often *not* a short period of time.

This mismatch leads to the fundamental challenge encountered by state-of-the-art GCs in data-intensive applications. Since newly-created (data) objects often do not have short life spans, and thus, unexpectedly survive several GC runs, much of GC effort is spent for tracing a huge number of objects, while they are not immediately reclaimable.

4.2 Object Characteristics in Real Systems

We have also conducted several experiments to verify the non-generational property of data items in Big Data applications. Figure 4.1 depicts the memory footprint and its correlation with epochs when Page Rank (PR) and Connected Components (CC) were executed on GraphChi [108] to process the Twitter graph [107] on a machine with 2 Intel(R) Xeon(R) CPU E5-2630 v2 processors running CentOS 6.6. The default Parallel Scavenge GC was used.

GraphChi is a high-performance disk-based graph processing system. In this system, graph data is partitioned into different shards, defined by a vertex interval. Each shard is first loaded into memory in each iteration, followed by the creation of many vertex objects to represent the data. These objects are long-lived and frequently visited to perform vertex updates. They cannot be reclaimed until the next vertex interval is processed. There can be dozens to hundreds of GC runs in each interval. The processing of an interval forms an epoch.

In this experiment, GraphChi finished respectively, in 2337 and 3227 seconds, of which 1289 (55.2%) and 1324 (41.1%) seconds were spent on GC. Each epoch lasts about 20 seconds (PR) and 40 seconds (CC), denoted by dotted lines in Figure 4.1. We can observe clear correlation between the end point of each epoch and each significant memory drop (Figure 4.1 (a)) as well as each large memory reclamation (Figure 4.1 (b)). During each epoch, many GC runs occur and each reclaims little memory (Figure 4.1 (b)).

For comparison, we also measured the memory usage of programs in the DaCapo benchmark suite [46], widely-used for evaluating JVM techniques. Figure 4.2 shows the memory footprint of Eclipse under large workloads provided by DaCapo. Eclipse is a popular development IDE and compiler front end. It is an example of applications that have complex logic but process

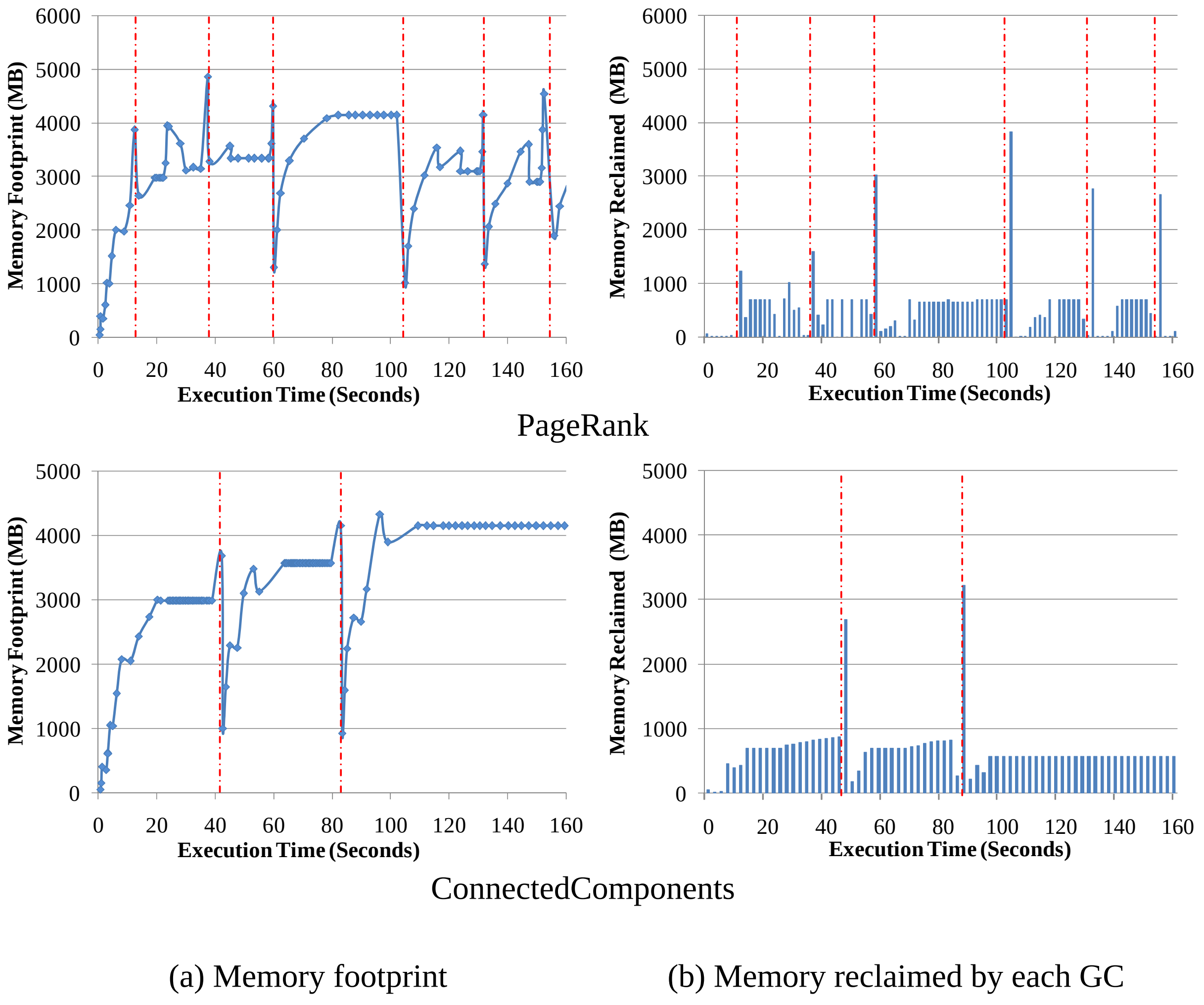


Figure 4.1: Memory footprint for the first 160 seconds of the executions when GraphChi runs PR (top) and CC (bottom). Each dot in (a) represents the memory consumption measured right after a GC; each bar in (b) shows how much memory is reclaimed by a GC; dotted vertical lines show the iteration boundaries.

small amounts of data. GC performs well for Eclipse, taking only 2.4% of total execution time and reclaiming significant memory in each GC run. We do not observe epochal patterns in Figure 4.2. Furthermore, while other DaCapo benchmarks may exhibit some epochal behavior, epochs in these programs are often not clearly defined and finding them is not easy for application developers who are not familiar with the system codebase.

Strawman Can we solve the problem by forcing GC runs to happen *only* at the end of epochs? This simple approach, unfortunately would not work due to the multi-threaded

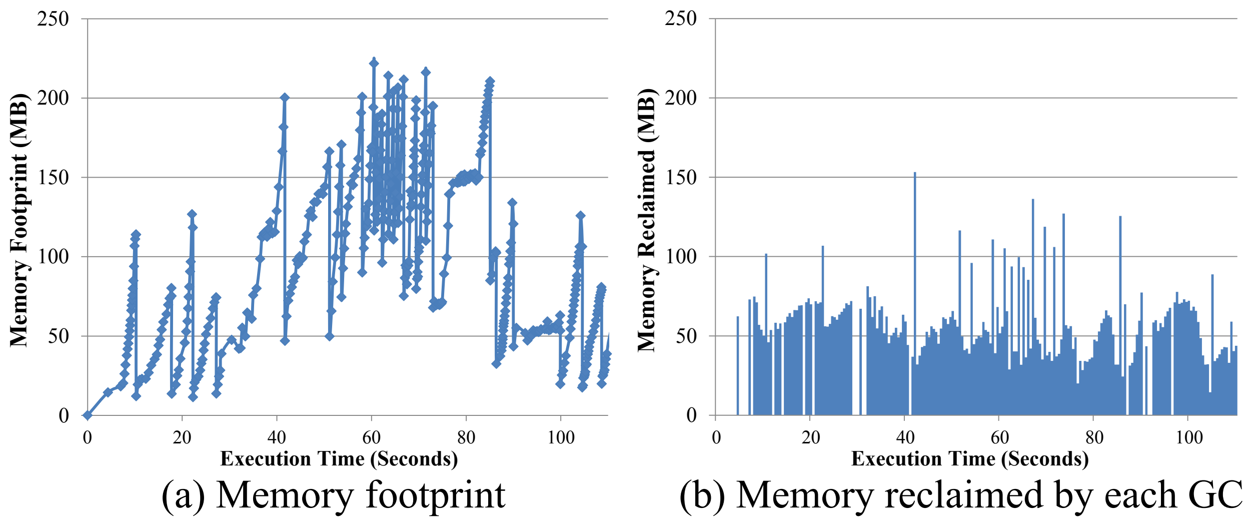
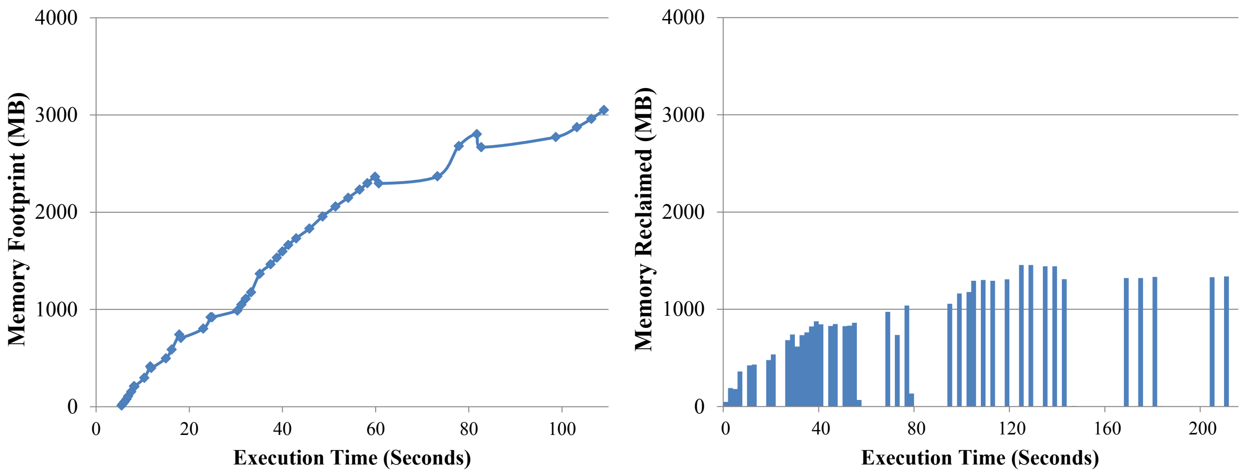


Figure 4.2: Eclipse execution (GC takes 2.4% of time).

nature of real systems. In systems like GraphChi, each epoch spawns many threads that collectively consume a huge amount of memory. Waiting until the end of an epoch to conduct GC could easily cause out-of-memory crashes. In systems like Hyracks [51], a distributed dataflow engine, different threads have various processing speeds and reach epoch ends at different times. Invoking the GC when one thread finishes an epoch would still make the GC traverse many live objects created by other threads, leading to wasted effort. This problem is illustrated in Figure 4.3, which shows memory footprint of one slave node when Hyracks performs word counting over a 14GB text dataset on an 11-node cluster. Each node was configured to run multiple Map and Reduce workers and have a 12GB heap. There are no epochal patterns in the figure, exactly because many worker threads execute in parallel and reach the end of an epoch at different times.

The two-paths-two-hypotheses phenomenon Big Data systems naturally motivates the need of developing a hybrid memory manager which can better manage objects in two different paths. The epochal behavior data objects is more suitable for region-based memory management — objects created in an epoch are allocated in a memory region and efficiently deallocated as a whole when the epoch ends.

Next, we will introduce our solution mitigating the mismatch by having two different management techniques coexist in harmony in one system, Yak.



(a) Memory footprint (b) Memory reclaimed by each GC

Figure 4.3: Hyracks WordCount (WC) execution (GC takes 33.6% of time).

4.3 Design Overview

The overall idea of Yak is to split the managed heap into a small control space (CS) and a much larger data space (DS), based on the key observation of a clear distinction between a control path and a data path, and use different mechanisms to manage these spaces. Objects created in the control path are allocated in the CS and subject to regular tracing GC. The lifetimes of objects in the data path often align with epochs creating them. They are thus allocated in the DS and subject to region-based memory management.

We present below two major design issues as well as our solutions.

Issue 1: When to Create & Deallocate DS Regions? A region is created (deallocated) in the DS whenever an epoch starts (ends). This region holds all objects created inside the epoch. An epoch is the execution of a block of data transformation code. Note that the notion of an epoch is well-defined in Big Data systems. For example, in Hyracks [51], the body of a dataflow operator is enclosed by calls to `open` and `close`. Similarly, a user-defined (Map/Reduce) task in Hadoop [35] is enclosed by calls to `setup` and `cleanup`.

To enable a unified treatment across different Big Data systems, Yak expects a pair of user annotations, `epoch_start()` and `epoch_end()`. These annotations are translated into two native function calls at run time to inform the JVM of the start/end of an epoch. Placing these annotations requires negligible manual effort. Even a novice, without much knowledge about the system, can easily find and annotate epochs in a few minutes. Yak guarantees execution correctness regardless of where epoch annotations are placed. Of course, the locations of epoch boundaries do affect performance: if objects in a designated epoch have very different life spans, many of them need to be copied when the epoch ends, creating overhead.

Often, the framework declared callbacks (on which developers cannot make mistakes) are already good enough for Yak to achieve performance gains. Once the managed heap is created (during JVM's bootstrapping), we reserve a range of virtual addresses as the DS. The actual memory will not be committed until needed. Each epoch is assigned a region, which consists of a set of fixed-length memory pages.

In practice, we need to consider a few more issues about the epoch concept. One is the nested relationships exhibited by epochs in real systems. A typical example is GraphChi [108], where a computational iteration naturally represents an epoch. Each iteration iteratively loads and processes all shards, and hence, the loading and processing of each memory shard (called `Interval` in GraphChi) forms a sub-epoch inside the computational iteration. Since a shard is often too large to be loaded entirely into memory, GraphChi further breaks it into several `subIntervals`, each of which forms a sub-sub-epoch.

Yak supports nested regions for performance benefits — unreachable objects inside an inner epoch can be reclaimed long before an outer epoch ends, preventing the memory footprint from aggressively growing. Specifically, if an `epoch_start()` is encountered in the middle of an already-running epoch, a sub-epoch starts; subsequently a new region is created, and considered a child of the existing region. All subsequent object allocations take place in the child region until an `epoch_end()` is seen. We do not place any restrictions on regions; objects in arbitrary regions are allowed to mutually reference one another.

The other issue is how to create regions when multiple threads execute the same piece of data-processing code concurrently. We could allow those threads to share one region. However, this would introduce complicated thread-synchronization problems; and might also delay memory recycling when multiple threads exit the epoch at different times, causing unnecessary memory pressure. Hence, we propose thread-local regions — Yak creates one region for each dynamic instance of an epoch. When two threads execute the same piece of epoch code, they each get their own regions, and thus, can allocate objects and deallocate the region without having to worry about synchronization.

Overall, at any moment of execution, multiple epochs and hence regions could exist. They can be partially ordered based on their nesting relationships, forming a semilattice structure. As shown in Figure 4.4, each node on the semilattice is a region of form $\langle r_{ij}, t_k \rangle$, where r_{ij} denotes the j-th execution of epoch r_i and t_k denotes the thread executing the epoch. For example, region $\langle r_{21}, t_1 \rangle$ is a child of $\langle r_{11}, t_1 \rangle$, because epoch r_2 is nested in epoch r_1 in the program and they are executed by the same thread t_1. Two regions (e.g., $\langle r_{11}, t_1 \rangle$ and $\langle r_{12}, t_2 \rangle$) are *concurrent* if their epochs are executed by different threads.

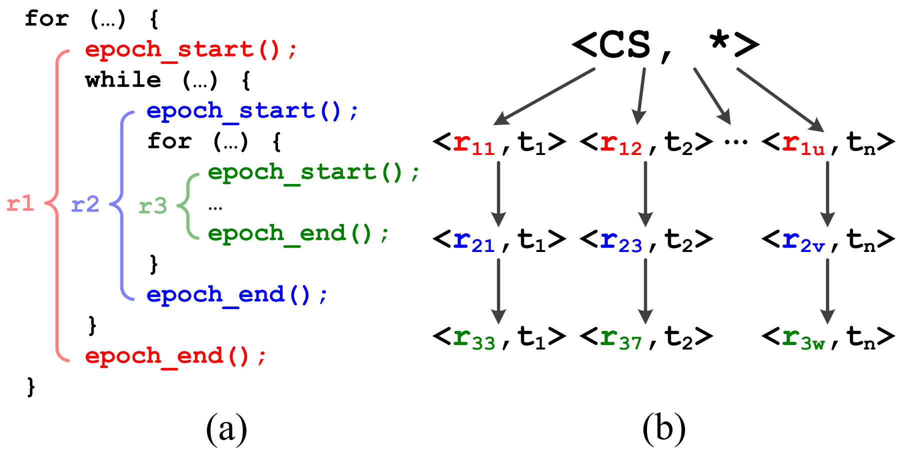


Figure 4.4: An example of regions: (a) a simple program and (b) its region semilattice at some point of execution.

Issue 2: How to Deallocate Regions Correctly and Efficiently? A small number of objects may outlive their epochs, and have to be identified and carefully handled during region deallocation. For example, GraphChi allocates many array objecys during sub-subepochs to hold vertex data, and these arrays are not released until the end of a sub-epoch when the updated data is written back to disk. We do not want to solve this problem by an iterative manual process of code refactoring and testing, which is labor-intensive as was done in Facade [135] or Broom [86]. Yak has to automatically accomplish two key tasks: (1) identifying escaping objects and (2) deciding the relocation destination for these objects.

For the first task, Yak uses an efficient algorithm to track cross-region/space references and records all incoming references at run time for each region. Right before a region is deallocated, Yak uses these references as the root set to compute a transitive closure of objects that can escape the region (details in §4.4.2).

For the second task, for each escaping object O, Yak tries to relocate O to a live region that will not be deallocated before the last (valid) reference to O. To achieve this goal, Yak identifies the source regions for each incoming cross-region/space reference to O, and uses them to find their closest common ancestor on the region semilattice. More precisely, we define a `Join` operation on the semilattice that takes as input a set of nodes and returns their least upperbound. This upperbound gives us the region or space into which we should relocate O. For example, in Figure 4.4, joining $\langle r_{21}, t_1 \rangle$ and $\langle r_{11}, t_1 \rangle$ returns $\langle r_{11}, t_1 \rangle$, while joining any two concurrent regions returns the CS. Intuitively, if O has references from its parent and grand-parent regions, O should be moved up to its grand-parent. If O has two references coming from regions created by different threads, it has to be moved to the CS for efficiency.

Upon deallocation, computing a transitive closure of escaping objects while other threads are accessing them may result in an incomplete closure. In addition, moving objects concurrently with other running threads is dangerous and may give rise to data races. Yak employs a lightweight "stop-the-world" treatment to guarantee memory safety in deallocation. When a thread reaches an `epoch_end()`, Yak pauses all running threads, scans their stacks, and computes a closure that includes all potential live objects in the deallocating region. These objects are moved to their respective target regions before all mutator threads are resumed.

4.4 Implementation

We have implemented Yak in Oracle's production JVM OpenJDK 8 (build 25.0-b70). In addition to implementing our own region-based technique, we have modified the two Just-in-Time (JIT) compilers (C1 and Opto), the interpreter, the object/heap layout, and the Parallel Scavenge collector (to manage the CS). Below, we discuss how to split the heap and create regions (§4.4.1); how to track inter-region/space references, how to identify escaping objects, and how to determine where to move them (§4.4.2); how to deallocate regions correctly and efficiently (§4.4.3); and how to modify the Parallel Scavenge GC to collect the CS (§4.4.4).

4.4.1 Region & Object Allocation

Region Allocation When the JVM is launched, it asks the OS to reserve a block of virtual addresses based on the maximum heap size specified by the user (i.e., via -Xmx). Yak divides this address space into the CS and the DS, with the ratio between them specified by the user via JVM parameters. Yak initially asks the OS to commit a small amount of memory, which will grow if the initial space runs out. Once an `epoch_start()` call is encountered, Yak creates a region in the DS. A region contains a list of pages whose size can also be specified by a JVM parameter.

Heap Layout Figure 4.5 illustrates the heap layout maintained by Yak. The CS is the same as the old Java heap maintained by a generational GC, except for the newly added remember set. The DS is much bigger, containing multiple regions, with each region holding a list of pages.

The remember set is a bookkeeping data structure maintained by Yak for every region and the CS space. It is used to determine what objects escape a region r and where to relocate them. The remember set of CS helps identify live objects in the CS. The remember set of a

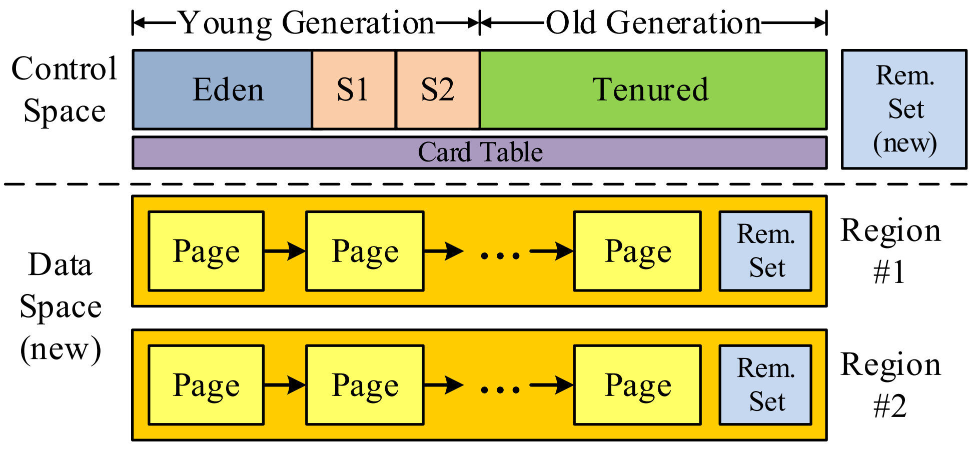


Figure 4.5: The heap layout in Yak.

region/space r is implemented as a hash table that maps an object O in r to all references to O that come from a different region/space.

