UNIVERSITY OF CALIFORNIA,
IRVINE

# Detecting and Fixing Memory-Related Performance Problems in Managed Languages

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Lu Fang

Dissertation Committee:
Professor Guoqing Xu, Chair
Professor Alex Nicolau
Professor Brian Demsky

2017

# DEDICATION

*To my supportive parents, Yaping Fang and Ling Gao,*
*and my amazing girlfriend, Yiran Xie.*

# TABLE OF CONTENTS


Page

**LIST OF FIGURES** **v**

**LIST OF TABLES** **vii**

**LIST OF ALGORITHMS** **viii**

**ACKNOWLEDGMENTS** **ix**

**CURRICULUM VITAE** **x**

**ABSTRACT OF THE DISSERTATION** **xiii**

**1 Introduction** **1**

**2 Background** **7**
2.1 Performance Problems . . . . . 7
2.2 Managed Languages . . . . . 8
2.3 Data-Parallel Systems . . . . . 9

**3 Memory-Related Performance Bug Description and Detection** **11**
3.1 ISL: Instrumentation Specification Language . . . . . 12
3.1.1 ISL Overview . . . . . 12
3.1.2 ISL Syntax . . . . . 17
3.1.3 Instrumentation Semantics . . . . . 18
3.2 Amplification and Deamplification . . . . . 24
3.3 More ISL Programs . . . . . 25
3.3.1 Allocation sites creating invariant objects. . . . . 25
3.3.2 Under-utilized Java maps. . . . . 27
3.3.3 Never-used return objects. . . . . 27
3.4 Summary and Interpretation . . . . . 30

**4 PerfBlower: A Performance Testing Framework** **31**
4.1 Overview . . . . . 31
4.2 Diagnostic Information Collection . . . . . 33
4.2.1 Heap Reference Paths . . . . . 33
4.2.2 Mirroring Reference Paths . . . . . 35

4.3 Evaluation . . . 41
4.3.1 Amplification Effectiveness . . . 42
4.3.2 Problems Found . . . 46
4.3.3 PerfBlower Completeness . . . 50
4.3.4 False Positive Elimination . . . 52
4.4 Limitations . . . 53
4.5 Summary and Interpretation . . . 54

**5 ITask: Helping Data-Parallel Tasks Survive Memory Pressure 58**
5.1 Memory Problems in the Real World . . . 58
5.2 Design Overview . . . 61
5.3 The ITask Programming Model . . . 66
5.3.1 Programming Model . . . 66
5.3.2 Instantiating ITasks in Existing Frameworks . . . 73
5.3.3 Discussion . . . 78
5.4 The ITasks Runtime System . . . 79
5.4.1 The IRS Overview . . . 79
5.4.2 Monitor . . . 80
5.4.3 Partition Manager . . . 81
5.4.4 Scheduler . . . 82
5.4.5 Fault Tolerance . . . 84
5.5 Evaluation . . . 84
5.5.1 ITasks in Hadoop . . . 85
5.5.2 ITasks in Hyracks . . . 88
5.6 Summary and Interpretation . . . 98

**6 Related Work 101**
6.1 Finding Performance Bugs . . . 101
6.2 Test Amplification Techniques . . . 102
6.3 Domain-Specific Languages . . . 103
6.4 Optimizations of Data-Parallel Systems . . . 103
6.5 Performance autotuning . . . 105

**7 Conclusions and Future Work 106**

**Bibliography 109**

# LIST OF FIGURES


Page

3.1 An ISL program for detecting memory leaks. . . . . 13
3.2 Abstract syntax of ISL. . . . . 17
3.3 Parameter lists of ISL events. . . . . 18
3.4 ISL semantic domains. . . . . 19
3.5 Auxiliary functions. . . . . 20
3.6 Instrumentation semantics of rule NEW. . . . . 21
3.7 Instrumentation semantics of rule LOAD. . . . . 22
3.8 Instrumentation semantics of rule STORE. . . . . 22
3.9 Instrumentation semantics of rule GC. . . . . 23
3.10 The definition of VSO. . . . . 24
3.11 An ISL program for detecting allocation sites creating invariant objects. . . . 26
3.12 An ISL program for detecting under-utilized Java hash maps. . . . . 28
3.13 An ISL program for detecting never-used return objects. . . . . 29

4.1 The architecture of PerfBlower. . . . . 32
4.2 A Java program with memory leaks. . . . . 34
4.3 An example heap reference path. . . . . 35
4.4 An example mirror path. . . . . 35
4.5 The layout of a tracked object. . . . . 36
4.6 The definition of the `Mirror` class. . . . . 36
4.7 VSOs for the memory leak amplifier with different heap sizes. . . . . 45
4.8 A partial reference path reported by our memory leak amplifier. . . . . 47

5.1 Memory footprints of executions with (red) and without (blue) ITask. . . . . 61
5.2 Handling the different memory components of an ITask. . . . . 62
5.3 The architecture of the ITask runtime system. . . . . 64
5.4 The `DataPartition` abstract class. . . . . 67
5.5 The `ITask` abstract class. . . . . 68
5.6 Setting input-output relationship. . . . . 70
5.7 The dataflow of an ITask execution. . . . . 71
5.8 The `MITask` abstract class. . . . . 72
5.9 `MapOperator` with ITask for the `WordCount` in Hyracks. . . . . 74
5.10 `ReduceOperator` with ITask for the `WordCount` in Hyracks. . . . . 75
5.11 MITask for the `WordCount` in Hyracks. . . . . 76
5.12 Modified `Mapper` in Hadoop. . . . . 77

5.13 Performance of the original WC with different threads. . . . . 91
5.14 Performance of the original HS with different threads. . . . . 91
5.15 Performance of the original II with different threads. . . . . 92
5.16 Performance of the original HJ with different threads. . . . . 92
5.17 Performance of the original GR with different threads. . . . . 92
5.18 Comparisons between the ITask `WC` and its Java counterpart. . . . . 94
5.19 Comparisons between the ITask `HS` and its Java counterpart. . . . . 95
5.20 Comparisons between the ITask `II` and its Java counterpart. . . . . 95
5.21 Comparisons between the ITask `HJ` and its Java counterpart. . . . . 95
5.22 Comparisons between the ITask `GR` and its Java counterpart. . . . . 96
5.23 The legend of Figure 5.18–5.22. . . . . 96
5.24 The performance of `WC` whiling changing the heap size. . . . . 97
5.25 The performance of `II` whiling changing the heap size. . . . . 97
5.26 The number of active ITask instances during the execution. . . . . 98

# LIST OF TABLES


Page

4.1 Space overheads reported by the amplifiers. . . . . 42
4.2 Space overheads for memory leaks with different history sizes. . . . . 44
4.3 Comparison of PerfBlower and Sleigh's reports. . . . . 51
4.4 Completeness of PerfBlower. . . . . 52
4.5 False warnings eliminated by deamplification. . . . . 54

5.1 Hadoop performance comparisons for five real-world programs. . . . . 86
5.2 A detailed breakdown of memory savings. . . . . 87
5.3 `WC`, `HS`, and `II`'s inputs: the Yahoo! Webmap and its subgraphs. . . . . 89
5.4 `HJ` and `GR`'s inputs: TPC-H data. . . . . 89
5.5 The configurations which provide the best scalability/performance. . . . . 91
5.6 A summary of the performance improvements from ITask. . . . . 93

# LIST OF ALGORITHMS


| | | Page |
|---|---|---|
| 1 | Incrementally creating virtual penalties and building mirror paths. . . . . | 38 |
| 2 | The major logic of the IRS monitor. . . . . | 80 |
| 3 | The major logic of the IRS partition manager. . . . . | 81 |
| 4 | The major logic of the IRS scheduler. . . . . | 83 |

# ACKNOWLEDGMENTS

My sincerest and deepest thanks go to my advisor, Professor Guoqing Xu, who has expended a tremendous amount of effort on training me into a good researcher. It has been my great honor to join Professor Xu's research group as his first Ph.D. student. During the past five years, from him, I have learned how to identify problems, solve problems, build systems, think deeply, write papers, make presentations, manage time, collaborate with people and be a good person. Without his guidance and help, my doctoral research and this dissertation would not have been possible. Since I can never repay Professor Xu for what he has given to me, I will try my best to make him proud of me in the future.

I would like to thank Professor Brian Demsky, who has served on my final defense committee and candidacy exam committee. Professor Demsky has provided many constructive comments on my research work, including PerfBlower, ITask, Yak and Skyway, from system architectures to implementations. From him, I have learned how to build practical systems in a right way.

I would like to thank Professor Alex Nicolau, Professor Michael Franz, and Professor Mohammad Al Faruque, for serving on my final defense committee and/or candidacy exam committee. Their valuable feedback helped me improve my work.

I would like to thank Dr. Gang Ren and Dr. Du Li for providing me opportunities to intern at Google and Hewlett Packard Labs, respectively. The internship has enlarged my visions and helped me understand how to align the research with real-world impact.

I would like to thank my fantastic colleague and best teammate, Khanh Nguyen, for his enormous help with my work and life. Like Professor Xu always says, when Khanh and I team up together, nothing is impossible. I will never forget the days and nights we spent together building systems and fighting against bugs.

I would like to thank my colleagues, Dr. Yingyi Bu, Dr. Zhiqiang Zuo, Dr. Keval Vora, Kai Wang, Jianfei Hu, Aftab Hussain, Cheng Cai, Bojun Wang, Christian Navasca, Matthew Hartz, Louis Zhang, Ankur Gupta, for their great support during the past five years.

Last but not least, I would like to thank my friends for enriching my life. My thanks go to Guannan Li, Dawei Guo, Man Xu, Rong Zhu, Yuanfang Sun, Yunshan Ke, Ximin Chen, Wei Zhao, Shuai Shao, Weihu Wang, Fangwei Si, Wan Shi, Xu Sun, Lang Wang, Xin Zhao, Lei Shen, Rui Yang, Xin Yong, Yanchao Xin, Ping Hu, Le Huang, Dr. Nan Hu, Dr. Bowen Meng, Dr. Ze Yu, Dr. Dimin Niu, Rundong Lv, Yuchen Liu, Haochen Tang, Kai Pan, Kailai Zhou, Lucas Sun, Hamilton Wen, Owen Ou, Fan Yin, Junjie Shen, Dr. Zhiqiang Jin, Haoli He, Albert Jiang, Penghang Yin, Zhaohao Zhou, Bingchen Yu, Dr. Xiyan Wang, Dr. Jiaxi Wang, Zhi Chen, Menglin Jiang, Dr. Chunyan Wang, Kan Wang.

The work presented in this dissertation has been supported by NSF under grants CNS-1321179, CCF-1409829, CNS-1613023, CCF-0846195, and by ONR under grants N00014-14-1-0549 and N00014-16-1-2913.

# CURRICULUM VITAE

## Lu Fang

### EDUCATION

**Doctor of Philosophy in Computer Science** **2012–2017**
University of California, Irvine *Irvine, California*

**Master of Science in Computer Science** **2009–2012**
Peking University *Beijing, China*

**Bachelor of Science in Computer Science** **2005–2009**
Peking University *Beijing, China*

### RESEARCH EXPERIENCE

**Graduate Research Assistant** **2012–2017**
University of California, Irvine *Irvine, California*

**System Research Intern** **Summer 2016**
Hewlett Packard Labs *Palo Alto, California*

**Graduate Research Assistant** **2009–2012**
Peking University *Beijing, China*

### TEACHING EXPERIENCE

**Teaching Assistant** **2012–2013**
University of California, Irvine *Irvine, California*

**Teaching Assistant** **2009–2011**
Peking University *Beijing, China*

### PROFESSIONAL EXPERIENCE

**Software Engineer Intern** **Summer 2015**
Google Inc. *Mountain View, California*

**Software Engineer Intern** **2010–2011**
Kuwo Inc. *Beijing, China*

**Software Engineer Intern** **Summer 2009**
IBM *Beijing, China*

## REFEREED PUBLICATIONS

**Yak: A High-Performance Big-Data-Friendly Garbage Collector** **Nov. 2016**
The 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)

**Low-Overhead and Fully Automated Statistical Debugging with Abstraction Refinement** **Nov. 2016**
The 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)

**Speculative Region-based Memory Management for Big Data Systems** **Oct. 2015**
The 8th Workshop on Programming Languages and Operating Systems (PLOS'15)

**Interruptible Tasks: Treating Memory Pressure As Interrupts for Highly Scalable Data-Parallel Programs** **Oct. 2015**
The 25th ACM Symposium on Operating Systems Principles (SOSP'15)

**PerfBlower: Quickly Detecting Memory-Related Performance Problems via Amplification** **Jul. 2015**
The 29th European Conference on Object-Oriented Programming (ECOOP'15)

**Facade: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications** **Mar. 2015**
The 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)

**An Exploratory Study of API Usage Examples on the Web** **Dec. 2012**
The 19th Asia-Pacific Software Engineering Conference (APSEC'12)

**Towards Automatic Tagging for Web Services** **Jun. 2012**
The 19th IEEE International Conference on Web Services (ICWS'12)

**APIExample: An Effective Web Search Based Usage Example Recommendation System for Java APIs** **Nov. 2011**
The 26th IEEE/ACM International Conference On Automated Software Engineering (ASE'11)

## SOFTWARE

**PerfBlower** `https://bitbucket.org/fanglu/perfblower-public`
*PerfBlower is a general performance testing framework which helps developers find and fix memory-related performance problems at the early stage.*

# ABSTRACT OF THE DISSERTATION

Detecting and Fixing Memory-Related Performance Problems in Managed Languages

By

Lu Fang

Doctor of Philosophy in Computer Science

University of California, Irvine, 2017

Professor Guoqing Xu, Chair

Performance problems commonly exist in many kinds of real-world applications, including smartphone apps, databases, web servers, as well as large-scale data analytical systems. The pervasive use of managed languages such as Java and C#, which often blow up memory usage in unpredictable ways, further exacerbates these problems. A great deal of evidence from various application communities shows that seemingly insignificant performance problems can lead to severe scalability reductions and even financial losses.

Performance problems are notoriously difficult to find and fix in real-world systems. First, visible performance degradation is often an accumulation of the effects of a great number of invisible problems that scatter all over the program. These problems only manifest for particular workloads or when the input size is large enough. Hence, it is extremely difficult, if not impossible, for developers to catch performance problems early during testing before they go out to production and are experienced by users. Fixing these problems is equally difficult. Developing a fix requires the understanding of the root cause, which can be both time-consuming and labor-intensive given the complexity of large software systems. Furthermore, for modern applications such as data analytical systems, application developers often write only a simple layer of user logic (e.g., Map/Reduce programs), treating the underlying system as a blackbox. It would not be possible to come up with a fix if the problem is located deeply

in the system code.

There is a rich literature in techniques that can find and fix performance problems in managed programs. However, the existing works all suffer from one or more of the following drawbacks: (1) lacking a general way to describe different kinds of performance problems, (2) lacking effective test oracles that can capture invisible performance problems under small workloads, (3) lacking effective debugging support that can help developers find the root cause when a bug manifests, and (4) lacking a systematic approach that can effectively tune memory usage in data-intensive systems. As a result, performance bugs still escape to production runs, hurt user experience, degrade system throughput, and waste computational resources. For modern Big Data systems, most of which are written in managed languages, performance problems become increasingly critical, causing scalability degradation, unacceptable latency, and even execution failures leading to wide-range crashes and financial losses.

In this dissertation, we propose a set of dynamic techniques that can help developers find and fix memory-related performance problems in both programs running on a single machine and data-intensive systems deployed on large clusters. Specifically, this dissertation makes the following three major contributions. The *first contribution* is the design of an instrumentation specification language (ISL) in which one can easily describe the symptoms and counter-evidence of performance problems. The language supports two primary actions *amplify* and *deamplify*, which can help developers capture invisible problems by "blowing up" the effect of the performance bugs and reduce false warnings. The *second contribution* is the development of a general performance testing framework named PerfBlower, which can efficiently profile program executions, continuously amplify the problems specified with the ISL language, and report reference-path-based diagnostic information, making it easy for the user to understand the root cause of a reported problem and develop a fix. The *third contribution* is the design and implementation of *interruptible task*, a new type of data-parallel tasks that can be interrupted upon memory pressure—with part or all of their used memory

reclaimed—and resumed when the pressure goes away, to help large-scale data analytical systems such as Hadoop and Hyracks survive memory pressure.

To evaluate these techniques, we have performed an extensive set of experiments using real-world programs and datasets. Our experimental results demonstrate that the techniques proposed in this dissertation can effectively detect and fix memory-related performance problems in both single-machine programs and distributed data-parallel system. These techniques are also easy to use — users can quickly describe performance problems using ISL and/or develop interruptible tasks to improve the performance of their application code without understanding the underlying systems. These techniques can be readily employed in practice to help real-world developers in various testing and tuning scenarios.

# Chapter 1

# Introduction

Performance problems commonly exist in many kinds of real-world applications, such as smartphone apps, databases, web servers, as well as large-scale analytical systems. The pervasive use of managed languages such as Java and C#, which often blow up memory usage in unpredictable ways [67, 124, 128], further exacerbates these problems. A great deal of evidence [52, 121, 68] shows that seemingly insignificant performance problems can lead to severe scalability reductions and even financial losses.

Performance problems are notoriously difficult to find and fix in real-world systems for a number of reasons. First, visible performance degradation is often an accumulation of the effects of a great number of invisible problems that scatter all over the program [121]. These problems only manifest for particular workloads or when the input size is large enough. Hence, it is extremely difficult, if not impossible, for developers to catch performance problems early during testing before they go out to production and are experienced by users. Fixing these problems is equally difficult. Developing a fix requires the understanding of the root cause, which can be both time-consuming and labor-intensive given the complexity of large software systems. Furthermore, for modern applications such as data analytical sys-

tems, application developers often write only a simple layer of user logic (e.g., Map/Reduce programs), treating the underlying system as a blackbox. It would not be possible to come up with a fix if the problem is located deeply in the system code.

To find and fix performance problems, researchers have proposed a wide variety of techniques in programming languages and systems. Many approaches [52, 75, 126, 127, 77, 73] fall into the category of *pattern-based bug detection* where patterns are summarized to find regions of inefficient code. *Heap analysis* [69, 68], *data mining* [46, 132] and *machine learning* [98, 140, 118] techniques have also been employed to detect performance problems. Much work [24, 44, 141, 32, 70, 63, 138, 109] has been done to improve scalability at the architecture level for distributed systems [36, 78, 32, 63, 3, 64, 58, 29, 26]. In addition to these research efforts, many open-source frameworks (e.g., [30, 42, 107, 92, 49]) have been designed to support performance testing, e.g., by generation of large tests/loads for triggering performance bugs[1].

However, all of these existing techniques suffer from one or more of the following drawbacks:

- **Drawback 1: Lacking a general specification to describe different kinds of performance problems.** Existing tools are designed for specific types of problems. To detect/fix a new type of problems, developers need to develop tools completely from scratch.

- **Drawback 2: Lacking effective test oracles that can capture invisible performance problems under small (testing) workloads.** Many tools detect performance problems by checking the absolute time elapsed and/or memory usage. Such approaches are often subjective and cannot reveal small performance bugs before they accumulate. As a result, performance bugs still escape to production runs, hurt user experience, degrade system throughput, and waste computational resources [19]. They affect even well-tested software such as Windows 7's Windows Explorer, which had sev-

[1]We use "performance bug" and "performance problem" interchangeably.

eral high-impact performance bugs that escaped detection for long periods of time [46].

- **Drawback 3: Lacking effective debugging support that can help developers find the root cause when a bug manifests.** Precise and useful diagnostic information is critical for identifying root causes and fixing performance problems. However, most existing detectors can only provide simple information such as object types or allocation sites. Without enough execution information, fixing the problem is as difficult as finding a needle in a haystack.

- **Drawback 4: Lacking a systematic way to reduce memory pressure in data-parallel systems.** Data-parallel systems such as Hadoop commonly experience high memory pressure that can cause them to fail or suffer from very high garbage collection (GC) overhead. These systems have many tuning parameters that can be used to adjust various configurations such as degree of parallelism or task granularity. Practical solutions to high memory pressure center around making "best practice" recommendations for manual tuning of framework parameters [9]. For example, the developer could reduce input size for each task (i.e., finer granularity) and/or degree-of-parallelism (i.e., fewer threads). However, it is impossible to find a one-size-fits-all configuration even for a single data-parallel program when considering data skewness and different behaviors of its tasks, not to mention finding an optimal configuration across different programs.

In this dissertation, we propose a set of dynamic techniques that can help developers find and fix memory-related performance problems in both programs running on a single machine and data-intensive systems deployed on large clusters. This dissertation makes the following contributions:

- **Contribution 1: A performance specification language.** To provide developers a general way to define symptoms, we propose a simple, event-based language, referred

to as ISL (i.e., short for instrumentation specification language). ISL is a domain-specific language tailored for describing symptoms of performance problems on a JVM. Since performance problems cannot be expressed using logical assertions, an important challenge in the design of ISL is what to do when a symptom is seen. Unlike a regular detector that immediately reports a problem, ISL provides a pair of commands *amplify* and *deamplify*, which allow developers to add *memory penalties* to an object upon observing a symptom and remove penalties when a *counter-evidence* is seen. Symptoms are only *indicators* of performance problems, not *definitive evidence*. If the specified symptom does not point to a real problem (e.g., the symptom stays for a while and then disappears), the developer calls *deamplify* to remove penalties. The symptom specified in ISL is periodically checked (often during garbage collection): objects that keep satisfying the specification will keep getting penalized; all existing penalties for an object are removed at the moment we observe that the object does not satisfy the specification.

- **Contribution 2: A general performance testing framework.** We build a novel framework PerfBlower with the support for ISL to help developers efficiently amplify memory-related performance bugs at the early stage. To collect useful diagnostic information, PerfBlower incrementally builds a mirror chain to reflects the major reference path leading to the problematic objects.

  We have implemented PerfBlower based on the Jikes Research Virtual Machine. Using PerfBlower, we have amplified *four different types of performance problems* (i.e., memory leaks, under-utilized containers, over-populated containers, and never-used return objects) on a set of real-world applications; our experimental results show that even under small workloads, there is a very clear distinction between the VSOs for executions with and without problems, making it particularly easy for developers to write simple space assertions and only inspect programs that have assertion failures. In fact, we have manually inspected each of our benchmarks for which PerfBlower reports a

high VSO, and found a total of *8 unknown problems*.

- **Contribution 3: A systematic approach helping data-parallel tasks survive memory pressure.** We propose a novel, systematic approach, called *interruptible task (ITask)*, that can help data-parallel programs survive memory pressure without needing (1) additional hardware resources (e.g., memory, nodes, etc.) or (2) manual parameter tuning. Inspired by how processors handle hardware interrupts, our idea is to *treat memory pressure as interrupts*: a data-parallel task can be interrupted upon memory pressure—with part or all of its consumed memory reclaimed—and re-activated when the pressure goes away.

  We have instantiated ITasks in the widely-used Hadoop framework [102] as well as Hyracks [17], a distributed data-parallel framework. We reproduced 13 Hadoop problems reported on StackOverflow and implemented their ITask versions. ITask was able to help all of these programs survive memory pressure and successfully process their entire datasets. For a diverse array of 5 problems, we performed an additional comparison between their ITask versions under default configurations and their original programs under the recommended configurations; the results show that the ITask versions outperform the manually-tuned versions by an average of 2×. For Hyracks, we selected 5 well-tuned programs from its repository and performed a detailed study on performance and scalability between their original versions and ITask versions: the ITask versions are 1.5–3× faster and scale to 3–24× larger datasets than their regular counterparts. In fact, the scalability improvement ITask provides can be even higher (e.g., hundreds of times) if further larger datasets are used.

**Impact.** The amplification technique proposed in this dissertation is the first attempt to reveal performance problems under small workloads by amplifying the effects of problematic objects. The success of this technique has demonstrated that it is possible to capture invisible

performance problems under small inputs during in-house testing, shedding new light on future development of performance testing/debugging techniques.

ISL and PerfBlower have immediate practical impact for three reasons. First, developers can quickly create detectors to amplify new performance problems as long as the problems have visible heap symptoms. Our techniques can save developers tremendous effort of building new detectors from scratch. Second, PerfBlower captures performance problems under small workloads, before they escape to production runs. This can help developers prevent the severe losses caused by performance bugs as well as improve the user experience of software products. Third, the object mirroring technique, which PerfBlower employs to store the major reference path leading to each suspicious objects, can be employed by any dynamic analysis to collect useful diagnostic information.

ITask is the first systematic approach to solve memory pressure problems in data-parallel systems. The design of ITask's programming model and runtime system provides developers new insights for future development of data-intensive systems. ITask can significantly improve the performance and scalability of existing data-parallel systems, such as Hadoop, which processes data for thousands of companies and has great influence on our daily lives. The non-intrusive feature of ITask makes it easy to apply our techniques for any existing frameworks.

**Organization.** The remainder of this dissertation is organized as follows. Chapter 2 introduces several basic concepts. The details of the problem specification language, called ISL, are described in Chapter 3. Chapter 4 presents our general performance testing framework PerfBlower and its experimental results. Chapter 5 proposes a systematic approach, named ITask, which helps data-parallel tasks survive memory pressure. Related work is discussed in Chapter 6. Finally, we conclude in Chapter 7.