Note that a remember set is one of the many possible data structures to record such references. For example, the generational GC uses a card table that groups objects into fixed-sized buckets and tracks which buckets contain objects with pointers that point to the young generation. Yak uses remember sets, because each region has only a few incoming references; using a card table instead would require us to scan all objects from the CS and other regions to find these references.

Allocating Objects in the DS When the execution is in an epoch, we redirect all allocation requests made to the Eden space (e.g., young generation) to our new `Region_Alloc` function. Yak filters out JVM meta-data objects, such as class loader and class objects, from getting allocated in the region. Using a quick bump pointer algorithm (which uses a pointer that points to the starting address of free space and bumps it up upon each allocation), the region's manager attempts to allocate the object on the last page of its page list. If this page does not have enough space, the manager creates a new page and appends it to the list. For a large object that cannot fit into one page, we request a special page that can fit the object. For performance, large objects are never moved.

Functions	**Description**
Addr(o)	Returns the address of an object
Offset(f)	Returns the offset of an object field
Space(r)	Returns the range of the addresses of a region
Region(o)	Reads object header and returns region ID
Target(ref)	Returns the pointee of a reference ref
Move(o, r)	Moves object o to region r
Join(r_1, r_2)	Joins two regions based on the semilattice
DeQueue(q)	Removes top object from a queue and returns it
EnQueue(o, q)	Adds an object o to a queue q
Threads()	Returns a set of all running threads
ScanStack(t, r)	Scans local stack of thread t and returns reachable objects of r from t
PauseOtherThreads()	Pauses all other threads
ResumePausedThreads()	Resumes all paused threads

Table 4.1: Auxiliary functions used in Algorithms 1 – 4

4.4.2 Tracking Inter-region References

Overview As discussed in Section 4.3, Yak needs to efficiently track all inter-region/space references. At a high level, Yak achieves this in three steps. First, Yak adds a 4-byte field *region* into the header space of each object to record the region information of the object. Upon an object allocation, its *region* field is updated to the corresponding region ID. A special ID is used for the CS.

Second, we modify the write barrier (i.e., a piece of code executed with each heap write instruction $a.f = b$) to detect and record heap-based inter-region/space references. Note that, in OpenJDK, a barrier is already required by a generational GC to track inter-generation references. We modify the existing write barrier as shown in Algorithm 1.

Finally, Yak detects and records local-stack-based inter-region references as well as remote-stack-based references when `epoch_end()` is triggered. These algorithms are shown in Lines 1 – 4 and Lines 5 – 10 in Algorithm 3. A list of auxiliary functions and their descriptions are shown in Table 4.1.

Algorithm 1: The write barrier $a.f = b$.

```
Input: Expression $a.f$, Variable $b$
1 if $\textsc{Addr}(O_a) \notin \textsc{Space}(CS)$ OR $\textsc{Addr}(O_b) \notin \textsc{Space}(CS)$ then
2     if $\textsc{Region}(O_a) \neq \textsc{Region}(O_b)$ then
3         Record the reference $\textsc{Addr}(O_a) + \textsc{Offset}(f) \xrightarrow{\textsc{Region}(O_a)} \textsc{Addr}(O_b)$ in the
          remember set $rs$ of $O_b$'s region
4 ... // Normal OpenJDK logic (for marking the card table)
```

Details We describe in detail how Yak can track all inter-region references, following the three places where the reference to an escaping object can reside in — the heap, the local stack, and a remote stack. The semantics of writes to static fields (i.e., globals) as well as array stores are similar to that of instance field accesses; we omit the details of their handling. Copies of large memory regions (e.g., `System.arraycopy`) are also tracked in Yak.

(1) In the heap. An object O_b can outlive its region r if its reference is written into an object O_a allocated in another (live) region r'. Algorithm 1 shows the write barrier to identify such escaping objects O_b. The algorithm checks whether the reference is an inter-region/space reference (Line 2). If it is, the pointee's region (i.e., $\textsc{Region}(O_b)$) needs to update its remember set (Line 3).

Each entry in the remember set is a reference which has a form $a \xrightarrow{r} b$ where a and b are the addresses of the pointer and pointee, respectively, and r represents the region the reference comes from. In most cases (such as those represented by Algorithm 1), r is the region in which a resides and it will be used to compute the target region to which b will be moved. However, if a is a stack variable, we need to create a placeholder reference with a special r, determined based on which stack a comes from. We will shortly discuss such cases in Algorithm 3.

To reduce overhead, we have a check that quickly filters out references that do not need to be remembered. As shown in Algorithm 1, if both O_a and O_b are in the same region, including the CS (Lines 1 – 2), we do not need to track that reference, and thus, the barrier proceeds to the normal OpenJDK logic.

(2) On the local stack. An object can escape by being referenced by a stack variable declared beyond the scope of the running epoch. Figure 4.6 (a) shows a simple example. The reference of the object allocated on Line 3 is assigned to the stack variable a. Because a is still alive after `epoch_end()`, it is unsafe to deallocate the object.

```
1 a = ...;
2 //epoch_start
3 b = new B();
4 if (/*condition */) {
5   a = b;
6 }
7 //epoch_end
8 c = a;
```

(a)

```
1  Thread t :
2  //epoch_start
3  a = A.f;
4  a.g = new O();
5  //epoch_end
6
7  Thread t' :
8  //epoch_start
9  p = A.f;
10 b = p.g;
11 p.g = c;
12 //epoch_end
```

(b)

Figure 4.6: (a) An object referenced by b escapes its epoch via the stack variable a; (b) An object O created by thread t and referenced by $a.g$ escapes to thread t' via the load statement $b = p.g$.

Yak identifies this type of escaping objects through an analysis at each `epoch_end()` mark. Specifically, Yak scans the local stack of the deallocating thread for the set of live variables at `epoch_end()` and checks if an object in r can be referenced by a live variable (Lines 1 – 4 in Algorithm 3). For each such escaping object O_{var}, Yak adds a placeholder incoming reference, whose source is from r's parent region (say p), into the remember set rs of r (Line 4). This will cause O_{var} to be relocated to p. If the variable is still live when p is about to be deallocated, this would be detected by the same algorithm and O_{var} would be further relocated to p's parent.

(3) On the remote stack. A reference to an object O created by thread t could end up in a stack variable in thread t'. For example, in Figure 4.6 (b), object O created on Line 4 escapes t through the store at the same line and is loaded to the stack of another thread

t' on Line 10. A naive way to track these references is to monitor every read (i.e., a read barrier), such as the load on Line 10 in Figure 4.6 (b).

Yak avoids the need for a read barrier, whose large overhead could affect practicality and performance. Before proceeding to discuss the solution, let us first examine the potential problems of missing a read barrier. The purpose of the read barrier is for us to understand whether a region object is loaded on a remote stack so that the object will not be mistakenly reclaimed when its containing region is deallocated. Without it, a remote thread which references an object O in region r, may cause two potential issues when r is deallocated (Figure 4.7).

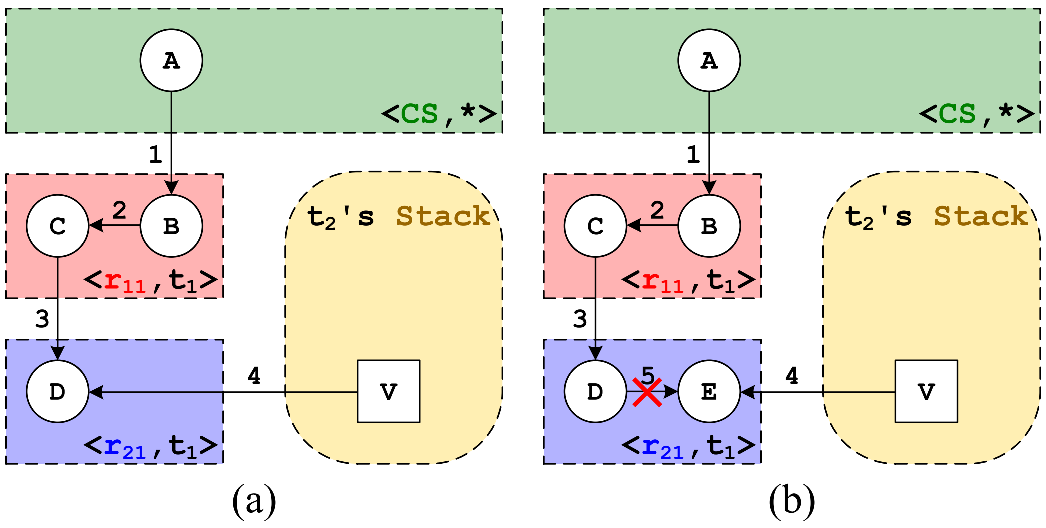


Figure 4.7: Examples showing potential problems with references on a remote stack: (a) moving object D is dangerous; and (b) object E, which is also live, is missed in the transitive closure.

Problem 1: Dangerous object moving. Figure 4.7 (a) illustrates this problem. Variable v on the stack of thread t_2 contains a reference to object D in region $\langle r_{21}, t_1 \rangle$ (by following the chain of references starting at object A in the CS). When this region is deallocated, D is in the escaping transitive closure; its target region, as determined by the semilattice, is its parent region $\langle r_{11}, t_1 \rangle$. Obviously, moving D at the deallocation of $\langle r_{21}, t_1 \rangle$ is dangerous, because we are not aware that v references it and thus cannot update v with D's new address after the move.

Problem 2: Dangerous object deallocation. Figure 4.7 (b) shows this problem. Object E is first referenced by D in the same region $\langle r_{21}, t_1 \rangle$. Hence, the remote thread t_2 can reach E by following the reference chain starting at A. Suppose t_2 loads E into a stack variable v and then deletes the reference from D to E. When region $\langle r_{21}, t_1 \rangle$ is deallocated, E cannot be included in the escaping transitive closure while it is being accessed by a remote stack. E thus becomes a "dangling" object that would be mistakenly treated as a dead object and reclaimed immediately.

Solution. Yak's solution to these problems is to pause all other threads and scan their stacks when thread t deallocates a region r. Objects in r that are also on a remote stack need to be explicitly marked as escaping roots before the escaping closure computation because they may be dangling objects (such as E in Figure 4.7 (b)) that are already disconnected from other objects in the region. §4.4.3 provides the detailed algorithms for region deallocation and thread stack scanning.

4.4.3 Region Deallocation

Algorithm 3 shows our region deallocation algorithm that is triggered at each `epoch_end()`. This algorithm computes the closure of escaping objects, moves escaping objects to their target regions, and then recycles the whole region.

Algorithm 2: Scanning the stack of thread t.

```
  Input: Thread $t$, Region $r$ to be deallocated
  Output: Map⟨Var, Object⟩ $stackObj$
1 $stackObj \leftarrow \{\}$
2 foreach $Variable\ var \in t.Stack$ do
3     if $\textsc{Region}(O_{var}) = r$ then
4         $stackObj \leftarrow stackObj \bigcup \{var, O_{var}\}$
5 return $stackObj$
```

Finding Escaping Roots There are three kinds of escaping roots for a region r. First, pointees of inter-region/space references recorded in the remember set of r. Second, objects

Algorithm 3: Region deallocation.

```
Input: Region $r$, Thread $t$
1  $Map\langle Var, Object\rangle\ stackObjs \leftarrow$ ScanStack$(t, r)$
2  foreach $\langle var, O_{var}\rangle \in stackObjs$ do
3      if Region$(O_{var}) = r$ then
4          Record a placeholder reference Addr$(var) \xrightarrow{r.parent}$ Addr$(O_{var})$ in $r$'s
             remember set $rs$
5  PauseOtherThreads()
6  foreach $Thread\ t' \in$ Threads() : $t' \neq t$ do
7      $Map\langle Var, Object\rangle remoteObjs \leftarrow$ ScanStack$(t', r)$
8      foreach $\langle var, O_{var}\rangle \in remoteObjs$ do
9          if Region$(O_{var}) = r$ then
10             Record a placeholder reference Addr$(var) \xrightarrow{CS}$Addr$(O_{var})$ in $r$'s
                 remember set $rs$
11 ClosureComputation()
12 ResumePausedThreads()
13 Put all pages of $r$ back onto the available page list
```

referenced by the local stack of the deallocating thread t. Third, objects referenced by the remote stacks of other threads.

Since inter-region/space references have already been captured by the write barrier (§4.4.2), here we first identify objects that escape the epoch via t's local stack, as shown in Lines 1 – 4 of Algorithm 3. Auxilary function ScanStack is shown in Algorithm 2.

Next, Yak identifies objects that escape via remote stacks. To do this, Yak needs to synchronize threads (Line 5). When a remote thread t' is paused, Yak scans its stack variables and returns a set of objects that are referenced by these variables and located in region r. Each such (remotely referenced) object needs to be explicitly marked as an escaping root to be moved to the CS (Line 10) before the transitive closure is computed (Line 11).

No threads are resumed until t completes its closure computation and moves all escaping objects in r to their target regions. Note that it is unsafe to let a remote thread t' proceed even if the stack of t' does not reference any object in r. To illustrate, consider the following scenario. Suppose object A is in the CS and object B is in region r, and there is a reference from A to B. Only A but not B is on the stack of thread t' when r is deallocated. Scanning the stack of t' would not find any new escaping root for r. However, if t' is allowed to proceed immediately, t' could load B onto its stack through A and then delete the reference between A and B. If this occurs before t completes its closure computation, B would not be included in the closure although it is still live.

After all escaping objects are relocated, the entire region is deallocated with all its pages put back onto the free page list (Line 13).

Closure Computation Algorithm 4 shows the details of our closure computation from the set of escaping roots detected above. Since all other threads are paused, closure computation is done together with object moving. The closure is computed based on the remember set *rs* of the current deallocating region r. We first check the remember set *rs* (Line 1): if *rs* is empty, this region contains no escaping objects and hence is safe to be reclaimed. Otherwise, we need to identify all reachable objects and relocate them.

We start by computing the target region to which each *escaping root* O_b needs to be promoted (Lines 2 – 4). We check each reference $addr \xrightarrow{r'} O_b$ in the remember set and then *join* all the regions r' based on the region semilattice. The results are saved in a map *promote*.

We then iterate through all escaping roots in topological order of their target regions (the loop at Line 5). The order is based on the region semilattice (e.g., CS is ordered before any DS region). For each escaping root O_b, we perform a breadth-first traversal inside the current region to identify a closure of *transitively escaping* objects reachable from O_b and put all of them into a queue *gray*. During this traversal (Lines 8 – 23), we compute the regions to which each (transitively) escaping object should be moved and conduct the move. We will shortly discuss the details.

Identifying Target Regions When a transitively escaping object O' is reachable from only one escaping root O_b, we simply use the target region of O_b as the target of O'. When O' is reachable from multiple escaping roots, which may correspond to different target re-

Algorithm 4: Closure computation.

```
   Input: Remember Set $rs$ of Region $r$
1  if The remember set $rs$ of $r$ is NOT empty then
2      foreach Escaping root $O_b \in rs$ do
3          foreach Reference $addr \xrightarrow{r'}$ Addr($O_b$) in $rs$ do
4              $promote[O_b] \leftarrow$ Join ($r'$, $promote[O_b]$)
5      foreach Escaping root $O_b$ in topological order of $promote[O_b]$ do
6          Region $tgt \leftarrow promote[O_b]$
7          Initialize queue $gray$ with $\{O_b\}$
8          while $gray$ is NOT empty do
9              Object $O \leftarrow$ Dequeue($gray$)
10             Write $tgt$ into the region field of $O$
11             Object $O^* \leftarrow$Move($O$, $tgt$)
12             Put a forward reference at Addr($O$)
13             foreach Reference $addr \xrightarrow{x}$Addr($O$) in $r$'s $rs$ do
14                 Write Addr($O^*$) into $addr$
15                 if $x \neq tgt$ then
16                     Add reference $addr \xrightarrow{x}$Addr($O^*$) into the remember set of region
                        $tgt$
17             foreach Outgoing reference $e$ of $O^*$ do
18                 Object $O' \leftarrow$ Target($e$)
19                 if $O'$ is a forward reference then
20                     Write the new address into $O^*$
21                 Region $r' \leftarrow$ Region($O'$)
22                 if $r' = r$ then
23                     Enqueue($O'$, $gray$)
24                 else if $r' \neq tgt$ then
25                     Add reference Addr($O^*$) $\xrightarrow{tgt}$ Addr($O'$) into the remember set of
                        region $r'$
26 Clear the remember set $rs$ of $r$
```

gions, we use the "highest-ranked" (based on the topological ordering of regions in the region semilattice) one among them as the target region of O'.

The topological order of our escaping root traversal is key to our implementation of the above idea. By computing closure for a root with a "higher-ranked" region earlier, objects reachable from multiple roots need to be traversed only once — the check at Line 22 filters out those that already have a region r' ($\neq r$) assigned in a previous iteration of the loop because the region to be assigned in the current iteration is guaranteed to be "lower-ranked" than r'. When this case happens, the traversal stops further tracing the outgoing references from O'.

Figure 4.8 (a) shows a simple heap snapshot when region $\langle r_{21}, t_1 \rangle$ is about to be deallocated. There are two references in its remember set, one from region $\langle r_{11}, t_1 \rangle$ and a second from $\langle r_{12}, t_2 \rangle$. The objects C and D are the escaping roots. Initially, our algorithm determines that C will be moved to $\langle r_{11}, t_1 \rangle$ and D to the CS (because it is reachable from a concurrent region $\langle r_{12}, t_2 \rangle$). Since the CS is higher-ranked than $\langle r_{11}, t_1 \rangle$ in the semilattice, the transitive closure computation for D occurs before C, which sets E's target to the CS. Later, when the transitive closure for C is computed, E will be ignored (since it has been visited).

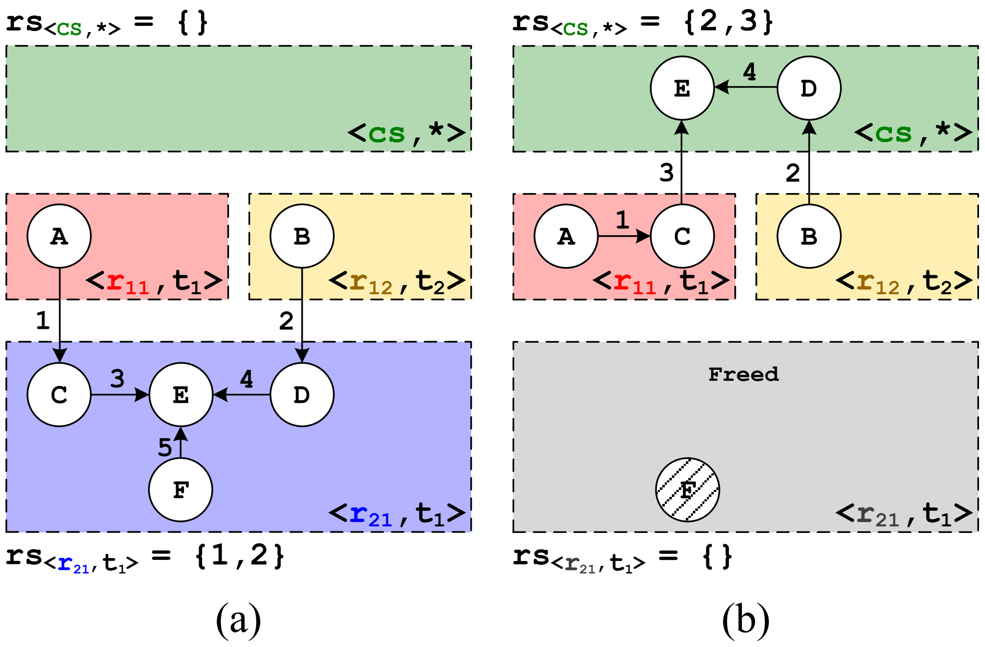


Figure 4.8: An example heap snapshot (a) before and (b) after the deallocation of region $\langle r_{21}, t_1 \rangle$.

Updating Remember Sets and Moving Objects Because we have pause all threads, object moving is safe (Line 11). When an object O is moved, we need to update *all* (stack and heap) locations that store its references. There can be three kinds of locations from which it is referenced: (1) intra-region locations (i.e., referenced from another object in r);

(2) objects from other regions or the CS; and (3) stack locations. We discuss how each of these types is handled by Algorithm 4.

1. **Intra-region locations.** To handle intra-region references, we follow the standard GC treatment by putting a special *forward reference* at O's original location (Line 12). This will notify intra-region incoming references of the location change — when this old location of O is reached from another reference, the forward reference there will be used to update the source of that reference (Line 20).

2. **Objects from another region.** References from these objects must have been recorded in r's remember set. Hence, we find all inter-region/space references of O in the remember set rs and update the source of each such reference with the new address O^* (Line 14). Since O^* now belongs to a new region tgt, the inter-region/space references that originally went into region r now go into region tgt. If the regions contain such a reference are not tgt, such references need to be explicitly added into the remember set of tgt (Line 16).

   When O's outgoing edges are examined, moving O to region tgt may result in new inter-region/space references (Lines 24 – 25). For example, if the target region r' of a pointee object O' is not tgt (i.e., O' has been visited from another escaping root), we need to add a new entry $\textsc{Addr}(O^*)\xrightarrow{tgt}\textsc{Addr}(O')$ into the remember set of r'.

3. **Stack locations.** Since stack locations are also recorded as entries of the remember set, updating them is performed in the same way as updating heap locations. For example, when O is moved, Line 14 would update each reference going to O in the remember set. If O has (local or remote) stack references, they must be in the remember set and updated as well.

After the transitive closure computation and object promotion, the remember set rs of region r is cleared (Line 26).

Figure 4.8 (b) shows the heap after region $\langle r_{21}, t_1 \rangle$ is deallocated. The objects C, D, and E are escaping objects and will be moved to the target region computed. Since D and E belong to the CS, we add their incoming references 2 and 3 into the remember set of the CS. Object F does not escape the region, and hence, is automatically freed.

4.4.4 Collecting the CS

We implement two modifications to the Parallel Scavenge GC to collect the CS. First, we make the GC run locally in the CS. If the GC tracing reaches a reference to a region object, we simply ignore the reference.

Second, we include references in the CS' remember set into the tracing roots, so that corresponding CS objects would not be mistakenly reclaimed. Before tracing each such reference, we validate it by comparing the address of its target CS object with the current content in its source location. If they are different, this reference has become invalid and is discarded. Since the Parallel Scavenge GC moves objects (away from the young generation), Yak also needs to update references in the remember set of each region when their source in the CS is moved.

4.5 Evaluation

This section presents an evaluation of Yak on widely-deployed real-world systems.

4.5.1 Methodology and Benchmarks

We have evaluated Yak on Hyracks [51], a parallel dataflow engine powering the Apache AsterixDB [32] stack, Hadoop [35], a popular distributed MapReduce [65] implementation, and GraphChi [108], a disk-based graph processing system. These three frameworks were selected due to their popularity and diverse characteristics. For example, Hyracks and Hadoop are distributed frameworks while GraphChi is a single-PC disk-based system. Hyracks runs one

JVM on each node with many threads to process data while Hadoop runs multiple JVMs on each node, with each JVM using a small number of threads.

For each framework, we selected a few representative programs, forming a benchmark set with nine programs — External Sort (ES), Word Count (WC), and Distributed Grep (DG) for Hyracks; In-map Combiner (IC), Top-word Selector (TS), and Distributed Word Filter (DF) for Hadoop; Connected Components (CC), Community Detection (CD), and Page Rank (PR) for GraphChi. Table 4.2 provides the descriptions of these programs.

FW	**P**	**Description**
Hyracks	ES	Sort a large array of data that cannot fit in main memory
	WC	Count word occurrences in a large document
	DG	Find matches based on user-defined regular expressions
Hadoop	IC	Count word frequencies in a corpus using local aggregation
	TS	Select a number of words with most frequent occurrences
	DF	Return text with user-defined words filtered out
GraphChi	CC	Identify strongly connected components (label propagation)
	CD	Detect communities (label propagation)
	PR	Compute webpage rankings (SpMV kernel)

Table 4.2: Our benchmarks and their descriptions.

Table 4.3 shows the datasets and heap configurations in our experiments. For Yak, the heap size is the sum of the sizes of both CS and DS. Since we fed different datasets to various frameworks, their memory requirements were also different. Evidence [48] shows that in general the heap size needs to be at least twice as large as the minimum memory size for the GC to perform well. We selected the heap configurations shown in Table 4.3 based on this observation — they are roughly 2× – 3× of the minimum heap size needed to run the original JVM.

FW	**Dataset**	**Size**	**Heap Configs**
Hyracks	Yahoo Webmap	72GB	20GB, 24GB
Hadoop	StackOverflow	37GB	2&1GB, 3&2GB
GraphChi	Sample twitter-2010	$E = 100M$, $V = 62M$	6GB, 8GB

Table 4.3: Datasets and heap configurations used to run our programs; for Hadoop, the configurations *a*&*b* GB are the max heap sizes for each map (*a*) and reduce task (*b*).

In a small number of cases, the JVM uses hand-crafted assembly code to allocate objects directly into the heap without calling any C/C++ function. While we have spent more than a year on development, we have not yet performed any assembly-based optimizations for Yak. Thus, this assembly-based allocation in the JVM would allow some objects in an epoch to bypass Yak's allocator. To solve the problem, we had to disable this option and force all allocation requests to go through the main allocation entrance in C++. For a fair comparison, we kept the assembly-level allocation option disabled for all experiments including both Yak and original GC runs. We saw a small performance degradation (2–6%) after disabling this option in the JVM.

We ran Hyracks and Hadoop on an 11-node cluster, each with 2 Xeon(R) CPU E5-2640 v3 processors, 32GB memory, 1 SSD, running CentOS 6.6. We ran GraphChi on one node of this cluster, since it is a single-PC system. For Yak, we let the ratio between the sizes of the CS and the DS be 1/10. We did not find this ratio to have much impact on performance as long as the DS is large enough to contain objects created in each epoch. The page size in DS is 32KB by default. We performed experiments with different DS-page sizes and report these results shortly. We focus our comparison between Yak and Parallel Scavenge (PS) — the Oracle JVM's default production GC.

We ran each program for three iterations. The first iteration warmed up the JIT. The performance difference between the last two iterations were negligible (e.g., less than 5%). This section reports the medians. We also confirmed that no incorrect results were produced by Yak.

4.5.2 Epoch Specification

We performed our annotation by strictly following existing framework APIs. For Hyracks, an epoch covers the lifetime of a (user-defined) dataflow operator (i.e., via `open/close`); for Hadoop, it includes the body of a Map or Reduce task (i.e., via `setup/cleanup`). For GraphChi, we let each epoch contain the body of a sub-interval specified by a `beginSubInterval` callback, since each sub-interval holds and processes many vertices and edges. A sub-interval creates many threads to load sliding shards and execute update functions. The body of each such thread is specified as a sub-epoch. It took us about ten minutes to annotate all three programs on each framework. Note that our optimization for these frameworks only scratches the surface; vast opportunities are possible if both user-defined operators and system's built-in operators are epoch-annotated.

4.5.3 Latency and Throughput

Figure 4.9 depicts the detailed performance comparisons between Yak and PS. Table 4.4 summarizes Yak's performance improvement by showing Overall run time, as well as GC and Application time and Memory consumption, all normalized to those of PS.

FW	Overall	GC	App	Mem
Hyracks	0.14 ∼ 0.64	0.02 ∼ 0.11	0.31 ∼ 1.05	0.67 ∼ 1.03
	(0.40)	(0.05)	(0.77)	(0.78)
Hadoop	0.73 ∼ 0.89	0.17 ∼ 0.26	1.03 ∼ 1.35	1.07 ∼ 1.67
	(0.81)	(0.21)	(1.13)	(1.44)
GraphChi	0.70 ∼ 0.86	0.15 ∼ 0.56	0.91 ∼ 1.13	1.07 ∼ 1.34
	(0.77)	(0.38)	(1.01)	(1.21)

Table 4.4: Summary of Yak performance normalized to baseline PS in terms of **Overall** run time, **GC** time, including Yak's region deallocation time, **App**lication (non-GC) time, and **Mem**ory consumption across all settings on each framework. The values shown depict Min ∼ Max and (Mean), and are normalized to PS. A lower value indicates better performance versus PS.

For Hyracks, Yak outperforms PS in all evaluated metrics. The GC time is collected by identifying the maximum GC time across runs on all slave nodes. Data-parallel tasks in Hyracks are isolated by design and they do not share any data structures across task instances. Hence, while Yak's write barrier incurs overhead, almost all references captured by the write barrier are intra-region references and thus they do not trigger the slow path of the barrier (i.e., updating the remember set). Yak also improves the (non-GC) application performance — this is because PS only performs thread-local allocation for small objects and the allocation of large objects has to be in the shared heap, protected by locks. In Yak, however, all objects are allocated in thread-local regions and thus threads can allocate objects completely in

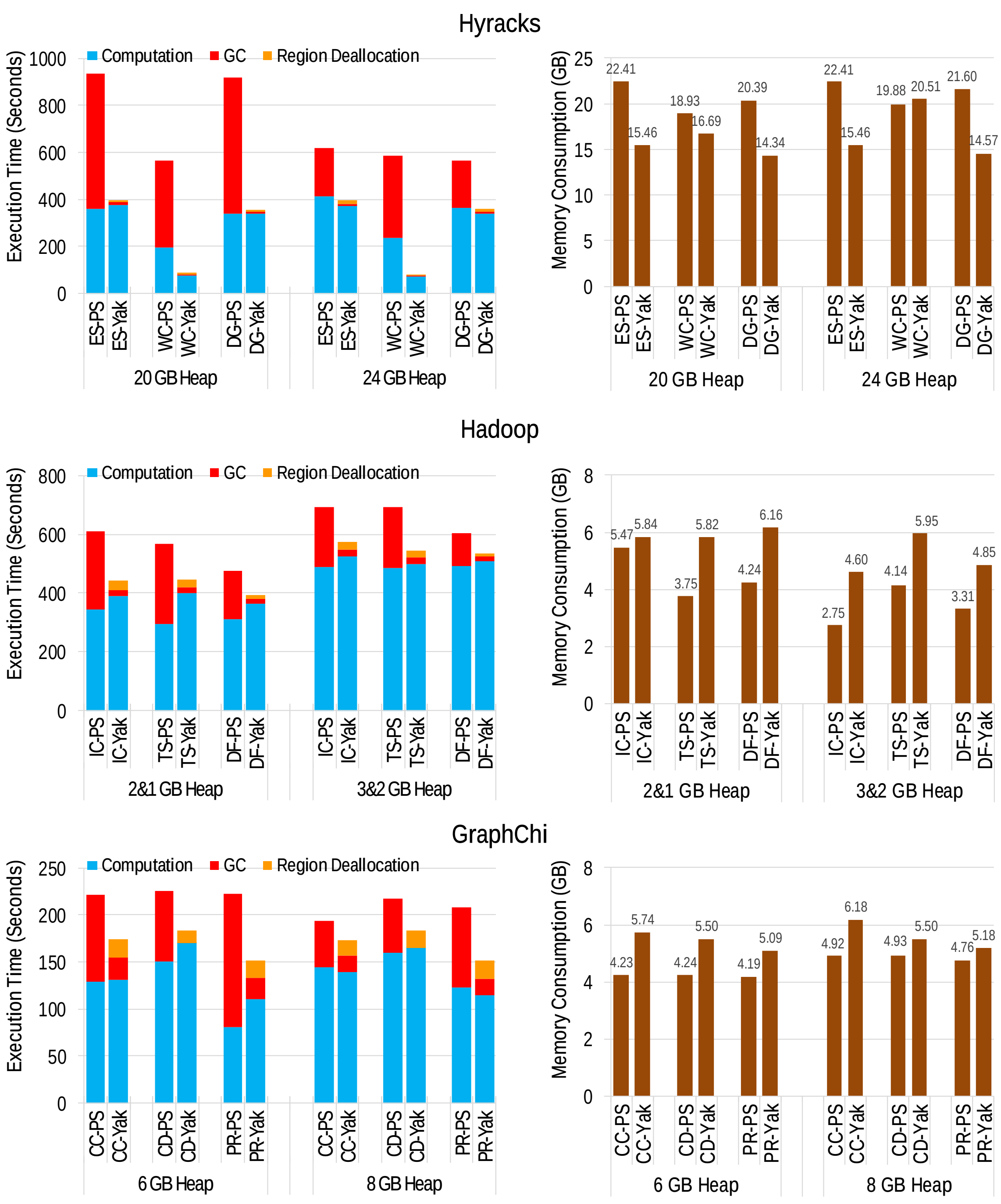


Figure 4.9: Performance comparisons between PS and Yak on various programs.

parallel. Lock-free allocation is the major reason why Yak improves application performance because large objects (e.g., arrays in HashMaps) are frequently allocated in such programs.

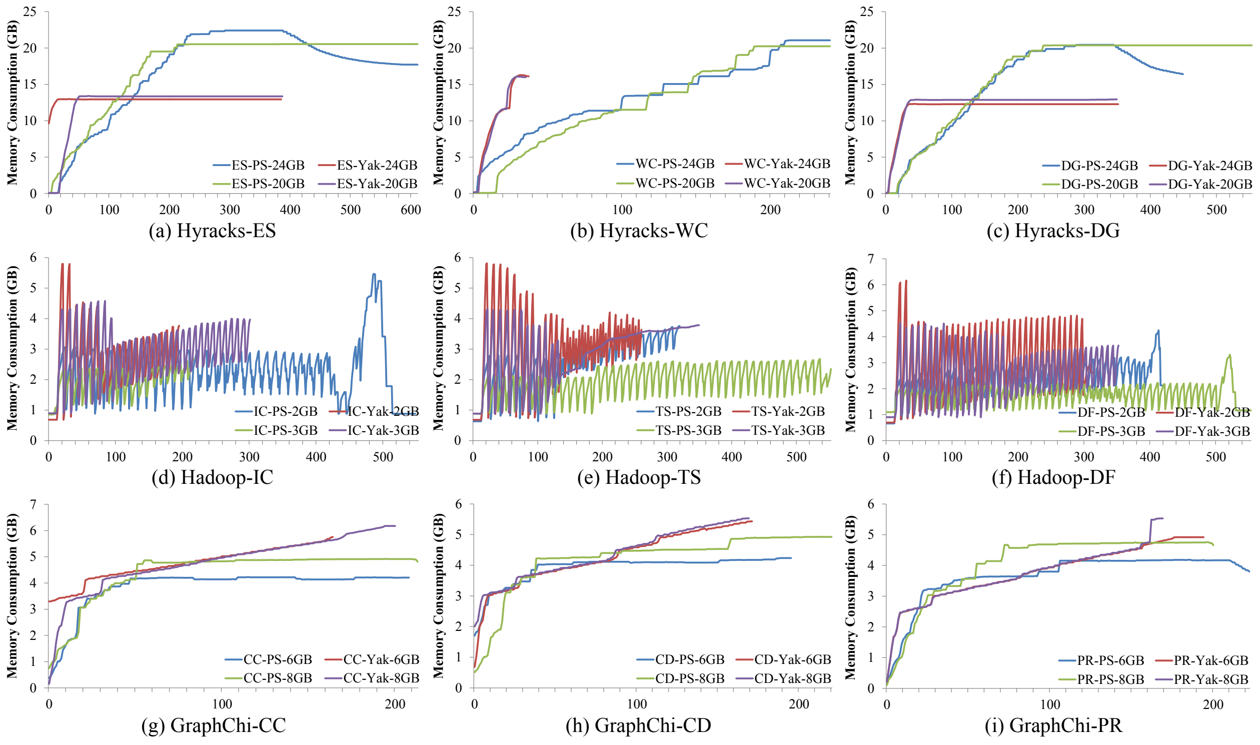


Figure 4.10: Memory footprints collected from `pmap`.

For Hadoop and GraphChi, while Yak substantially reduces the GC time and the overall execution time, it increases the application time and memory consumption. Longer application time is expected because (1) memory reclamation (i.e., region deallocation) shifts from the GC to the application execution, with Yak, and (2) the write barrier is triggered to record a large number of references. For example, Hadoop has a state object (i.e., `context`) in the control path that holds objects created in the data path, generating many inter-space references. In GraphChi, a number of large data structures are shared among different data-loading threads, leading to many inter-region references (e.g., reported in Table 4.5). Recording all these references makes the barrier overhead stand out.

We envision two approaches that can effectively reduce the write barrier cost. First, existing GCs all have manually crafted/optimized assembly code to implement the write barrier. As mentioned earlier, we have not yet investigated assembly-based optimizations for Yak. We expect the barrier cost to be much lower when these optimizations are implemented. Second, adding extra annotations that define finer-grained epochs may provide further performance improvement. For example, if objects reachable from the state object can be created in the CS in Hadoop, the number of inter-space references can be significantly reduced. In this experiment, we did not perform any program restructuring, but we believe significant performance potential is possible with that: it is up to the developer to decide how much annotation effort she can afford to expend for how much extra performance gain she would like to achieve.

Yak greatly shortens the pauses caused by GC. When Yak is enabled, the maximum (deallocation or GC) pauses in Hyracks, Hadoop, and GraphChi are, respectively, 1.82, 0.55, and 0.72 second(s), while the longest GC pauses under PS are 35.74, 1.24, and 9.48 seconds, respectively.

As the heap size increases, there is a small performance improvement for PS due to fewer GC runs. The heap increase has little impact on Yak's overall performance, given that the CS is small anyways.

4.5.4 Memory Usage

We measured memory usage by periodically running `pmap` to understand the overall memory consumption of the Java process (for both the application and GC data). Figure 4.10 compares the memory footprints of Yak and PS under different heap configurations. For Hyracks and GraphChi, memory footprints are generally stable, while Hadoop's memory consumption fluctuates. This is because Hadoop runs multiple JVMs and different JVM instances are frequently created and destroyed. Since the JVM never returns claimed memory back to the OS until it terminates, the memory consumption always grows for Hyracks and GraphChi. The amount of memory consumed by Hadoop, however, drops frequently due to the frequent creation and termination of its JVM processes.

Note that the end times of Yak's memory traces on Hadoop in Figure 4.10 are earlier than the execution finish time reported in Figure 4.9. This is because Figure 4.10 shows the memory trace of the node that has the *highest memory consumption*; the computation on this node often finishes before the entire program finishes.

Yak constantly has lower memory consumption than PS for Hyracks. This is primarily because Yak can recycle memory immediately when a data processing thread finishes, while there is often a delay before the GC reclaims memory. For Hadoop and GraphChi, Yak has slightly higher memory consumption than PS. The main reason is that there are many control objects created in the data path and allocated in regions. Those objects often have shorter lifespans than their containing regions and, therefore, PS can reclaim them more efficiently than Yak. We plan to solve this problem by developing *feedback-directed allocation* – if objects created by an allocation site keep getting allocated in regions but later copied to the CS, these objects are likely to be control objects and their allocation sites can be modified to directly allocate such objects in the CS in future executions.

Space Overhead To understand the overhead of the extra 4-byte field *region* in each object header, we ran the GraphChi programs with the unmodified HotSpot 1.8.0_74 and compared peak heap consumption with that of Yak (by periodically running `pmap`). We found that the difference (i.e., the overhead) is relatively small. Across the three GraphChi benchmarks, this overhead varies from 1.1% to 20.8%, with an average of 12.2%.

4.5.5 Performance Breakdown

Yak's heap To provide a deeper understanding of Yak's performance, Table 4.5 reports various statistics on Yak's heap. Yak was built based on the assumption that in a typical Big Data system, only a small number of objects escape from the data path to the control path. This assumption has been validated by the fact that the ratios between numbers in **#CSR** and **#TR** are generally very small. As a result, each region has only very few objects (**%CSO**) that escape to the CS when the region is deallocated.

Sensitivity with page sizes Figure 4.11 depicts execution time (top row) and memory (bottom row) performance of Yak, when different page sizes are used on GraphChi with 8GB

Program	#CSR	#CRR	#TR	%CSO	#R
Hyracks-ES	2051	243	3B	0.0028%	103K
Hyracks-WC	2677	4221	213M	0.0043%	148K
Hyracks-DG	2013	16	2B	0.0034%	101K
Hadoop-IC	60K	0	2B	0%	598
Hadoop-TS	60K	0	2B	0%	598
Hadoop-DF	33K	0	1B	0%	598
GraphChi-CC	53K	25K	653M	0.044%	2699
GraphChi-CD	52K	14M	614M	1.3%	2699
GraphChi-PR	54K	24K	548M	0.060%	2699

Table 4.5: Statistics on Yak's heap: numbers of cross-space references (CSR), cross-region references (CRR), and total references generated by stores (TR); average percentage of objects escaping to the CS (CSO) among all objects in a region when the region retires; and total number of regions created during execution (R).

heap. Execution time under different page sizes does not vary much across all three programs, while the peak memory consumption generally goes up when page size increases.

Scalability To understand how Yak and PS perform when datasets of different sizes are processed, we ran Hyracks ES with four subsets of the Yahoo Webmap with sizes of 9.4GB, 14GB, 18GB, and 44GB respectively. Figure 4.12 shows that Yak consistently outperforms PS and its performance improvement increases with the size of the dataset processed.

The write barrier and region deallocation are the two major sources of Yak's application overhead. As shown in Figure 4.9, region deallocation time accounts for 2.4%-13.1% of total execution time across the benchmarks. Since all of our programs are multi-threaded, it is difficult to pinpoint the exact contribution of the write barrier to execution time. To get an idea of the sensitivity to this barrier's cost, we manually modified GraphChi's execution engine to enforce a barrier between threads that load sliding shards and execute updates. This has the effect of serializing the threads and making the program sequential. For all three programs on GraphChi, we found that the mutator time (i.e., non-pause time) increased by an overall of 24.5%. This shows that the write barrier is a major bottleneck, providing strong motivation for us to hand optimize it in assembly code in the near future.

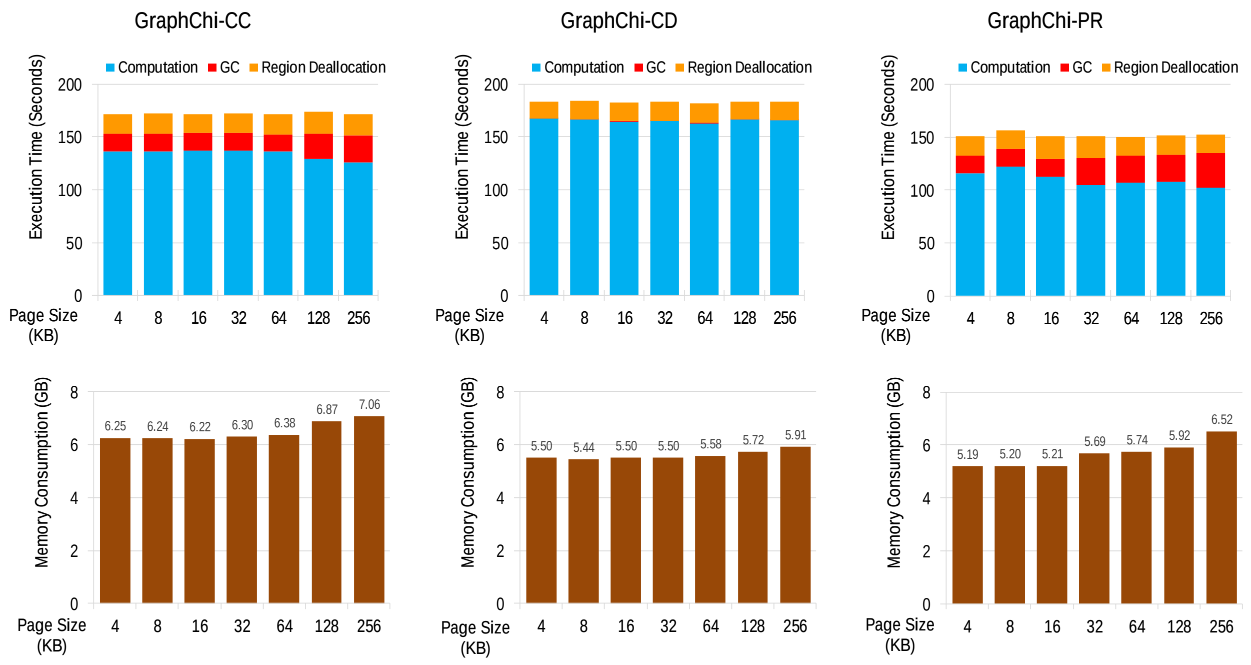


Figure 4.11: Yak performance comparisons on GraphChi between different page sizes

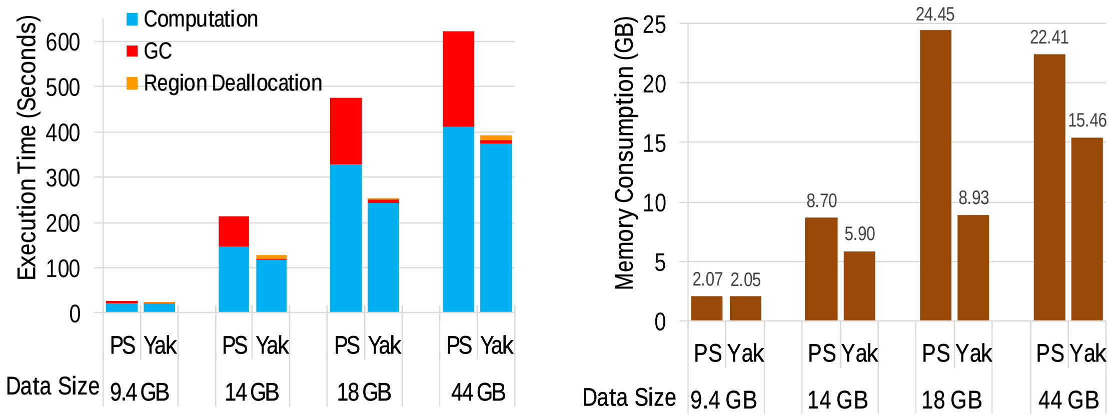


Figure 4.12: Performance comparisons between Yak and PS when datasets of various sizes were sorted by Hyracks ES on a 24GB heap

4.5.6 Comparison with G1

Garbage First (G1) [67] is an emerging, high-performance collector that is expected to become the default production GC, replacing the PS. G1 also features a space-partitioned heap with many fixed-size regions. The runtime collects statistics on each region and direct GC effort on regions with highest percentage of dead objects (hence the name Garbage First), avoiding the cost of whole-heap tracing.