# Chapter 2

# Background

## 2.1 Performance Problems

Performance problems, also known as performance bugs usually do not affect the correctness of programs. Instead, they may significantly degrade the performance of software systems and waste computational resources [52]. Because even expert programmers may introduce performance problems into their products, performance bugs widely exist in real world applications, including many well-tested products such as Internet Explorer, Mozilla Firefox, Windows 10, etc. It seems that performance problems are harmless; but actually they can lead to disasters [31], such as badly hurting user experience [66, 65], substantial financial losses [13, 61], etc.

Most existing performance debugging techniques require users to provide test cases which can trigger the bugs. This means that many performance problems do not manifest at the development stage, and they escape to production runs. So we propose an amplification framework to help developers capture the performance problems at the early stage. With precise diagnostic information, developers can easily fix most of the performance bugs with

changing several lines of code [52, 74]. But some performance problems (e.g., the scalability issues in Big Data systems caused by memory pressure) are extremely difficult to solve, even though we know the root causes of the problems. We will attack the memory problems in data-parallel systems in Chapter 5.

## 2.2 Managed Languages

If a programming language runs in its own container (i.e., a virtual machine), we call it managed language. For example, Java is a managed language, because we run Java programs in Java Virtual Machines (JVM). There are many other managed languages such as Scala, C#, etc. Compared with unmanaged languages, managed languages have the following advantages:

- The design of managed languages can greatly reduce the cost of supporting different languages on many platforms. In managed languages, source code is first compiled to some intermediate representation (IR), and executes under the management of a virtual machine. For example, we would like to develop support of 4 languages on 5 platforms. Without IR, we have to develop the compilation and runtime support for each language on every platform. There are totally 20 compilers to build. On the other hand, if these languages are first compiled to the unique IR, we only need to build 4 compilers which translate the original programs to the IR, and 5 runtime systems which execute the IR code. From the above case, we clearly see that this design can dramatically improve the scalability of the development environment.

- Managed languages are usually platform independent. For example, once Java code is compiled to bytecode, it can run in compatible JVMs on different platforms. So managed languages are more portable than unmanaged ones.

- Compared to native code, managed languages are safer. Unlike unmanaged code, which is directly executed by CPU, IR code runs in a managed environment. Thus the code is checked to be safe, and it cannot crash the host.

- In managed languages, the memory management is automated by Garbage Collection (GC), which can significantly alleviate developers' burden, and substantially increase their productivity.

Due to the above reasons, managed languages are widely used in the real world applications. According to TIOBE index [110], Java is the most popular programming language over the past decade. Most popular Big Data systems are also written in managed languages. For example, Hadoop [102], Hive [104], Pig [106], and Cassandra [101] are implemented in Java.

The mainstream garbage collectors in modern virtual machines are tracing based collectors. In one collection, a tracing collector traces live objects by following references, starting from a set of root objects that are directly reachable from live stack variables and global variables. After tracing, the unreachable objects are guaranteed to be dead and will be reclaimed. Although GC automates the memory management for developers, it requires additional resources, and may greatly slow down the execution of the programs. We will continue discussing this problem in Chapter 5.

## 2.3 Data-Parallel Systems

Data-parallel systems distribute the data across different workers, which operate on the data in parallel [35]. MapReduce [36] has inspired a body of research on distributed data-parallel computation, including Hyracks [105], Hadoop [102], Spark [137], or Dryad [50]. The MapReduce model has been extended [28] with Merge to support joins and adapted to support pipelining [32]. Yu *et al.* propose a programming model [133] for distributed

aggregation for data-parallel systems.

A number of high-level declarative languages for data-parallel computation have been proposed, including Sawzall [81], Pig Latin [78], SCOPE [23], Hive [109], and DryadLINQ [134]. All of these frameworks and languages except SCOPE and Dryad were implemented in JVM-based languages such as Java and Scala and thus can immediately benefit from the ITask optimization proposed in this dissertation. SCOPE and Dryad were implemented in C#, which also runs on top of a managed runtime system; we expect the ideas proposed in this thesis can also be adapted to optimize their applications.

# Chapter 3

# Memory-Related Performance Bug Description and Detection

As mentioned in Chapter 1, existing performance testing frameworks lack a general way to describe performance problems and effective test oracles to capture performance bugs. Finding general performance problems is infeasible, as it requires finding an alternative, more efficient computation in a possibly infinite solution space. We first narrow our focus onto a class of performance problems that have *observable symptoms*; their symptoms can be expressed by logical statements over a history of heap updates. These problems include, to name a few, memory leaks, inefficiently-used collections, unused return values, or loop-invariant data structures. One common axis these problems center around is that they manifest in the form of inefficient data usage, and their symptoms can be identified by capturing and comparing heap snapshots. To describe and detect such performance bugs, we propose a specification language named and an automated test oracle, and this chapter will focus on them.

Many performance problems are caused by redundant computations; although they are not

directly data-related, data inefficiencies can still be seen as a result of redundant computations. For example, one problem in Mozilla studied in [52] is due to the over-general implementation of an API—the `draw` method performs heavy-weight computations to draw high-quality images while the client only needs transparent ones. Even if the proposed technique cannot directly describe redundant computations, most fields of the resulting image object are never used; these redundant data can be captured and reported.

## 3.1 ISL: Instrumentation Specification Language

In this section, we will start with an example to explain the major components in a typical ISL program. Then the formal syntax and instrumentation semantics will be provided.

### 3.1.1 ISL Overview

As the first step of performance debugging, the developer writes an ISL program to describe (1) symptoms of a performance problem (for which amplification is needed) and (2) counter-evidence that shows what is being tracked is not a performance problem (for which deamplification is needed). ISL explicitly models heap partitioning, history of heap updates, as well as collaborations between the collector and the mutator, allowing developers to easily write simple Java-like code to amplify and deamplify performance problems.

To illustrate, Figure 3.1 shows a sample ISL program that describes the symptom of memory leaks caused by unnecessary strings held in object arrays, as well as how their effects can be amplified. A typical ISL description consists of a set of constructs that specifies how heap updates can be tracked and where the tracking information should be stored, as well as a set of *events* that uses imperative constructs in regular Java to define actions to be taken when the events are triggered.

```
1   Context ArrayContext {
2       sequence = "*.main,*";
3       type = "java.lang.Object[]";
4   }
5
6   Context TrackingContext {
7       sequence = "*.main,*";
8       path = ArrayContext;
9       type = "java.lang.String";
10  }
11
12  History UseHistory {
13      type = "boolean";
14      size = UserParameter;  // Defined by the user.
15  }
16
17  Partition P {
18      kind = all;
19      history = UseHistory;
20  }
21
22  TObject MyObj {
23      include = TrackingContext;
24      partition = P;
25      instance boolean useFlag = false;  // Instance field.
26  }
27
28  Event on_rw(Object o, Field f, Word w1, Word w2) {
29      o.useFlag = true;
30      deamplify(o);
31  }
32
33  Event on_reachedOnce(Object o) {
34      UseHistory h = getHistory(o);
35      h.update(o.useFlag);
36      if (h.isFull() && !h.contains(true))
37          amplify(o);
38      o.useFlag = false;
39  }
```

Figure 3.1: An ISL program for detecting memory leaks.

**Context.** Many dynamic analyses do not need to track every single detail of the execution. For different performance problems, the developer may focus on different aspects of the execution, such as the behavior of an object when it is created under a certain calling context (e.g., control-related) or referenced by a certain data structure (e.g., data-related). A `Context` construct is designed for the developer to express the scope of the objects of interest. `Context` has three properties: `sequence`, `type`, and `path`, each of which defines a constraint in a different aspect. They can be used to specify, respectively, the calling context, the type, and the reference path for the objects to be tracked. For example, context `ArrayContext` narrows the scope of the heap to objects of type `java.lang.Object[]` created under calling contexts that match string "`*.main, *`", where the asterisk (`*`) is used as a wildcard. This string matches all call chains that start with an invocation of method "`*.main`".

Next, `ArrayContext` is used to define the reference path in a new context `TrackingContext`, which imposes the constraint that we are only interested in the `java.lang.String` objects that are (1) created under context "`*.main, *`" (i.e., defined by `sequence`), and (2) reachable directly from the array objects created under the same calling context in the object graph (i.e., defined by `path`). Note that this context unifies two orthogonal object naming schemes (i.e., call-chain-based and access-path-based), providing much flexibility for the developer to narrow down interesting objects.

**History.** Since we target heap-related performance problems, their symptoms can often be identified by comparing old and new values in the tracked objects. A `History` construct is such an abstraction that models the heap update history (i.e., in an execution window) that needs to be tracked on the objects of interest. A `History` has two important properties: the type of a history element (i.e., `type`) and the length of the execution window tracked by the history (i.e., `size`). The element type has to be a primitive type because tracking a program execution often requires collaborative work between the mutator and the collector; creating tracking objects during a GC is not allowed. The length of the execution window specifies

how many (user-defined) state updates can be recorded in the history. The recorded updates will be used to determine whether a symptom is seen. A history has to be attached to a partition, allowing for the tracking of heap updates at different heap abstraction levels. We provide a few functions to manipulate a history; these functions will be discussed shortly.

**Partition.** Different analyses may need to collect tracking information at different abstraction levels. For example, analyses that report information regarding allocation sites can maintain a piece of tracking information for each allocation site, while analyses designed to identify typestate bugs have to keep per-object tracking information. A `Partition` construct can be used to define the partitioning of the heap needed by an analysis. Properties `kind` and `history` specify, respectively, how the heap should be partitioned and what history should be attached to each heap partition. In other words, one history instance (as defined in `History`) will be created and maintained for each partition of the heap defined by `kind`. `kind` can have five different values—`all`, `context[`$i$`]`, `alloc`, `type`, and `single`—which specify heap abstractions with increasing levels of granularity.

In particular, `single` means that all heap objects will be in the same partition, and therefore will share the same tracking information (i.e., a history instance). `type` and `alloc` specify that our runtime system will create one history instance per Java type and per allocation site, respectively; `context[`$i$`]` informs the runtime system to create one history instance for each allocation site executed under a distinct calling context; since each allocation site can have an extremely large number of distinct contexts, we allow the developer to define a (statically-fixed) integer number $i$ to limit the number of contexts for each allocation site; an encoding function (described shortly) is used to map a full context to a number in `[0,` $i$ `- 1]`; this option can be used to enable context-sensitive tracking of certain heap properties; finally, `all` means that one history instance will be created per heap object.

**TObject.** `TObject` defines the type of objects to be tracked, using two properties, `include` and `partition`, that specify the tracking context and the heap partitioning, respectively.

In our example, our system tracks objects under the context `TrackingContext` and creates a history for each tracked object. In addition, each tracked object has an instance field `useFlag`, which stores object-local state information necessary for identifying the symptom. The keyword `instance` can be used to declare (primitive-typed) per-object metadata information at runtime.

**Event.** At the center of an ISL program is a set of `Event` constructs that define what to do when important events occur on tracked objects. ISL supports seven different events: `read`, `write`, `rw`, `call`, `alloc`, `reached`, and `reachedOnce`. The first five are regular mutator events while the last two are GC events. `reached` is triggered every time an object is reached during a GC object graph traversal. Since an object may be reached multiple times (through different references), we use `reachedOnce` to define actions that need to be taken only once on the object during each GC—`reachedOnce` is triggered only at the first time the object is reached. An event construct takes a few parameters exposing the related run-time values at the event. For example, in the `rw` event, we have access to the base object, the field being accessed, and the value being read/written. `Word` is a special type representing a 32-bit value (regardless of its Java type). If this event reads/writes a 64-bit value, this value will be broken into two parts $w_1$ (i.e., the high 32 bits) and $w_2$ (i.e., the low 32 bits).

In our example, the events `rw` and `reachedOnce` specify the symptom of a memory leak and when to amplify its effect. Once a tracked object $o$ is used (i.e., read or written), we set its `useFlag` true. Since we see a use of the object, we call a library method `deamplify` to cancel all the space penalty previously added to $o$. When $o$ is reached in a `reachedOnce` event, the history associated with $o$'s partition is updated with $o.useFlag$. Since the history is defined to track only the most recent `UserParameter` [1] updates, the `update` operation will add the current boolean value and discard the oldest value from the history (if all of the `UserParameter` slots have been taken). Finally, if the history has a full record of

[1] `UserParameter` is defined by the user.

`UserParameter` updates (i.e., `isFull` returns `true`) and the record does not contain any `true` value, object $o$ is stale and thus we add space penalty to amplify the staleness effect (i.e., time since its last use) by calling the library method `amplify`.

### 3.1.2 ISL Syntax

Figure 3.2 shows the core syntax of ISL. An ISL program is a set of contexts, histories, partitions, tracked objects, and events. Since most of these productions are defined in expected ways, we discuss only a few interesting ones.

| | | | |
|---|---|---|---|
| Context IDs | *cxtID* | $\in$ | `StringLiterals` |
| Calling contexts | *mSeq* | $\in$ | `StringLiterals` |
| Class names | *tStr* | $\in$ | `StringLiterals` |
| History IDs | *hID* | $\in$ | `StringLiterals` |
| Partition IDs | *ptID* | $\in$ | `StringLiterals` |
| TObject IDs | *oID* | $\in$ | `StringLiterals` |
| Field names | *varID* | $\in$ | `StringLiterals` |
| Parameter Names | *oName* | $\in$ | `StringLiterals` |
| Contexts | $c$ | ::= | `Context` *cxtID* { *cb* } |
| Context bodies | $cb$ | ::= | `sequence` = *mSeq*; *cb* \| `type` = *tStr*; *cb* |
| | | \| | `path` = *cxtID*; *cb* \| $\epsilon$ |
| Histories | $h$ | ::= | `History` *hID* { *hb* } |
| History bodies | $hb$ | ::= | `type` = *tStr*; *hb* \| `size` = `int`; *hb* \| $\epsilon$ |
| Partitions | $p$ | ::= | `Partition` *ptID* { *pb* } |
| Partition bodies | $pb$ | ::= | `kind` = *pk*; *pb* \| `history` = *hID*; *pb* \| $\epsilon$ |
| Partition kinds | $pk$ | ::= | `single` \| `type` \| `alloc` \| `context[int]` \| `all` |
| Tracked objects | $o$ | ::= | `TObject` *oID* { *tb* } |
| TObject bodies | $tb$ | ::= | `include` = *cxtID*; *tb* \| `partition` = *ptID*; *tb* |
| | | \| | *fd tb* \| $\epsilon$ |
| Field decls | $fd$ | ::= | `instance` *tStr varID*; *fd* \| $\epsilon$ |
| Events | $e$ | ::= | `Event` *ek* ( *pl* ) { *eb* } |
| Event kinds | $ek$ | ::= | `on_read` \| `on_write` \| `on_rw` \| `on_call` |
| | | \| | `on_alloc` \| `on_reached` \| `on_reachedOnce` |
| Event parameters | $pl$ | ::= | *oID oName*, ... |
| Event bodies | $eb$ | ::= | `amplify` (*oName*); *eb* \| `deamplify` (*oName*); *eb* |
| | | \| | ... \| $\epsilon$ |
| ISL program | $s$ | ::= | $c\ s \mid p\ s \mid o\ s \mid e\ s \mid h\ s \mid \epsilon$ |

Figure 3.2: Abstract syntax of ISL.

The `path` field of a context $c_1$ is defined by another context $c_2$, which specifies that the objects constrained by $c_1$ has to be directly reachable from the objects constrained by $c_2$ on the object graph. If we wish to specify transitive reachability, e.g., via a path of $n$ nodes, we can define $n$ contexts from $c_2$ to $c_{n+1}$, each constraining a node on the path.

A `History` object supports the following seven operations: `void update(T)`, `T get(int)`, `boolean contains(T)`, `int length()`, `boolean isFull()`, `boolean containsSameValue()`, and `History subHistory(int, int)`. The first five operations are defined in expected ways. `containsSameValue` returns true if all elements of the history have the same value, and `subHistory` converts a sub-region of the current history into another history instance.

ISL supports seven different events, and each type of event has a different parameter list, shown in Figure 3.3. `Field` represents an object field, which contains the offset and ID of the field (i.e., from 0 to the total number of fields - 1). `Method` contains the information of the invoked method.

```
        on_read  (Object o, Field f, Word w1, Word w2);
       on_write  (Object o, Field f, Word w1, Word w2);
          on_rw  (Object o, Field f, Word w1, Word w2);
        on_call  (Object o, Method m, Word return1, Word return2);
       on_alloc  (Object o);
    on_reached   (Object o, Field f);
on_reachedOnce   (Object o);
```

Figure 3.3: Parameter lists of ISL events.

### 3.1.3 Instrumentation Semantics

We first discuss the basic instrumentation semantics and Section 3.2 will extend the semantics to cover library call `amplify` and `deamplify`. Figure 3.4 shows a list of semantic domains that will be used in the instrumentation rules. The first five domains (i.e., the domains of allocation sites, types, fields, variables, and partition entities) are defined straightforwardly.

| | | | | | |
|---|---|---|---|---|---|
| Allocation sites | $l$ | | | $\in$ | $\mathbb{A}$ |
| Types | $t$ | | | $\in$ | $\mathbb{T}$ |
| Fields | $f$ | | | $\in$ | $\mathbb{F}$ |
| Variables | $v$ | | | $\in$ | $\mathbb{V}$ |
| Partition entities | $e$ | | | $\in$ | $\mathbb{E}$ |
| Context instances | $c$ | $::=$ | $\langle ms, type \rangle$ | $\in$ | $\mathbb{C}$ |
| Context reference relations | $\gamma$ | $::=$ | $\langle c_1, c_2 \rangle$ | $\in$ | $\mathbb{C} \times \mathbb{C}$ |
| History instances | $h$ | | | $\in$ | $\mathbb{H}$ |
| History stores | $\tau$ | | | $\in$ | $\mathbb{E} \cup \{\perp\} \rightarrow \mathbb{H}$ |
| Objects | $\hat{o}$ | $:=$ | $o^{\langle c,h \rangle}$ | $\in$ | $\Phi$: $\mathbb{O} \times \mathbb{C} \times \mathbb{H}$ |
| Environments | $\rho$ | | | $\in$ | $\mathbb{V} \rightarrow \Phi \cup \mathbb{V} \cup \{\perp\}$ |
| Heaps | $\sigma$ | | | $\in$ | $\Phi \times \mathbb{F} \rightarrow \mathbb{V} \cup \{\perp\}$ |
| Call stacks | $s$ | | | $\in$ | $\mathbb{N} \rightarrow \mathbb{M}$ |

Figure 3.4: ISL semantic domains.

At runtime, a context instance is maintained for each `Context` construct. Each context has a method sequence *ms* and a type constraint *type*. If the `path` field of context $c_1$ references context $c_2$, a pair $\langle c_2, c_1 \rangle$ exists in the context reference relation $\gamma$. Among all context constructs in an ISL program, there is a main context, denoted as $\hat{c} \in \mathbb{C}$, which defines the `include` field of `TObject`. We maintain a history store $\tau$ that maps each program entity $e$ in domain $\mathbb{E} \cup \{\perp\}$ to a history instance. The type of $e$ depends on the `kind` field of the partition construct. For example, $e$ is $\perp$, a type, an allocation site, an allocation site plus an encoded context, and a heap object if `kind` is `single`, `type`, `alloc`, `context`, and `all`, respectively. Each object in our system is a base heap object $o$ labeled with a pair of a context $c$ to which $o$ belongs and a history instance $h$. Environment $\rho$ maps each variable to its runtime (primitive-typed or reference-typed) value. Heap $\sigma$ maps each object field to the value stored in the field. $\tau$, $\rho$ and $\sigma$ are augmented with a special symbol $\perp$, which denotes a `null` value for pointers. Finally, a call stack $s$ is a map that maps a natural number $\in \mathbb{N}$ to a method $\in \mathbb{M}$ on the stack.

The goal of the instrumentation is to identify objects specified by the `TObject` construct and invoke user-defined handlers on them upon events. An object is a specified object if the following two conditions hold: (1) it is created under the main context $\hat{c}$; and (2) if a

context reference chain of the form $c_1, c_2, \ldots, \hat{c}$ exists for $\hat{c}$, this reference chain must have been validated. While condition (1) can be easily verified at each allocation site (e.g., by checking whether the allocation type and the current call stack match the specification in the main context), reference path validation can only be done when GC traverses the heap. The validation algorithm will be discussed shortly in Rule GC. However, GC may occur after many mutator events; if we do not execute handlers until the context reference path has been validated, we may potentially lose important information of the tracked objects. We solve the problem by relaxing our requirement—we check only condition (1) to decide whether to execute a handler for a mutator event; both condition (1) and (2) are checked to determine whether a GC event handler should be executed. Because `amplify` is invoked only when the symptom specification is checked in GC, this handling will not mistakenly amplify objects whose reference relationships do not hold.

In Figure 3.5, we define the auxiliary functions which will be used in the semantics rules. In the definition of function $match(c, s, t)$, $\preceq$ is a subtype relation, and $a \vdash_r b$ if string $b$ is represented by the regular expression $a$. This function returns true if the specified type in $c.type$ is $t$ or a supertype of $t$, and the regular expression in $c.ms$ represents the sequence of the method invocations on the stack. We design the function *encode* based on the computation of the probabilistic calling context proposed in [15]: suppose $s_k$ is the $k$-th method on the stack; the encoding function *encode* is a recursive function that does the computation based on the encoded value of the prefix of $s$ up to method $s_{k-1}$. Despite its simplicity, the function can be implemented efficiently and has been demonstrated to have a small conflict rate.

$$
\begin{array}{rcl}
match(c, s, t) & = & \begin{cases} true & \text{if } t \preceq c.type \wedge c.ms \vdash_r s \\ false & \text{otherwise} \end{cases} \\
encode(s, i) & = & encode(s_{|s|-1}) \texttt{ mod } i \\
encode(s_k) & = & 3 \times encode(s_{k-1}) + methodID(s_k)
\end{array}
$$

Figure 3.5: Auxiliary functions.

Figure 3.6-3.9 define the instrumentation semantics at four important operations: allocation

sites, object reads, object writes, and reference traversals in GC, respectively. A judgment of the form $\delta$, $\rho$, $\tau$, $\sigma$, $s \Downarrow \rho'$, $\tau'$, $\sigma'$ starts with operation $\delta$, which is followed by environment $\rho$, history store $\tau$, heap $\sigma$, and call stack $s$. The execution of $\delta$ terminates with a final environment $\rho'$, history store $\tau'$, and heap $\sigma'$. Stack $s$ can only be modified by method calls and returns; rules at calls and returns are usual and thus omitted.

$$\hat{o}.c = \begin{cases} c & \text{if } \exists\, c \in \mathbb{C} : match(c, s, t) \\ \bot & \text{otherwise} \end{cases} \quad h = \begin{cases} e.h & \text{if } \exists\, \langle e, h\rangle \in \tau \\ fresh & \text{otherwise} \end{cases}$$

$$\hat{o}.e = e \qquad e.h = h \qquad \rho' = \rho[a \mapsto \hat{o}]$$

$$\tau' = \begin{cases} \tau \cup \{\langle \bot, h\rangle\} & \text{if } c = \hat{c} \ \wedge\ \texttt{kind} = \texttt{single} \\ \tau \cup \{\langle t, h\rangle\} & \text{if } c = \hat{c} \ \wedge\ \texttt{kind} = \texttt{type} \\ \tau \cup \{\langle l, h\rangle\} & \text{if } c = \hat{c} \ \wedge\ \texttt{kind} = \texttt{alloc} \\ \tau \cup \{\langle l_{encode(s,i)}, h\rangle\} & \text{if } c = \hat{c} \ \wedge\ \texttt{kind} = \texttt{context[}i\texttt{]} \\ \tau \cup \{\langle o, h\rangle\} & \text{if } c = \hat{c} \ \wedge\ \texttt{kind} = \texttt{all} \\ \tau & \text{otherwise} \end{cases}$$

$$\sigma' = \sigma[\lambda f.(\hat{o}.f \mapsto \bot)]$$

$$on\_alloc(\hat{o}), \rho', \tau', \sigma', s \Downarrow \rho'', \tau'', \sigma''$$

$$\rho''' = \begin{cases} \rho'' & \text{if } c = \hat{c} \\ \rho' & \text{otherwise} \end{cases} \quad \tau''' = \begin{cases} \tau'' & \text{if } c = \hat{c} \\ \tau' & \text{otherwise} \end{cases} \quad \sigma''' = \begin{cases} \sigma'' & \text{if } c = \hat{c} \\ \sigma' & \text{otherwise} \end{cases}$$

$$\frac{}{a = \text{new } \mathrm{t}^l, \rho, \tau, \sigma, s \Downarrow \rho''', \tau''', \sigma'''} \quad \text{(New)}$$

Figure 3.6: Instrumentation semantics of rule New.