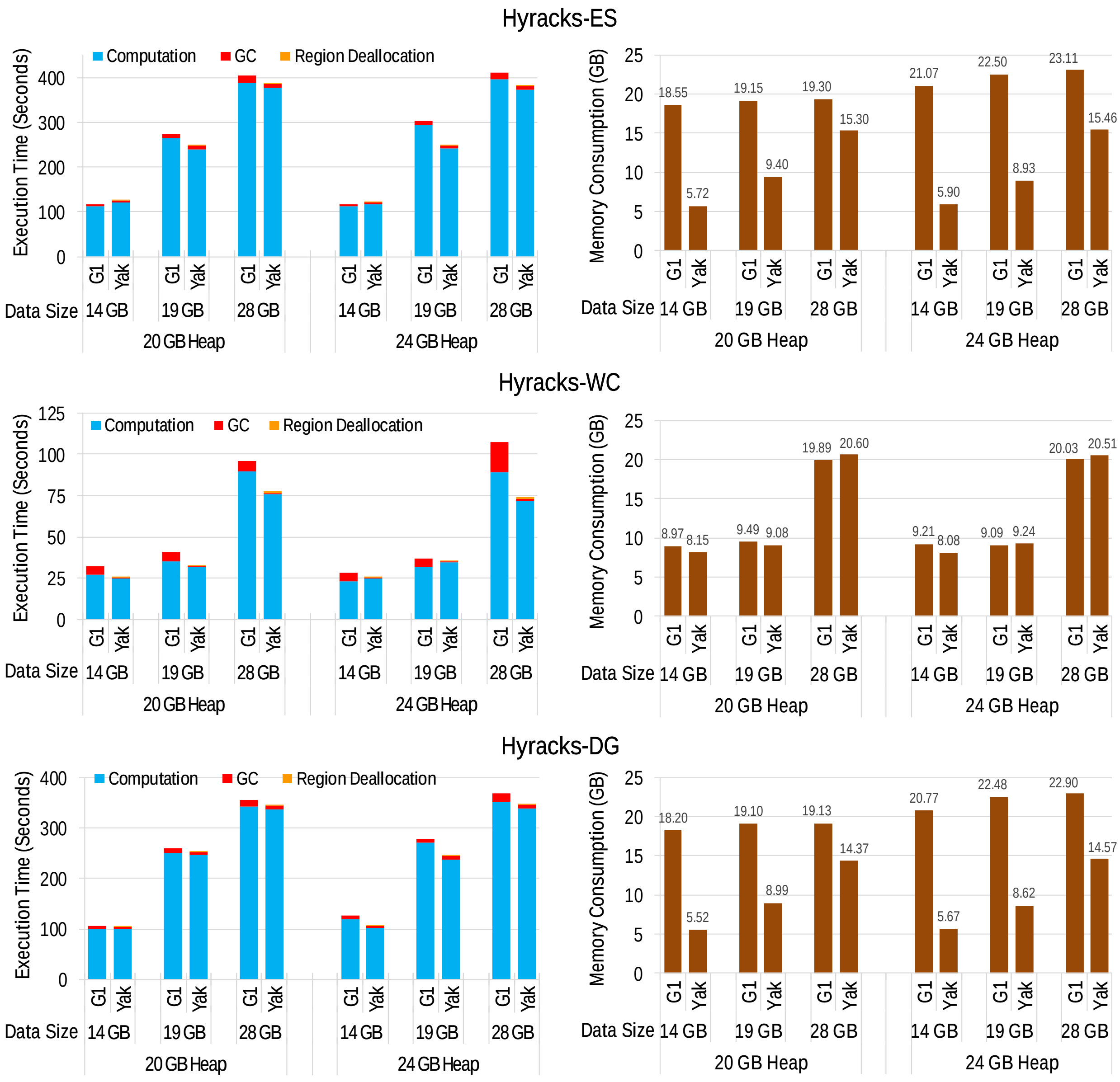


Figure 4.13: Performance comparisons between G1 and Yak on Hyracks

We extensively compared Yak with G1 using Hyracks as a case study. We used the same three programs — External Sort (ES), Word Count (WC), and Distributed Grep (DG) on the same cluster mentioned earlier. For input, we used three different subsets of the Yahoo Webmap [4] with sizes of 14GB, 19GB, and 28GB. Two heap configurations used were 20GB and 24GB. Yak's configuration stayed unchanged (e.g., ratio of CS/DS is 1/10 and page size is 32KB). Each program is run 3 times and the median is reported. Table 4.6 gives a summary of performance of both G1 and Yak, normalized to that of PS. We report the detailed breakdown of runtime median between G1 and Yak in Figure 4.13. For simplicity and better constrast, the bars of PS are omitted from the figure.

	Overall	**GC**	**App**	**Mem**
G1	0.13 ∼ 0.67	0.02 ∼ 0.10	0.19 ∼ 1.07	0.86 ∼ 2.42
	(0.43)	(0.06)	(0.68)	(1.33)
Yak	0.12 ∼ 0.62	0.01 ∼ 0.08	0.16 ∼ 1.04	0.37 ∼ 1.03
	(0.40)	(0.04)	(0.64)	(0.74)

Table 4.6: Summary of G1 and Yak performance normalized to baseline PS. Each cell shows Min ∼ Max and (Mean), and are normalized to PS. Lower is better.

The experiment shows G1 is able to mitigate the GC overhead. Under G1, the system sees great improvement. GC only accounts for 3–18% (mean 7%) of the execution time, saving more than half (i.e., 57%) of the end-to-end time. The ability to direct GC effort on regions with mostly dead objects enables the runtime strategically and efficiently reclaim memory. Despite such performance, Yak still constantly outperforms G1. Comparing to G1, we have seen on average 10.6% running time reduction (and up to 31.1%), and 1.5–26.3× (mean 6.9×) GC time reduction. Moreover, unlike G1 where we see an increased of the peak memory consumption due to the accumulation of stale data, the peak memory consumption on Yak has been reduced also. On average, Yak uses 35.71% less memory, suggesting Yak can scale to much larger input datasets.

4.6 Summary

While GC has been extensively studied, existing research centers around the generational hypothesis, improving various aspects of the collection/application performance based on this hypothesis. Yak, in contrast, is a high-throughput, low-latency GC tailored for managed Big Data systems. Yak adapts to the two very different types of object behavior (generational and epochal) observed in modern data-intensive workloads. Yak is the *first hybrid GC* that splits the heap into a control space (CS) and a data space (DS), which respectively employ generation-based and region-based algorithms to automatically manage memory.

Yak offers an *automated and systematic solution*, requiring *zero* code refactoring. Yak asks the developer to mark the beginning and end points of each epoch in the program. This is a simple task that even novices can do in minutes, and is already required by many Big Data infrastructures (e.g., the `setup/cleanup` APIs in Hadoop). Objects created inside each epoch are allocated in the DS, while those created outside are allocated in the CS. Yak manages all data-space objects using epoch-based regions and deallocates each region as a whole at the end of an epoch, while efficiently tracking the small number of objects whose lifetimes span region boundaries. Yak uses a lattice-based promotion algorithm to migrate escaping objects during region deallocation. This handling completely frees the developers from the stress of understanding object life spans, making Yak practical enough to be used in real settings. Since the number of objects to be traced in the CS is very small and only escaping objects in the DS need tracing, the memory management cost can be substantially reduced compared to a state-of-the-art generational GC.

CHAPTER 5

Mitigating Data Transfer Mismatch

The key problem with existing serialization/deserialization (S/D) libraries is that, within 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. We advocate to build a "skyway" between managed heaps so that data objects no longer need to be pushed down to a lower level for transfer.

We will start with an empirical study of data transfer cost (Section 5.1), followed by an overview of our solution to reduce such cost in Section 5.2. Section 5.3 includes several implementation details of the system. After that, we present our experimental results in Section 5.4, before closing this chapter in Section 5.5.

5.1 Serialization/Deserialization Costs in Real World

In existing serializers, the number of times serialization/deserialization functions are invoked is proportional to the dataset cardinality; every data transfer can easily require several millions invocations, potentially taking a significant fraction of the execution time. To understand the serialization/deserialization costs in the real world, we have performed a set of experiments using Apache Spark [182] as a representative platform. We execute Spark on a small cluster of 3 worker nodes, each with 2 Xeon(R) CPU E5-2640 v3 processors, 32GB memory, 1 SSD, running CentOS 6.8, connected via InfiniBand. We ran a Triangle Counting (TC) algorithm over the Live Journal graph [42] that counts the number of triangles induced by graph edges. It is widely used in social network analysis for analyzing the graph connectivity properties [164]. We used Oracle JDK 8 (build 25.71) and let each slave run one single executor — the single-thread execution on each slave made it easy for us to measure the breakdown of performance. The size of the input graph was around 1.2GB and we gave each JVM a 20GB heap — a large enough heap to perform in-memory computation — as is the recommended practice in Spark. Tungsten sort was used to shuffle data.

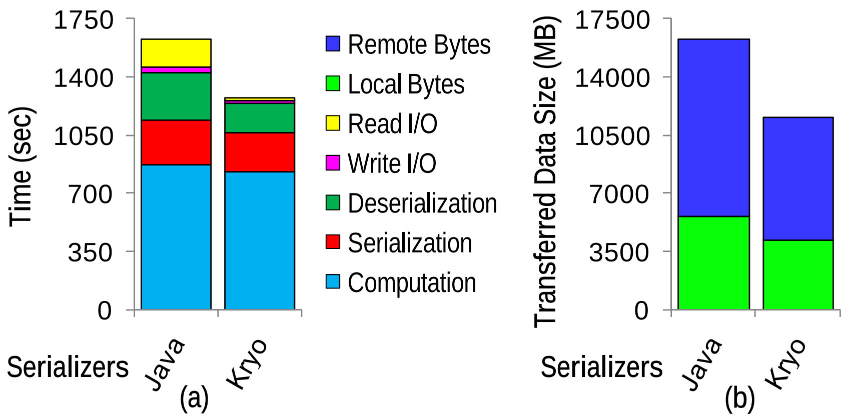


Figure 5.1: Spark serialization/deserialization costs: (a) performance breakdown and (b) bytes shuffled under the Java and Kryo serializer. partitions.

Figure 5.1(a) shows Spark's performance under the Kryo and Java serializers. Before transferring data over the network, Spark shuffles and sorts records, and saves the sorted records as disk files. The cost is thus broken down into five components: computation time, serialization time (measured as time spent on turning RDD records into byte sequences), write I/O (measured as the time writing bytes onto disk), deserialization time (measured as time spent on reconstructing RDD record objects from bytes), and read I/O (measured as time reading bytes). Since each JVM has a large heap compared to the amount of data processed, the garbage collection cost is less than 2.1% and thus not shown on the figure. The network cost is included in the read I/O. Local Bytes and Remote Bytes in Figure 5.1(b) show the number of bytes fetched from the local and remote RDDs, repectively.

One observation is that the invocation of serialization/deserialization functions accounts for a huge portion (more than 30%) of the total execution time under both Kryo and the Java serializer. Under Kryo, the invocations of the serialization and deserialization take 18.2% and 14.1% of the total execution time, respectively; under the Java serializer, these two take

16.3% and 17.8%. The actual write and read I/O time is much shorter in comparison, taking 1.4% and 1.1% under Kryo, and 2.3% and 9.9% under the Java serializer. The read I/O is significantly increased under the Java serializer primarily because the Java serializer needs to read many type strings. For example, serializing an object containing a 1-byte data field can generate a 50-byte sequence [168] — in addition to its own field and the fields in its superclasses, the serializer needs to (1) write out the class name and (2) recursively write out the description of the superclasses of the object's class until it reaches `java.lang.Object` (i.e., the root of all classes).

Another observation is that the S/D process is a bottleneck that cannot be easily removed by upgrading hardware. Unlike other bottlenecks such as GC (that can be eliminated almost entirely by using a large heap) or I/O (that can be significantly reduced by using fast SSDs and InfiniBand networks), S/D is a memory- and compute-intensive process that turns heap objects into bytes and vice versa. The inefficiencies inherent in the process strongly call for system-level optimizations.

5.2 Skyway Design Overview

To combat data transfer mismatch, we propose Skyway to enhance the JVM, as shown in Figure 5.2. By building a direct connection between two managed heaps, we would avoid completely the process of serialization and deserialization that are required to transform data format. We advocate transferring object graphs *as-is* from heap to heap, enabling them to be used on a remote node right after the transfer. Specifically, given a root object o specified by the application (e.g., the RDD object in Spark), the Skyway-enhanced JVM performs a GC-like heap traversal starting from o, copies every reachable object into an output buffer, and conducts lightweight adjustment to machine-dependent meta data stored in an object without changing the object format. This output buffer can then be copied as a whole directly into the remote heap and used almost immediately after the transfer.

The benefit provided by Skyway to existing and future Big Data systems is two-fold: (1) Skyway completely eliminates the cost of accessing fields and types, saving computation costs; and (2) the developer does not need to handcraft any S/D functions, yet enjoying efficient data transfer.

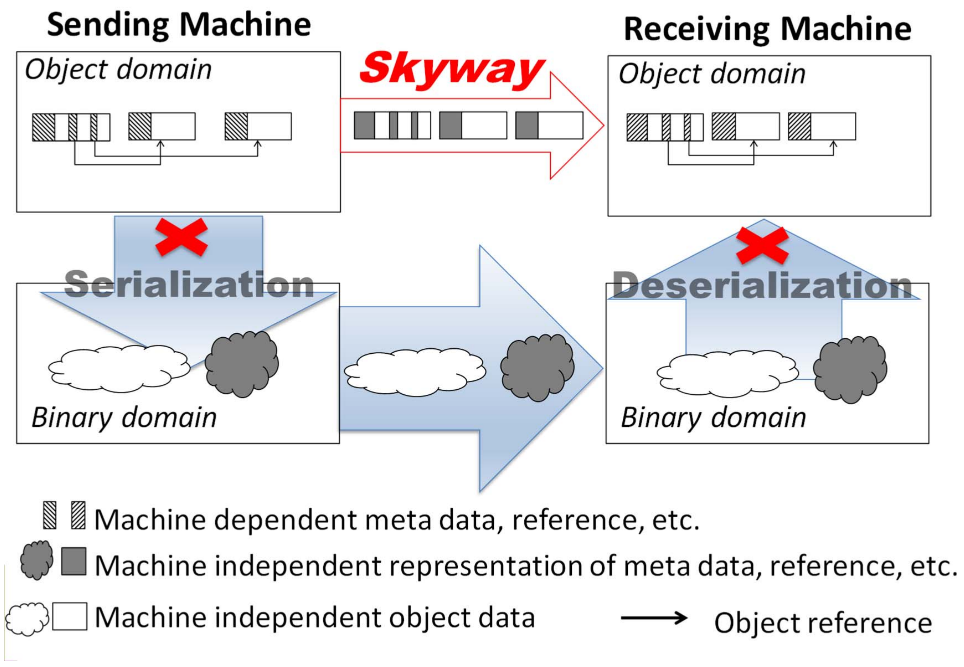

Sending Machine
Object domain
Skyway
Receiving Machine
Object domain
Serialization
Deserialization
Binary domain
Binary domain
Machine dependent meta data, reference, etc.
Machine independent representation of meta data, reference, etc.
Machine independent object data
Object reference

Figure 5.2: Skyway's overview.

Next we explain how Skyway is designed towards three performance goals — correctness, efficiency, and ease of integration. Figure 5.3 shows the system architecture of Skyway.

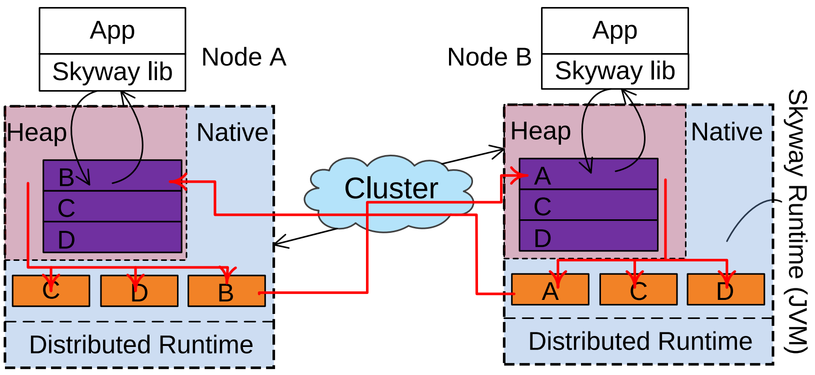


Figure 5.3: Skyway's system architecture. Purple and orange rectangles represent input (in-heap) buffers and output (native) buffers, respectively; objects flow along red arrows.

5.2.1 Correctness

Skyway adjusts machine-specific parts of each transferred object to guarantee execution correctness. First, Skyway fills the type field of an object header with an automatically maintained global type-ID during sending, and later replaces it with the correct type representation on the receiving node. The details are presented in §5.3.1. Second, Skyway replaces the references stored in all non-primitive fields of an object with relativized references during sending, and turns them back to the correct absolute references during receiving. The details are presented in §5.3.2. Finally, certain meta data such as GC bits and lock bits need to be reset when objects are moved to another machine. Skyway resets these flags at sending, and does *not* need to access them at receiving.

Skyway also provides support for heterogeneous clusters where JVMs on different machines may support different object formats. If the sender and receiver nodes have different JVM specifications, Skyway adjusts the format of each object (e.g., header size, pointer size, or header format) when copying it into the output buffer. This incurs an extra cost only on the sender node while the receiver node pays *no* extra cost for using the transferred objects. For homogeneous clusters, such platform-adjustment cost is not incurred on any nodes. The only assumption Skyway uses is that the sender and the receiver use the same version of each transfer-related class — if two versions of the same class have different fields, object reading would fail. However, this assumption is not unique for Skyway; all other serializers assume the same.

5.2.2 Efficiency

Skyway uses a GC-like traversal to discover the object graph reachable from a set of root objects specified by developers. A naive approach could transfer each object individually as it is being visited during the traversal. This approach, however, would generate a great number of small network packets, leading to performance degradation. To improve efficiency, Skyway uses buffering — Skyway copies every object encountered during the traversal into a buffer on the sending node (i.e., output buffer) and streams the buffer content to the corresponding buffer(s) on the receiving node (i.e., input buffer). Both output and input buffers are carefully designed for efficiency concerns. Multi-threaded data transfer is also supported (cf. §5.3).

Skyway output buffers are segregated by receivers — objects with the same destination are put into the same output buffer. Only one such output buffer exists for each destination. The output buffer can be safely cleared after its objects are sent. Skyway input buffers are segregated by senders, so that data objects coming from different senders can be written simultaneously without synchronization. Note that the heap of a receiver node may actually contain multiple input buffers for each sender, each holding objects sent in a different round of shuffling from the sender. Skyway does not reuse an old input buffer unless the developer explicitly frees the buffer using an API — frameworks such as Spark cache all RDDs in memory and thus Skyway keeps all input buffers.

Output buffers are located in off-the-heap, native memory — they will not interfere with the GC, which could reclaim data objects before they are sent if these buffers were in the managed heap. Input buffers are allocated from the managed heap so that data coming from a remote node is directly written into the heap and can be used right away. Furthermore, while each input buffer is shown as consuming contiguous heap space in Figure 5.3, we allow it to span multiple small memory chunks for two reasons. First, due to streaming, the receiver may not have the knowledge of the number of sent bytes, and hence, determining the input-buffer size is difficult. Second, allocating large contiguous space can quickly lead to memory fragmentation, which can be effectively mitigated by using smaller memory chunks. Details can be found in §5.3.3.

Streaming is an important feature Skyway provides for these buffers: for an output buffer, it is both time-inefficient and space-consuming if we do not send data until all objects are in; for an input buffer, streaming would allow the computation to be performed in parallel with data transfer. Supporting streaming creates many challenges, e.g., how to adapt pointers without multiple scans and how to manage memory on the receiver node. Details can be found in §5.3.2.

5.2.3 Ease of Integration

Skyway aims to provide a simple interface for application developers. Skyway should support not only the development of brand new systems but also easy S/D library integration for existing systems such as Spark. To this end, Skyway provides a set of high-level Java APIs that are directly compatible with the standard Java serializer.

Skyway provides `SkywayObjectOutputStream` and `SkywayObjectInputStream` classes that are subclasses of the standard `ObjectOutputStream` and `ObjectInputStream`. These two classes create an interface for Skyway's (native) implementation of the `readObject` and `writeObject` methods. A `SkywayObjectOutputStream/SkywayObjectInputStream` object is associated with an output/input buffer. We have also created our `SkywayFileOutput-Stream` / `SkywayFileInputStream` and `SkywaySocketOutputStream`/ `SkywaySocketInput-Stream` classes – one can easily program with Skyway in the same way as programming with the Java serializer.

Switching a program from using its original library to using Skyway requires light code modifications. For example, we do not need to change object-writing/reading calls such as `stream.writeObject(o)` at all. The only modification is to (1) instantiate `stream` to be a `SkywayFileOutputStream` object instead of any other type of `ObjectOutputStream` objects and (2) identify a shuffling phase with an API function `shuffleStart`. Since all of our output buffers need to be cleared before the next shuffling phase starts (§5.3), Skyway needs a mark from the developer to know when to clear the buffers. Identifying shuffling phases is often simple – in many systems, a shuffling phase is implemented by a `shuffle` function and the developer can simply place a call to `shuffleStart` in the beginning of the function. Also note that, user programs written to run on Big Data systems mostly do not directly use S/D libraries and hence can benefit from Skyway without changes.

Finally, Skyway provides an interface that allows developers to easily update some object fields after the transfer, such as re-initializing some fields for semantic reasons. For example, the code snippet below updates field `timestamp` in the class `Record` with the value returned by the user-defined function `updateTimeStamp` when a `Record` object is transferred. Of course, we expect this interface to be used rarely — the need to update object data content after a transfer never occurs in our experiments.

```java
/*Register the update function*/
registerUpdate(Record.class, Record.class.getField("timeStamp"),
    SkywayFieldUpdateFunctions.getFunction(SkywayUpdate.class, "
    updateTimeStamp", "()[B");
...
class SkywayUpdate{
  /*The actual update function*/
  public byte[] updateTimeStamp(){
    return new byte[]{0};
  }
}
```

5.3 Implementation

We implemented Skyway in Oracle's production JVM OpenJDK 1.8.0 (build 25.71). In addition to implementing our object transfer technique, we have modified the classloader subsystem, the object/heap layout, and the Parallel Scavenge garbage collector, which is the default GC in OpenJDK 8. We have also provided a Skyway library for developers.

5.3.1 Global Class Numbering

Skyway develops a distributed type-registration system that automatically allows different representations of the same class on different JVM instances to share the same integer ID. This system completely eliminates the need of using strings to represent types during data transfer (as in the standard Java serializer) or the involvement of human developers to understand and register classes (as in Kryo).

Skyway type registration runs inside every JVM and maintains a type registry, which maps every type string to its unique integer ID. The driver JVM assigns IDs to all classes; it maintains a complete type registry covering all classes that have been loaded in the cluster and made known to the driver since the computation starts. Every worker JVM has a registry view, which is a subset of the type registry on the driver; it checks with the driver to obtain the ID for every class that it loads and does not yet exist in the local registry view. An example of these registries is shown in Figure 5.4.

Algorithm 5 describes the algorithms running on the driver and worker JVMs. The selection of the driver is done by the user through an API call inserted in the client code. For example, for Spark, one can naturally specify the JVM running the Spark driver as the Skyway driver, and all the Spark worker nodes run Skyway workers. Fault tolerance is provided by the application — e.g., upon a crash, Spark restarts the system on the Skyway-equipped JVMs; Skyway's driver JVM will be launched on the node that hosts Spark's driver.

At the beginning, the driver populates the registry by scanning its own loaded classes after the JVM finishes its startup logic (Lines 4 – 8). Next, the driver switches to background by running a daemon thread that listens on a port to process lookup requests from the workers (Lines 10 – 19).

Skyway uses a *pull*-based communication between the driver and workers. Upon launching a worker JVM, it first requests (Line 22) and obtains (Line 12) the current complete type registry from the driver through a "REQUEST_VIEW" message. This provides each worker JVM with a *view* of all classes loaded so far in the cluster at its startup. The rationale behind this design is that most classes that will be needed by this worker JVM are likely already registered by the driver or other workers. Hence, getting their IDs in a *batch* is much more efficient than making individual remote-fetch requests.

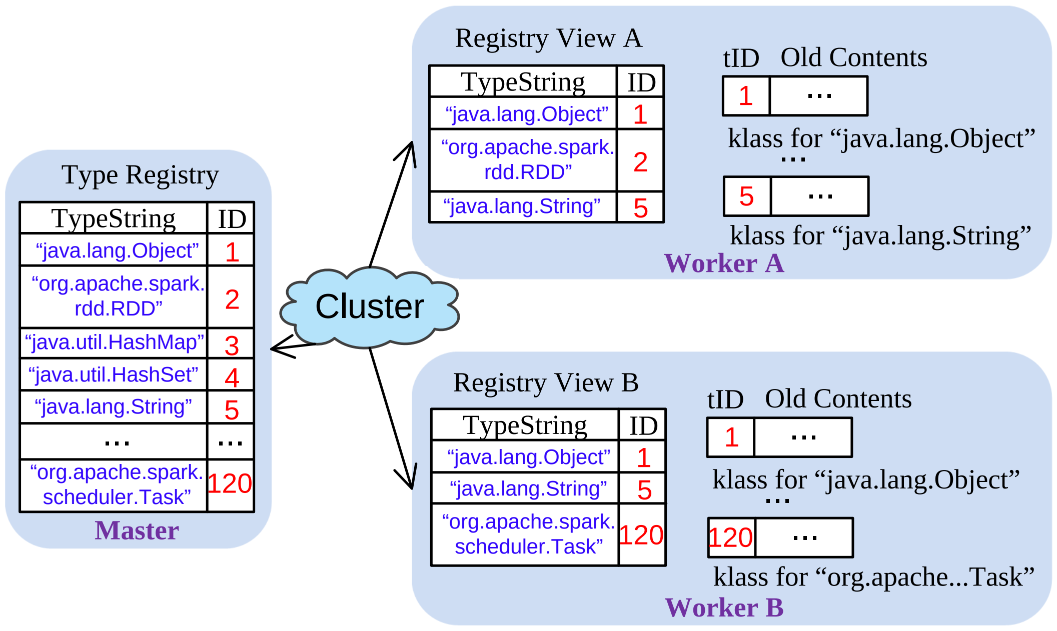


Figure 5.4: Type registries used for global class numbering.

We modify the class loader on each worker JVM so that during the loading of a class, the loader obtains the ID for the class. The loader first consults the registry view in its own JVM. If it cannot find the class, it goes on to communicate with the driver (Lines 29 – 34) by a "LOOKUP" message with the class name string. The driver returns the ID if the string exists in its own registry or creates a new ID and registers it with the class name (Line 18). Once the worker receives this ID, it updates its registry view (Line 34). Finally, the worker JVM writes this ID into the meta object of the class (Line 35). In the JVM terminology, a meta object is called a "klass" (as shown in Figure 5.4). We add an extra field in each klass to accommodate its ID.

During deserialization, if we encounter an unloaded class on the worker JVM, Skyway instructs the class loader to load the missing class since the type registry knows the full class name. While other options (e.g., low-collision hash functions such as the MD and SHA families) can achieve the same goal of assigning each class a unique ID, Skyway cannot use them as they cannot be used to recover class names.

Algorithm 5: Driver and worker algorithms for global class numbering.

```
1  /* Driver Program */
2  /*Part 1: right after the JVM starts up*/
3  JVMStartUp() /*Normal JVM startup logic*/
4  /*Initialize the type registry*/
5  $globalID \leftarrow 0$
6  $registry \leftarrow EMPTY\_MAP$
7  foreach class k loaded in the driver JVM do
8      $registry \leftarrow registry \cup \{(\text{Name}(k), globalID{+}{+})\}$
9  /*Part 2: a daemon thread that constantly listens*/
10 while Message m = ListenToWorkers() do
11     if m.type == "REQUEST_ VIEW" then
12         SendMsg(m.workerAddr, registry)
13     else if m.type == "LOOKUP" then
14         /*The content of a "LOOKUP" message from worker to driver is a class
            string*/
15         $id \leftarrow$ LookUp(registry, m.content)
16         if id == Null then
17             $id \leftarrow globalID{+}{+}$
18             $registry \leftarrow registry \cup \{(m.content, id)\}$
19         SendMsg(m.workerAddr, id)

20 /* Worker Program*/
21 /* Part 1: inside the JVM startup logic*/
22 SendMsg(driverAddr, ComposeMsg("REQUEST_VIEW", Null, myAddr))
23 Message m = ListenToDriver()
24 /*The content of a "LOOKUP" message is the registry map*/
   $registryView \leftarrow m.content$
25 /* Part 2: after the class loading routine*/
26 $clsName \leftarrow$ GetClassName()
27 $metaObj \leftarrow$ LoadClass(clsName)
28 $id \leftarrow$ LookUp(registryView, clsName)
29 if id == Null then
30     SendMsg(driverAddr, ComposeMsg("LOOKUP", clsName, myAddr))
31     Message m = ListenToDriver()
32     /*The content of a message from driver to worker is an ID*/
33     $id \leftarrow m.content$
34     $registryView \leftarrow registryView \cup \{(clsName, id)\}$
35 WriteTID(metaObj, id)
```

Comparing with the standard Java serializer that sends a type string over the network together with every *object*, Skyway sends a type string at most once for every *class* on each machine during the whole computation. Naturally, the number of strings communicated under Skyway is several orders-of-magnitude smaller. Comparing with Kryo, Skyway automatically registers all classes, and eliminates the need for developers to understand what classes will be involved in data transfer, leading to significantly reduced human effort.

5.3.2 Sending Object Graph

When `writeObject(root)` is invoked on a `SkywayObjectOutputStream` object, Skyway starts to traverse and send the object graph reachable from `root`. Algorithm 6 describes the single-threaded logic of copying the object graph reachable from a user-specified *root*, and we discuss the multi-threaded extension later in this section.

At a high level, Skyway mimics a BFS-based GC traversal. It maintains a queue `gray` holding records of every object that has been visited but not yet processed, as well as the location `addr` at which this object will be placed in the output buffer `ob`. Every iteration of the main loop (Line 8) processes the top record in `gray` and conducts three tasks.

First, based on the object-address pair (s, *addr*) retrieved from `gray`, an object s is cloned into buffer *ob* at a location calculated from *addr* (Line 10). CloneInBuffer would also adjust the format of the clone if Skyway detects that the receiver JVM has a different specification from the sender JVM, following a user-provided configuration file that specifies the object formats in different JVMs. Second, the header of the clone is updated (Lines 12 – 22). Third, for every reference-typed field f of s, Skyway pushes the referenced object o into the working queue `gray` if o has not been visited yet and then updates f with a relativized address (i.e., o's position in output buffer), which will enable a fast reference adjustment on the receiver machine (Lines 15 – 27).

As objects are copied into the buffer, which is in native memory, the buffer may be flushed (i.e., the streaming process). A flushing is triggered by an allocation at Line 10 — the allocation first checks whether the buffer still has space for the object s; if not, the buffer ob is flushed and the value of *ob.flushedBytes* is increased by the size of the buffer.

Algorithm 6: Copying the object graph reachable from object *root* and relativizing pointers for a single thread.

```
Input: Shuffling phase ID sID, a top object root, output buffer ob
1  ob.allocableAddr ← 0
2  Word w ← Read(root, OFFSET_BADDR)
3  pID ← HighestByte(w)
4  /*root has not been visited in the current phase*/
5  if pID < sID then
6      /*gray is a list of pairs of objects and their buffer addresses*/
7      gray ← {(root, ob.allocableAddr)}
8      while gray ≠ ∅ do
9          Object-Address pair (s, addr) ← RemoveTop(gray)
10         CloneInBuffer(s, ob, addr − ob.flushedBytes)
11         /*Update the clone of s in the buffer*/
12         Write(addr, OFFSET_BADDR, 0)
13         ResetMarkBits(addr)
14         Write(addr, OFFSET_KLASS, s.klass.tID)
15         foreach Reference-typed field f of s do
16             Object o ← s.f
17             if o ≠ Null then
18                 Word v ← Read(o, OFFSET_BADDR)
19                 phaseID ← HighestByte(v)
20                 if phaseID < sID then
21                     /* o has not been copied yet*/ newAddr ← ob.allocableAddr
22                     Write(o, OFFSET_BADDR, Compose(sID, newAddr))
23                     PushToQueue(gray, {(o, newAddr)})
24                     ob.allocableAddr += GetSize(o)
25                 else
26                     newAddr ← Lowest7Bytes(v)
27                 Write(addr, OFFSET(f), newAddr)
28 else
29     oldAddr ← Lowest7Bytes(w)
30     WriteBackwardReference(oldAddr)
31 SetTopMark()
```

Reference Relativization Imagine that a reference field f of an object s points to an object o. Skyway needs to adjust f in the output buffer, as o may be put at a different address on the receiver node. Skyway replaces the cloned field f with the relative address in *ob* where o will be cloned to. This will allow the receiver node to easily calculate the correct absolute value for every reference in an input buffer, once the input buffer's starting address is determined.

We first describe the overall relativization algorithm, and then discuss how Skyway addresses the three challenges caused by streaming and multi-phase data shuffling.

As shown on Lines 15 – 27 of Algorithm 6, for each reference-type field $s.f$, Skyway follows the reference to find the object (o). Skyway determines whether o has been visited in the current data-shuffling phase; details are discussed shortly. If not (Line 20), we know o will be cloned to the end of the output buffer at location $ob.allocableAddr$. We use this location to fill the `baddr` field of o (Line 22), and bump up $ob.allocableAddr$ by the size of o to keep tracking the starting address of the next cloned object in *ob*. If o has been visited (Line 26), we retrieve its location in the output buffer from the lowest seven bytes of the `baddr` field in its object header, which we will explain more later. We then update the clone of f with this buffer location *newAddr* at which the clone of o will be or has already been placed (Line 27).

The first challenge is related to streaming. When Skyway tries to update f with the output-buffer location of o's clone (f points to o), this clone may have been streamed out and no longer exists in the physical output buffer. Therefore, Skyway has to carefully store such buffer-location information, making it available throughout a data-shuffling phase. Skyway saves the buffer location in the header of the original object, not the clone, using an extra field `baddr`. The modified object layout is shown in Figure 5.5(a). When o is reached again via a reference from another object o', the `baddr` in o will be used to update the reference in the clone of o'.

The second challenge is also related to streaming. The buffer location stored in `baddr` of an object s and in its record in `gray`-queue both represent the *accumulative* bytes that have been committed to other objects in output buffer before s. However, when Skyway clones o into the buffer, it needs to account for the streaming effect that the physical buffer may have

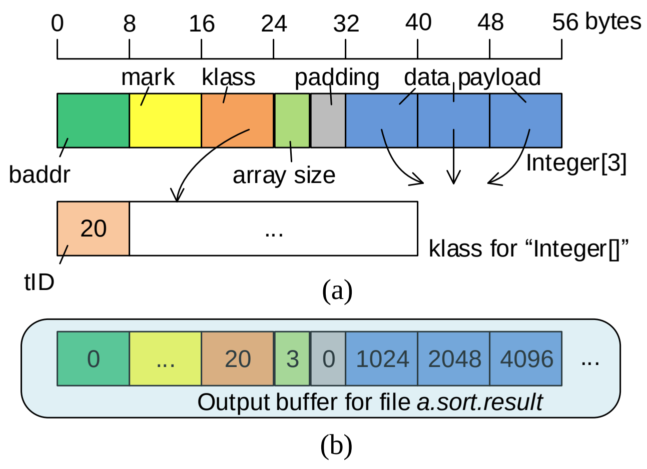


Figure 5.5: Skyway object layout in the heap (a) and an output buffer (b). This is an `Integer` array of three elements on a 64-bit HotSpot JVM. `mark` contains object locks, hash code of the object, and GC bits. `klass` points to the meta object representing the object's class. What follows is the data payload – three references to `Integer` objects. `baddr` and `tID` are both added by Skyway.

been flushed multiple times. Therefore, Skyway subtracts the number of bytes previously flushed *ob.flushedBytes* from *addr* when computing the actual address in the buffer to which *s* should be copied (Line 10).

The third challenge is due to multi-phase data shuffling. Since one object may be involved in multiple phases of shuffling, we need to separate the use of its `baddr` field for different shuffling phases. Skyway employs an *sID* to uniquely identify a shuffling phase. Whenever Skyway updates the `baddr` field, the current *sID* is written to as a prefix to the highest byte of `baddr`. Thus, Skyway can easily check whether the content in a `baddr` field is computed during the same phase of data shuffling (i.e., valid) or an earlier phase (i.e., invalid). Examples are on Lines 2 – 5 and Lines 19 – 20 of Algorithm 6. In the former case, if *root* has already been copied in the same shuffling phase (due to a copy procedure initiated by another root object), Skyway simply creates a *backward reference* pointing to its location in the buffer (Line 30). Skyway provides an API function `shuffleStart` that can be used by developers to mark a shuffling phase. *sID* is incremented when `shuffleStart` is invoked.

Header Update Lines 12 – 14 update the header of the cloned object in buffer. Following Figure 5.5, Skyway first clears the `baddr` field of the cloned object; this field will be used later to restore the object on the receiver side. Second, Skyway processes the `mark` word in the header, resetting the GC and lock bits while preserving the object hashcode. Since hashcodes are used to determine the layout of a hash-based data structure (e.g., HashMap or HashSet), reusing them on the receiver side enables the immediate reuse of the data structure layout without rehashing. Third, Skyway replaces the `klass` pointer with the type ID stored in the `klass` meta object (Line 14).

Root Object Recognition After copying all objects reachable from *root* into the buffer, we set a top mark, which is a special byte indicating the starting point of the next top-level object. The reason for setting this mark is the following. For the original implementation of `writeObject`, an invocation of the function on a top object would in turn invoke the function itself recursively on the fields of the object to serialize the referenced objects. The deserialization process is exactly a reverse process – each invocation of `readObject` in the deserialization processes the bytes written in by its corresponding invocation of `writeObject` in serialization. However, Skyway's implementation of `writeObject` works in a different way — one invocation of the function on a top object triggers a system-level graph traversal that finds and copies all of its reachable objects. Similarly, Skyway's `readObject` also reads one object from the byte sequence instead of recursively reading out all reachable objects.

Although on the receiver side we can still compute all reachable objects for a root, this computation also needs a graph traversal and is time-consuming. As an optimization, we let the sender explicitly mark the root objects so that the receiver-side computation can be avoided. This is achieved by top marks. With these top marks, Skyway can easily skip the lower-level objects in the middle and find the next top object. Note that this treatment does not affect the semantics of the program — all data structures reachable from top objects are recovered by the system, not by the application APIs.

Support for Threads Algorithm 6 does not work in cases that multiple threads on one node try to transfer the same object concurrently (i.e., shared objects). Since each data-transfer thread has its own output buffer and the `baddr` field of a shared object can only store the relative buffer address for one thread t at a time, when other threads visit the object later, they would mistakenly use this address that is specific to t. To solve the problem, we let the lower seven bytes of `baddr` store both stream/thread ID (with the two highest bytes) and relative address (with the five lowest bytes).

When an object is first visited by t, t's thread ID is written into `baddr` together with the address specific to t's buffer. When the object is visited again, Skyway first checks whether the ID of the visiting thread matches the thread ID stored in its `baddr`. If it does, `baddr` of the object is used; otherwise, Skyway switches to a *hash table-based* approach – each thread maintains a thread-local hash table; the object and its buffer address for the thread are added into the hash table as a key and a value. Compare-and-swap (CAS) is used to provide thread safety when updating each `baddr`.

This approach prevents a thread from mistakenly using the object's buffer address for another thread. An object will have distinct copies in multiple output buffers when visited by different threads; these copies will become separate objects after delivered to a remote node. This semantics is consistent with that of the existing serializers.

5.3.3 Receiving Object Graph

With the careful design on sending, the receiving logic is much simpler. To receive objects from a sender, the receiver JVM first prepares an input buffer, whose size is user-tunable, for the sender in its managed heap to store the transferred objects. A subtle issue here is that a sender node may use multiple streams (in multiple threads) to send data to the same receiver node simultaneously. To avoid race conditions, the receiver node creates an input buffer for each stream of each sender so that different streams/threads can transfer data without synchronizations. We create oversized buffers to fit large objects whose size exceeds the size of a regular buffer.

After the input buffer is filled, Skyway performs a linear scan of the buffer to absolutize types and pointers. For the `klass` field of each object, Skyway queries the local registry view to get the correct `klass` pointer based on the type ID and writes the pointer into the field. For a relative address a stored in a reference field, Skyway replaces it with $a + s$ where s is the starting address of this input buffer.

There is one challenge related to streaming. Since Skyway may not know the total size of the incoming data while allocating the buffer, one buffer of a fixed length may not be large enough. Skyway solves this by supporting linked chunks – a new chunk can be created and linked to the old chunk when the old one runs out of space. Skyway does not allow an object to span multiple chunks for efficiency. Furthermore, when a buffer contains multiple chunks, the address translation discussed above needs to be changed. We first need to calculate which chunk i a relative address a would fall in. Then, because previous chunks might not be fully filled, we need to calculate the *offset* of a in the i-th chunk. Suppose s_i is the starting address of chunk i and hence, $s_i + \mathit{offset}$ is the final absolute address for a. This address will be used to replace a in each pointer field.

As each input buffer corresponds to a distinct sender, we can safely start the computation to process objects in each buffer for which streaming is finished. This would not create safety issues because objects that come from different nodes cannot reference each other. However, we do need to block the computation on buffers into which data is being streamed until the absolutization pass is done.

Interaction with GC After receiving the objects, it is important for the Skyway client on the receiver JVM to make these objects reachable in the garbage collection. Skyway allocates all input buffers in the old generation (tenured) of the managed heap. In Skyway, we use the Parallel Scavenge GC (i.e., the default GC in OpenJDK 8), which employs a *card table* that groups objects into fixed-sized buckets and tracks which buckets contain objects with young pointers. Therefore, we add support in Skyway that updates the card table appropriately to represent new pointers generated from each data transfer.