Rule New defines our instrumentation at each allocation site. If the current call stack $s$ and the type of object $t$ match a specified context $c$, we store $c$ in the object $\hat{o}$. The definition of function *match* can be found in Figure 3.5. Each tracked object is associated with one partition entity, and if the partition entity has no history instance, we create a $fresh$ history for it. If the context $c$ that matches the call stack is the main context $\hat{c}$, we need to update the history store $\tau$; the update is done based on the definition of `kind` in the `TObject` construct. The most interesting case here is to update $\tau$ for `kind = context[`$i$`]`. In this case, we create $i$ history instances for each allocation site, where $i$ is given by the developer as a parameter. We use a function *encode*, which is defined in Figure 3.5, to encode the current call stack into an integer value, which is then mapped into a number in [0, $i$ - 1]. Eventually, the user-defined event `on_alloc` is executed. Note that if there is no matching

context or the current context is not the main context, we do not store the history and do not invoke the event. This can be seen through the last three selective state updates in the rule. It is important to note that our system tracks objects that belong to each context (i.e., not just the main context) declared in an ISL program—objects that belong to a non-main context will be used to identify the reference path leading to the objects that belong to the main context, and therefore, they also need to be tracked. However, we do not execute an event if the context of the tracked object is not the main context.

$$\frac{\begin{array}{c} w = \sigma(\rho(b).f) \qquad \rho' = \rho[a \mapsto w] \\ \mathit{on\_read}(\rho(b), f, \mathit{h32}(w), \mathit{l32}(w)), \rho', \tau, \sigma, s \Downarrow \rho'', \tau', \sigma' \\ \mathit{on\_rw}(\rho(b), f, \mathit{h32}(w), \mathit{l32}(w)), \rho'', \tau', \sigma', s \Downarrow \rho''', \tau'', \sigma'' \\ \rho'''' = \begin{cases} \rho''' & \text{if } \rho(b).c = \hat{c} \\ \rho' & \text{otherwise} \end{cases} \\ \tau''' = \begin{cases} \tau'' & \text{if } \rho(b).c = \hat{c} \\ \tau & \text{otherwise} \end{cases} \qquad \sigma''' = \begin{cases} \sigma'' & \text{if } \rho(b).c = \hat{c} \\ \sigma & \text{otherwise} \end{cases} \end{array}}{a = b.f, \rho, \tau, \sigma, s \Downarrow \rho'''', \tau''', \sigma'''} \quad (\textsc{Load})$$

Figure 3.7: Instrumentation semantics of rule Load.

Figure 3.7 defines the semantics of object reads. In rule Load, we need to invoke both events `on_read` and `on_rw` if the base object (i.e., $\rho(b)$) is tracked (i.e., its context $\rho(b).c$ is the main context). Again, we use three selective updates to determine whether the post states can be affected by the two events. Functions *h32* and *l32* return the high 32 bits and the low 32 bits of a value. If the value only has 32 bits, *l32* returns 0. Rule Store in Figure 3.8 is defined in a similar manner.

$$\frac{\begin{array}{c} w = \rho(a) \qquad \sigma' = \sigma[\rho(b).f \mapsto w] \\ \mathit{on\_write}(\rho(b), f, \mathit{h32}(w), \mathit{l32}(w)), \rho, \tau, \sigma', s \Downarrow \rho', \tau', \sigma'' \\ \mathit{on\_rw}(\rho(b), f, \mathit{h32}(w), \mathit{l32}(w)), \rho', \tau', \sigma'', s \Downarrow \rho'', \tau'', \sigma''' \\ \rho''' = \begin{cases} \rho'' & \text{if } \rho(b).c = \hat{c} \\ \rho & \text{otherwise} \end{cases} \\ \tau''' = \begin{cases} \tau'' & \text{if } \rho(b).c = \hat{c} \\ \tau & \text{otherwise} \end{cases} \qquad \sigma'''' = \begin{cases} \sigma''' & \text{if } \rho(b).c = \hat{c} \\ \sigma' & \text{otherwise} \end{cases} \end{array}}{b.f = a, \rho, \tau, \sigma, s \Downarrow \rho''', \tau''', \sigma''''} \quad (\textsc{Store})$$

Figure 3.8: Instrumentation semantics of rule Store.

In Figure 3.9, the last rule GC defines how the analysis states are updated when object $\hat{o}_2$ is reached from field $f$ of object $\hat{o}_1$ during a GC. Here we introduce two new domains: $\phi$ and $\mu$, where $\phi$ denotes the set of objects already traversed in the current GC and $\mu$ denotes the set of objects on the reference path specified in the main context. We update $\mu$ with $\hat{o}_2$ if one of the two conditions holds: (1) $\hat{o}_1$ is already in $\mu$ and there is a context reference between $\hat{o}_1.c$ and $\hat{o}_2.c$ specified by the `path` field (as indicated by $\langle \hat{o}_1.c, \hat{o}_2.c \rangle \in \gamma$); or (2) $\hat{o}_1$ does not have a context but $\hat{o}_2$ has one, and there is no reference path specified for $\hat{o}_2$ in the program; this indicates $\hat{o}_2$ is the starting point of the specified reference path. Eventually, we execute event `reachedOnce` if (1) the context of $\hat{o}_2$ is the main context, (2) object $\hat{o}_2 \in \mu$ (which indicates that this edge is indeed on the reference path), and (3) this is the first time $\hat{o}_2$ is reached (which can be seen by $\hat{o}_2 \notin \phi$). If conditions (1) and (2) are satisfied but condition (3) is not, we execute `reached` instead. Currently, the language supports specification of only one main context and one `TObject`. While it is easy to extend the language to support multiple main contexts and `TObject` constructs, we find that the language is already expressive enough to define and amplify various kinds of performance problems.

$$\frac{\begin{array}{c} \mathit{on\_reachedOnce}(\hat{o}_2), \sigma \Downarrow \sigma' \qquad \mathit{on\_reached}(\hat{o}_2, f), \sigma \Downarrow \sigma'' \\ \mu' = \begin{cases} \mu \cup \{\hat{o}_2\} & \text{if } \langle \hat{o}_1.c, \hat{o}_2.c \rangle \in \gamma \quad \wedge \quad \hat{o}_1 \in \mu \\ \mu \cup \{\hat{o}_2\} & \text{if } \hat{o}_1.c = \bot \quad \wedge \quad \hat{o}_2.c \neq \bot \quad \wedge \quad \nexists\, \langle c', \hat{o}_2.c \rangle \in \gamma,\, c' \in \mathbb{C} \\ \mu & \text{otherwise} \end{cases} \\ \sigma''' = \begin{cases} \sigma' & \text{if } \hat{o}_2 \in \mu \quad \wedge \quad \hat{o}_2.c = \hat{c} \quad \wedge \quad \hat{o}_2 \notin \phi \\ \sigma'' & \text{if } \hat{o}_2 \in \mu \quad \wedge \quad \hat{o}_2.c = \hat{c} \quad \wedge \quad \hat{o}_2 \in \phi \\ \sigma & \text{otherwise} \end{cases} \end{array}}{\mathit{processEdge}(\hat{o}_1, f, \hat{o}_2), \sigma, \phi, \mu \Downarrow \sigma''', \phi \cup \{\hat{o}_2\}, \mu'} \quad \text{(GC)}$$

Figure 3.9: Instrumentation semantics of rule GC.

## 3.2 Amplification and Deamplification

The goal of amplification is to "blow up" the effect of performance problems, especially small performance bugs, so that they can be easily captured during testing. Amplification of a performance problem is done on the objects that satisfy the symptom specification of the problem. Method `amplify`($o$) takes an object $o$ as input and increases the penalty counter associated with $o$ by the size of the object. Deamplification is performed when the counter-evidence of the problem is found. When `deamplify` is executed on $o$, we set $o$'s penalty counter back to zero.

$$VSO = \frac{Sum_{penalty} + Size_{live\ heap}}{Size_{live\ heap}}$$

Figure 3.10: The definition of VSO.

During each GC, we modify the reachability analysis to compute the sum of the penalty counters of all live objects ($Sum_{penalty}$). $Size_{live\ heap}$ is the live heap size. We define VSO (i.e., short for virtual space overhead) in Figure 3.10. It simulates the (peak) overhead of an execution that had the real memory penalty been used. VSO is a metric that measures the severity of performance problems relative to the amount of memory consumed by a program—the same problem is more serious if the program only consumes small amounts of memory. Since $Sum_{penalty}$ has a time component (e.g., objects that keep satisfying the symptom specification will make $Sum_{penalty}$ keep growing), a program may have increasingly large VSOs as it executes. Eventually, the VSOs computed at all GCs are compared and the maximal VSO is reported to indicate the severity of the performance problems. We will explain how to update penalty counters during GC in Section 4.2.2.

## 3.3 More ISL Programs

In addition to the memory leak example given in Section 3.1, Figure 3.11, Figure 3.12, and Figure 3.13 show, respectively, the (simplified) ISL descriptions for amplifying three other types of performance problems: (1) extensive creation of objects with invariant data contents, (2) under-utilized Java hash maps, and (3) unused return values. While there are multiple ways to detect these problems, we show an amplification approach here to demonstrate the generality of ISL.

### 3.3.1 Allocation sites creating invariant objects.

The symptom is that certain allocation sites keep creating objects with identical data values. The goal of the ISL program in Figure 3.11 is to identify these allocation sites and then penalize their objects. The partitioning is based on allocation site, which means all objects created by the same allocation site share one history instance. We define the history as a list of `long` values (as `long` can express values of any type) over an execution window of `UserParameter` updates. Here we are interested in values stored in each primitive-typed field of an object. Since our `History` construct can model only scalar values, we *linearize* histories of all fields into one single history whose size is `UserParameter` $*$ `MAX_NUM_FIELDS` where `MAX_NUM_FIELDS` is the maximal number of fields in a class (e.g., 100 is a reasonably large number). In other words, for a field with index $i$, elements from $i *$ `UserParameter` to $(i+1) *$ `UserParameter` in the history model the updates of the field.

In the `on_write` event, we first obtain the sub-history for the field being accessed from the history associated with the object, and then update the sub-history with the value to be written into the field. If a different value is seen in the sub-history, the object is no longer invariant and we cancel all the penalty previously added to the object. In the

```
1   History UpdateHistory {
2       type = "long";
3       // UserParameter is defined by the user.
4       size = UserParameter * MAX_NUM_FIELDS;
5   }
6
7   Partition P {
8       kind = alloc;
9       history = UpdateHistory;
10  }
11
12  TObject MyObj {
13      partition = P;
14  }
15
16  Event on_write(Object o, Field f, Word w1, Word w2) {
17      if (!f.isPrimitive())
18          return;
19      long l = combine(w1, w2);
20      UpdateHistory allHis = getHistory(o);
21      UpdateHistory fieldHis = allHis.subHistory(f.index() *
22                                                 UserParameter,
23                                                 UserParameter);
24      fieldHis.update(l);
25      if (fieldHis.isFull() && !fieldHis.containsSameValue())
26          deamplify(o);
27  }
28
29  Event on_reachedOnce(Object o) {
30      UpdateHistory allHis = getHistory(o);
31      int i = 0;
32      for (; i < allHis.length(); i += UserParameter) {
33          UpdateHistory fieldHis = allHis.subHistory(
34                                              i, UserParameter);
35          if (!fieldHis.isFull()
36              return;
37          if (!fieldHis.containsSameValue())
38              return;
39      }
40      amplify(o);
41  }
```

Figure 3.11: An ISL program for detecting allocation sites creating invariant objects.

`on_reachedOnce` event, we check two conditions for each field: (1) the sub-history has a full record and (2) all values in the sub-history are the same. If both conditions hold for all the fields, we start penalizing the object. Note that we can make amplification even finer-grained by using a context-based partitioning (e.g., `kind = context[`$i$`]`), which will enable us to identify certain contexts that are more likely to create identical objects than others and only amplify objects created under such contexts.

### 3.3.2 Under-utilized Java maps.

The goal of the ISL program in Figure 3.12 is to penalize Java hash maps that take a large memory space but contain only a very small number of valid elements. Using two contexts, we narrow our focus onto `java.util.Map.Entry` arrays that are referenced by `java.util.HashMap` object. We use a per-object partitioning with a history that tracks the most recent `UserParameter` *utility rates* of each array. A utility rate of an array is defined as the ratio between the number of non-null elements and the total length of the array. Every time a tracked array is traversed in the GC, we compute its utility rate and update the history. If the rate is greater than a user-provided threshold value (i.e., 0.3), the array is in good health and thus we cancel all its previous penalty. If the history has a full record in which all elements are less than or equal to 0.3, we start penalizing this array.

### 3.3.3 Never-used return objects.

In Figure 3.13, our goal is to detect and penalize objects that are never used after being returned by a method. Such objects are often indicators of wasteful computation done during the method invocation. Amplifying this problem is similar to amplifying a memory leak. The only difference is that we use a boolean instance field `returned` in each object to indicate whether the object has been returned by a method call. This flag is set in event `on_call` if

```
Context MapContext {
    type = "java.util.HashMap";
}

Context TrackingContext {
    path = MapContext;
    type = "java.util.Map.Entry[]";
}

History UtilityRates {
    type = "double";
    size = UserParameter;  // Defined by the user.
}

Partition P {
    kind = all;
    history = UtilityRates;
}

TObject MyObj {
    include = TrackingContext;
    partition = P;
}

Event on_reachedOnce(Object o) {
    UtilityRates rates = getHistory(o);
    Object[] array = (Object[]) o;
    int numNonNull = 0;
    for (Object element : array) {
        if (element != null)
            ++ numNonNull;
    double rate = ((double) numNonNull) / array.length;
    rates.update(rate);
    if (rate > 0.3)
        deamplify(o);
    else if (rates.isFull()) {
        for (int i = 0; i < UserParameter; ++ i) {
            if (rates.get(i) > 0.3)
                return;
        amplify(o);
    }
}
```

Figure 3.12: An ISL program for detecting under-utilized Java hash maps.

a method call returns an object.

```
1   History UseHistory {
2       type = "boolean";
3       size = UserParameter;  // Defined by the user.
4   }
5
6   Partition P {
7       kind = all;
8       history = UseHistory;
9   }
10
11  TObject MyObj {
12      partition = P;
13      instance boolean returned = false;  // Instance field.
14  }
15
16  Event on_rw(Object o, Field f, Word w1, Word w2) {
17      if (o.returned) {
18          getHistory(o).update(true);
19          deamplify(o);
20      }
21  }
22
23  Event on_call(Object o, Method m, Word return1, Word return2) {
24      if (m.getReturnType().isReference()) {
25          Object obj = addressToObject(return1, return2);
26          obj.returned = true;
27      }
28  }
29
30  Event on_reachedOnce(Object o) {
31      UseHistory history = getHistory(o);
32      if (o.returned && history.isFull() && !h.contains(true))
33          amplify(o);
34  }
```

Figure 3.13: An ISL program for detecting never-used return objects.

## 3.4 Summary and Interpretation

To define and detect memory-related performance problems, we have proposed an event-based domain-specific language, called instrumentation specification language (ISL). In summary, real-world developers can benefit from the following features of ISL.

First, by exploiting a combination of instrumentation and runtime system support, ISL provides a unified way of expressing various memory-related performance problems, which were previously defined separately. Developers can quickly build detectors for different types of performance problems using ISL instead of building everything from scratch; the amount of the work of developing detectors can be significantly reduced.

Second, ISL has a Java-like syntax and thus is easy to use in real-world development; amplification for even complicated performance problems can often be specified using only a few lines of ISL code. In contrast, had a JVM been modified manually to implement amplification, developing a testing tool for each problem would have needed modification of thousands of lines of code, which can take a skillful programmer several months or even longer.

Third, while an ISL program penalizes individual objects, it is easy to attribute penalties to a data structure as a whole by defining reference-path contexts. For example, although array objects are penalized in the detection of under-utilized containers, reference-path contexts are used to connect those low-level arrays with high-level maps, stacks, queues, lists, and sets, and hence, our detector is able to report not only individual objects but also logical data structures.

# Chapter 4

# PerfBlower: A Performance Testing Framework

Based on the technique proposed in the previous chapter, we build a general performance testing framework, PerfBlower, which can "blow up" the effect of small performance problems so that they can be easily captured during testing. In this chapter, we will first present the overview of PerfBlower's design and implementation. And then we will propose a novel algorithm that collects useful diagnostic information at runtime. After that the experimental results will be shown and the limitations of our framework will be discussed.

## 4.1 Overview

Figure 4.1 depicts the architecture of PerfBlower. It has two major components: an ISL compiler and a runtime system. The compiler parses an ISL program provided by the developer and automatically generates instrumentation code for a JVM. The JVM is built and a program to be tested is then executed on the modified VM. During the execution, the runtime

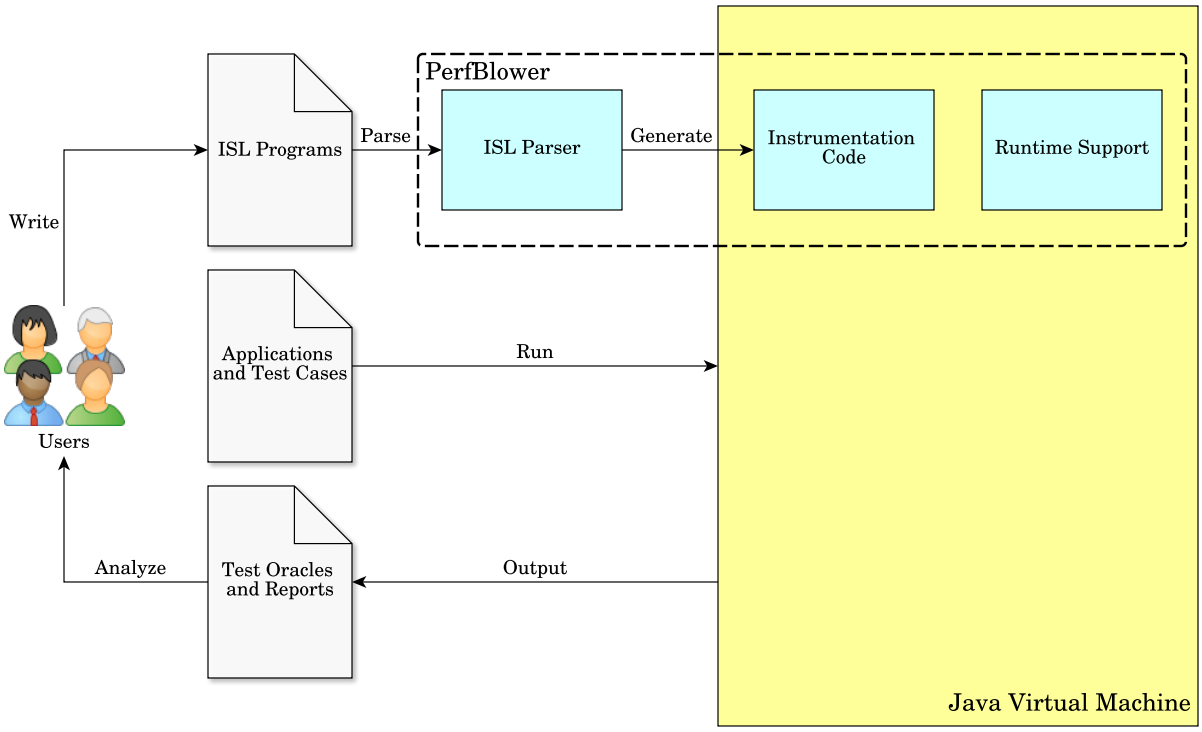


Figure 4.1: The architecture of PerfBlower.

system monitors the behavior of the program, checks symptom specifications, and performs amplification and deamplification. PerfBlower reports detailed diagnostic information for programs with high virtual overhead.

We have developed our ISL parser using ANTLR v4. During compilation, contexts are broken into a set of $\langle ms, type \rangle$ [1] pairs. A new `History` class is generated from a template based on the properties declared in the ISL program. An additional header space is requested per object to accommodate instance fields declared in `TObject`. Declaring many instance fields would create tremendous space overhead and thus developers are encouraged to declare as few fields as possible. Note that we do not generate a Java class for a `TObject` construct. The contexts used in the construct are checked at runtime to determine whether an event needs to be invoked. We do not typecheck the imperative statements in an event construct; these events are directly translated into Java methods that are invoked at various program points;

[1] *ms* and *type* mean a method sequence and a type constraint, respectively.

the generated methods will be typechecked when the instrumentation code is compiled.

The runtime system is built in Jikes Research Virtual Machine (RVM) [51], version 3.1.3. To implement the proposed technique, we have modified both the baseline compiler and the optimizing compiler in RVM [80], and additional logic is added to MarkSweep GC. PerfBlower supports simultaneous checking of multiple problems. This is done by assigning a unique ID to each ISL program specifying a problem. The ID will be passed (implicitly) as a parameter to `amplify`($o$)—it is written into each mirror object created for $o$ so that when the mirror path for $o$ is traversed and printed, the problem to which $o$ is related is reported as well.

## 4.2 Diagnostic Information Collection

When a test fails, it is important to provide highly-relevant diagnostic information that can help developers quickly find the root cause and fix the problem. In this chapter, we will first introduce heap reference path, which is an important piece of diagnostic information, and then propose a novel method to efficiently collect reference path information.

### 4.2.1 Heap Reference Paths

Evidence [122, 131] shows that heap reference paths leading to suspicious objects are very important information as they reveal contexts and data structures containing these objects. We will further explain how reference paths reveal the causes of the performance problems by example. Figure 4.2 shows a Java memory leak example from [12], and we add method `main` to the original program to trigger the performance problem. In method `main`, we first create a `Stack` $s$ (Line 32) and then allocate an `Object` *obj* (Line 33). When object *obj* is pushed onto the stack (Line 34), a reference to *obj* is stored in $s$'s field *elements* (Line 13).

After *obj* is popped off the stack (Line 35), *elements* still maintains an *obsolete reference* to *obj*, which prevents the garbage collector from reclaiming *obj*. In this case, since *obj* will

```
1   public class Stack {
2       private Object[] elements;
3       private int size = 0;
4       private static final int DEFAULT_INITIAL_CAPACITY = 16;
5
6       public Stack() {
7           // Allocation site 1.
8           elements = new Object[DEFAULT_INITIAL_CAPACITY];
9       }
10
11      public void push(Object e) {
12          ensureCapacity();
13          elements[size++] = e;
14      }
15
16      public Object pop() {
17          if (size == 0)
18              throw new EmptyStackException();
19          return elements[--size];
20      }
21
22      /**
23       * Ensure space for at least one more element, roughly
24       * doubling the capacity each time the array needs to grow.
25       */
26      private void ensureCapacity() {
27          if (elements.length == size)
28              elements = Arrays.copyOf(elements, 2 * size + 1);
29      }
30
31      public static void main(String args[]) {
32          Stack s = new Stack();  // Allocation site 2.
33          Object obj = new Object();  // Allocation site 3.
34          s.push(obj);
35          s.pop();  // obj becomes a leaking object.
36          ...  // Do something.
37      }
38  }
```

Figure 4.2: A Java program with memory leaks.

never be accessed, it becomes a leaking object. With the allocation site information, we can easily figure out that the leaking object is created at Line 33. However this is not enough for us to identify the root cause of the problem. A reference path leading to the problematic object *obj* is presented in Figure 4.3. From this reference path, we can clearly see that the obsolete object *obj* is referenced by the field of `Stack` *s*. With this piece of information, we can easily find the cause of the memory leak—*elements* holds an obsolete reference. The fix for this problem is simple: when an element is popped, null out its reference in *elements*.

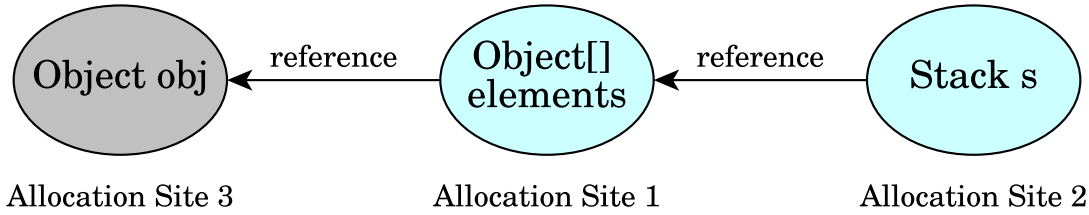


Figure 4.3: An example heap reference path.

### 4.2.2 Mirroring Reference Paths

As we discussed in the previous section, heap reference paths are very helpful for fixing performance bugs. However, it can be quite expensive for the runtime system to identify and report reference paths for objects. First, although the GC traverses all live heap objects, path information is not easy to obtain—to be scalable, the reachability analysis in the GC

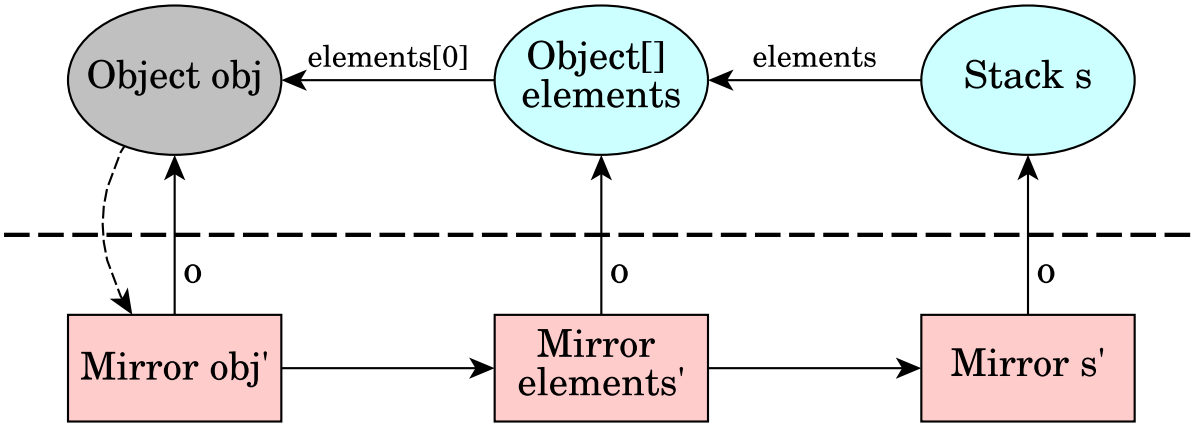


Figure 4.4: An example mirror path.