5.4 Evaluation

To thoroughly evaluate Skyway, we have conducted three sets of experiments, one on a widely-used suite of benchmarks and the other two on widely-deployed systems Apache Spark [182] and Apache Flink [37]. The first set of experiments focuses on comparing Skyway with all existing S/D libraries — since most of these libraries **cannot** be directly plugged into a real system, we used the Java serializer benchmark set (JSBS) [153], which was designed specifically to evaluate Java/Scala serializers, to understand where Skyway stands among existing S/D libraries. JSBS was initially designed to assess single-machine S/D. We modified this program to make it work in a distributed setting; details are discussed shortly.

In the second and third set of experiments, we modified the Spark and Flink code to replace the use of Kryo and the Java serializer (in Spark) and built-in serializers (in Flink) with Skyway in order to assess the benefit of Skyway to real-world distributed systems. All of our experiments were run on a cluster with 11 nodes, each with 2 Xeon(R) CPU E5-2640 v3 processors, 32GB memory, 1 100GB SSD, running CentOS 6.8 and connected by a 1000Mb/s Ethernet. Each node ran 8 job instances. The JVM on each node was configured to have a 30GB heap.

5.4.1 Java Serializer Benchmark Set

The JSBS contains several workloads under which each serializer and deserializer is repeatedly executed. Each workload contains several media content objects which consist of primitive `int` and `long` fields as well as reference-type fields. The driver program creates millions of such objects, each of which is around 1KB in JSON format. These objects are serialized into in-memory byte arrays, which are then deserialized back to heap objects. To understand the cost of transferring the byte sequences generated by different serializers, we modified the benchmark, turning it into a distributed program — each node serializes these objects, broadcasts the generated bytes to all the other nodes, and deserializes the received bytes back into objects. To execute this program, we involved five nodes and executed this process 1000 times repeatedly. The average S/D time for each object and the network cost are reported.

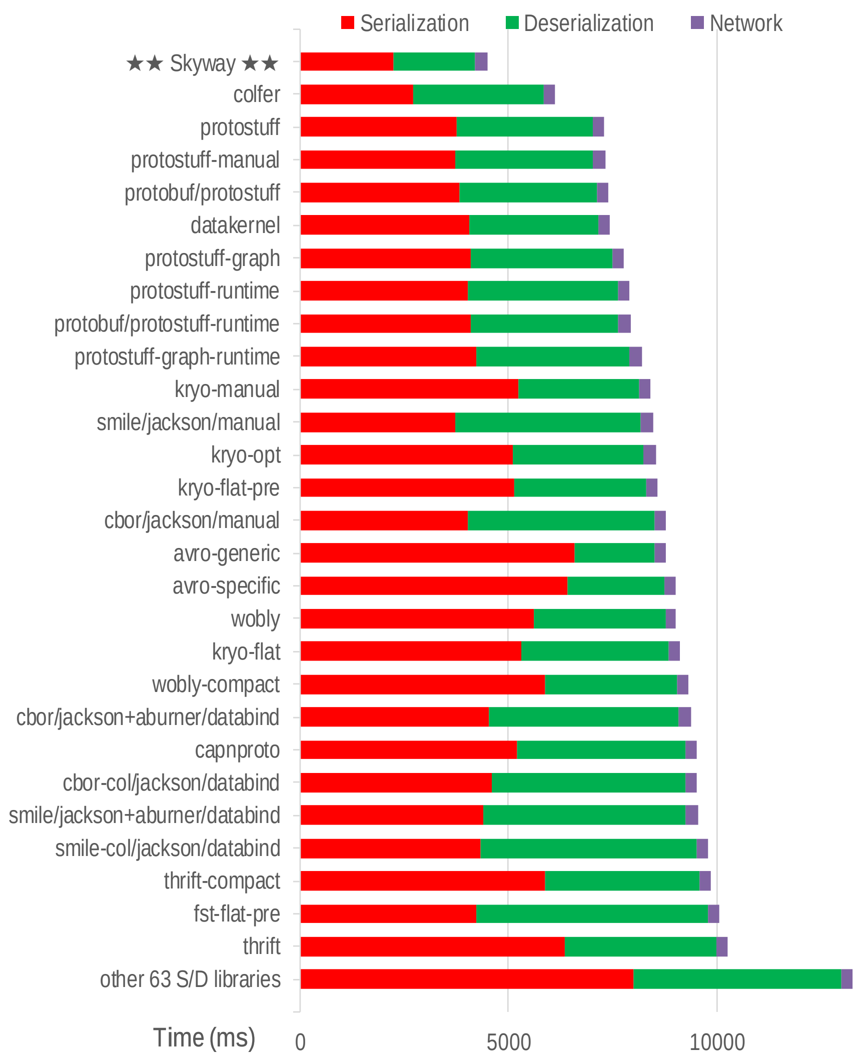


Figure 5.6: Serialization, deserialization, and network performance of different S/D libraries. Although we have compared Skyway with 90 existing libraries, we include in this figure Skyway and 27 fastest-performing libraries. The last bar is simply a placeholder of the 63 libraries that perform slowly.

We have compared Skyway exhaustively with 90 existing S/D libraries. For simplicity, we excluded from the figure 63 slower libraries whose total S/D time exceeds 10 seconds. The performance of the fastest 28 libraries is shown in Figure 5.6. Skyway, without needing any user-defined S/D functions, is the fastest of all of them. For example, it is 2.2× faster than

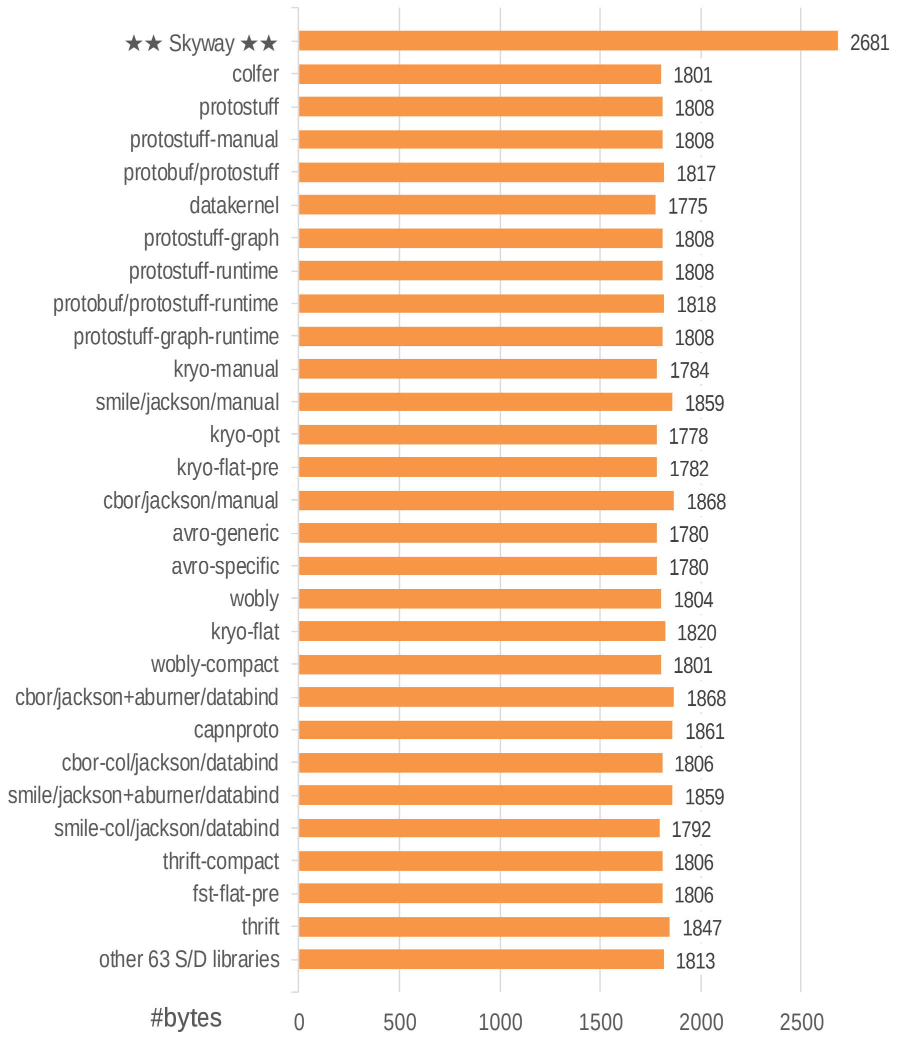


Figure 5.7: Size of serialized object by different libraries.

Kryo-manual, which requires manual development of S/D functions. It is more than 67× faster than the Java serializer, which is also not shown in the figure.

Colfer [63] is the only serializer whose performance is close to (but still 1.5× slower than) that of Skyway. It employs a compiler colf(1) to generate serialization source code from schema definitions to marshal and unmarshal data structures. Hence, the use of colf(1) requires user-defined schema of data formats, which, again, creates a practicality obstacle if data structures are complicated and understanding their layouts is a daunting task.

Skyway's faster S/D speed is achieved at the cost of greater numbers of bytes serialized. Figure 5.7 shows the number of bytes genereated by each S/D library in the JSBS. On average, Skyway generates 50% more bytes than existing serializers. However, as shown earlier in Figure 5.6, the increased data amount does not cause the network cost to change much even on a 1000Mb/s Ethernet, whereas the computation cost in S/D is significantly reduced, beyond 20%.

5.4.2 Improving Apache Spark with Skyway

Experience We have modified Spark version 2.1.0 (released December 2016) to replace the use of Kryo-manual with the Skyway library. Spark was executed under Hadoop version 2.6.5 and Scala version 2.11. Our experience shows that the library replacement was rather straightforward — to use Skyway, we created a Skyway serializer that wraps the existing Input/OutputStream with our `SkywayInput/OuputStream` objects. We modified the Spark configuration (*spark.serializer*) to invoke the Skyway serializer instead of Kryo. Since data serialization in Spark shuffles orders of magnitude more data than closure serialization, we only used Skyway for data serialization. The Java serializer was still used for closure serialization. The entire `SkywaySerializer` class contains less than 100 lines of code, most of which was adapted directly from the existing `JavaSerializer` class. The number of lines of new code we wrote ourselves was only 10: 2 lines to wrap the I/O stream parameters, 3 lines to modify calls to `readObject`, and 5 lines to specify tuning parameters (e.g., buffer size).

Programs and Datasets We ran Spark with four representative programs: Word Count (WC), Page Rank (PR), Connected Components (CC), and Triangle Counting (TC). Word Count is a simple MapReduce application that needs only one round of data shuffling. The other three programs are iterative graph applications that need to shuffle data in each iteration. We used four real-world graphs as input – Live Journal (LJ) [42], Orkut (OR) [26], UK-2005 (UK) [50], and Twitter-2010 (TW) [107]; Table 5.1 lists their details.

For PR over Twitter-2010, Spark could not reach convergence in a reasonable amount of time (i.e., 10 hours) for all configurations. We had to terminate Spark at the end of the 10th iteration and thus the performance we report is w.r.t. the first 10 iterations. All the other iterative applications ran to complete convergence. We have experimented with three serializers: the Java serializer, Kryo, and Skyway.

Graphs	**#Edges**	**#Vertices**	**Description**
LiveJournal [42]	69M	4.8M	Social network
Orkut [26]	117M	3.0M	Social network
UK-2005 [50]	936M	39.5M	Web graph
Twitter-2010[107]	1.5B	41.6M	Social network

Table 5.1: Graph inputs for Spark.

Spark Performance Figure 5.8 reports the running time comparisons among three serializers over the four input graphs. Since different programs have very different performance numbers, we plot them separately on different scales. For each dataset, WC and CC finished much more quickly than PR and TC. This is primarily due to the nature of the application — WC has one single iteration and one single round of shuffling; it is much easier for CC (i.e., a label propagation application, which finishes in 3-5 iterations) to reach convergence than the other two applications that often need many more iterations.

It is the same reason that explains why Skyway performs better for PR and TC — since they perform many rounds of data shuffling, a large portion of their execution time is taken by S/D and thus the savings in data transfer achieved by Skyway are much more significant for these two applications than the other two.

A detailed summary of each run-time component is provided in Table 5.2. Network time is included in **Read**. On average, Skyway makes Spark run 36% and 16% faster than the Java serializer and Kryo. Compared to the Java serializer, Kryo achieves most of its savings from avoiding reading/writing type strings since Kryo relies on developers to register classes. As a result, the I/O in network and local reads has been significantly reduced. Skyway, on the contrary, benefits most from the reduced deserialization cost. Since the transferred objects can be immediately used, the process of recreating millions of objects and calling

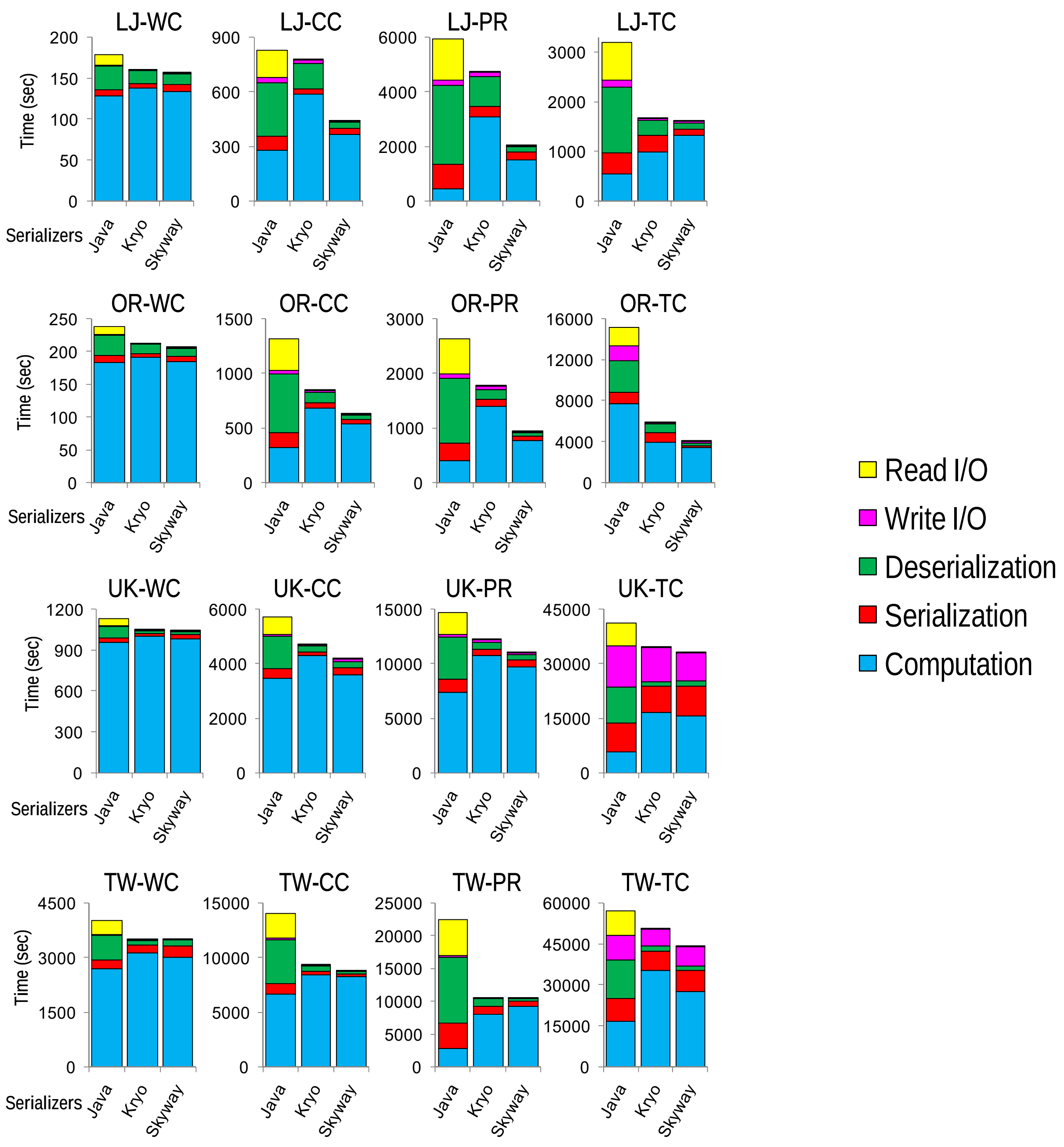


Figure 5.8: Spark performance with Java serializer, Kryo, and Skyway.

their constructors is completely eliminated. Furthermore, it is worth noting, again, that Kryo achieves its benefit via heavyweight manual development — there is a package of more than 20 classes (several thousands of lines of code) in Spark developed to use Kryo, while Skyway completely eliminates this manual burden and simultaneously achieves even higher performance gains.

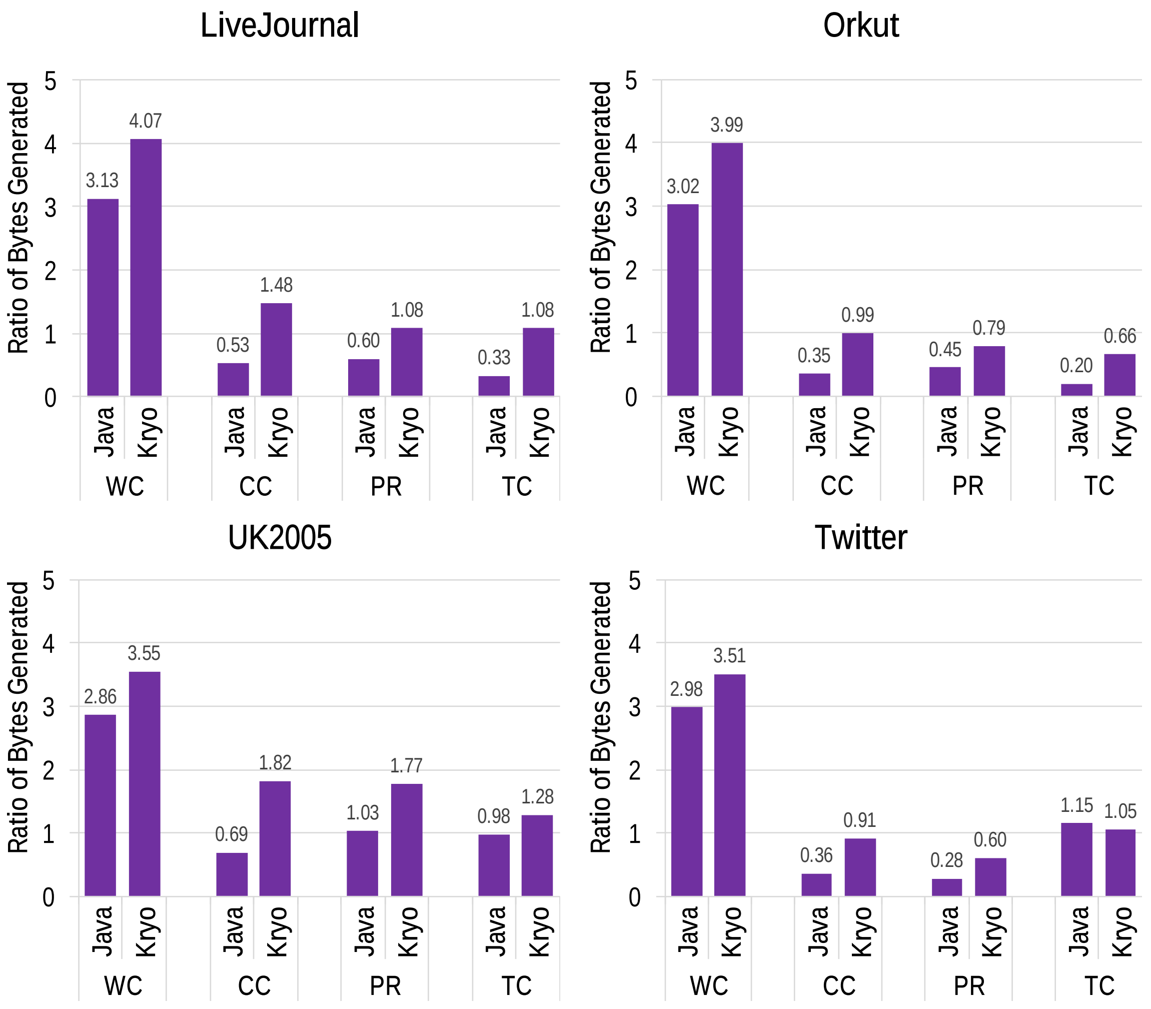


Figure 5.9: Ratio between the number of bytes generated by Skyway and the number of bytes generated by Java serializer and Kryo.

System	Overall	Ser	Write	Des	Read	Size
Kryo	0.39 ~ 0.94 (0.76)	0.33 ~ 0.89 (0.59)	0.12 ~ 0.83 (0.61)	0.11 ~ 0.55 (0.26)	0.01 ~ 0.03 (0.02)	0.31 ~ 1.09 (0.52)
Skyway	0.27 ~ 0.92 (0.64)	0.19 ~ 1.29 (0.62)	0.12 ~ 1.61 (0.97)	0.04 ~ 0.43 (0.16)	0.01 ~ 0.05 (0.02)	0.20 ~ 3.13 (1.18)

Table 5.2: Performance summary of Skyway and Kryo on Spark: normalized to baseline (Java serializer) in terms of **Overall** running time, **Ser**ialization time, **Write** I/O time, and **Des**erialization time, **Read** I/O time (including the network cost), and the **Size** of byte sequence generated. A lower value indicates better performance. Each cell shows a percentage range and its geometric mean.

Because the number of bytes generated in the benchmark suite varies dramatically from one program as well as and from one input dataset to another (e.g., from dozens GBs to thousands GBs), we plot the ratio between the number of bytes generated by Skyway and that of Java serializer and Kryo instead in Figure 5.9. On average, the number of bytes transferred under Skyway is about the same as the Java serializer, but 77% more than Kryo due to the transferring of the entirety of each object. The increased data size is also reflected in the increased write I/O. Skyway's read I/O time is shorter than that of the Java serializer. This is primarily due to the elimination of object creation — we only need one single scan of each buffer instead of reading in individual bytes to create objects as done in Kryo. Skyway's read I/O is longer than that of Kryo because Kryo transfers much less bytes.