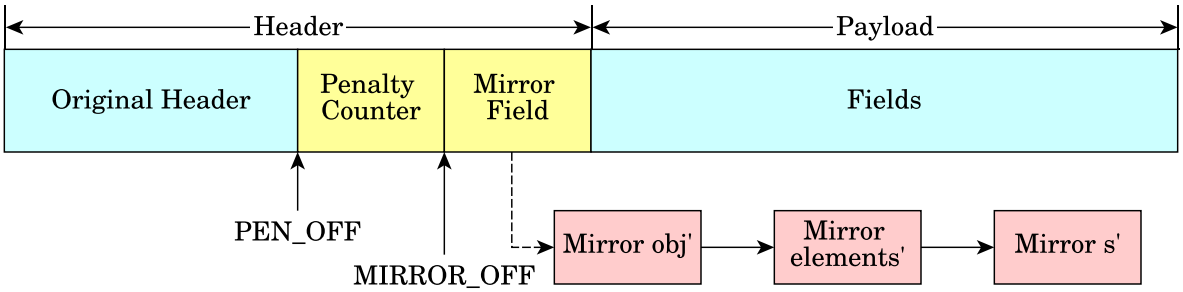


Figure 4.5: The layout of a tracked object.

does not record any backward information. In addition, reporting a reference path requires recording the source code information (e.g., allocation site) of each object on the path, which can introduce significant overhead.

We solve the problem by building a *mirror object chain* for object *obj* that reflects the major reference path leading to *obj*. Figure 4.4 shows an example mirror path. The mirror object of an original object *name* is annotated with *name*′. As such, to report object *obj*'s reference path, we only need to go forward to traverse its mirror chain (as apposed to going backward on the graph) and print information contained in the mirror objects. This eliminates the need to either modify the GC algorithm or increase the size of a regular object to store source code information. Our algorithm tracks only one single reference path for each suspicious object. We find it to be sufficient in most cases. After the execution finishes, allocation sites are ranked based on the total size of the memory penalties added to their objects; their related reference paths are reported as well.

Figure 4.5 shows the layout of a tracked object at runtime. We add two words in each

```
1  public class Mirror {
2      Object o;  // The actual object.
3      AllocSiteInfo info;  // The source code info.
4      Field fInfo;  // The information of the corresponding edge.
5      Mirror next;  // Points to the next mirror object.
6  }
```

Figure 4.6: The definition of the `Mirror` class.

object's header space to store (1) its penalty count (in numbers of bytes) and (2) a pointer pointing to the head of its mirror chain. Constants `PEN_OFF` and `MIRROR_OFF` are used to locate the offsets of these two locations upon accesses. Figure 4.6 shows the definition of the `Mirror` class. Each mirror object contains two outgoing edges, one pointing to the actual object it mirrors (i.e., via field $o$) and the other pointing to the next mirror object (i.e., via field `next`). The mirror object also contains the source code information of the object it mirrors as well as the field information of the corresponding reference edge in the original path. In the example mirror chain shown in Figure 4.4, field `fInfo` in object $s'$ and object $elements'$ record field $elements$ and $elements[0]$ respectively.

Algorithm 1 shows our algorithm for incrementally adding penalties and building mirror paths. The algorithm has three major procedures: PreGC, InGC, and PostGC. PreGC resets the penalty counter, and sets two mark values that will be used later in GC to mark objects (Lines 10–14). *lastMark* and *currMark*, which represent, respectively, the mark values used in the last and the current GC, alternate between 1 and 2. Procedure InGC traverses the object graph to build mirror reference paths (Lines 15–40). Because the object graph traversal cannot go backward, it is impossible to mirror a reference path with multiple edges in one single GC run. The basic idea of the incremental algorithm is *to mirror one edge on the path in each GC, starting from the one closest to the penalized object.* For example, to build the mirror path in Figure 4.4, we first mirror the edge going directly to object *obj* (i.e., $elements \xrightarrow{elements[0]} obj$) by creating an edge between $obj'$ and $elements'$. Next, we mark object *elements* with a special mark value so that *elements* will be recognized in the next GC and the edge between $s$ and *elements* can be mirrored. If object *obj* keeps being penalized (e.g., indicating that we are amplifying a true problem), a long mirror chain will be constructed and, hence, more complete path information will be reported.

During the graph traversal, if the object being reached happens to be the one to be penalized (Lines 17–34), we first increase the object's penalty counter by the size of the object (Lines 18

**Algorithm 1** Incrementally creating virtual penalties and building mirror paths.

```
1: global variables
2:     Map newRefEdges                          ▷ Reference edges to be added in the mirror chain.
3:     Set objToBeMirrored                      ▷ Objects to be mirrored.
4:     List mirrorObjCurrGC                     ▷ Mirror objects at the end of each mirror chain.
5:     List mirrorObjLastGC                     ▷ Chain-ending mirror objects found in the previous GC.
6:     long VP                                  ▷ The accumulated virtual penalty size.
7:     int currMark ← 1                         ▷ The mark bit used in the current GC.
8:     int lastMark                             ▷ The mark bit used in the previous GC.
9: end global variables
10: procedure PreGC
11:     VP ← 0                                  ▷ VP is reset before each GC.
12:     lastMark ← currMark
13:     currMark ← currMark % 2 + 1
14: end procedure
15: procedure InGC
16:     for each live object currObj do         ▷ currObj is reachable in this GC.
17:         if currObj ≡ obj then               ▷ If currObj is the target.
18:             s ← Load(obj, PEN_OFF + Size(obj)
19:             Store(obj, PEN_OFF, s)
20:             head ← Load(obj, MIRROR_OFF)    ▷ head is a mirror object.
21:             if head ≡ null then             ▷ The mirror chain does not exist yet.
22:                 objToBeMirrored ← objToBeMirrored ∪ {obj}
23:             else
24:                 m ← head                    ▷ m is a mirror object.
25:                 n ← m.next                  ▷ n is a mirror object.
26:                 while n ≠ null do
27:                     if refExists(n.o, m.o) then   ▷ Verify the reference.
28:                         m ← n
29:                         n ← n.next
30:                     else
31:                         m.next ← null       ▷ The recorded path has changed.
32:                         break
33:                 SetMark(n.o, currMark)
34:                 mirrorObjCurrGC ← mirrorObjCurrGC ∪ {n}
35:         for each object p referenced by currObj do
36:             if GetMark(p) ≡ lastMark then
37:                 newRefEdges ← newRefEdges ∪ {⟨currObj, p⟩}
38:                 Unmark(p)
39:         VP ← VP + Load(obj, PEN_OFF)
40: end procedure
```

```
41: procedure PostGC
42:     for each object m ∈ objToBeMirrored do                ▷ m is a mirror object.
43:         m′ ← CreateMirror()                               ▷ m′ is a mirror object.
44:         WriteSourceCodeInfo(m, m′)
45:         Store(m, , MIRROR_OFF, m′)
46:     for each object p ∈ mirrorObjLastGC do                ▷ p is a mirror object.
47:         if ∃ ⟨i, p.o⟩ ∈ newRefEdges then
48:             i′ ← CreateMirror()                           ▷ i′ is a mirror object.
49:             WriteSourceCodeInfo(i, i′)
50:             p.next ← i′
51:     mirrorObjLastGC ← mirrorObjCurrGC
52:     mirrorObjCurrGC, objToBeMirrored, newRefEdges ← ∅
53:     VSO ← (VP + LiveHeapSize()) / LiveHeapSize()          ▷ Compute virtual space
    overhead.
54: end procedure
```

and 19), and then find the head object of its mirror path (Lines 20). If no mirror object has been created (i.e., it is the first time to penalize this object), we remember the object in set *objToBeMirrored*—since we cannot create mirror objects during a GC, we will do it later after the GC. If the head mirror object is found, we need to create a new mirror object and append it to the existing mirror path. Since this existing mirror path was built in the previous GCs, certain reference edges being mirrored may have changed. To detect such changes, we traverse the mirror path (Lines 24–32) to validate if each edge in the path still mirrors a valid heap reference (Line 27). When the traversal finishes, $n$ is either the last node of the mirror path or the first node at which the old reference relationship breaks. A new mirror object will be created and linked to $n$. We mark the object that $n$ mirrors (i.e., $n.o$) with *currMark* (Line 33) so that $n.o$ will be recognized in the next GC and a new reference edge pointing to $n.o$ will be mirrored. $n$ is then remembered in set *mirrorObjCurrGC* and we leave the mirror object creation to the PostGC procedure.

If any child of the object being traversed has been marked (by the previous GC) (Line 36), the reference edge connecting the object and this child needs to be mirrored. We remember the pair $\langle currObj, p\rangle$ in map *newRefEdges* for further processing (Line 37) and then unmark object $p$ (Line 38). It is important to note that the correctness of existing mirror chains

is verified when their root nodes are traversed during each GC (Lines 24–32). If its corresponding reference path is invalid, the invalid part will be removed and the reference path will be built based on the remaining part.

All mirror objects are created right after the GC is done (Lines 42–50). We first create a mirror object $m'$ for each object $m \in$ *objToBeMirrored* that does not have a mirror chain (Line 43, store $m$'s allocation site information into $m'$ (Line 44), and write a reference of $m'$ into $m$'s header space (Line 45). Next, we check each chain-ending mirror object $p \in$ *mirrorObjLastGC* and see whether there exists a new heap reference edge recorded in *newRefEdges* whose target is mirrored by $p$ (Line 47). If this is the case, we need to create one more mirror object to mirror the source of the edge (Lines 48–50).

All objects in list *mirrorObjCurrGC* are added to list *mirrorObjLastGC* before the mutator execution resumes, so that these objects will be used to mirror new edges in the next GC. *mirrorObjCurrGC*, *objToBeMirrored* and *newRefEdges* are cleared in PostGC (Line 52). Finally, virtual space overhead is computed (Line 53) based on the total penalty size *VP* (which is updated every time a node is traversed at Line 39).

Our incremental algorithm enables an important feature of our detector: *the completeness of diagnostic information provided for a "suspicious" object is proportional to the degree of its suspiciousness* (i.e., how long it gets penalized). Every time the object is penalized, the algorithm records one additional level of the reference path for the object. While a regular detector can also provide diagnostic information, it maintains a tracking space of the same size and records the same amount of information for all objects. For example, the reference path associated with the top object in `xalan` (shown in Section 4.3.2) reported by our leak amplifier has 10 nodes; all of the nodes on this long path are necessary for us to understand the cause of the leak. A regular detector can never afford to store a 10-node path for each object.

Our algorithm can mirror only one reference path for each object. We find it to be sufficient in most cases: there is often one single reference path in which an object is inappropriately used and causes a problem, although the object may be referenced by many paths at various points; if the object keeps being penalized, the penalty chain will eventually get stabilized to mirror the problematic path necessary for the developer to fix the problem. Furthermore, for many types of performance problems, reporting *any* reference path will be helpful to find their root causes. For instance, for leak detection, any reference path that holds a suspicious object and makes it reachable is problematic. As another example, for detection of under-utilized containers, our detector tracks inefficiently-used arrays and the key to producing a high-quality report is to find the data structures (e.g., HashMap, ArrayList, etc.) that reference those arrays. Since an array is often *owned* by a high-level data structure, any reference path that leads to the array must pass the data structure, and hence, reporting any path will be sufficient for the developer to find the data structure and understand the problem.

## 4.3 Evaluation

Using ISL, we have implemented four different amplifiers that target, respectively, memory leaks, under-utilized containers, over-populated containers and never-used return objects. In our evaluation, no application-related information was added into these descriptions, although a future user may describe a symptom in a way that is specific to an application. These ISL programs were easy to write and took us only a few hours. All experiments were executed on a quadcore machine with an Intel Xeon E5620 2.40GHZ processor, running Linux 2.6.32. To reduce the influence of execution noise, we repeat each test twenty times and report the medians.

| *Bench* | (a) MEM | | (b) UUC | | (c) OPC | | (d) NUR | |
|---|---|---|---|---|---|---|---|---|
| | *Real* | $m = 10$ | *Real* | $m = 10$ | *Real* | $m = 10$ | *Real* | $m = 10$ |
| antlr | 1.20 | 1.1 | 1.22 | 2.6 | 1.20 | 1.0 | 1.33 | 1.0 |
| bloat | 1.35 | 1.1 | 1.72 | 5.9 | 1.58 | 1.1 | 1.65 | 1.0 |
| eclipse | 1.28 | 5.6 | 1.21 | 8.7 | 1.23 | 3.5 | 1.26 | 1.0 |
| fop | 1.15 | 1.6 | 1.22 | 3.3 | 1.16 | 1.1 | 1.14 | 1.0 |
| **hsqldb** | 1.24 | **26.2**→1.3 | 1.19 | **16.6**→2.2 | 1.19 | **29.2**→1.7 | 1.19 | 1.0 |
| **jython** | 1.16 | **22.7**→6.4 | 1.06 | **48.3**→5.3 | 1.06 | **16.3**→3.8 | 1.06 | 1.0 |
| luindex | 1.18 | 1.2 | 1.16 | 1.7 | 1.13 | 1.1 | 1.09 | 1.0 |
| lusearch | 1.60 | 1.4 | 1.70 | 1.5 | 1.69 | 1.2 | 1.71 | 1.0 |
| pmd | 1.18 | 1.5 | 1.12 | 1.4 | 1.12 | 1.1 | 1.12 | 1.0 |
| **xalan** | 1.15 | **53.1**→7.1 | 1.09 | **117.7**→7.9 | 1.08 | 3.4 | 1.11 | 1.0 |
| GeoMean-H | | **31.6** | | **45.5** | | **21.8** | | — |
| GeoMean-L | | 1.6 | | 2.9 | | 1.5 | | 1.0 |

Table 4.1: Space overheads reported by the amplifiers.

### 4.3.1 Amplification Effectiveness

The first set of experiments focuses on understanding of whether our technique is effective in exposing performance problems in a testing environment. Since the DaCapo benchmark [11] provides three different kinds of workloads (i.e., small, default, and large), we ran each amplifier on their small workloads to simulate how a developer would use our tool during testing—it is much more difficult to reveal performance bugs under small workloads than large workloads. The number of iterations for each program is such that the execution can have at least 20 GCs. We make no assumption about test inputs—real-world developers can enable amplification when executing a regular test suite; whenever a test case triggers a true problem, amplification will incur a large virtual overhead.

**Indication of performance bugs.** Table 4.1 reports the (real and virtual) space overheads (in times) for memory leaks (column (a)), under-utilized containers (column (b)), over-populated containers (column (c)) and never-used return objects (column (d)) over the DaCapo benchmarks (2006 release) using a 500MB heap. Each column reports the real space overheads (*Real*) and the VSOs when $m = 10$. Highlighted in the table are the programs

whose maximal VSO is greater than 15. Notation of the form $a \rightarrow b$ means a program with a VSO $a$ has a new VSO $b$ after its performance problems are fixed. GeoMean-H and GeoMean-L report the average VSOs for the highlighted and non-highlighted programs, respectively.

Clearly, the VSOs indicate that leaks may exist in `hsqldb`, `jython` and `xalan`; under-utilized containers in `hsqldb`, `jython` and `xalan`; and over-populated containers in `hsqldb` and `jython`. An important observation made here is that there is a clear distinction between the overheads of programs with and without problems, making it particularly easy for developers to determine whether manual inspection is needed during testing. For example, to find over-populated containers, there are only two programs whose memory consumption is significantly amplified. Developers only need to read reports for these two programs; they would otherwise have to inspect all programs and check each warning had a regular detector been used. The differences in the VSOs of programs with and without problems are clearly demonstrated by the numbers reported in the GeoMean-H and GeoMean-L rows. We have manually inspected the programs whose VSO is $> 15$ and found problems in all of them[2]. Details of these problems will be discussed shortly in Section 4.3.2. As also shown in Table 4.2, the VSOs of `mysql` and `mckoi` became much smaller after the problems were fixed.

**Space and time overheads.** Section *Real* of Table 4.1 reports the real space overheads of our infrastructure, computed as $S_1/S_0$, where $S_0$ and $S_1$ are, respectively, the peak post-GC heap consumptions of the unmodified and modified RVM. The average space overheads incurred by the four amplifiers (i.e., memory leaks, under-utilized containers, over-populated containers, and never-used return objects) under the configuration $m = 10$ are 1.23×, 1.23×, 1.25×, and 1.25× respectively. The overall time overheads under the same configuration are 2.39×, 2.74×, 2.73×, and 2.80× respectively. To understand the scalability of the tool, we

[2]15 is just a number in a large range (between 10 and 20) that can easily distinguish these overheads.

| *Bench* | $m = 5$ | $m = 10$ | $m = 15$ | $m = 20$ |
|---|---|---|---|---|
| antlr | 1.2 | 1.1 | 1.2 | 1.2 |
| bloat | 1.1 | 1.1 | 1.1 | 1.1 |
| eclipse | 5.3 | 5.6 | 6.3 | 6.3 |
| fop | 1.6 | 1.6 | 1.7 | 1.7 |
| **hsqldb** | **26.5**→1.7 | **26.2**→1.3 | **25.9**→1.2 | **25.7**→1.2 |
| **jython** | **22.7**→6.3 | **22.7**→6.4 | **22.7**→6.4 | **24.5**→6.3 |
| luindex | 1.2 | 1.2 | 1.2 | 1.2 |
| lusearch | 1.4 | 1.3 | 1.3 | 1.3 |
| pmd | 1.5 | 1.5 | 1.5 | 1.5 |
| **xalan** | **53.1**→7.3 | **53.2**→7.1 | **40.4**→2.7 | **37.0**→1.9 |
| **mysql** | **155.5**→2.3 | **108.9**→2.3 | **70.4**→1.5 | **30.1**→1.2 |
| **mckoi** | **111.6**→6.9 | **72.4**→6.6 | **85.6**→3.3 | **42.6**→1.7 |
| GeoMean-H | **56.1** | **47.8** | **42.7** | **31.3** |
| GeoMean-L | 1.6 | 1.6 | 1.6 | 1.6 |

Table 4.2: Space overheads for memory leaks with different history sizes.

have also run these amplifiers using the DaCapo large workloads. The time overheads for the four amplifiers are all between 2.5× and 3.2×.

**Sensitivity to the execution window length.** We ran our memory leak amplifier with $m = 5$, $m = 10$, $m = 15$ and $m = 20$ to show whether our framework is sensitive to execution window length. Table 4.2 shows the results over the DaCapo benchmarks with different lengths of the execution window using a 500MB heap. We added two extra programs `mysql` and `mckoi` to our benchmark set—they have known leaks studied before and we are thus interested in how our amplifier performs on these programs. Each column $m = i$ reports the VSOs of the amplifier when the size of the execution window is $i$ (i.e., the value of the field `size` in the `History` construct). Highlighted in the table are the programs whose maximal VSO is greater than 15. Notation of the form $a \rightarrow b$ means a program with a VSO $a$ has a new VSO $b$ after its performance problems are fixed. GeoMean-H and GeoMean-L for a column $m = i$ report the average VSOs under $m = i$ for the highlighted and non-highlighted programs, respectively.

We make two observations based on the amplification results in Table 4.2. First, the VSOs

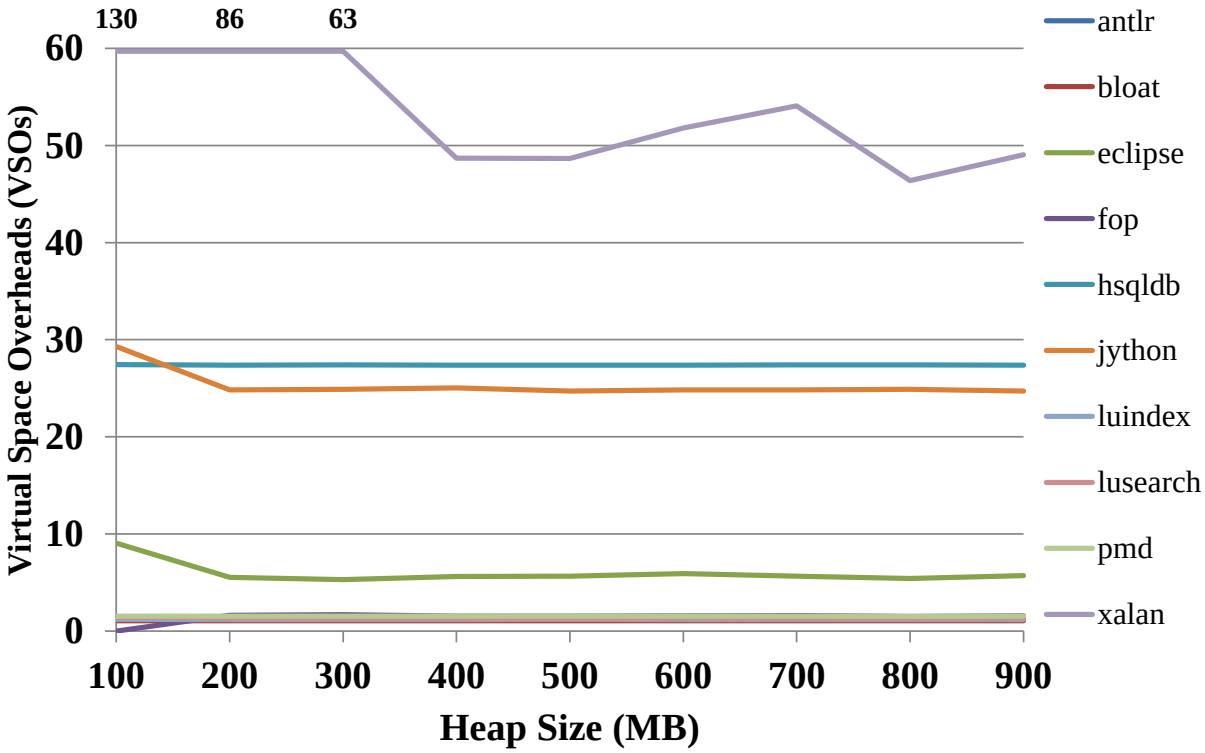


Figure 4.7: VSOs for the memory leak amplifier with different heap sizes.

under different parameters are generally stable; the size of an execution window (i.e., user-provided threshold) does not have much impact on the amplification results. On the contrary, the effectiveness of an existing symptom-based performance problem detector relies heavily on such a threshold. Second, as the size of the execution window increases, the space overhead of the tool decreases in general. This is straightforward—the larger the size of the history is, the fewer objects are amplified, and the smaller effect amplification creates.

**Sensitivity to the heap size.** Figure 4.7 shows the VSOs reported by our memory leak amplifier with different heap sizes. Based on the data in Figure 4.7, we make two observations. First, the VSOs reported by our leak amplifier with differnt heap sizes are general stable; the heap size does not have much impact on the amplification results. Second, as the heap size increases, the space overhead of the tool decreases in general. With a larger heap, GC is triggered less frequently. As we mentioned in Section 3.2, the same memory problem is more serious if the normal parts of the program consumes less resource.

### 4.3.2 Problems Found

We have manually inspected our reference path report for each highlighted program in Table 4.1 and Table 4.2. We have not found any highlighted program to be mistakenly amplified, that is, real performance problems are identified in all of them. We have studied a total of 10 amplification reports and found 8 unknown problems that have never been reported in any prior work. Furthermore, all of these problems are uncovered under small workloads; their executions do not exhibit any abnormal behavior without amplification. In this section, we discuss our experiences with 8 unknown problems. For each problem found, we have developed a fix. As shown in Table 4.1 and Table 4.2, the VSOs of these programs are significantly reduced after their fixes are applied. This demonstrates that PerfBlower is sensitive only to true memory issues, not false problems.

For each program, we have also run an existing leak detector, Sleigh [14], and compared our reports with Sleigh's reports. The reason we chose Sleigh is because it is the only publicly available performance problem detector that does not require user involvement. While there exist other tools such as LeakChaser [122] and a few inefficient data structure detectors [89], they are either unavailable or require heavy user annotations to understand the program semantics. Hence, we compare PerfBlower only with Sleigh in this dissertation.

**Memory leaks in `xalan`.** The No. 1 allocation site in the report has a very long reference path (with 10 nodes); the top 3 objects on the path are in Figure 4.8.

The reference path shows us that the largest penalty is added to objects created and referenced by the instances of an XML transformer class `TransformerImpl`. This immediately catches our attention, since `TransformerImpl` objects should be reclaimed after each transformation. The reference path directs us to class `StyleSheetRoot`, which implements interface `Templates` and has a reference to an `IteratorPool` object. Each `IteratorPool` object maintains a vector of `DTMAxisTraverser` objects in order to reuse those objects. However,

```
#0. id: 5561 ObjectVector.java: 106
Method: org.apache.xml.utils.ObjectVector.<init>
Object Type: java.lang.Object[]
Number of suspicious objects: 27
Size: 14173056 bytes
Path: ID: 5573 XPathContext.java: 920
   Method: org.apache.xpath.XPathContext.<init>
   Object Type: org.apache.xml.utils.ObjectStack
   <= ID: 5557 TransformerImpl.java: 402
      Method: org.apache.xalan.transformer.TransformerImpl.<init>
      Object Type: org.apache.xpath.XPathContext
      <= ID: 5543 StylesheetRoot.java: 212
         Method: ...templates.StylesheetRoot.newTransformer
         Object Type: org.apache.xalan.transformer.TransformerImpl
         <= ...
```

Figure 4.8: A partial reference path reported by our memory leak amplifier.

upon the reuse of an old `DTMAxisTraverser` object, the object still keeps a reference of its previous `SAX2DTM` object, which, in turn, references the `TransformerImpl` object. Hence, each reused `DTMAxisTraverser` object in the `IteratorPool` leaks its previous `TransformerImpl`. This is a serious problem, because the size a `TransformerImpl` object is often very large (at the megabyte level). We come up with a quick fix in which one line of code is commented out to disallow reuse of `DTMAxisTraverser` objects. **The fix has resulted in a 25.4% reduction in memory consumption and a 14.6% reduction in execution time.** A subsequent search in the `xalan` bug repository reveals that the same bug was found and fixed in version 2.5.1 [37] (with a different approach), while the DaCapo xalan version in which we detect the bug is a much earlier one (2.4.1). We find that every object on this long path is necessary for understanding the cause of the bug. It would not be possible to record such complete information with a regular detector/profiler (that uses a fixed-size space to store metadata for each object).

For `xalan`, Sleigh reports a few stale objects, including their types and numbers of instances. Most of these types are Java library classes or VM classes, such as `java.lang.String` and

`com.ibm.JikesRVM.classloader.VM_Atom`, which have nothing to do with the leak. In this case, Sleigh provides neither allocation sites nor last access sites for these stale objects. This is primarily because Sleigh uses only one bit per object to encode information statistically; it needs a long execution to gather sufficient data for producing a diagnostic report, and thus does not work well for test executions, which are often small-scale and very short.

**Memory leaks in `jython`.** From the report, it is easy to identify that most space penalties are added to stale `PyJavaPackage` objects. One use of these objects is to cache the relationships between Java classes and Jar files. When a jar file is loaded, Jython iterates over all the classes and creates PyJavaPackage objects. Many classes are never used during execution, and thus, their corresponding `PyJavaPackage` objects become leaks. We fix the problem by (1) employing lazy loading and (2) removing unnecessary jars from classpath. **The fix has led to a 24.3% peak memory reduction and a 7.4% time reduction for the warm up phase.**

In Sleigh's report, there is only one stale allocation site. Although this allocation site is related to the cause of the leak, it is still far away and it would take a developer a fair amount of effort to find the cause from the allocation site. On the contrary, PerfBlower provides much more precise diagnostic information, significantly alleviating the developer's debugging burden.

**Memory leaks in `hsqldb`.** Most of the reported objects are related to data values computed but not used at all in the DaCapo execution. For example, `hsqldb` maintains four maps between paths and databases. Although in DaCapo only the memory database is used, `hsqldb` also keeps maps for the file database and the resource database. One way to solve the problem is to provide configuration options that allow users to specify the databases needed. We change these stale objects to lazy initialization, reducing the memory consumption by **15.6%**.

**Under-utilized containers in `hsqldb`.** The report clearly shows most under-utilized containers are due to inappropriate initial capacity. For example, the top object in our report points to arrays created during the initialization of `BaseHashMap`. The reference path quickly leads us to inspect ValuePool.java. In this class, `hsqldb` maintains six `ValuePoolHashMap` objects in static fields, which are used to support singleton patterns. However, all of these maps have 10000 as their initial capacity, which is way too large for many clients. We fix the problem by making these containers grow dynamically as necessary and changing their initial capacity to 32. **This optimization has led to a 17.4% space reduction.**

**Under-utilized containers in `jython`.** Our report indicates that the largest penalty is added to arrays created in `PyStringMap.resize` method. By inspecting `PyStringMap`, we find that the class uses a string array to store keys and a `PyObject` array to store values. Before each insertion, it checks the load factor of its key array (i.e., how full the map is allowed to get before its capacity is increased). In this case, if the load rate is higher than 0.5, its capacity is doubled. Actually, since the hash function of `String` works generally well and the key array has a very small collision rate, using 0.5 as load factor seems too aggressive. Modifying it to 0.75 results in a **19.1% space reduction**.

**Under-utilized containers in `xalan`.** The top warning in the report points to `ObjectStack` objects created in `XPathContext` and `TransformerImpl` objects. The initial capacity of `ObjectStack` is defined by `XPathContext`. `RECURSIONLIMIT`, a static final field with a value 4096. A detailed inspection of the source code reveals that ObjectStack is implemented based on ObjectVector, which is a data structure designed to support random accesses. The initial capacity defines a tradeoff between space and time: using a smaller value saves space at the cost of increased lookup computation. Finding the sweet spot requires deep understandings of the program and empirical studies. When we use 2048 as the initial capacity, the memory consumption is reduced by **5.4%**, and the execution time is reduced by **34.1%**.

**Over-populated containers in `hsqldb`.** Before `hsqldb` executes an `insert` query, it

invokes a method called `getNewRowData` to create an object array that represents the row to be inserted. A column in a database table often has a default value—when a new row specified by an `insert` query does not have a value for the column, this default value will be filled in. We find that method `getNewRowData` creates a large number of objects and fill them into the object array as default values. These objects are never retrieved from this array until later they are replaced by the actual values. We fix the problem by performing a lazy default value assignment, resulting in a **14.9% space reduction.**

**Over-populated containers in `jython`.** The cause of the problem here is the same as in jython-leak. The top object in our report directs us to inspect method `PyJavaPackage.resize`, in which a large array is created to contain Java classes, most of which are never retrieved from the map. This same problem has two different manifestations.

**Comparison with Sleigh.** Table 4.3 summarizes the comparison of the leak reports of PerfBlower and Sleigh. PerfBlower found memory leaks in six benchmarks (including four new leaks and two known leaks), and for each memory leak, PerfBlower provided very precise diagnostic information (leaks$^+$). Sleigh reported that five programs contain memory leaks, including two new leaks, two known leaks, and a false warning. Furthermore, Sleigh did not provide any useful diagnostic information for three leaking programs (leaks$^-$). The comparison demonstrates that PerfBlower is able to find more (new) leaks, has fewer false positives, and generates more precise diagnostic information.

### 4.3.3 PerfBlower Completeness

Our amplifiers are able to find bugs in all of the highlighted programs; but do they miss bugs? To answer this question, we performed an additional experiment. In this experiment, the benchmark set includes 14 programs (shown in Table 4.4), which have known performance problems reported in all prior works [53, 14, 16, 122, 126] except Chameleon [89]. Chameleon

| *Bench* | PerfBlower | Sleigh |
|---|---|---|
| antlr | no leak | no leak |
| bloat | no leak | no leak |
| eclipse | leaks$^+$ | no leak |
| fop | no leak | no leak |
| hsqldb | leaks$^+$ | leaks$^+$ |
| jython | leaks$^+$ | leaks$^-$ |
| luindex | no leak | false positive |
| lusearch | no leak | fail to run |
| pmd | no leak | no leak |
| xalan | leaks$^+$ | leaks$^-$ |
| mysql | leaks$^+$ | fail to run |
| mckoi | leaks$^+$ | leaks$^-$ |
| true unknown leaks | **4** | 2 |
| true known leaks | **2** | **2** |
| false positives | **0** | 1 |
| useful information | **6** | 1 |

Table 4.3: Comparison of PerfBlower and Sleigh's reports.

is a dynamic technique that can detect inefficiently-used containers. However, it is based on the commercial J9 VM and not publicly available. In addition, the description of the bugs in [89] is at a very high level and does not contain detailed location information (such as classes, methods, and line numbers). Hence, those bugs are not considered.

Among these benchmarks, three programs cannot be executed. Two of them are about memory leaks in Sun JDK library, which is not used by Jikes RVM. Another benchmark is `chart`, which could not be executed because some AWT libraries were missing. For the rest of the bugs, all but one are captured by PerfBlower. The missing one is an under-utilized container in the DaCapo benchmark `bloat`, reported by a static analysis-based container problem detector [126]. Because PerfBlower is based on dynamic analysis, its ability of finding bugs depends on the coverage of test cases. We inspect the source code of benchmark `bloat` and conclude that this bug cannot be triggered by the DaCapo input. Based on this experiment, we observe that PerfBlower does not miss any bug that can be triggered by a test case.

| *Work* | *Bugs Studied* | *PerfBlower* |
|---|---|---|
| Cork [53] | SPECjbb's leak | Reported |
| | Eclipse #115789 | Reported |
| Sleigh [14] | SPECjbb's leak | Reported |
| | Eclipse #115789 | Reported |
| Container Profiler [125] | SPECjbb's leak | Reported |
| | JDK #6209673 | Not available in RVM |
| | JDK #6559589 | Not available in RVM |
| LeakChaser [122] | SPECjbb's leak | Reported |
| | Eclipse #115789 | Reported |
| | Eclipse #155889 | Reported |
| | MySQL's leak | Reported |
| | Mckoi's leak | Reported |
| Leak Pruning [16] | SPECjbb's leak | Reported |
| | Eclipse #115789 | Reported |
| | Eclipse #155889 | Reported |
| | MySQL's leak | Reported |
| | JbbMod's leak | Reported |
| | Mckoi's leak | Reported |
| | Delaunay's leak | Reported |
| | List leak | Reported |
| | Swap leak | Reported |
| | Dual leak | Reported |
| Static Bloat Finder[126] | Bloat's UUC | Not triggered |
| | Chart's OPC | Not available in RVM |

Table 4.4: Completeness of PerfBlower.

### 4.3.4 False Positive Elimination

In order to understand if our technique successfully eliminates false warnings of a regular performance problem detector, we measure the number of objects that have experienced both amplification and deamplification. These objects are amplified because they satisfy the symptom specification for a sufficiently long period; they are deamplified later when the symptom disappears. Although they are not true causes of performance problems, a symptom-based detector would report all of them.

False warnings eliminated by our memory leak amplifier are shown in Table 4.5. The setup for this experiment is exactly the same as the one for the experiments reported in Table 4.1

and Table 4.2. For each program under each history size $m = i$, we report four numbers *FO*, *FS*, *TO*, and *TS*; *FO* is the number of such objects that their staleness exceeds $i$ but they are used afterwards, *FS* is the number of allocation sites creating objects in *FO*, *TO* is the total number of objects whose staleness exceeds $i$, and *TS* is the number of allocation sites creating objects in *TO*. *FO* and *FS* are, respectively, the numbers of false warning objects and allocation sites that are eliminated by deamplification but would have otherwise been reported by a regular memory leak detector.

We make three observations based on these numbers. First, the number of false warnings is reduced as we increase the threshold $i$. This is straightforward because true leaking objects often have larger staleness than false leaks. However, increasing $i$ can easily lead to *false negatives* because the staleness of objects created late during execution may not reach $i$ before the execution finishes. This can be observed from the fact that the total numbers of suspicious objects *TO* and suspicious allocation sites *TS* drop significantly, especially for `lusearch`, `xalan`, `mysql`, and `mckoi`. For example, setting $i$ to 10 during an execution with 20 GCs would miss all stale objects created in the second half of the execution. Our algorithm frees developers from choosing a perfect threshold $i$: one can always use a small $i$ (such as 5) to ensure no true problem is missing without worrying about false positives—most of them are automatically eliminated by deamplification. Finally, PerfBlower has eliminated not only false leaking objects (*FO*), but also false leaking allocation sites (*FS*). Since a leak detector often ranks and reports allocation sites, elimination of false leaking allocation sites leads directly to more precise reports as well as reduced manual inspection effort.

## 4.4 Limitations

While PerfBlower captures commonalities of different memory-related problems and makes it easy for them to manifest, it has a few limitations that leave room for improvement. First,

| *Bench* | $m = 5$ | | $m = 10$ | | $m = 15$ | | $m = 20$ | |
|---|---|---|---|---|---|---|---|---|
| | *FO/FS* | *TO/TS* | *FO/FS* | *TO/TS* | *FO/FS* | *TO/TS* | *FO/FS* | *TO/TS* |
| antlr | 26/2 | 103/68 | 10/2 | 87/68 | 3/2 | 80/68 | 2/1 | 79/68 |
| bloat | 19/3 | 72/29 | 4/3 | 56/29 | 3/3 | 54/29 | 2/2 | 53/29 |
| eclipse | 5992/498 | 12062/1279 | 4697/246 | 10572/1030 | 4648/235 | 9747/1018 | 4339/201 | 8852/958 |
| fop | 27/2 | 823/723 | 9/1 | 806/722 | 4/1 | 797/722 | 0/0 | 793/722 |
| hsqldb | 386/4 | 1866/105 | 188/4 | 1236/105 | 68/3 | 793/105 | 13/3 | 493/105 |
| jython | 28/3 | 17864/1072 | 21/3 | 17709/1034 | 11/2 | 17634/951 | 7/2 | 17410/830 |
| luindex | 9/2 | 94/72 | 5/2 | 90/72 | 2/2 | 90/72 | 2/2 | 88/72 |
| lusearch | 183/15 | 1713/113 | 32/14 | 206/67 | 5/4 | 178/66 | 3/2 | 174/64 |
| pmd | 28/2 | 238/156 | 6/2 | 210/156 | 1/1 | 204/156 | 1/1 | 203/156 |
| xalan | 120/12 | 19795/637 | 65/11 | 14657/637 | 17/8 | 11304/636 | 10/7 | 10540/628 |
| mysql | 4457/3 | 54937/192 | 1933/3 | 40844/192 | 2/2 | 26547/192 | 2/2 | 12877/192 |
| mckoi | 116/76 | 12004/298 | 116/76 | 11547/292 | 111/76 | 10133/292 | 1/1 | 5058/236 |
| jbb | 8192/55 | 54358/59 | 27/8 | 49018/55 | 5/4 | 27913/55 | 4/4 | 25081/53 |

Table 4.5: False warnings eliminated by deamplification.

it can only find heap-related inefficiencies, while there are many different sources for real-world performance problems, such as idle threads, redundant computations, or inappropriate algorithms. Second, the JVM needs to be rebuilt every time a new checker is added. In practice, this is not an issue, since most modern JVMs support fast and incremental building. For example, it takes only one minute to build the Jikes RVM on which PerfBlower is implemented. One future direction would be to make PerfBlower a JVMTI-like library that can be dynamically registered during bootstrapping of the JVM without needing to build the JVM. Third, since PerfBlower piggybacks on the GC, its effectiveness may be affected by the GC frequency. PerfBlower may report a higher overhead if a program is executed with a smaller heap (due to more GCs). While this introduces uncertainties, our experimental results (in Figure 4.7) demonstrate, for most programs, the VSOs reported by PerfBlower are very stable across different heap sizes. Finally, PerfBlower may become less effective when a generational GC is used. Since a generational GC does not frequently perform full-heap scanning, objects that stay in the old generation space may not be effectively amplified.

## 4.5 Summary and Interpretation

Based on ISL, we have built a general performance testing framework, PerfBlower, which can help developers capture performance bugs in a testing environment. Using PerfBlower,

we have amplified *four different types of performance problems* (i.e., memory leaks, under-utilized containers, over-populated containers, and never-used return objects) on a set of real-world applications; our experimental results show that even under small workloads, there is a very clear distinction between the VSOs for executions with and without problems, making it particularly easy for developers to write simple space assertions and only inspect programs that have assertion failures. In fact, we have manually inspected each of our benchmarks for which PerfBlower reports a high VSO, and found a total of *8 unknown problems*.

We have compared the quality of leak reports between PerfBlower and an existing leak detector Sleigh [14] executed under the same (small) workloads: PerfBlower reported 6 leaks with no false positive and very detailed diagnostic information, while Sleigh reported 4 leaks with 1 false positive, and generated zero diagnostic information for 2 programs and very high-level information for the other 2 programs. We additionally performed an exhaustive study of the 14 performance bugs reported in the literature: PerfBlower found all but 4 bugs; for these bugs, we either could not run the program or did not have a triggering test case. These promising results clearly show that PerfBlower is useful in quickly building a large number of checkers that can effectively find performance bugs before they manifest in production runs.

PerfBlower is more tolerant to benign problems than existing symptom-based detectors [14, 53, 89], which often have many false warnings due to heavy reliance on user parameters and heuristics. For instance, in a simple staleness-based memory leak detector, an object whose staleness exceeds a parameter $i$ will be reported as a leak regardless of whether the object is used afterwards. A fundamental difficulty in an existing detector lies in the choice of the appropriate threshold: on one hand, a small $i$ leads to many false positives because a stale object may be used after a warning is reported; on the other hand, a big $i$ may cause the profiler to miss a true problem because the execution may have already finished before $i$ is reached. The difficulty of finding the right $i$ is further increased in a testing environment

(with small workloads), wherein the behaviors of true and benign problems are often very similar and cannot be easily distinguished.

Although PerfBlower still needs a user threshold to determine when to perform amplification, its reliance on finding a perfect $i$ is significantly reduced by turning a symptom into a *cancelable penalty*. Using ISL, developers specify not only *when to amplify* (i.e., when the symptom specification is satisfied in an execution window), but also *when to deamplify* (i.e., when counter-evidence is seen). In our experiments, we observe that a small $i$ often works very well—if a program only has benign problems, although penalties are still created when a symptom appears, these penalties will be removed later by deamplification and thus not accumulate. As demonstrated in Section 4.3.4, for large applications such as `eclipse` and `mysql`, many thousands of false warnings are eliminated by deamplification but would otherwise have been reported by an existing leak detector.

In our framework, the amplification is done on a *per-object* basis: a memory penalty is created and associated with each object that satisfies the symptom specified in ISL (i.e., at the moment the object becomes "suspicious"). Instead of requesting actual memory as penalties (which incurs significant space overhead), we create *virtual penalties* by maintaining a *penalty counter* inside each object to track the size of the penalty created for the object. During each garbage collection (GC), we identify the real heap usage of the program and then compute a *virtual heap consumption* by adding up the penalty counter for each object and the real heap consumption. The virtual heap consumption is then compared to the real heap consumption to compute a *virtual space overhead* (VSO). VSO provides an automated test oracle: our experimental results show that the overall VSOs for benchmarks with and without real performance problems are, respectively, 20+ times and 1.5 times. The gap is sufficiently large so that test runs triggering bugs can be easily identified.

In PerfBlower, we have implemented the algorithm shown in Section 4.2.2 which *incrementally* builds a *mirror object chain* that reflects the major reference path in the object graph

leading to the "suspicious" object $o$. This chain records the source code information of each object on the reference path; identifying and reporting the reference path for $o$ gets reduced to traversing *forward* $o$'s *mirror chain*, a task significantly easier than performing a *backward* traversal on the object graph which has only unidirectional edges. Our incremental algorithm enables an important feature of the framework: *the completeness of diagnostic information provided for a "suspicious" object is proportional to the degree of its suspiciousness* (i.e., how long it gets penalized). Every time the object is penalized, the algorithm presented in Section 4.2.2 incrementally records one additional level of the reference path for the object. While a regular detector can also provide diagnostic information, it maintains a tracking space of the same size and records the same amount of information for all objects. Developers are very often interested in only a few (top) warnings in a diagnostic report; hence, it is much better to record more diagnostic information for top warnings than those that are low down on the list. Our algorithm dynamically determines the metadata space size based on how "interesting" an object is, making it possible to *prioritize* object tracking. For example, the reference path associated with the top object in `xalan` (shown in Section 4.3.2) reported by our leak amplifier has 10 nodes; all of the nodes on this long path are necessary for us to understand the cause of the leak. A regular detector can never afford to store a 10-node path for all objects in the heap.

# Chapter 5

# ITask: Helping Data-Parallel Tasks Survive Memory Pressure

In Big Data era, since the workload becomes very large, data-related performance problems are easy to manifest. However, some performance issues are extremely difficult to resolve. In this chapter, we will attack one of such difficult problems—memory pressure in data-parallel applications, since many Big Data applications suffer from this type of memory problems and no existing work can systematically address it. We will start with the memory problems in the real world to understand memory pressure in Big Data systems. Then we will propose our solution ITask. The experimental results will be shown in the evaluation section.

## 5.1 Memory Problems in the Real World

To understand memory problems and their root causes in real-world data-parallel systems, we searched in StackOverflow for the two keywords "out of memory" and "data parallel". The search returned 126 memory problems, from which we discarded those that can be easily fixed

or whose problem description was not sufficiently clear for us to reproduce. We eventually obtained 73 relevant posts with clear problem descriptions. A detailed investigation of them identified two common root-cause patterns:

- **Hot keys.** In a typical data-parallel framework, data are represented as key-value pairs, and some particular keys may have large numbers of associated values. 22 problems fall into this category, including the example discussed in Chapter 1—a join operation in the program builds an XML object that represents a post (with all its comments) on StackOverflow. Certain posts are much longer than others; holding one such long and popular post can consume an extremely large amount of memory. Consequently, the out-of-memory error (OME) can only be avoided if the processing of a long post does not occur simultaneously with that of other posts, which, in an existing framework, can only be done by making the entire framework sequential.

- **Large intermediate results.** For 51 problems, the semantics of their programs require intermediate results to be cached in memory, waiting for subsequent operations before the final results can be produced. In many cases, developers hold these results in large Java collections (e.g., HashMap or ArrayList), which have non-trivial space overhead [68]. One post [2] describes a scenario in which the developer uses the Stanford Lemmatizer [97] (i.e., part of a natural language processor) to preprocess customer reviews before calculating the lemmas' statistics. The task fails to preprocess a dataset of 31GB at a very early stage. The large memory consumption of the lemmatizer is the cause: due to the temporary data structures used for dynamic programming, for each sentence processed, the amount of memory needed by the lemmatizer is 3 orders of magnitude larger than the sentence. Furthermore, the Stanford Lemmatizer is a third-party library: the developer is unlikely to know either its memory behavior or how to control it.

Among the 73 posts we studied, only 25 have recommended fixes. We also investigated these recommendations and classified them into two major categories:

- **Configuration tuning.** There are 16 problems for which the recommended fixes are to change framework parameters. However, framework parameters in data-parallel systems are extremely complex. For example, Hadoop 2.6.0 has about 190 framework parameters, such as data split size, number of workers, buffer size, etc. Even experienced developers may have difficulties finding the appropriate configurations. The tuning process is often labor-intensive, consisting of repetitive tests and trials. In fact, almost every discussion thread contains multiple proposed parameter changes and there is no confirmation whether they have actually worked.

- **Skew fixing.** There are 9 posts in which the recommended fixes are to fix skews in the datasets. For instance, to fix the lemmatizer problem, one recommended a thorough profiling to find all long sentences in the dataset and break them into short sentences. However, it is nearly impossible to manually break sentences in such a large dataset. While there exist tools that can manage skews [57], it is difficult to apply them to general datasets, where there is often huge diversity in data types and distributions.

One common complaint the developers had is that they "lost control" over their programs after these programs start running in a framework: the developers are constrained within the given interfaces (such as Map/Reduce); they can do nothing more than writing efficient implementations of those interfaces, hoping that the system will clean up the mess and perform intelligent resource management. Since memory problems are difficult to predict during development, many developers wish they could have a more flexible programming interface (e.g., a hook) that would allow them to reason about what to do upon memory pressure.

The complexity of real-world memory problems as well as the difficulty of manually coming

up with fixes strongly call for system support that can automatically release memory upon extreme memory pressure. The design of ITask is motivated exactly by this real-world need. We have implemented ITask versions for a representative subset of the problems we have studied. Without any manual parameter tuning or skew fixing, the ITask versions successfully processed the datasets on which their original versions crashed with OMEs.

## 5.2 Design Overview

The high level idea behind ITasks is shown in Figure 5.1. When the system detects memory pressure, a selected task is interrupted, with part or all of its consumed memory reclaimed. This process is repeated until the pressure disappears to direct the execution of other tasks on the same node back to the "safe zone" of memory usage.