Header optimization To understand what constitutes the extra bytes produced by Skyway, we analyzed these bytes for our Spark applications. Our results show that, on average, object headers take 51%, object paddings take 34%, and the remaining 15% are taken by pointers. Since headers and paddings dominate these extra bytes, we have implemented an aggressive space optimization — during serialization, we take off the header (except the type ID and the hashcode) and padding of each object and only copy its payload into the buffer. The receiver node cannot directly use data upon receiving them – we need one extra copy for each object to add its header as well as some computation to calculate the size of the padding needed for the object. Compared to Skyway, this space optimization reduces the amount of serialized data by 40%, and thus reduce the read and write time as well. However this is at the cost of much increased deserialization time (by 20%). However, since the time spent on read/write I/O is much smaller than the S/D time, this optimization actually causes the overall execution time to increase by 7%.

Memory Overhead To understand the overhead of the extra word field *baddr* in each object header, we ran the Spark programs with the unmodified HotSpot and compared peak heap consumption with that of Skyway (by periodically running `pmap`). We found that the difference (i.e., the overhead) is relatively small. Across our four programs, this overhead varies from 2.1% to 21.8%, with an average of 15.4%.

5.4.3 Improving Apache Flink with Skyway

We evaluated Skyway with the latest version of Flink (1.3.2, released August 2017) executing under Hadoop version 2.6.5. Flink has both streaming and batch processing models. Here we focus on the batch-processing model, and particularly, query answering applications.

Flink reads input data into a set of tuples (e.g., rows in relational database); the type of each field in a tuple must be known at compile time. Flink can thus select a built-in serializer for each field to use when creating tuples from the input. Flink falls back to the Kryo serializer when encountering a type with neither a Flink-customized nor a user-defined serializer available. Since the read/write interface is clearly defined, we could easily integrate Skyway into Flink.

We used the TPC-H [163] data generator to generate a 100GB dataset as our input. Next, we transformed 5 representative SQL queries generated by TPC-H into Flink applications. The description of these queries can be found in Table 5.3. They were selected due to the diverse operations they perform and database tables they access.

	Description
QA	Report pricing details for all items shipped within the last 120 days.
QB	List the minimum cost supplier for each region for each item in the database.
QC	Retrieve the shipping priority and potential revenue of all pending orders.
QD	Count the number of late orders in each quarter of a given year.
QE	Report all items returned by customers sorted by the lost revenue.

Table 5.3: Descriptions of the queries used in Flink.

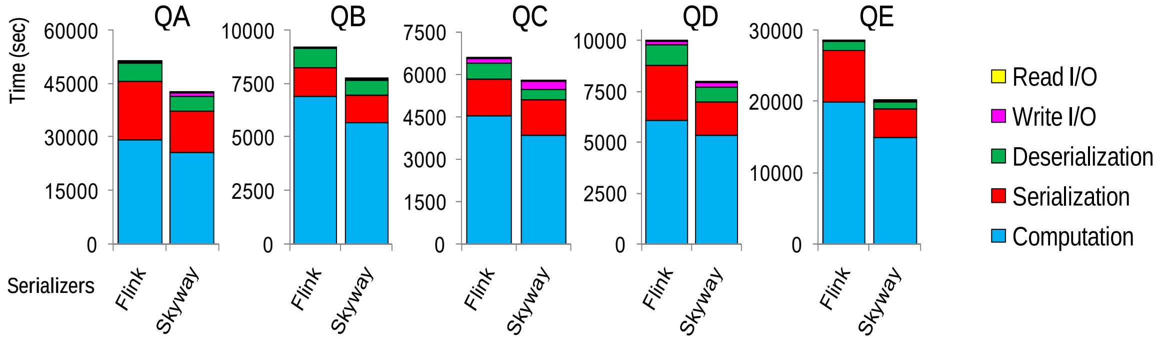


Figure 5.10: Flink performance with Flink's built-in serializer and Skyway.

Overall	Serialization	Write	Deserialization	Read	Size
$0.71 \sim 0.88$	$0.56 \sim 1.06$	$0.51 \sim 1.76$	$0.58 \sim 0.82$	$0.49 \sim 1.13$	$1.23 \sim 2.03$
(0.81)	(0.77)	(0.96)	(0.75)	(0.61)	(1.68)

Table 5.4: Performance improvement summary of Skyway on Flink: normalized to Flink's built-in serializer. Each cell shows a percentage range and its geometric mean. A lower value indicates better performance.

Figure 5.10 shows Flink's performance improvement using Skyway. Performance summary is also shown in Table 5.4. In Flink, the amount of time in deserialization (8.7%) is much less than that in serialization (23.5% on average). This is because Flink does not deserialize all fields of a row upon receiving it — only those involved in the transformation are deserialized. Despite this lazy mechanism, Skyway could improve Flink's performance by, an overall of 19%, compared to Flink's built-in serializer. The total number of bytes written by Skyway is also higher than the baseline — on average, Skyway emits 68% more bytes. It is worth noting that Skyway is compared with Flink's highly optimized built-in serializer; it is statically chosen and optimized specifically for the data types involved in the queries, and has been shown to outperform generic serializers such as Kryo.

5.5 Summary

This chapter presents Skyway, a 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 provides performance benefits to any JVM-based system by *completely eliminating* the need (1) of invoking serialization/deserialization functions, thus saving CPU time, and (2) of requiring developers to hand-write serialization functions.

Skyway addresses the inefficiencies in data transfer much more efficiently than all the existing libraries in three aspects as follows.

- First, Skyway, by changing the JVM, transfers every object as a whole, completely eliminates the need of accessing individual data fields. Furthermore, since the hashcode of an object is cached in the header of the object, transferring the entirety of each object preserves the original hashcode of the object, so that hash-based data structures can be used on the receiver node without rehashing—a process that takes a great amount of time in traditional serialization/deserialization.

- Second, Skyway represents types by employing an automated global type-numbering procedure — the master node maintains a registry of all types and their IDs, and each worker node communicates with the master to obtain IDs for its classes upon class loading. This process enables all classes across the cluster to be globally numbered without any developer intervention and thus each ID can be used to uniquely identify the same class on different nodes.
- Third, Skyway employs an efficient "relativization" technique to adjust references. As objects are copied into the output buffer, pointers stored in them are relativized in linear time—they are changed from *absolute addresses* to *relative addresses*. Upon receiving the buffer, the Skyway client on the receiver node performs another linear scan of the input buffer to *absolutize* the relative information in the buffer.

Skyway may push more bytes over the network than other serialization/deserialization libraries, because it transfers the entirety of each object yet others do not transfer object headers. However, much evidence [142, 180] shows that bottlenecks in real systems are shifting from I/O to computing, and hence, we believe this design strikes the right design tradeoff—the savings on the computation cost significantly outweigh the extra network I/O cost incurred by the extra bytes transferred on a modern network.

It is important to note that Skyway is *not* a general-purpose serializer. Our insight why Skyway would work well for Big Data processing is two-fold. First, data processing applications frequently shuffle many millions of objects and do so in strongly delimited phases. Hence, sending objects *in batch* without changing their formats provides significant execution efficiency—it not only reduces the format-changing efforts, but also enables computation on the receiver node to be performed simultaneously with objects being streamed in. Second, the use of modern network technology enables extra bytes to be quickly transferred without incurring much overhead.

CHAPTER 6

Related Work

6.1 Software Bloat Analysis

Bloat exists widely in object-oriented Big Data systems and incurs a severe performance penalty when large amounts of data are processed. Software bloat analysis [33, 125–128, 150, 169, 171–176] attempts to find, remove, and prevent performance problems due to inefficiencies in the code execution and the use of memory. Prior work [127, 128] proposes metrics to provide a performance assessment of the use of data structures. Their observation that a large portion of the heap is not used to store data is also confirmed in our study. Our work proposes optimizations specifically targeting the problems we found and our experimental results show that these optimizations are very effective.

Work by Dufour et al. [73] uses a blended escape analysis to characterize and find excessive use of temporary data structures. By approximating object lifetimes, the analysis has been shown to be useful in classifying the usage of newly created objects in the problematic areas. Work by Xu et al. [172] detects memory bloat by profiling copy activities, and their later work [171, 175] looks for high-cost-low-benefit data structures to detect execution bloat. Facade performs effective optimizations to remove bloat in data-intensive environments.

Dufour et al. propose dynamic metrics for Java [71], which provide insights by quantifying runtime bloat. Mitchell et al. [128] analyzes structure behavior according to the flow of information, though using a manual technique. Their aim is to allow programmers to place judgments on whether certain classes of computations are excessive. Their follow-up work [127] introduces a way to find data structures that consume excessive amounts of memory. Work by Dufour et al. finds excessive use of temporary data structures [72, 73] and summarizes the shape of these structures. They employ a blended escape analysis, which applies static analysis to a region of dynamically collected calling structure with observed performance problem. By approximating object effective lifetimes, the analysis has been shown to be useful in classifying the usage of newly created objects in the problematic program region.

JOLT [150] is a VM-based tool that uses a new metric to quantify object churn and identify regions that make heavy use of temporary objects, in order to guide aggressive method inlining. Jin et al. [101] studies performance bugs in real-world software systems. These bugs are analyzed to extract efficiency rules, which are then applied to detect problems in other applications. Nistor et al. [138] detects performance problems using similar memory-access patterns. Object Equality Profiling [123] is a run-time technique that discovers opportunities for replacing a set of equivalent object instances with a single representative object to save space. Xu [169] proposes a technique to find reusable data structures.

These work, unfortunately, cannot scale to large data-intensive systems that have a large codebases or a large heap due to the well-known limitation of static and dynamic analyses. Our work shares the same goal of removing system bloat but is designed with scalability and practicality in minds to work with large-scale, distributed frameworks that have to process massive amounts of data.

6.2 Reducing the number of objects via program analysis

Traditional optimization techniques for object-oriented programs use various approaches to reduce the number of heap objects and their management costs, ranging from programming guidelines [80] through static program analyses [49, 61, 68, 70, 83, 84, 90, 114, 152, 166, 167, 176] to low-level systems support [170].

Object inlining [70, 114] is a technique that statically inlines objects in a data structure into its root to reduce the number of pointers and headers. Free-Me [90] adds compiler-inserted frees to a GC-based system. Pool-based allocation proposed by Lattner et al. [109–111] uses a context-sensitive pointer analysis to identify objects that belong to a logical data structure and allocate them into the same pool to improve locality. Design patterns [80] such as `Singleton` and `FlyWeight` aim to reuse objects. However, these techniques have limited usefulness in Big Data systems that exhibit strong epochal behavior. Even if we can reuse data objects across iterations, the number of heap objects in each iteration is not reduced and these objects still need to be traversed frequently by the GC.

Shuf et al. [152] propose a static technique that splits objects into *prolific types*—types that have large numbers of instances— and non-profilic types, to enable aggressive optimizations and fast garbage collection. Objects with prolific types are allocated in a prolific region, which is frequently scanned by GC (analogous to to a nursery in a generation collector); objects with non-prolific types are allocated in a regular region, which is less frequently scanned (analogous to an old generation). The insight is that the instances of prolific types are usually temporary and short-lived. Facade and Yak both are motivated by a completely opposite observation: data classes have great numbers of objects, which are often long-lived; frequently scanning those objects can create prohibitively high GC overhead. Hence, in Facade we allocate data records in native memory without creating objects to represent them, and in Yak data objects are allocated in data space that is non-GCed. Moreover, Facade adopts a new execution model and does not require any profiling.

Facade shares similarity with object pooling which is a well-known technique for reducing the number of objects by reusing a set of objects. For example, Java 7 supports the use of thread pools to save thread instances. Our facade pool differs from traditional object pooling in three important aspects. First, while they have the same goal of reducing objects, they achieve the goal in completely different ways: Facade moves data objects out of the heap to native memory while object pooling recycles and reuses instances after they are no longer used by the program. Second, the facade pool has a bound; we provide a guarantee that the number of objects in the pool will not exceed the bound. On the contrary, object pooling does not provide any bound guarantee. In fact, it will hurt performance if most of the objects from the pool cannot be reused, because the pool will keep growing and consume a lot of memory. Finally, retrieving/returning facades from/to the pool is automatically done by the compiler while object pooling depends on the developer's insight—the developer has to know what objects have disjoint lifetimes and write code explicitly to recycle them.

6.3 Region-based Memory Management

The epochal behavior of the data path which we call the *epochal hypothesis* is not a new concept. This hypothesis has been leveraged in *region-based memory management*. Region-based memory management was first used in the implementations of functional languages [31, 162] such as Standard ML [91], and then was extended to Prolog [120], C [81, 82, 88, 96], Java [60, 146], as well as real-time Java [45, 53, 105]. In these, objects created in an *epoch* are allocated in a memory region and efficiently deallocated as a whole when the epoch ends.

One drawback of these work is that they focus on the detection of region-allocatable objects, assuming that a new programming model will be used to allocate them and there already exists a modified runtime system (e.g., a new JVM) that supports region-based allocation, limiting their usefulness and practicality. On the contrary, Facade is a non-intrusive technique that compiles the program and allocates objects based on an existing JVM, without needing developers to write new programs as well as any JVM modification. Yak, while results in a new JVM, does not require developers to use a new API for object allocation and deallocation.

Furthermore, existing region-based techniques need either sophisticated static analyses [31, 45, 53, 68, 81, 82, 86, 88, 154, 159]. Specifically, these analyses examine the whole program to identify region-allocable objects, which cannot scale to Big Data systems that all have large codebases and thus, cannot scale to Big Data systems that all have large code bases. Or they require heavy human intervention: developers are required to use a brand new programming model, such as region types [45, 53], or heavy manual refactoring [86, 135], to guarantee that objects created in an epoch are indeed unreachable at the end of the epoch. Our work does not rely on any heavy interprocedural analyses exactly because of that. Yak is a pure dynamic technique that easily scales to large systems and requires only straightforward marking of epochs from users. The design of Yak frees developers from the burden of understanding object lifetimes to use regions.

6.4 Garbage Collection

Yak belongs to the extensive family of tracing garbage collectors [40, 43, 44, 47, 48, 48, 52, 59, 62, 67, 78, 98, 104, 124, 148, 155, 156]. At first glance, Yak is similar to generational GC in that it promotes objects reachable after an epoch and then frees the entire epoch region. However, the regions in Yak have completely different and much richer semantics than the two generations in a generational GC. Consequently, Yak encounters completely different challenges and uses a design that is different from a generational GC. Specifically, in Yak, regions are thread-private; they reflect nested epochs; many regions could exist at any single moment. Therefore, to efficiently check which objects are escaping, we cannot rely on a traditional tracing algorithm; escaping objects may have multiple destination regions, instead of just the single old generation.

Connectivity-based garbage collection (CBGC) [97] is a family of algorithms that place objects into disjoint partitions by performing conservative, static connectivity analyses on the object graph. A connectivity analysis can be based on types, allocations, or the partitioning introduced by Harris [93]. HiStar [183] organizes space usage into a hierarchy of containers with quotas. Any object not reachable from the root container is garbage collected. Garbage First (G1) [67] is a generational algorithm that divides the heap into many fixed-size small regions and gives higher collection priority to regions with more garbage. N-Generational Garbage Collector (NG2C) [54] is an extension of the G1 collector. It extends the 2-generation heap layout with a young and an old generation into an N-generation heap layout. While CBGC, G1, NG2C, and Yak each uses a notion of *region*, each has completely different semantics for the region and hence a different design. For example, objects inside a G1 region are *not* expected to have lifespans that are similar to each other.

NumaGiC [85] is a new GC for "Big Data" on NUMA machines. It considers data location when performing allocation and collection. However, as a generational GC, NumaGiC shares with modern GCs the problem of unnecessarily traverse a large volume of data objects in the heap, which could inevitably cause long pauses during the execution.

Another orthogonal line of research on reducing GC pauses is building a holistic runtime for distributed Big Data systems [118, 119]. The runtime collectively manages the heap on different nodes, coordinating GC pauses to make them occur at times that are convenient for applications in a centralized manner. Different from these techniques, Yak focuses on optimizing memory management efficiency on each node.

Zing [5] is a high-performance Java runtime developed by Azul for large heaps. It uses the Continuously Concurrent Compacting Collector (C4) [158] that aims to reduce latency induced by garbage collection in real-time analytics — it does not need to stop the mutators for space compaction. However, it is still a generational algorithm and needs to scan most of the heap in each GC run. Yak aims for batch-processing programs and thus, was optimized for throughput. Furthermore, stop-the-word period for region deallocation is very short.

6.5 Effort in programming languages community

Most of the existing efforts for language development focus on providing support for data representation (such as the PADS project [79, 122] or [41, 94]). Expanded types in Eiffel and value types in C# are used to declare data with simple structures. Value types can be stack allocated or inlined into heap objects. While using value types to represent data items appears to be a promising idea, its effectiveness is actually rather limited. For example, if data items are stack allocated, they have limited scope and cannot easily flow across multiple functions. One the other hand, always inlining data items into heap objects can significantly increase memory consumption, especially when a data structure grows (e.g., resizing of a hash map) and two copies of the data structure are needed simultaneously. Moreover, these data items are no longer amenable to iteration-based memory management—they cannot be released until their owner objects are reclaimed, leading to significant memory inefficiencies.

Yu et al. propose a programming model [178] for distributed aggregation for Big Data systems. A number of high-level declarative languages for data-intensive computation have been proposed in the last decade, including Sawzall [144], Pig Latin [141], SCOPE [56], Hive [160], and DryadLINQ [179]. Rust [2] is a systems programming language designed by Mozilla that advocates for the elimination of garbage collection. Rust enforces ownership propagation through new syntax and type system to prevent memory issues.

Our work is at a lower (i.e., system) level of the computing stack, and thus, complements these work. For example, Skyway can optimize programs written in Rust to reduce data transfer cost.

6.6 Memory Management Optimizations in Big Data Systems

A variety of data computation models and processing systems have been developed in the past decade [35, 51, 56, 64, 65, 100, 141, 144, 160, 177–179, 182]. MapReduce [66] has inspired much research on distributed data-parallel computation, including Hyracks [51], Hadoop [35], Spark [182], and Dryad [100]. It has been extended [177] with Merge to support joins and adapted [64] to support pipelining.

There exists also a large body of work on optimizing data-intensive applications, but focus on domain-specific optimizations, including, for example, data pipeline optimizations [30, 51, 56, 57, 89, 100, 108, 140, 179, 184] , query optimizations [64, 69, 112, 129, 139, 141], and Map-Reduce-related optimizations [29, 66, 116, 144, 160, 161, 177].

Cascading [6] is a Java library built on top of Hadoop. It provides abstractions for developers to explicitly construct a dataflow graph to ease the challenge of programming data-parallel tasks. Similarly to Cascading, FlumeJava [57] is another Java library that provides a set of immutable parallel collections. These collections present a uniform abstraction over different data representations and execution strategies for MapReduce. StarFish [95] is a self-tuning framework for Hadoop that provides multiple levels of tuning support. At the heart of the framework is a Just-In-Time optimizer that profiles Hadoop jobs and adaptively adjusts various framework parameters and resource allocation.

These frameworks are all developed in managed languages and perform their computations on top of the managed runtime. Hence, these systems can benefit immediately all of our work. Facade and Yak can reduce memory management cost both in term of space overhead and garbage collection effort. Skyway significantly speeds up data shuffling tasks.

Recently, there has been much interest in optimizing memory management in language runtimes for efficient data processing [55, 77, 86, 118, 119]. These works are largely orthogonal to all of our work. Although Yak and Skyway also fit in the category of language runtime optimizations. ITask [77] provides a library-based programming model for developing interruptible tasks in data-parallel systems. ITask solves the memory management problem using an orthogonal approach that interrupts tasks and dumps live data to disk. Tungsten [22] is an effort under the Spark [182] umbrella that aims to explicitly manage memory and eliminate the overhead of the JVM object model and GC.

Bu et al. studied several data processing systems [55] and showed that a “bloat-free” design (i.e., no objects allowed in data processing units), which can make the system orders of magnitude more scalable. This insight has inspired work, like Broom [86], lifetime-based memory management [117], as well as our work of Facade and Yak. Broom aims to replace the GC system by using regions with different scopes to manipulate objects with similar lifetimes. Broom ask developers to explicitly use Broom APIs to allocate objects in regions. Deca [117] is very similar to Facade. It is a conservative transformation technique that tries to group and inline objects with same expected lifetime into large byte arrays. Deca also requires heavyweight code refactoring from the user. After our work, most recently, Kedia et al. [103] and project Snowflake [143] propose safe manual memory management for C# by adding a delete operator to free memory explicitly. An exception is thrown if the program dereferences a pointer to freed memory. Flare [76] transforms Spark programs for moderate-sized workload using C memory layout and C-like semantics.

While promising, most the work requires extensive programmer intervention, much negated the benefits of managed languages. Users of Facade must annotate the code and determine “data classes” and “boundary classes” to use the compiler. Despite the compiler already does a lot of heavy lifting behind the scene to make the transformation easy, user effort is considerable. Comparing to those, Yak is much more user-friendly. It was designed to free developers from the burden of understanding object lifetimes to use regions, making region-based memory management part of the managed runtime by requiring only the identification of epoch starts and ends, making region-based memory management part of the managed runtime, a vital choice for the adoption in real world.

Furthermore, it is worth noting that a lot of these work was designed specifically for a particular system. For example, Broom is designed specifically for Naiad’s actor semantics. Flare [76] also tries to transform Spark program into C program but it is limited to only the Spark system, same as Deca. Our work is much more general than these, providing benefits to *any* JVM-based language such as Java, Scala, C# or Python.

6.7 Object Sharing in the OS

Skyway is related to a range of work in the system community. The idea of sharing memory segments across processes has been studied in the OS design [58, 74, 92, 115, 147]. An object can exist in different address spaces, allowing the system to share memory across simultaneously executing processes. Mach [147] introduces the concept of a memory object mappable by various processes. The idea was later adopted in the Opal [58] and Nemesis [92] operating systems to describe memory segments characterized by fixed virtual offsets. Lindstrom [115] expands these notions to shareable containers that contain code segments and private memory, leveraging a capability model to enforce protection. Although most contemporary OSes allow one process to be associated with a single virtual address space, there exist systems that support multiple virtual address space abstractions.

The idea of multiple address spaces has mainly been applied to achieve protection in a shared environment [58, 74, 157]. More recently, to support the vast physical memory whose capacity may soon exceed the virtual address space size supported by today's CPUs, SpaceJMP [74] provides a new operating system design that promotes virtual address spaces to first-class citizens, which enables process threads to attach to, detach from, and switch between multiple virtual address spaces. Although this line of work is not directly related to Skyway, they share a similar goal of achieving memory efficiency when objects are needed by multiple processes. XMem [165] is a JVM-based technique that shares heap space across JVM instances. Unfortunately, none of these techniques target object transfer in distributed systems, which Skyway aims for.