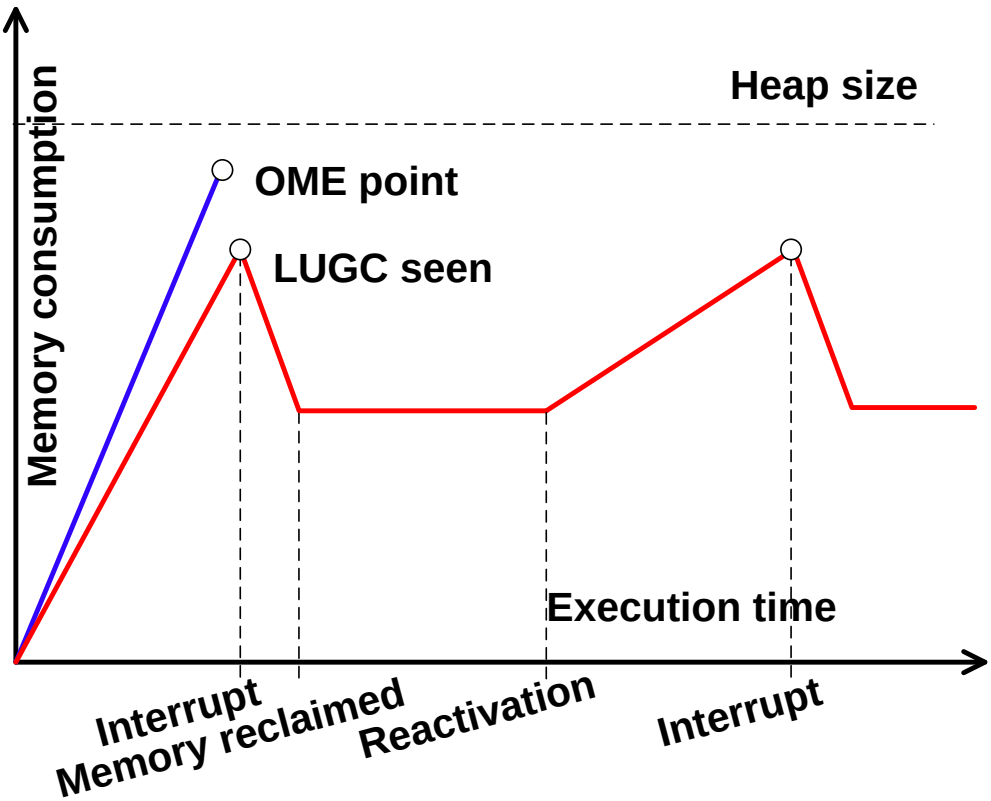


Figure 5.1: Memory footprints of executions with (red) and without (blue) ITask.

We present below three key challenges in carrying out this high-level idea, our high-level solutions to these challenges, as well as the ITask system architecture.

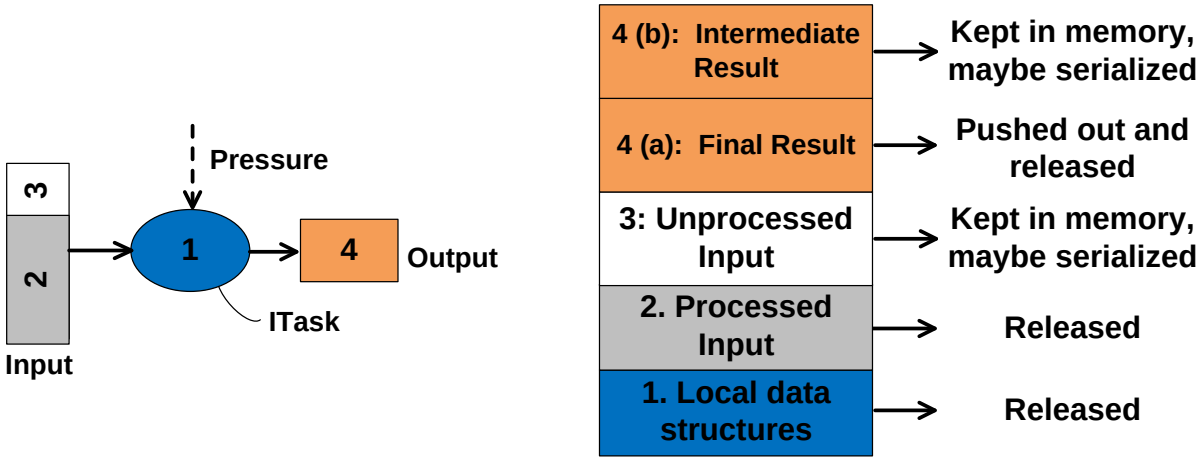


Figure 5.2: Handling the different memory components of an ITask.

**How to lower memory usage when a task is interrupted?** Figure 5.2 illustrates of an ITask execution at an interrupt with numbers showing different components of its memory consumption (the left part) and how these components are handled (the right part). As shown in Figure 5.2, the memory consumption of a data-parallel task instance consists of the following four components: (1) local data structures created by the task, (2) the processed input data before the interrupt, (3) the unprocessed part of the input, and (4) the partial results produced. Simply blocking a thread running the task without swapping data would not change the task's memory consumption at all; naïvely terminating the thread can completely eliminate the task's memory consumption, but would also completely waste the computation already performed by the thread.

Our design carefully handles different memory components differently when terminating the task-running thread, as shown in Figure 5.2. For Component 1 and Component 2, it is safe to discard them when the corresponding thread is terminated. For Component 3, we will try to keep the unprocessed input in memory and serialize it (lazily) when needed.

For Component 4, we differentiate two sub-types of result data, represented by 4(a) and 4(b) in Figure 5.2. Immediately useful results, referred to as *final results*, can be pushed to the next operator in the data pipeline immediately (e.g., another set of MapReduce tasks). Where that next operator is executed is determined by the framework scheduler and may be on a different node. Results that are not immediately useful and need further aggregation, referred to as *intermediate results*, will stay in memory and wait to be aggregated until all intermediate results for the same input are produced. These results can be lazily serialized under severe memory pressure. Which result is final and which is intermediate depends on the task semantics. For example, in MapReduce, an interrupt to a Map task generates a final result, which can be forwarded immediately to the shuffle phase; an interrupt to a Reduce task generates an intermediate result, which cannot be used until all intermediate results from the same hash bucket are aggregated.

**When to interrupt a task?** The best timing has to consider two factors: per-process system memory availability and per-thread/task data processing status. Specifically, we want to interrupt a task when the overall memory pressure comes and when its execution arrives at a *safe state* where it is not in the middle of processing an input data item. The former avoids unnecessary interrupts. The latter allows terminating a task by recording only minimum local information of the execution. During task re-activation, a new task instance can simply work on the unprocessed part of the original input without missing a beat.

To handle the first factor, our system leverages an observation that long and useless GC (LUGC)—that scans the whole heap without reclaiming much memory—is a good indicator of memory pressure, and uses an LUGC as a signal to trigger interrupts. To handle the second factor, we need to understand the data processing status, which is related to the semantics of a task.

**How to interrupt a task?** Interrupting in our system involves much more than terminating a random thread in a memory-pressured process. In a system with many tasks running, determining which thread(s) to terminate is challenging and requires precise global runtime assessment. Even if we know which task to interrupt, conducting the interrupt is still non-trivial, involving recording the progress of the task, such as which part of the input has been processed and what results have been generated. Like the two challenges stated above, addressing this challenge requires both whole-system coordination and understanding of per-task semantics.

**ITask system architecture.** The ITask system includes two main components that work together to address the challenges discussed above: an ITask programming model and an ITask runtime system (IRS).

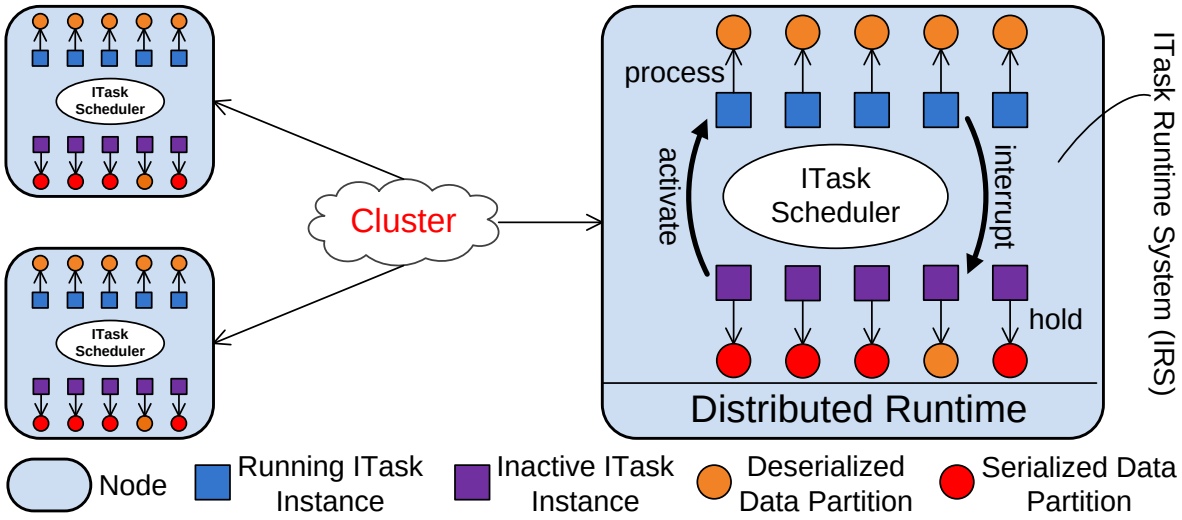


Figure 5.3: The architecture of the ITask runtime system.

The ITask programming model provides abstractions for developers to program interrupt logic and allows the IRS to conduct operations that require semantic information about tasks. Our programming model is carefully designed to provide a *non-intrusive* way to realize ITasks, making it possible to quickly modify existing data-parallel programs and enable ITasks in different frameworks. Specifically, developers only need to implement a

few additional functions by restructuring code from an existing task. The details of the programming model are discussed in Section 5.3.

The IRS is implemented as a Java library. This library-based implementation makes ITask immediately applicable to all existing Java-based data-parallel frameworks. The same idea can be easily applied to other managed languages such as C# and Scala. More details of the IRS implementation are presented in Section 5.4.

Figure 5.3 shows a snapshot of a running ITask-based data-parallel job. The IRS sits on top of the distributed runtime on each node. Whenever the job scheduler assigns a job, which contains a set of ITasks implementing a logical functionality, to a machine, it *submits* the job to the IRS instead of directly running the ITasks. The IRS maintains a task scheduler that decides when to interrupt or re-activate an ITask instance. As a result, the number of running ITask instances dynamically fluctuates in response to the system's memory availability. Each task instance is associated with an input data partition. Running tasks have their inputs in the deserialized form in memory (e.g., map, list, etc.), while interrupted tasks may have their inputs in the serialized form (e.g., bytes in memory or on disk) to reduce memory/GC costs.

Figure 5.1 illustrates an over-the-time memory-usage comparison between executions with and without ITasks. In a normal execution, the memory footprint keeps increasing; after a few LUGCs, the program crashes with an OME. In an ITask execution, the IRS starts interrupting tasks at the first LUGC point; the memory usage is brought down by the ITask interrupt and stays mostly constant until the next re-activation point at which new task instances are created. Without any manual configuration tuning, ITask-enhanced data-parallel jobs can keep their memory consumption in the safe zone throughout their executions, effectively avoiding wasteful GC effort and disruptive OMEs.

**Other design choices.** Besides the proposed approach, there are several other design choices. The first one is to develop a language with new constructs to allow developers to express interrupt logic. However, as with all language design efforts, there is an inherent risk that developers lack strong motivation to learn and use the new language. Another choice is to develop a data-parallel system from scratch with all the ITask-related features embedded. This choice shares a similar risk to language design: migrating programs from existing frameworks to a new framework is a daunting task that developers would be reluctant to do. Hence, it is clear to us that the most practical way to systematically reduce memory pressure is to combine an API-based programming model with a library-based runtime system—as proposed in the thesis—that can extend the performance benefit to a large number of existing systems and programs independently of their computation models.

## 5.3 The ITask Programming Model

This section describes the ITask programming model as well as how it is implemented in two state-of-the-art data-parallel frameworks: Hyracks [105] and Hadoop [102].

### 5.3.1 Programming Model

To turn an existing data-parallel task into an ITask, the developer needs to make the following three changes to the task's original implementation. First, implement the `DataPartition` interface (shown in Figure 5.4). `DataPartition` objects wrap around the framework's existing data representation (e.g., key-value buffer) and are used as an ITask's input and output. Second, make the original task's Java class inherit the `ITask` abstract class (shown in Figure 5.5) and implement its four abstract methods. This can be easily done by restructuring existing code and adding a small amount of new code to handle interrupts. Third, add

a few lines of glue code to specify the input-output relationships between data partitions and ITasks. The development effort is insignificant because most of the work is moving existing code into different methods. For instance, for the 13 StackOverflow problems we reproduced, it took us only a week in total to implement their ITask versions although we had never studied these programs before.

```
1  // The DataPartition abstract class in the library.
2  abstract class DataPartition {
3      // Partition state.
4      int tag, cursor;
5      abstract boolean hasNext();
6      abstract Tuple next();
7      abstract void serialize();
8      abstract DataPartition deserialize();
9  }
```

Figure 5.4: The `DataPartition` abstract class.

**Input and output.** For a data-parallel task, the input dataset is a vector of *data tuples*. A `DataPartition` object wraps around an interval of tuples in the input and different partitions never overlap. Existing data-parallel frameworks already have their own notions of partitions; to use ITask, the developer only needs to wrap an existing partition in a `DataPartition` object.

The `DataPartition` class provides a unified interface for the runtime to track the state of data processing. The internal state of a partition object (Line 4) has two major components: (1) a *tag* that specifies how partial results should be aggregated, and (2) a *cursor* that marks the boundary between the processed and unprocessed parts of the input. We will discuss these two components shortly.

The `DataPartition` class also provides an interface to iterate over data items through the `next` and `hasNext` methods as well as `serialize` and `deserialize` methods to convert data format. These methods are highlighted in red in Figure 5.4. It is up to the developer how

```
1   // The ITask abstract class in the library.
2   abstract class ITask {
3       // Initialization logic.
4       abstract void initialize();
5
6       // Interrupt logic.
7       abstract void interrupt();
8
9       // Finalization logic.
10      abstract void cleanup();
11
12      // Process a tuple; this method should be side-effect-free.
13      abstract void process(Tuple t);
14
15      // Scalable loop.
16      boolean scaleLoop(DataPartition dp) {
17          initialize();
18          while (dp.hasNext()) {
19              if (Monitor.hasMemoryPressure() &&
20                  ITaskScheduler.terminate(this)) {
21                  // Invoke the user-defined interrupt logic.
22                  interrupt();
23                  // Push the partially processed input to the queue.
24                  ITaskScheduler.pushToQueue(dp);
25                  return false;
26              }
27              process(dp.next());
28          }
29          cleanup();
30          return true;
31      }
32  }
```

Figure 5.5: The `ITask` abstract class.

to implement `serialize` and `deserialize`: the data partition can be serialized to disk and the deserialization brings it back into memory; for applications that cannot tolerate disk I/O, the partition can be serialized to large byte arrays and the deserialization recovers the object-based representation. These two methods will be invoked by the IRS to (de)serialize data in response to memory availability (see Section 5.4).

**The ITask abstract class.** An existing task needs to extend the `ITask` abstract class to become an interruptible task, as shown in Figure 5.5. The extension forces the task to implement four new methods which are highlighted in red : `initialize`, `interrupt`, `cleanup`, and `process`. As we will show shortly, the implementation can be easily done by simply re-structuring existing code. Rather than changing the original core semantics of a task (e.g., map or reduce in Hadoop), these methods provide a way for the developer to reason about interrupts.

The `initialize` method loads the input and creates local (auxiliary) data structures before starting data processing; `interrupt` specifies the interrupt handling logic; `cleanup` contains the finalization logic when the *entire* input is processed; and `process` implements the main data processing logic. The `scaleLoop` method is implemented in the library. It iterates over the input data tuples and invokes the `process` method to process a tuple in each iteration (Line 27). It checks memory availability at the beginning of each iteration, ensuring that interrupts can only occur at safe points (i.e., not in the middle of processing a tuple).

When a running task is interrupted, its input and output data partitions still stay in memory unless explicitly released by the developer (such as Line 12 in Figure 5.9). Serialization is not immediately performed. Instead, it is done in a *lazy* manner by the partition manager (Section 5.4.3) when needed.

**Input-output relationship.** The invocation of an ITask has a *dataflow semantics*, which aligns with the dataflow nature of the framework: as long as (1) there exists a `DataPartition` object in the partition queue, which contains the inputs of all tasks, and (2) the cursor of the partition does not point to the end, the task will be automatically invoked to process this partition. To illustrate, consider the code snippet in Figure 5.6, in which `ITaskA` and `ITaskB` are two `ITask` classes; and `DataPartitionA` and `DataPartitionB` are two `DataPartition` classes. These four statements make `ITaskB` a successor of `ITaskA`: whenever

a DataPartitionB is produced by ITaskA, it can be immediately processed by ITaskB.

```
1    ITaskA.setInputType(DataPartitionA.class);
2    ITaskA.setOutputType(DataPartitionB.class);
3    ITaskB.setInputType(DataPartitionB.class);
4    ITaskB.setOutputType(DataPartitionC.class);
```

Figure 5.6: Setting input-output relationship.

Note that the DataPartitionB object can be produced in two scenarios: (1) ITaskA successfully runs to the end, and thus it is the complete result from the processing of DataPartitionA; or (2) ITaskA is interrupted, and thus it is a partial result (e.g., a fragment). In either case, a thread of ITasksB will be launched to process the DataPartitionB object, so that the tasks are fully pipelined.

**ITask state machine.** Putting it all together, Figure 5.7 shows the ITask state machine. The input of the ITask is a DataPartition object in the partition queue (see Section 5.4.3). The object represents either a new, unprocessed partition or a partially processed partition that was pushed into the queue at a previous interrupt. As shown in Figure 5.5, after initialization (Line 4), the data is forwarded to scaleLoop (Line 16), which invokes the user-defined process (Line 13) method to process tuples. Each iteration of the loop processes one tuple and increments the input cursor. If there are no more data items to process (i.e., the cursor points to the end of the partition), the execution proceeds to cleanup (Line 10), which releases resources and outputs a new DataPartition object. This partition will become the input of the next ITask in the pipeline.

Upon memory pressure (Line 19), if the IRS determines that the current ITask instance needs to be terminated (Line 20, based on a set of priority rules discussed in Section 5.4), the user-defined interrupt logic is executed (Line 7) and the input data partition is pushed into the partition queue. The scaleLoop is then exited, and this terminates the thread and produces an output partition representing a result. The input partition cursor marks the

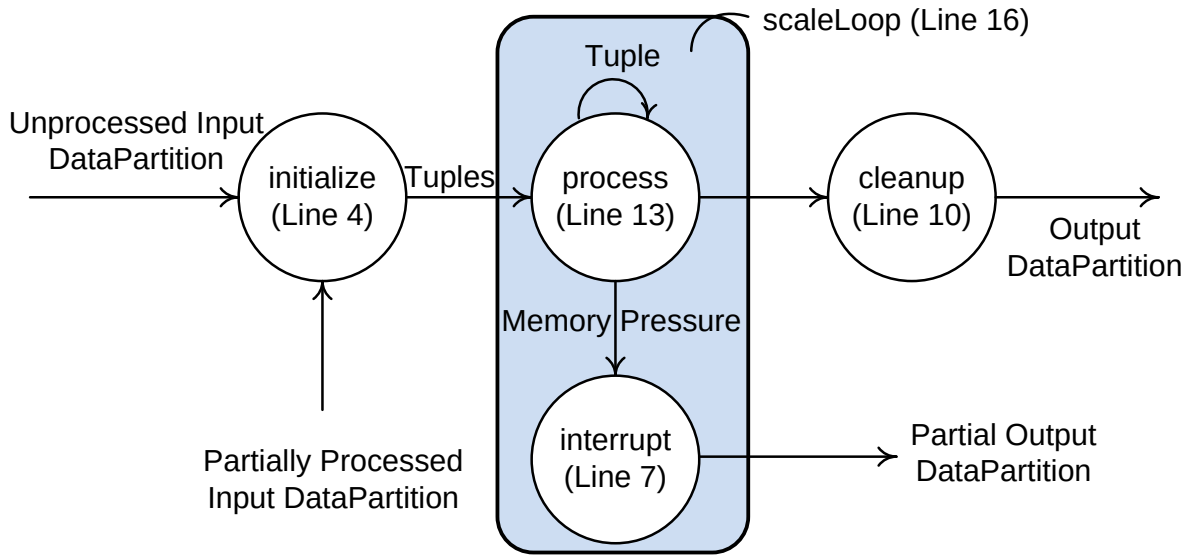


Figure 5.7: The dataflow of an ITask execution.

boundary between processed and unprocessed tuples. Future processing of this partition will start at the cursor when memory becomes available.

An important requirement here is that the `process` method cannot have side effects: it is only allowed to write the output partition and internal objects, which guarantees that the processing of a partially-processed data partition can be resumed without needing to restore a particular external state. Note that this is not a new requirement since side-effect freedom has already been enforced in many existing data-parallel tasks, such as Map/Reduce in Hadoop.

**ITask with multiple inputs.** It is necessary to allow an ITask to process multiple inputs at the same time, especially when a task produces an intermediate result that cannot be directly fed to the next task in the pipeline, as shown by component 4 (b) in Figure 5.2. As discussed earlier, an interrupt to a Reduce task in MapReduce would produce such an intermediate result; an additional follow-up task is needed to aggregate all intermediate results before a final result can be produced and further processed. To enable aggregation, we design an abstract class called `MITask`, which takes multiple data partitions as input.

This class differs from `ITask` only in the definition of the `scaleLoop` method as shown in Figure 5.8.

```
1   abstract class MITask {
2       boolean scaleLoop(PartitionIterator<DataPartition> i) {
3           initialize();
4           while (i.hasNext()) {
5               DataPartition dp = (DataPartition) i.next();
6               while (dp.hasNext()) {
7                   if (...) { // The same memory availability check.
8                       interrupt();
9                       ITaskScheduler.pushToQueue(i);
10                      return false;
11                  }
12                  process(dp.next());
13              }
14          }
15          cleanup();
16          return true;
17      }
18
19      // Other logic is the same as ITask, so omitted here.
20      ...
21  }
```

Figure 5.8: The `MITask` abstract class.

In `MITask`, `scaleLoop` processes a set of `DataPartition` objects. Since the partition queue may have many partition objects available, a challenge here is how to select partitions to invoke an MITask. We overcome the challenge using tags (Line 4 in Figure 5.4): each instance (thread) of an MITask is created to process a set of `DataPartition` objects that have the same tag. Tagging should be done in an earlier ITask that produces these partitions. Obviously, multiple MITask instances can be launched in parallel to process different groups of data partitions with distinct tags. Note that a special iterator `PartitionIterator` is used to iterate over partitions. It is a *lazy*, *out-of-core* iterator that does not need all partitions to be simultaneously present in memory; a partition on disk is loaded only if it is about to be visited.

### 5.3.2 Instantiating ITasks in Existing Frameworks

**Hyracks.** Hyracks [105] is a carefully crafted distributed dataflow system, which has been shown to outperform Hadoop and Mahout for many workloads [17]. In Hyracks, the user specifies a dataflow graph in which each node represents a data operator and each edge represents a connection between two operators. Processing a dataset is done by pushing the data along the edges of the graph. For example, given a big text file for `WordCount` (`WC`), Hyracks splits it into a set of (disjoint) partitions, each of which will be assigned to a worker for processing. Hyracks users dictate the data processing by implementing the `MapOperator` and `ReduceOperator`, connected by a "hashing" connector. The main entry of each operator's processing is the `nextFrame` method.

The `Map` operator does local word counting in each data partition. Its ITask implementation is shown in Figure 5.9, and Hyracks classes/methods are highlighted in blue. . To launch ITasks from Hyracks, we only need to write five lines of code as shown in Lines 24–33 in Figure 5.9: in the `nextFrame` method, we create a `BufferPartition` object using the `ByteBuffer` provided by the framework, set the input-output relationship, and start the ITask execution engine. The `nextFrame` method will be invoked multiple times by the Hyracks framework, and hence, multiple `BufferPartition` objects will be processed by threads running `MapOperator` under the control of the IRS. Upon memory pressure, the `MapPartition` object, *output*, contains a *final result*, and thus the invocation of `interrupt` can directly send it to the shuffle phase (Line 11).

The `Reduce` operator re-counts words that belong to the same hash bucket. Figure 5.10 shows its ITask implementation, we highlight all the Hyracks classes and methods in blue. `Reduce` has a similar implementation to `Map`. However, when memory pressure occurs and a `Reduce` thread is terminated, *output* contains *intermediate results* that are not immediately useful. Hence, before the thread is terminated, we tag *output* with the ID of the channel

```
1   // The Map ITask for Hyracks.
2   class MapOperator extends ITask implements HyracksOperator {
3       MapPartition output;
4       void initialize() {
5           // Create output partition.
6           output = new MapPartition();
7       }
8
9       void interrupt() {
10          // The output can be sent to shuffling at any time.
11          Hyracks.pushToShuffle(output.getData());
12          PartitionManager.release(output);
13      }
14
15      void cleanup() {
16          Hyracks.pushToShuffle(output.getData());
17      }
18
19      void process(Tuple t) {
20          addWordInMap(output, t.getElement(0));
21      }
22
23      // A method defined in HyracksOperator.
24      void nextFrame(ByteBuffer frame) {
25          // Wrap the buffer into a partition object.
26          BufferPartition b = new BufferPartition(frame);
27          // Set input and output.
28          MapOperator.setInputType(BufferPartition.class);
29          MapOperator.setOutputType(MapPartition.class);
30          // Push the partition to the queue and run the ITask.
31          ITaskScheduler.pushToQueue(b);
32          ITaskScheduler.start();
33      }
34  }
```

Figure 5.9: `MapOperator` with ITask for the `WordCount` in Hyracks.

that the input `ByteBuffer` comes from (Lines 12, 17). Because a distinct channel is used for each hash bucket, tagging facilitates the future recognition of partial results that belong to the same hash bucket.

Next, we develop an MITask `MergeTask` that aggregates the intermediate results produced

```
1   // The Reduce ITask for Hyracks.
2   class ReduceOperator extends ITask implements HyracksOperator {
3       MapPartition output;
4
5       void initialize() {
6           // Create output partition.
7           output = new MapPartition();
8       }
9
10      void interrupt() {
11          // Tag the output with the ID of hash bucket.
12          output.setTag(Hyracks.getChannelID());
13          ITaskScheduler.pushToQueue(output);
14      }
15
16      void cleanup() {
17          output.setTag(Hyracks.getChannelID());
18          ITaskScheduler.pushToQueue(output);
19      }
20
21      void process(Tuple t) {
22          addWordInMap(output, t.getElement(0));
23      }
24
25      void nextFrame(ByteBuffer frame) {
26          BufferPartition b = new BufferPartition(frame);
27          // Connect ReduceOperator with MergeTask.
28          ReduceOperator.setInputType(BufferPartition.class);
29          ReduceOperator.setOutputType(MapPartition.class);
30          MergeTask.setInputType(MapPartition.class);
31          ITaskScheduler.pushToQueue(b);
32          ITaskScheduler.start();
33      }
34  }
```

Figure 5.10: `ReduceOperator` with ITask for the `WordCount` in Hyracks.

by the interrupted `Reduce` instances. If a `MergeTask` instance is interrupted, it would create further intermediate results; since these results are tagged with the same tag as their input (Line 11), they will become inputs to `MergeTask` itself when memory becomes available in the future.

```
1   // The Merge MITask for Hyracks.
2   class MergeTask extends MITask {
3       MapPartition output;
4
5       void initialize() {
6           // Create output partition.
7           output = new MapPartition();
8       }
9
10      void interrupt() {
11          output.setTag(input.getTag());
12          ITaskScheduler.pushToQueue(output);
13      }
14
15      void cleanup() {
16          Hyracks.outputToHDFS(output);
17      }
18
19      void process(Tuple t) {
20          aggregateCount(output, t.element(0), t.element(1));
21      }
22  }
```

Figure 5.11: MITask for the `WordCount` in Hyracks.

The careful reader may notice that `MergeTask` has a similar flavor to the merge phase proposed in the Map-Reduce-Merge model [28]. Here the merge task is simply an example of the more general MITask that does not have a specific semantics while the merge phase in [28] has a fixed semantics to merge the reduce results. MITask can be instantiated to implement any M-to-N connector between different data partitions. In this work, an MITask assumes *associativity* and *commutativity* among its input partitions (with the same tag). Because these partitions can be accessed in an arbitrary order, the MITask may compute wrong results if some kind of ordering exists among them. Note that this is a natural requirement for any data-parallel task—any ordering may create data dependencies, making it difficult to decompose the input and parallelize the processing.

Note that the code that achieves the core functionality of the two operators already exists

in the original implementations of their `nextFrame` method. Turning the two operators to ITasks requires only lightweight code restructuring.

**Hadoop.** Hadoop is a MapReduce framework in which tasks only need to extend the `Mapper`/`Reducer` abstract classes, and therefore have narrower semantics than Hyracks tasks. To enable ITask in Hadoop, we let `Mapper` and `Reducer` extend `ITask`, so that all user-defined tasks automatically become ITasks. In addition, the `run` method in `Mapper`/`Reducer` is modified to become a driver to invoke the ITask state machine; its original functionality is moved into the `scaleLoop` method, as shown in Figure 5.12.

```
1   class Mapper extends ITask {
2      // Implementations of the ITask methods.
3      ...
4
5      void run() {
6          initialize();
7          if(!scaleLoop()) return;
8          cleanup();
9      }
10  }
11
12  class MyMapper extends Mapper {
13      void map(T key, K value, ...) {
14          // Data processing logic.
15          ...
16      }
17
18      void process(Tuple t) {
19          map(t.getElement(0), t.getElement(1));
20      }
21  }
```

Figure 5.12: Modified `Mapper` in Hadoop.

**Other frameworks.** It is possible to instantiate ITasks in other data-parallel frameworks as well, such as Spark [137] and Dryad [50]. In general, this can be done by embedding the

ITask state machine into the semantics of existing tasks, an easy effort that can be quickly done by experienced programmers. Furthermore, the majority of the IRS code can be reused across frameworks; slight modification is needed only for the *boundary code* where the IRS interacts with the framework.

### 5.3.3 Discussion

Through these examples, it is clear to see the necessity of providing abstractions for developers to reason about interrupts. In fact, for the three tasks in the Hyracks `WC` example, the handling of interrupts is completely different: when a Map thread is interrupted, the output can be directly sent to shuffling; when a Reduce thread is interrupted, its output needs to be tagged with the hash bucket ID so that it can be appropriately processed by the Merge task; when a Merge thread is interrupted, its output also needs to be tagged appropriately so that it will become its own input. Without the ITask programming model, it is difficult to customize interrupt handling based on the task semantics.A one-to-one mapping does not always exist between an existing task and an ITask. In many cases, the developer may want to represent an existing task using multiple ITasks. The `ReduceOperator` in Hyracks shows such an example. If an existing task is too large, breaking it into multiple ITasks can also better utilize parallelism.

Our experience shows that the effort of writing ITasks is small—since the data processing logic already exists, refactoring a regular task into an ITask usually requires the developer to manually write less than 100 lines of code. For example, the ITask version of `WC` has 309 more lines of code than its regular counterpart. 226 lines, including function skeletons and glue code, can be automatically generated by an IDE or our static analyzer. By default, we employ a third-party library called Kryo [56] to perform data serialization and deserialization; the use of this library only requires a few lines of code for object creation and method calls.

The developer may also write their own serialization and deserialization logic to optimize I/O performance.

Many data-parallel tasks are generated from high-level declarative languages. For example, Hyracks hosts the AsterixDB [100] software stack while Hadoop has an increasingly large number of query languages built on top of it, such as Hive [109] and Pig Latin [78]. Currently, we rely on developers to manually port existing tasks to ITasks. Once the usefulness of ITasks is demonstrated, an important and promising future direction is to modify the compilers of those high-level languages to make them automatically generate ITask code.

## 5.4 The ITasks Runtime System

Once enabled, the IRS manages task scheduling. The IRS contains three components: the partition manager, the scheduler, and the monitor. It determines (1) when to interrupt or re-activate an ITask (monitor, Section 5.4.2), (2) when to serialize or deserialize data partitions (partition manager, Section 5.4.3), and (3) which ITask to interrupt or re-activate (scheduler, Section 5.4.4).

### 5.4.1 The IRS Overview

The IRS starts with a *warm-up* phase in which a slow-start parallelism model is used to gradually scale up the number of threads: initially one thread is created to run the entry task; as the task executes, our system gradually increases the number of threads until it reaches the optimal execution point. If the heap is sufficiently large, the optimal execution point is where the number of threads equals the number of logical cores in the system, which defines the maximum amount of parallelism one can exploit. Otherwise, the IRS stops increasing the number of threads at the moment the available memory size falls below a user-defined

threshold percentage (e.g., $N\%$ of the total heap) to guarantee that the program is executed in a pressure-free environment. The warm-up phase serves as an initial guard to managing memory consumption: although the system memory utilization may change later, the IRS is unlikely to need to tune memory usage immediately after the program starts.

From the task code, we develop a static analysis that builds a *task graph* based on the input/output relationship of the ITasks in the program. The task graph will be used later to determine which ITask instances should be interrupted and re-activated.

### 5.4.2 Monitor

The IRS monitors the global memory usage and notifies the scheduler of the system's memory availability. Algorithm 2 shows the main logic of our monitor. As discussed earlier, we design the monitor by focusing on LUGCs. Specifically, we consider a GC as a LUGC if the GC cannot increase the free memory size above $M\%$ of the heap size, where $M$ is a user-specified parameter. The monitor also watches the execution to identify periods in which extra memory is available. These periods occur when the size of free memory is $\geq N\%$ of the heap size. If such a period is detected, the monitor sends a "GROW" signal (Line 5) to instruct the scheduler to increase the number of ITask instances. The scheduler then picks a task, finds a partition object that can serve as its input, and creates a new thread to run it (Lines 7–11 in Algorithm 4). We used $N = 20$ and $M = 10$ in our experiments and they

**Algorithm 2** The major logic of the IRS monitor.

```
1: repeat
2:     if Long and Useless GC occurs then
3:         SignalScheduler("REDUCE")
4:     if freeHeap ≥ N% * totalHeap then
5:         SignalScheduler("GROW")
6:     Sleep(interval)                    ▷ interval is a system parameter.
7: until The whole system is shut down.
```

worked well.

### 5.4.3 Partition Manager

Once a partition object is created, such as in the `nextFrame` method in Figure 5.9, it is registered with the partition manager. The manager puts the object into a global partition queue (as mentioned in Section 5.3.1) that contains all partially-processed and unprocessed partitions. These data partitions may be in serialized or deserialized form. How a partition is serialized depends on the `serialize` method defined in the partition class. While there can be multiple ways to implement the method, in our current prototype, data serialization writes a partition to disk and deserialization brings it back to memory. To avoid thrashing, we keep track of each partition's latest serialization and deserialization timestamps. A data partition is not allowed to be serialized if a deserialization of the partition was performed recently within a given time period, unless there are no other data partitions with earlier deserialization timestamps. If thrashing still occurs, the partition manager notifies the monitor, which then sends a "REDUCE" signal to the scheduler to terminate threads.

**Algorithm 3** The major logic of the IRS partition manager.

```
1: repeat
2:     Message m ← Listen()                          ▷ Wait for the next message.
3:     if m ≡ "SERIALIZE" then                       ▷ Memory pressure occurs.
4:         ScanAndDump()
5:         if IsThrashing() ≡ true then
6:             SignalScheduler("REDUCE")
7:     else if m ≡ "GET" then                        ▷ The scheduler asks for a partition.
8:         SendAnAvaiablePartition()
9: until The whole system is shut down.
```

Upon receiving a "REDUCE" signal from the monitor, the scheduler first checks with the partition manager to see if it can serialize some data partitions that are associated with already interrupted tasks (Lines 3–6 in Algorithm 3). In many cases, this is sufficient to remove memory pressure so that we do not need to interrupt more tasks. The partition

manager uses the following rules to determine which partitions to serialize first; background threads then write the data to disk.

- **Temporal Locality Rule:** Partitions that serve as inputs to the ITasks that are closer to the currently executed ITask on the task graph have a higher priority to stay in memory.
- **Finish Line Rule:** A fast turn-around from the initial inputs to the final output is a desired property of any system. To optimize for this property, the inputs to the ITasks that are closer to the finish line (i.e., lower on the task graph) have a higher priority to be retained in memory.

### 5.4.4 Scheduler

The scheduler determines which ITasks and how many instances of them to run. If serialization done by the partition manager cannot alleviate the memory pressure, the scheduler will reduce the number of task instances (Lines 5–6). The selection of ITask instances to interrupt is based on the following three rules:

- **MITask First Rule:** Threads running MITasks have the highest priority to continue running. Since an MITask often performs data merging, terminating the thread would create a large number of input/output fragments.
- **Finish Line Rule**: A thread running an ITask closer to the finish line has a higher priority to continue to run.
- **Speed Rule**: For a set of threads running the same ITask, the slowest thread will be terminated first. The processing speed of a thread is determined by the number

of `scaleLoop` iterations executed between two consecutive memory usage checks (performed by the monitor). The speed of a thread depends primarily on the characteristics of the input it processes. For example, it is quicker for a word count task to process a partition with more duplicated words than a partition in which most words appear only once. The scheduler favors fast-running threads so that they can quickly finish, giving more memory for the rest of the (slower) tasks to finish.

**Algorithm 4** The major logic of the IRS scheduler.

```
1:  repeat
2:      Message m ← Listen()                          ▷ Wait for the next message.
3:      if m ≡ "REDUCE" then                          ▷ Memory pressure is detected.
4:          SignalPartitionMananger("SERIALIZE")
5:          while freeHeap < M% * totalHeap do
6:              InterruptTaskInstance()
7:      else if m ≡ "GROW" then                       ▷ Enough memory for a new task instance.
8:          while freeHeap ≥ N% * totalHeap do
9:              SignalPartitionMananger("GET")        ▷ Ask for a partition.
10:             partition = ReceivePartitionFromManager()
11:             IncreaseTaskInstance(partition)
12: until The whole system is shut down.
```

When a task thread is selected to be interrupted by the scheduler, for this thread, the method call `ITaskScheduler.terminate(this)` (Line 20 in Figure 5.5) will return true and its `interrupt` method will be executed. The scheduler continues to terminate threads until the memory usage goes below the threshold. Upon receiving a "GROW" signal from the monitor, the scheduler creates a new thread to run an ITask based on the following two rules (Lines 8–11 in Algorithm 4):

- **Spatial Locality Rule:** We favor an ITask that has *in-memory* inputs. These partitions can be processed first before the manager needs to load partitions from disk.
- **Finish Line Rule:** We favor an ITask that is closer to the finish line.

When an ITask is selected and its input partition is on disk, the partition manager loads the partition back into memory transparently to the scheduler.

### 5.4.5 Fault Tolerance

Since ITask does not add new semantics onto existing tasks, we can reuse the fault tolerance mechanism provided by the underlying framework. In addition, we develop our own ITask-based fault tolerance support inside the IRS, which can be used by frameworks that do not provide fault tolerance. In particular, we add a checkpointer inside the partition manager, which, once enabled, can record all data partitions in the queue to disk. If the node fails and restart is needed, the IRS can quickly locate the most recent checkpoint, load all partitions into memory, and resume ITasks from there. Since data partitions all have cursor information, restarting the execution from a failure is the same as resuming the execution from an interrupt.

## 5.5 Evaluation

We have implemented the ITask library and the IRS on Hadoop and Hyracks. These implementations have approximately 30,000 lines of Java code. We ran Hadoop and Hyracks on a 11-node Amazon EC2 cluster. Each node (a c3.2x large instance) has 2 quad-core Intel Xeon E5-2680 2.80GHz processors, 15GB of RAM, and one RAID-0 comprised of 2 80GB SSDs. The cluster runs Linux 3.10.35 with enhanced networking performance. We used Java HotSpot(TM) 64-bit Server VM (build 24.71-b01) for all experiments. The state-of-the-art parallel generational garbage collector was used.

**Methodology.** Since Hadoop is a popular data-parallel framework that has many OMEs reported on StackOverflow, we focus our first set of experiments (Section 5.5.1) on reproducing real-world problems in Hadoop and understanding whether the ITask implementations can help these programs survive OMEs and successfully process their entire datasets. The second set of experiments (Section 5.5.2) focuses on comparing performance and scalability between the ITask programs and their original versions on Hyracks over various heap configurations and data sizes. For Hadoop and Hyracks, we used version 2.6.0 and 0.2.14 in our experiments, respectively. YARN was enabled when Hadoop was run.

### 5.5.1 ITasks in Hadoop

Reproducing real-world problems requires running the programs under the same configurations and with the same datasets as their original settings. This effort is extremely labor-intensive and time-consuming, especially in a distributed environment.

We have successfully reproduced and implemented ITasks for thirteen problems among the 73 problems we have studied. On average, it took us about a week to set up a distributed configuration as described on StackOverflow, manifest the problem, understand its semantics, and develop its ITask version. For all of these 13 problems [71, 135, 76, 83, 2, 90, 4, 88, 27, 8, 22, 82, 1], their ITask versions successfully survived memory pressure and processed the given datasets. We report detailed experimental results for a diverse array of five problems including map-side aggregation (MSA) [71], in-map combiner (IMC) [135], inverted-index building (IIB) [76], word cooccurrence matrix (WCM) [83], customer review processing (CRP) [2]. The StackOverflow data dump [96] and Wikipedia data dump [116] are used in this set of experiments. All the programs except `CRP` run on the full StackOverflow/Wikipedia dump, and `CRP` uses a sample of the Wikipedia data dump. For all the executions, the HDFS block size is set to 128MB.

| *Name* | `MSA` | `IMC` | `IIB` | `WCM` | `CRP` |
|---|---|---|---|---|---|
| *Dataset* | StackOverflow | Wikipedia | Wikipedia | Wikipedia | Wikipedia* |
| *Size* | 29GB | 49GB | 49GB | 49GB | 5GB |
| *Map Heap* | 1GB | 0.5GB | 0.5GB | 0.5GB | 1GB |
| *Reduce Heap* | 1GB | 1GB | 1GB | 1GB | 1GB |
| *# Mapper* | 6 | 13 | 13 | 13 | 6 |
| *# Reducer* | 6 | 6 | 6 | 6 | 6 |
| *CTime* | 1047 | 5200 | 1322 | 2643 | 567 |
| *PTime* | **48** | 337 | 2568 | 2151 | 6761 |
| *ITime* | 72 | **238** | **1210** | **1287** | **2001** |

Table 5.1: Hadoop performance comparisons for five real-world programs.

Table 5.1 shows the configurations in which the problems manifest and the execution times of different versions of the programs. The datasets used and their sizes are reported in Row *Data* and *Size*; the max heap size for each Map and Reduce task is shown in Row *Map Heap* and *Reduce Heap*; the maximum numbers of Mappers and Reducers on each machine are reported in Row *# Mapper* and *# Reducer*. Time elapsed before each original program ran out of memory is also reported in Row *CTime*. For each problem, we carefully read its recommended fixes on StackOverflow. For all the problems but `CRP`, the recommended fixes were changing parameters (# Map/Reduce workers on each node or task granularity). After a long and tedious tuning process, we observed that reducing worker numbers and/or sizes of data splits was indeed helpful. For these four problems, we ended up finding configurations under which the programs could successfully run to the end. The *shortest* running time under different working configurations we found is reported in Column *PTime*. For `CRP`, since the recommended fix was to break long sentences, we developed a tool that automatically breaks sentences whose length exceeds a threshold. Since we are not experts in natural language processing, this tool broke sentences in a naïve way and might be improved when considering domain knowledge.

The ITask versions of these problems were executed under the same Hadoop configuration as their original versions (as shown in Table 5.1). Their running time is shown in Section *ITime*. The numbers highlighted in bold are the lowest running times for the successful executions.

Comparing *PTime* and *ITime*, we can observe that the ITask versions have much better performance (i.e., on average 2× faster) than manually-tuned versions in most cases. The only exception is for `MSA`, where its *ITime* is 1.5× longer than *PTime*. An investigation identified the reason: since the first Map task loads a very large table to perform hash join, the program has to be executed with only a small degree of parallelism. Manual tuning sets the maximum number of workers to 1 thus paying no additional runtime cost while its ITask version alternates the number of workers between 1 and 2—the tracking overhead cannot be offset by the exploited parallelism. Another observation is that *CTime* may be much longer than *PTime* and *ITime*. This is because these programs all suffered from significant GC overhead as well as many restarts before YARN eventually gave up retrying and reported the crash.

**Memory savings breakdown.** Table 5.2 shows a detailed breakdown of memory savings from releasing various parts of an ITask's consumed memory. Note that different programs have different semantics and therefore they benefit from different optimizations. For instance, the OME in `MSA` occurs in a Map task; the task has an extremely large key-value buffer, which contains final results that can be pushed to the next stage. Hence, `MSA` benefits mostly from pushing out and releasing final results. As another example, `WCM` crashes in a Reduce task; therefore, it has large amounts of intermediate results that can be swapped out and merged later. These promising results clearly suggest that ITask is effective in reducing memory pressure for programs with different semantics and processing different datasets.

| *Name* | *Processed Input* | *Final Results* | *Intermediate Results* | *Lazy Serialization* |
|---|---|---|---|---|
| `MSA` | 14.9KB | 33.7GB | 0.0GB | 6.0GB |
| `IMC` | 18.4KB | 23.1GB | 0.0GB | 0.0GB |
| `IIB` | 70.1MB | 0.0GB | 7.1GB | 2.3GB |
| `WCM` | 192.6MB | 0.0GB | 14.3GB | 1.5GB |
| `CRP` | 1.0KB | 1.2GB | 112.8MB | 0.0GB |

Table 5.2: A detailed breakdown of memory savings.

With these programs, we have also compared ITask executions with naïve techniques that (1) kill a task instance upon memory pressure and later reprocess the same data partition from scratch (without using ITasks) and (2) randomly pick threads to terminate and data partitions to resume (without using our priority rules in Section 5.4.4). The results show that the ITask executions are up to 5× faster than these naïve techniques.

### 5.5.2 ITasks in Hyracks

The goal of this set of experiments is to understand the improvement in performance and scalability ITask provides for a regular data-parallel program. The 11-node cluster was still used; unless specified, each node used a 12GB heap as the default configuration.

**Benchmarks.** We selected the following five already hand-optimized applications from Hyracks' code repository and ported their tasks to ITasks. These programs include word count (`WC`), heap sort (`HS`), inverted index (`II`), hash join (`HJ`), and group-by (`GR`). Note that these applications were selected because (1) they provide a basis for many high-level applications built on top of Hyracks, such as AsterixDB [5, 100] and Preglix [20]; and (2) they were used extensively in prior work [17, 18, 10] to evaluate Hyracks and other high-level applications. Since Hyracks does not allow the use of Java objects, these programs are already well-tuned and expected to have high performance.

`WC` returns to the user the number of times each word occurs in a corpus. Section 5.3 already provided a description. `HS` heap-sorts a set of numbers (i.e., id of vertices in the Yahoo Webmap). `II` maps each word back to the document containing the word. `HJ` takes as input TPC-H's Customer and Order tables, in which each record represents a customer and an order, respectively. The application matches customer information with the order based on the ID of the customer. Its functionality is equivalent to the following SQL query:

```
SELECT * FROM customer, order WHERE customer.ID=order.custID;
```

GR takes as input the TPC-H's LineItem table, each record of which represents an item included in an order. The goal is to calculate the total item counts in each order and the total price of that order. It is functionally equivalent to the following SQL query:

```
SELECT orderID, SUM(quantity), SUM(price) FROM lineitem GROUP BY orderID;
```

| *Size* | *# Vertices* | *# Edges* |
|---:|---:|---:|
| 72GB | 1,413,511,390 | 8,050,112,169 |
| 44GB | 992,128,706 | 4,474,491,119 |
| 27GB | 587,703,486 | 2,441,014,870 |
| 14GB | 143,060,913 | 1,470,129,872 |
| 10GB | 75,605,388 | 1,082,093,483 |
| 3GB | 24,973,544 | 313,833,543 |

Table 5.3: WC, HS, and II's inputs: the Yahoo! Webmap and its subgraphs.

On average, each program has around 2K lines of Java code—e.g., the smallest program WC has 550 lines of code (LOC) and the largest program HJ has 3.2K LOC. It took us one person week to convert these five programs into ITask programs.

Our datasets came from two sources: the Yahoo Webmap [129], which is the largest publicly available graph with 1.4B vertices and 8.0B edges, and the TPC-H data generator [111], which is the standard data warehousing benchmark tool popularly used in the data management community. For TPC-H, we used the following three tables: Customer, Order, and LineItem. Table 5.3 and Table 5.4 show their statistics.

| *Scale* | *Size* | *# Customer* | *# Order* | *# LineItem* |
|---:|---:|---:|---:|---:|
| 150× | 150.4GB | $2.25\times10^7$ | $2.25\times10^8$ | $9.00\times10^8$ |
| 100× | 99.8GB | $1.50\times10^7$ | $1.50\times10^8$ | $6.00\times10^8$ |
| 50× | 49.6GB | $7.50\times10^6$ | $7.50\times10^7$ | $3.00\times10^8$ |
| 30× | 29.7GB | $4.50\times10^6$ | $4.50\times10^7$ | $1.80\times10^8$ |
| 20× | 19.7GB | $3.00\times10^6$ | $3.00\times10^7$ | $1.20\times10^8$ |
| 10× | 9.8GB | $1.50\times10^6$ | $1.50\times10^6$ | $6.00\times10^7$ |

Table 5.4: HJ and GR's inputs: TPC-H data.

**Scalability of the original programs.** We first ran each original version with various numbers of threads (between 1 and 8). Detailed performance comparisons among these configurations are shown in Figures 5.13–5.17. The task granularity is 32KB, and the configurations in which the program ran out of memory are omitted. In each graph, bars are grouped under input sizes. From left to right, they show the execution times with growing numbers of threads. For each bar, the time is further broken down to the GC time (the upper part) and the computation time (the lower part). These graphs clearly demonstrate that increasing thread number does not always lead to better performance. For example, for `HS` and `GR` at their largest inputs (27GB and 50×, respectively), their fastest executions used 6 threads.

We have also varied task granularities (between 8KB and 128KB). Table 5.5 summarizes the largest datasets in our experiments to which these programs could scale and the configurations under which the best performance was obtained over these datasets. In this table, *DS* reports the largest datasets in our experiments to which the programs scaled; # *K* and # *T* report the numbers of threads and the task granularities for which the best performance was obtained when processing the datasets shown under *DS*.