Distributed shared memory (DSM) enables physically separated memories to share the same logical address space. Skyway can also be used to improve data transfer in DSM systems.

CHAPTER 7

Conclusions and Future Directions

As modern computing progresses into the era of Big Data, developing systems that can scale efficiently to massive amounts of data is a key challenge faced by both researchers and professionals. When building infrastructure system for Big Data processing, a managed language such as Java or Scala is prevalently of choice due to simple programming interface and automatic memory management, enabling a fast software development cycle.

However, due to the mismatches between the ancient runtime abstractions and the characteristic of modern Big Data workloads, the cost of the runtime system has been magnified and became the performance bottleneck.

This dissertation tackles those mismatches via a series of solutions spanning different levels of the computing stack, leveraging compiler and runtime system techniques to improve many different aspects of Big Data processing including memory efficiency, scalability, latency, and throughput. Specifically, at the compiler level, Facade solves the memory usage mismatch due the grossly excessive object creations in Big Data systems. Facade transforms existing programs into a new execution model that breaks away from the long-held object-oriented principle which uses objects for both data storage and data manipulation. Under Facade's model, data are stored in native memory pages, not in managed heap. To operate on those data, Facade maps each such data item to a facade object in the heap, which then direct access to native memory. Facade guarantees via transformation that the number of heap-based facade objects are statically bounded regardless of how much data items a program must process; each facade object can be reused to represent multiple data items.

At the system level, Yak mitigates the mismatch between the heuristic used by state-of-the-art garbage collectors and the runtime object behaviors in Big Data systems. Although generational hypothesis assumes objects are short-lived and is the foundation on which garbage collectors are built upon to manage all objects, only a small number of runtime objects in Big Data applications conform to such conventional wisdom. Instead, most of the objects follow what we called "epochal hypothesis" — their lifetime aligns well with an epoch, which is a computational event. Epochal behavior is much better managed by region-based memory management techniques. Yak is the first hybrid garbage collector that marries generational GC and region-based techniques so that they can co-exist harmoniously in one single system to manage memory. Yak offers an automated and systematic solution, requiring zero user effort. Objects created in an epoch are allocated in memory regions that are not managed by the generational GC. These memory regions are deallocated as a whole as the end of the epoch. Yak automatically tracks, identifies, and moves region-allocated escaping objects at region deallocation point.

Skyway is also a solution built inside virtual machine, tackling the mismatch between slow serialization interface and frequent data shuffling across worker nodes in clusters. The high data transfer costs in distributed environment come from the lengthy and expensive process of pushing objects in heap format down to lower-level byte format in serialization procedure on the sender; and pulling byte sequence up to managed heap in deserialization procedure on the receiver. Skyway is tailored for Big Data sytems, advocating transferring objects as-is without changing much for their format. Objects are made immediately available in remote nodes, significantly reducing the data transfer costs, and improving work pipelining further by overlapping computation with transferring.

Although Facade, Yak, and Skyway are designed targeting a different mismatch, and thus, having different performance goals built-in, they are not isolated. On the contrary, they are complementary to each other. They are parts of one cohesive, organized research agenda to strive for efficient and scalable Big Data processing. This line of work has sparked interests and follow-up efforts across several communities of programming languages, systems, and software engineering as well as attracted attentions from industrial entities such as Oracle and Huawei.

Some potential future expeditions could be pursued sharing the overarching goal of building a Big-Data-friendly runtime system that let developers enjoy the managed language features without sacrificing much of performance as techniques in this dissertation are as follows:

- **System solutions in emerging environments.** NVM (non-volatile memory) is an emerging technology [27]. The embrace of NVM in Big Data systems is inevitable — NVM offers larger capacity compared to DRAM and consumes less energy compared to SSD. With NVM, 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, naively using NVM will result in significant performance degradation because NVM's access latency is much higher (e.g., 2-4×) than that of DRAM. The key challenge is thus how to develop intelligent data placement and 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 GC which is not aware of the physical memories. To efficiently support Big Data processing over hybrid memories, a series of runtime techniques that can make holistic allocation/migration decisions across multiple layers of the compute stack is required.

- **Leveraging machine learning in memory management.** Poor JVM performance on new workloads is due to the discrepancy between the design, what was known and the characteristic of workload which is unknown. For instance, the development of modern garbage collectors is based on the generational hypothesis — the heap is divided into an old and a young generation and the GC runs frequently in the young generation to move long-lived objects to the old generation while infrequently performing full-heap collection. However, as shown earlier, as new workloads (e.g., data analytics or machine learning) emerge, the generational hypothesis does not hold any more, leading to poor performance in memory management. 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 reflect the object characteristics under different (seen and unseen) workloads. The learned models can be used to predict object lifetimes to significantly reduce the cost of memory management. 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 ideas 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 iterative 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 decision of whether or not to cache a dataset is made entirely at compile time, and thus the system cannot adapt effectively at run time to the execution environment and the data characteristics. Developing 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 simultaneously improve performance. A preliminary experiment in Spark shows that a performance gain of up to 44% can be achieved if the system can automatically cache and uncache data at run time. It is interesting to develop and throughout evaluate such technique.

Bibliography

[1] Help understanding - not enough space to cache RDD. http://apache-spark-user-list.1001560.n3.nabble.com/Help-understanding-Not-enough-space-to-cache-rdd-td20186.html, 2014.

[2] The Rust programming language. http://www.rust-lang.org/, 2014.

[3] Soot framework. http://www.sable.mcgill.ca/soot/, 2014.

[4] Yahoo! webscope program. http://webscope.sandbox.yahoo.com/, 2014.

[5] Zing: Java for the real time business. http://www.azulsystems.com/products/zing/whatisit, 2014.

[6] Cascading. http://www.cascading.org, 2015.

[7] Choose c++ or java for applications requiring huge amounts of ram? http://programmers.stackexchange.com/questions/130108, 2015.

[8] For big data, java or c++. https://www.quora.com/For-big-data-Java-or-C++, 2015.

[9] Out of memory error due to appending values to stringbuilder. http://stackoverflow.com/questions/12831076/, 2015.

[10] Out of memory error due to large spill buffer. http://stackoverflow.com/questions/8464048/, 2015.

[11] Out of memory error in a web parser. http://stackoverflow.com/questions/17707883/, 2015.

[12] Out of memory error in building inverted index. http://stackoverflow.com/questions/17980491/, 2015.

[13] Out of memory error in computing frequencies of attribute values. http://stackoverflow.com/questions/23042829/, 2015.

[14] Out of memory error in customer review processing. http://stackoverflow.com/questions/20247185/, 2015.

[15] Out of memory error in efficient sharded positional indexer. http://stackoverflow.com/questions/1362460/, 2015.

[16] Out of memory error in hash join using distributedcache. http://stackoverflow.com/questions/15316539/, 2015.

[17] Out of memory error in map-side aggregation. http://stackoverflow.com/questions/16684712/, 2015.

[18] Out of memory error in matrix multiplication. http://stackoverflow.com/questions/16116022/, 2015.

[19] Out of memory error in processing a text file as a record. http://stackoverflow.com/questions/12466527/, 2015.

[20] Out of memory error in word cooccurrence matrix stripes builder. http://stackoverflow.com/questions/12831076/, 2015.

[21] The performance comparison between in-mapper combiner and regular combiner. http://stackoverflow.com/questions/10925840/, 2015.

[22] Project Tungsten. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html, 2015.

[23] Reducer hange at the merge step. http://stackoverflow.com/questions/15541900/, 2015.

[24] Spark worker insufficient memory. http://stackoverflow.com/questions/31830834, 2015.

[25] Why is java more popular than c++. http://www.cplusplus.com/forum/general/79656, 2015.

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

[27] Intel launches Optane DIMMs up to 512GB. https://www.anandtech.com/show/12828, 2018.

[28] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[29] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *International Conference on Extending Database Technology (EDBT)*, pages 99–110, 2010.

[30] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. *Proceedings of the VLDB Endowment*, 1(1):958–969, 2008.

[31] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: improving region-based analysis of higher-order languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 174–185, 1995.

[32] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A scalable, open source BDMS. *Proceedings of the VLDB Endowment*, 7(14):1905–1916, 2014.

[33] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 739–753, 2010.

[34] Giraph: Open-source implementation of Pregel. http://giraph.apache.org/, 2014.

[35] Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org/, 2014.

[36] The Hive Project. http://hive.apache.org, 2014.

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

[38] Storm: dstributed and fault-tolerant realtime computation. http://storm.apache.org/, 2014.

[39] Apache Thrift. http://thrift.apache.org/, 2017.

[40] A. W. Appel. Simple generational garbage collection and fast allocation. *Software - Practice and Experience (SPE)*, 19(2):171–183, 1989.

[41] G. Back and W. C. Hsieh. The KaffeOS Java Runtime System. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):583–630, 2005.

[42] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *ACM Conference on Knowledge Discovery and Data Minming (KDD)*, pages 44–54, 2006.

[43] H. G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[44] BEA Systems Inc. Using the Jrockit runtime analyzer. http://edocs.bea.com/wljrockit/docs142/usingJRA/looking.html, 2007.

[45] W. S. Beebee and M. C. Rinard. An implementation of scoped memory for real-time java. In *International Conference on Embedded Software (EMSOFT)*, pages 289–305, 2001.

[46] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage,

and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

[47] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 153–164, 2002.

[48] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 22–32, 2008.

[49] B. Blanchet. Escape analysis for object-oriented languages. Applications to Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 20–34, 1999.

[50] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *International World Wide Web Conference (WWW)*, pages 595–601, 2004.

[51] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *International Conference on Data Engineering (ICDE)*, pages 1151–1162, 2011.

[52] S. Borman. Sensible sanitation – understanding the IBM Java garbage collector. http://www.ibm.com/developerworks/ibm/library/i-garbage1/, 2002.

[53] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 324–337, 2003.

[54] R. Bruno, L. P. Oliveira, and P. Ferreira. NG2C: Pretenuring garbage collection with dynamic generations for Hotspot Big Data applications. In *International Symposium on Memory Management (ISMM)*, pages 2–13, 2017.

[55] Y. Bu, V. Borkar, G. Xu, and M. J. Carey. A bloat-aware design for big data applications. In *International Symposium on Memory Management (ISMM)*, pages 119–130, 2013.

[56] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[57] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2010.

[58] J. Chase, M. Baker-Harvey, H. Levy, and E. Lazowska. Opal: A single address space system for 64-bit architectures. *SIGOPS Oper. Syst. Rev.*, 26(2):9, 1992.

[59] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[60] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *International Symposium on Memory Management (ISMM)*, pages 85–96, 2004.

[61] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.

[62] J. Cohen and A. Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):532–553, 1983.

[63] Colfer. The Colfer serializer. https://go.libhunt.com/project/colfer, 2017.

[64] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 21–21, 2010.

[65] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[66] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[67] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *International Symposium on Memory Management (ISMM)*, pages 37–48, 2004.

[68] I. Dillig, T. Dillig, E. Yahav, and S. Chandra. The CLOSER: Automating resource management in Java. In *International Symposium on Memory Management (ISMM)*, pages 1–10, 2008.

[69] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of VLDB Endow.*, 3:515–529, 2010.

[70] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 345–357, 2000.

[71] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 149–168, 2003.

[72] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 118–128, 2007.

[73] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 59–70, 2008.

[74] I. El Hajj, A. Merritt, G. Zellweger, D. Milojicic, R. Achermann, P. Faraboschi, W.-m. Hwu, T. Roscoe, and K. Schwan. SpaceJMP: Programming with multiple virtual address spaces. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 353–368, 2016.

[75] Executive summary: Data growth, business opportunities, and the IT imperatives. http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm.

[76] G. Essertel, R. Tahboub, J. Decker, K. Brown, K. Olukotun, and T. Rompf. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 799–815, 2018.

[77] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 394–409, 2015.

[78] Y. Feng and E. D. Berger. A locality-improving dynamic memory allocator. In *Workshop on Memory System Performance (MSP)*, pages 68–77, 2005.

[79] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 2–15, 2006.

[80] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[81] D. Gay and A. Aiken. Memory management with explicit regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 313–323, 1998.

[82] D. Gay and A. Aiken. Language support for regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 70–80, 2001.

[83] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction (CC)*, LNCS 1781, pages 82–93, 2000.

[84] O. Gheorghioiu, A. Salcianu, and M. Rinard. Interprocedural compatibility analysis for static object preallocation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 273–284, 2003.

[85] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. NumaGiC: A garbage collector for big data on big NUMA machines. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 661–673, 2015.

[86] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *15th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[87] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[88] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293, 2002.

[89] Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. Spotting code optimizations in data-parallel pipelines through periscope. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 121–133, 2012.

[90] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-Me: a static analysis for automatic individual object reclamation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 364–375, 2006.

[91] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 141–152, 2002.

[92] S. M. Hand. Self-paging in the nemesis operating system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 73–86, 1999.

[93] T. Harris. Early storage reclamation in a tracing garbage collector. *SIGPLAN Notice*, 34(4):46–53, Apr. 1999.

[94] C. Hawblitzel and T. von Eicken. Luna: A flexible Java protection system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 391–403, 2002.

[95] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, pages 261–272, 2011.

[96] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in Cyclone. In *International Symposium on Memory Management (ISMM)*, pages 73–84, 2004.

[97] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 359–373, 2003.

[98] R. L. Hudson and J. E. B. Moss. Incremental collection of mature objects. In *International Workshop on Memory Management (IWMM)*, pages 388–403, 1992.

[99] Hyracks: A data parallel platform. `http://code.google.com/p/hyracks/`, 2014.

[100] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, pages 59–72, 2007.

[101] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 77–88, 2012.

[102] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *International Symposium on Computer Architecture (ISCA)*, pages 158–169, 2015.

[103] P. Kedia, M. Costa, M. Parkinson, K. Vaswani, D. Vytiniotis, and A. Blankstein. Simple, fast, and safe manual memory management. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 233–247, 2017.

[104] H. Kermany and E. Petrank. The Compressor: Concurrent, incremental, and parallel compaction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 354–363, 2006.

[105] S. Kowshik, D. Dhurjati, and V. Adve. Ensuring code safety without runtime checks for real-time control systems. In *International Conference on Architecture and Synthesis for Embedded Systems (CASES)*, pages 288–297, 2002.

[106] The Kryo serializer. `https://github.com/EsotericSoftware/kryo`, 2017.

[107] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *International World Wide Web Conference (WWW)*, pages 591–600, 2010.

[108] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[109] C. Lattner. *Macroscopic Data Structure Analysis and Optimization.* PhD thesis, University of Illinois at Urbana-Champaign, 2005.

[110] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 129–142, 2005.

[111] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 278–289, 2007.

[112] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another SQL-to-MapReduce translator. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 25–36, 2011.

[113] O. Lhoták and L. Hendren. Scaling Java points-to analysis using SPARK. In *International Conference on Compiler Construction (CC)*, pages 153–169, 2003.

[114] O. Lhoták and L. Hendren. Run-time evaluation of opportunities for object inlining in Java. *Concurrency and Computation: Practice and Experience*, 17(5-6):515–537, 2005.

[115] A. Lindstrom, J. Rosenberg, and A. Dearle. The grand unified theory of address spaces. In *HotOS*, pages 66–71, 1995.

[116] J. Liu, N. Ravi, S. Chakradhar, and M. Kandemir. Panacea: Towards holistic optimization of MapReduce applications. In *International Symposium on Code Generation and Optimization (CGO)*, pages 33–43, 2012.

[117] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng. Lifetime-based memory management for distributed data processing systems. *Proceedings of the VLDB Endowment*, 9(12):936–947, 2016.

[118] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz. Trash Day: Coordinating garbage collection in distributed systems. In *5th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[119] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 457–471, 2016.

[120] H. Makholm. A region-based memory manager for Prolog. In *International Symposium on Memory Management (ISMM)*, pages 25–34, 2000.

[121] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *International Conference on Management of Data (SIGMOD)*, pages 135–146, 2010.

[122] Y. Mandelbaum, K. Fisher, D. Walker, M. F. Fernández, and A. Gleyzer. PADS/ML: a functional data description language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 77–83, 2007.

[123] D. Marinov and R. O'Callahan. Object equality profiling. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 313–325, 2003.

[124] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, Apr. 1960.

[125] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 77–97, 2009.

[126] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to Java runtime bloat. *IEEE Software*, 27(1):56–63, 2010.

[127] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 245–260, 2007.

[128] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behavior in framework-based applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 429–451, 2006.

[129] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 121–131, 2011.

[130] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 439–455, 2013.

[131] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *USENIX Annual Technical Conference (ATC)*, pages 291–305, 2015.

[132] K. Nguyen, L. Fang, C. Navasca, G. Xu, B. Demsky, and S. Lu. Skyway: Connecting managed heaps in distributed big data systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 56–69, 2018.

[133] K. Nguyen, L. Fang, G. Xu, and B. Demsky. Speculative region-based memory management for big data systems. In *Workshop on Programming Languages and Operating Systems (PLOS)*, pages 27–32, 2015.

[134] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 349–365, 2016.

[135] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 675–690, 2015.

[136] K. Nguyen, K. Wang, Y. Bu, L. Fang, and G. Xu. Understanding and combating memory bloat in managed data-intensive systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(4):12:1–12:41, 2018.

[137] K. Nguyen and G. Xu. Cachetor: detecting cacheable data to remove bloat. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 268–278, 2013.

[138] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *International Conference on Software Engineering (ICSE)*, pages 562–571, 2013.

[139] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: sharing across multiple queries in MapReduce. *Proceedings of the VLDB Endowment*, 3(1-2):494–505, 2010.

[140] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX Annual Technical Conference (ATC)*, pages 267–273, 2008.

[141] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *International Conference on Management of Data (SIGMOD)*, pages 1099–1110, 2008.

[142] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–307, 2015.

[143] M. Parkinson, D. Vytiniotis, K. Vaswani, M. Costa, P. Deligiannis, D. McDermott, A. Blankstein, and J. Balkind. Project Snowflake: Non-blocking safe manual memory management in .NET. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):95:1–95:25, Oct. 2017.

[144] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal*, 13(4):227–298, 2005.

[145] Protocol Buffers. `https://developers.google.com/protocol-buffers/docs/javatutorial`, 2017.

[146] F. Qian and L. Hendren. An adaptive, region-based allocator for Java. In *International Symposium on Memory Management (ISMM)*, pages 127–138, 2002.

[147] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 31–39, 1987.

[148] N. Sachindran, J. E. B. Moss, and E. D. Berger. MC^2: High-performance garbage collection for memory-constrained environments. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 81–98, 2004.

[149] S. Salihoglu and J. Widom. GPS: A graph processing system. In *Scientific and Statistical Database Management*, pages 22:1–22:12, July 2013.

[150] A. Shankar, M. Arnold, and R. Bodik. JOLT: Lightweight dynamic analysis and removal of object churn. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 127–142, 2008.

[151] X. Shi, Z. Ke, Y. Zhou, L. Lu, X. Zhang, H. Jin, L. He, and F. Wang. Deca: a garbage collection optimizer for in-memory data processing. *ACM Transactions on Computer Systems (TOCS)*, 2018.

[152] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimizations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 295–306, 2002.

[153] E. Smith. The java serialization benchmark set. https://github.com/eishay/jvm-serializers, 2017.

[154] C. Stancu, C. Wimmer, S. Brunthaler, P. Larsen, and M. Franz. Safe and efficient hybrid memory management for Java. In *International Symposium on Memory Management (ISMM)*, pages 81–92, 2015.

[155] D. Stefanović, M. Hertz, S. M. Blackburn, K. S. McKinley, and J. E. B. Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *Workshop on Memory System Performance (MSP)*, pages 25–36, 2002.

[156] D. Stefanović, K. S. McKinley, and J. E. B. Moss. Age-based garbage collection. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 370–381, 1999.

[157] M. Takahashi, K. Kono, and T. Masuda. Efficient kernel support of fine-grained protection domains for mobile code. In *ICDCS*, pages 64–73, 1999.

[158] G. Tene, B. Iyengar, and M. Wolf. C4: The continuously concurrent compacting collector. In *International Symposium on Memory Management (ISMM)*, pages 79–88, 2011.

[159] D. Terei, A. Aiken, and J. Vitek. M^3: High-performance memory management from off-the-shelf components. In *International Symposium on Memory Management (ISMM)*, pages 3–13, 2014.

[160] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[161] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *International Conference on Data Engineering (ICDE)*, pages 996–1005, 2010.

[162] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lamda-calculus using a stack of regions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 188–201, 1994.

[163] TPC. The standard data warehousing benchmark. http://www.tpc.org/tpch, 2014.

[164] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[165] M. Wegiel and C. Krintz. XMem: Type-safe, transparent, shared memory for cross-runtime communication and coordination. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 327–338, 2008.

[166] M. Wegiel and C. Krintz. Dynamic prediction of collection yield for managed runtimes. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 289–300, 2009.

[167] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 187–206, 1999.

[168] J. World. The Java serialization algorithm revealed. http://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html, 2017.

[169] G. Xu. Finding reusable data structures. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1017–1034, 2012.

[170] G. Xu. Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 111–130, 2013.

[171] G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 174–186, 2010.

[172] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 419–430, 2009.

[173] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-

oriented applications. In *ACM SIGSOFT FSE/SDP Working Conference on the Future of Software Engineering Research (FoSER)*, pages 421–426, 2010.

[174] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *International Conference on Software Engineering (ICSE)*, pages 151–160, 2008.

[175] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 160–173, 2010.

[176] G. Xu, D. Yan, and A. Rountev. Static detection of loop-invariant data structures. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 738–763, 2012.

[177] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *International Conference on Management of Data (SIGMOD)*, pages 1029–1040, 2007.

[178] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 247–260, 2009.

[179] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, 2008.

[180] M. Zaharia. What is changing in big data? `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/Zaharia_Matei_Big_Data.pdf`, 2016. MSR Faculty Summit.

[181] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for

in-memory cluster computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 2–2, 2012.

[182] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[183] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

[184] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *International Conference on Data Engineering (ICDE)*, pages 1060–1071, 2010.