Most of these programs suffered from significant GC effort when large datasets were processed. For instance, for `HS` and `GR`, their GC time accounts for 49.14% and 52.27% of their execution time, respectively. Among the programs, `II` has the worst scalability due to the large in-memory maps it maintains: `II` was only able to process the smallest dataset (3GB). Even the single-threaded version of `II` could not process the 10GB dataset on the cluster. `HJ` scales the best: each slave was able to process up to 10GB input with a 12GB heap. The bottleneck that prevents `HJ` from processing the 150× dataset is its in-memory Customer table. This table consumes much of the heap leaving the program with little space for the actual join. However, the bottleneck is difficult to optimize away because the table is required to be fully constructed and present throughout the execution. During the processing

of the 150× dataset, the table was too large and the program was always under memory pressure, making the GC time exceed the JVM's limit.

| *Name* | *DS* | *# K* | *# T* |
|---|---|---|---|
| Word Count (`WC`) | 14GB | 2 | 32KB |
| Heap Sort (`HS`) | 27GB | 6 | 32KB |
| Inverted Index (`II`) | 3GB | 8 | 16KB |
| Hash Join (`HJ`) | 100× | 8 | 32KB |
| Group-By (`GR`) | 50× | 6 | 16KB |

Table 5.5: The configurations which provide the best scalability/performance.

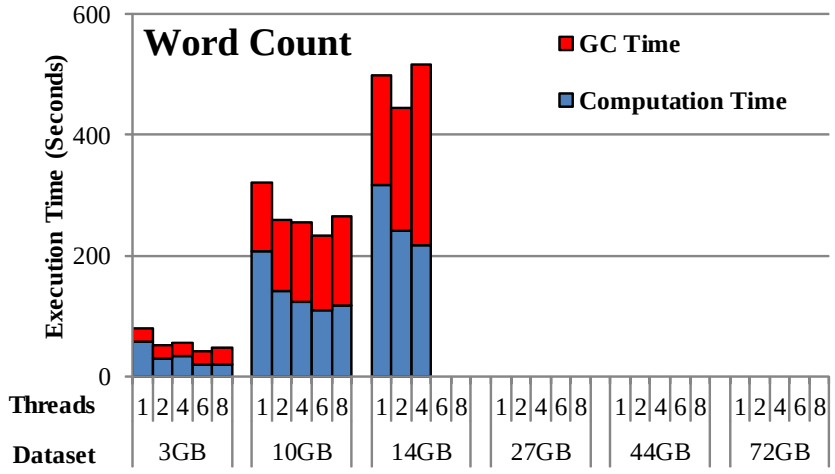


Figure 5.13: Performance of the original WC with different threads.

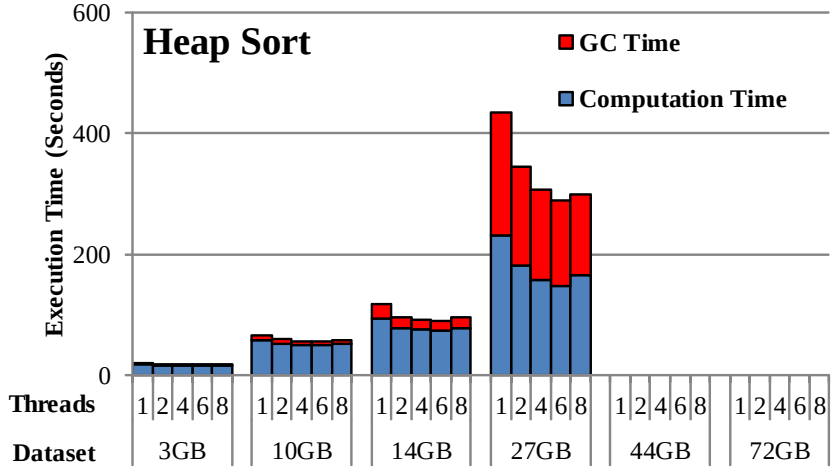


Figure 5.14: Performance of the original HS with different threads.

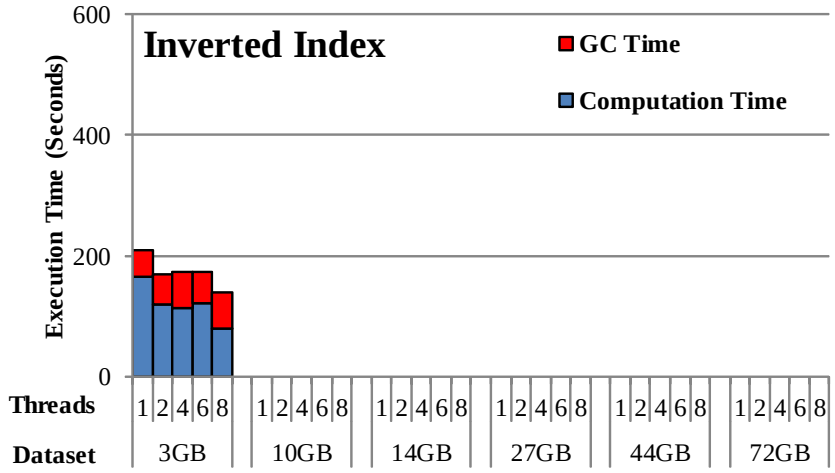


Figure 5.15: Performance of the original II with different threads.

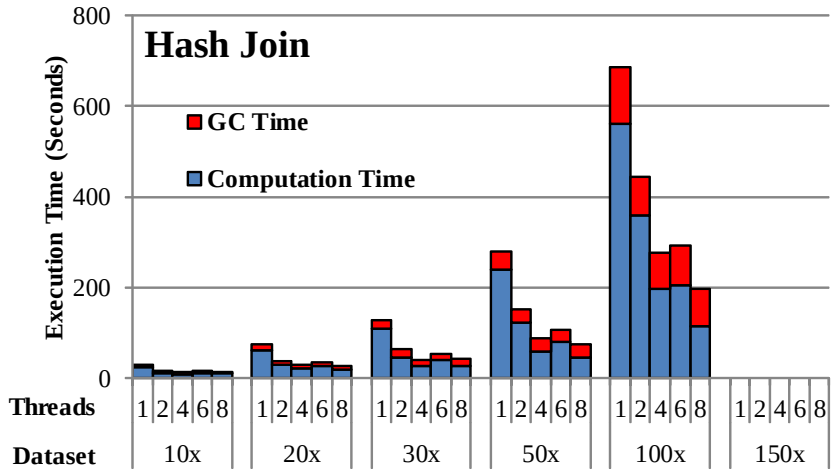


Figure 5.16: Performance of the original HJ with different threads.

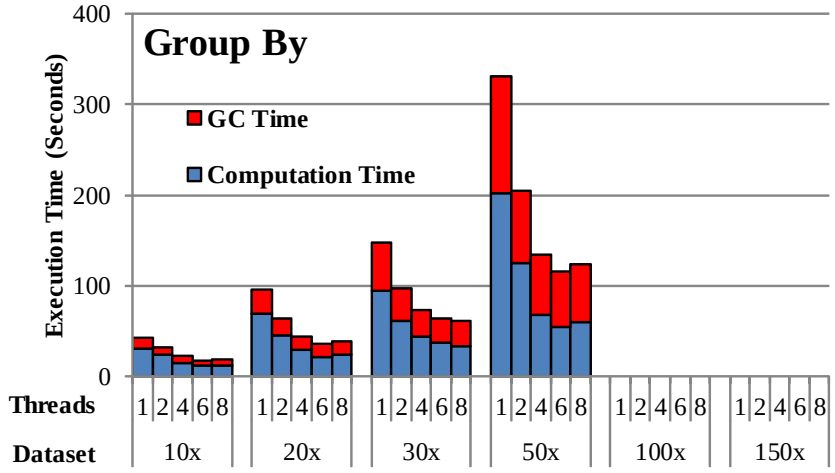


Figure 5.17: Performance of the original GR with different threads.

**Performance improvements.** For each program, we next compare its ITask version with the original version under the configuration that yields the best performance (as shown in Table 5.5). We have measured both running time and heap consumption in this experiment. To eliminate the execution noise on the cluster, we ran the ITask version 5 times with a 12GB heap. Figures 5.18–5.22 report the geometric means of these measurements. Bars represent running time of successful executions and are grouped by input sizes. In each group, the first/left bar corresponds to the best configuration for the original program and the second/right bar corresponds to the ITask version. Each bar is also broken down into GC (the upper part) and computation time (the lower part). The heap consumptions are represented by lines, each reporting the maximum heap usage of the program across all the slaves.

| *Name* | *# TS* | *% TS* | *# HS* | *% HS* | *Scalability* |
|---|---|---|---|---|---|
| `WC` | 5/6 | 39.63% | 5/6 | 13.81% | 5.14× |
| `HS` | 4/6 | 10.85% | 5/6 | 7.57% | 2.67× |
| `II` | 6/6 | 27.53% | 5/6 | -9.28% | 24.00× |
| `HJ` | 6/6 | 66.45% | 3/6 | -5.16% | 6.00× |
| `GR` | 6/6 | 61.35% | 5/6 | 26.62% | 5.00× |
| GeoMean | 27/30 | 44.95% | 23/30 | 7.65% | 6.29× |

Table 5.6: A summary of the performance improvements from ITask.

Table 5.6 summarizes the time and space savings from ITask; *# TS* and *# HS* report ratios at which an ITask program outperforms its regular counterpart in execution time and heap consumption, respectively; *% TS* and *% HS* report the ITask's reductions in time and heap consumption, respectively, for the inputs both versions have successfully processed; *Scalability* reports the ratios between the sizes of the largest datasets the two versions can scale to.

Among the 30 (both failed and successful) executions of these programs, their ITask versions were faster than their original versions in 27 of them. The 3 executions (for `WC` and `HS`) in which the ITask version was slower all processed very small datasets, plus the time differences

are negligible (i.e., 1.61%). The average time reduction ITask has achieved across the 17 successful executions is 44.95%. The majority of these savings stems from significantly reduced GC costs.

The ITask programs are also memory-efficient. In 23/30 executions, the maximum heap consumption is smaller than that of the Java version. This is due to IRS's ability to move data in and out. However, in the cases that the inputs are small, the ITask version consumes more memory because of the tracking/bookkeeping performed in the IRS. The overall memory space reduction across the executions in which both versions succeeded is 7.65%; furthermore, the original programs failed in 13 out of the 30 executions while the ITask version succeeded in all of them.

**Scalability improvements.** The last column of Table 5.6 shows how well the ITask programs scale. These measurements are computed as the ratios between the sizes of the largest inputs the ITask-based and the original programs can process. As shown in Figures 5.18–5.22, all the ITask programs successfully processed all input sizes in our experiments while none of the original programs could. Even for the highly-scalable `HJ` program, its ITask version well outperforms its original version. Overall, ITask has achieved a 6.29× scalability

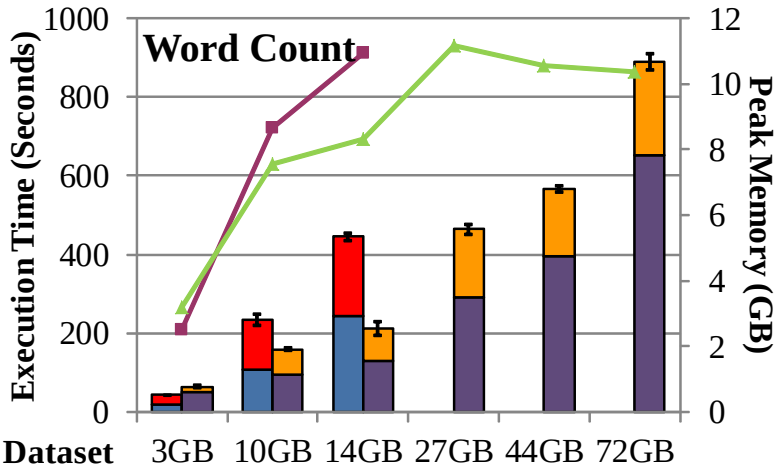


Figure 5.18: Comparisons between the ITask `WC` and its Java counterpart.

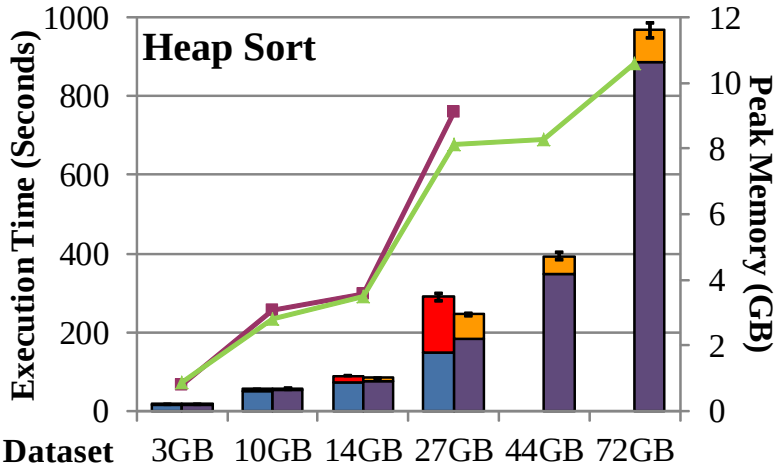


Figure 5.19: Comparisons between the ITask `HS` and its Java counterpart.

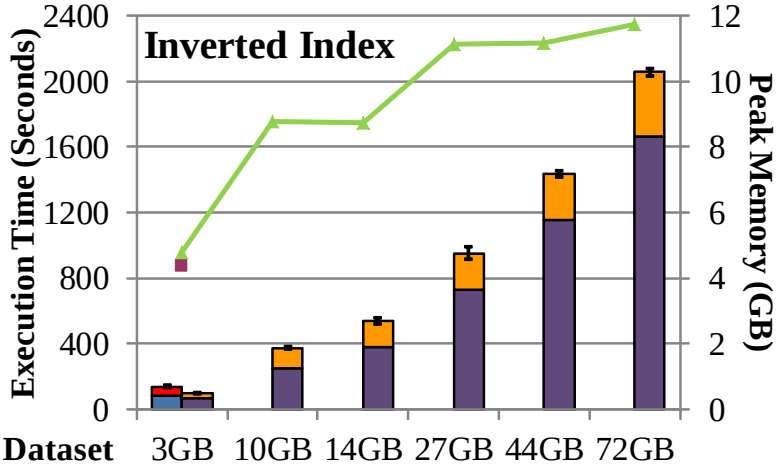


Figure 5.20: Comparisons between the ITask `II` and its Java counterpart.

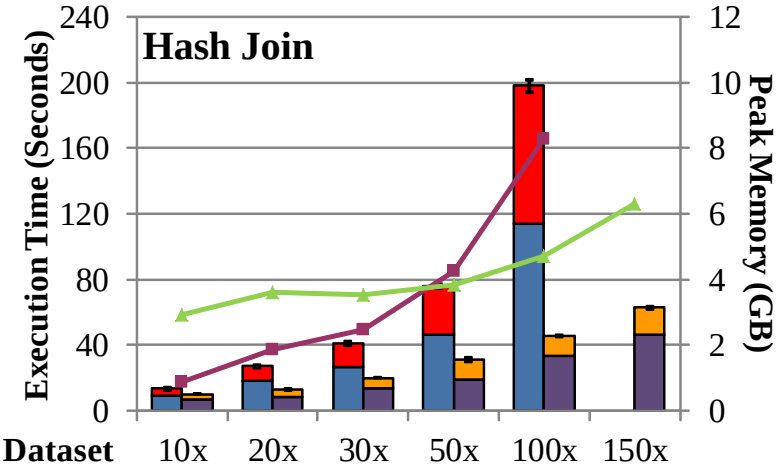


Figure 5.21: Comparisons between the ITask `HJ` and its Java counterpart.

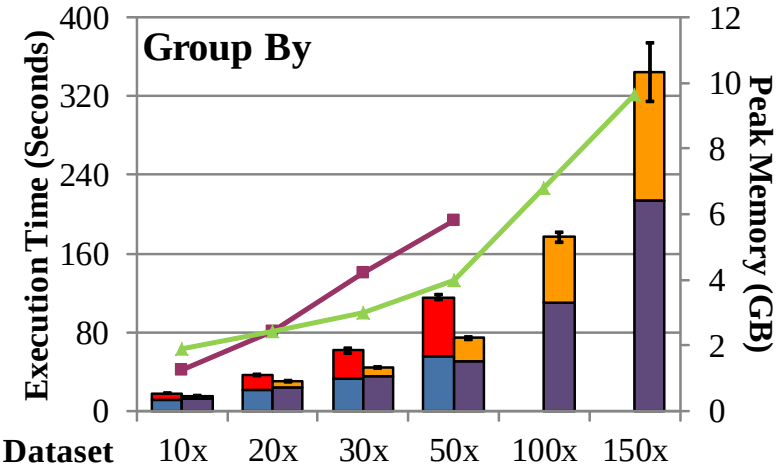


Figure 5.22: Comparisons between the ITask `GR` and its Java counterpart.

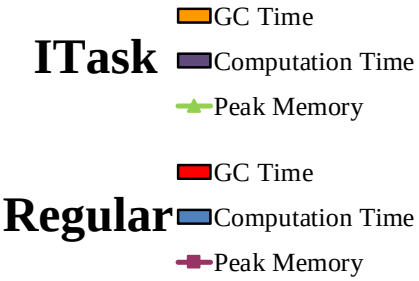


Figure 5.23: The legend of Figure 5.18–5.22.

improvement. We performed an additional test on further larger input sizes to understand the scalability upper bound of these programs. This experiment shows that the ITask versions of `HJ` and `GR` could successfully process a 600× and a 250× dataset, respectively. These results indicate that the scalability improvement ITask provides may be even larger when bigger datasets are used.

**Using different heaps.** To understand how an ITask program behaves under different heaps, we ran `WC` and `II` on the 10GB dataset, under a 12GB, 10GB, 8GB, and 6GB heap. Their detailed performance is shown in Figure 5.24 and 5.25. To summarize, when the input size is fixed, the performance of an ITask program does not change much with the heap size, while a Java program can easily crash when the heap size is reduced. For example, the original program of `WC` could not process the 10GB dataset with the 8GB and 6GB heap,

while its ITask version successfully processed the whole dataset with the 6GB heap, yielding a running time comparable to that with the 10GB heap. In addition, the GC component is less than 10% of the total time.

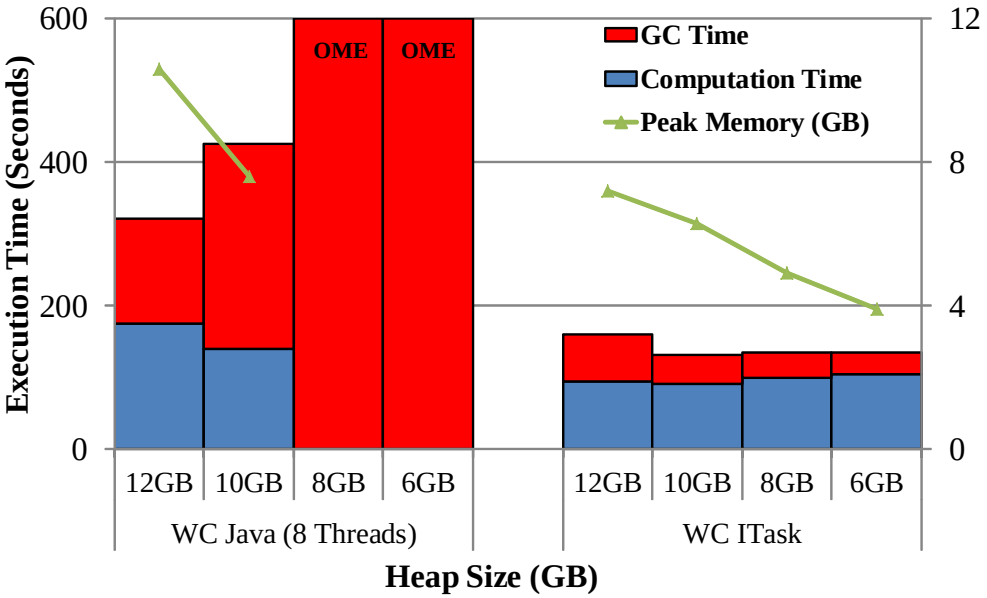


Figure 5.24: The performance of `WC` whiling changing the heap size.

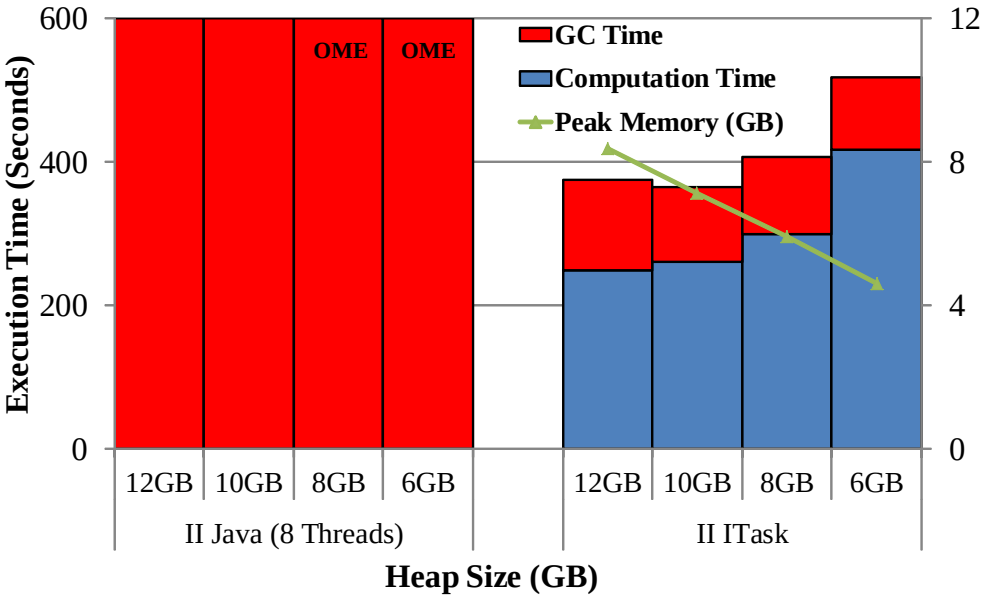


Figure 5.25: The performance of `II` whiling changing the heap size.

In order to closely examine ITask's adaptive execution, we counted the number of active ITask instances during the execution of `WC` on the 14GB dataset. Figure 5.26 shows how

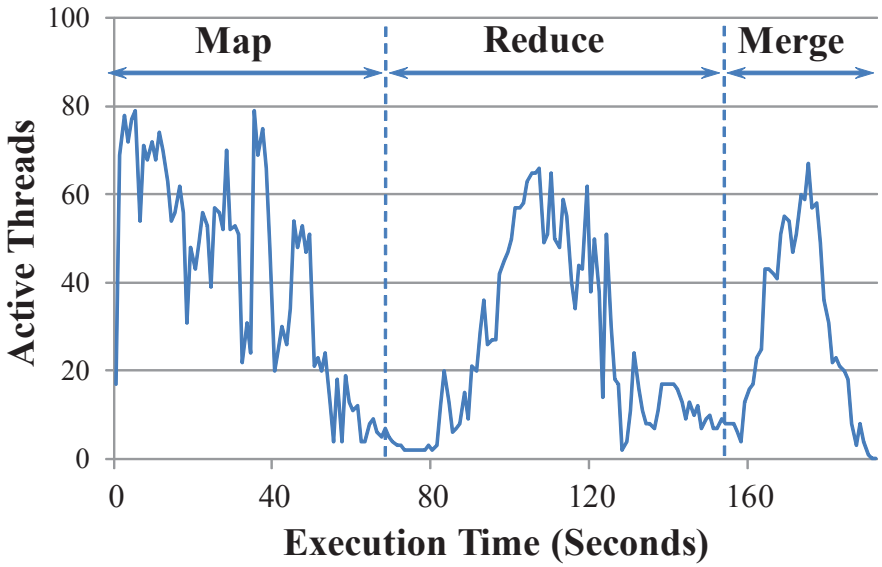


Figure 5.26: The number of active ITask instances during the execution.

the number of threads changes as the execution progresses on the cluster. The cluster has a maximum of 80 workers. Threads for Map and Reduce can overlap. The program finished in 192 seconds and the average number of active threads on each slave was 3.16. Figure 5.26 clearly shows that an ITask execution is very dynamic and our runtime system can automatically adapt the active worker numbers to memory availability while keeping as many active workers as possible. It would be interesting to also measure the cost of disk I/O. However, we create background threads for disk operations, making it difficult to separate out the I/O cost. Writing data partitions occurs simultaneously with the data processing while reading data can introduce stalls. These stalls contribute to 5-8% of the execution time.

## 5.6 Summary and Interpretation

Since no existing technique can systematically address the individual node memory pressure problem, we propose ITask, which treats memory pressure as interrupt, to help data-parallel applications survive memory pressure. Its non-intrusive design makes it easy for developers

to integrate this approach into a distributed framework. The experimental results show that ITask can solve real-world memory pressure problems without manually tuning parameters, and dramatically improve the performance and scalability of data-parallel programs.

ITask provides the following unique features unseen in existing systems. First, ITask works *proactively* in response to memory pressure. We take actions to interrupt tasks and reclaim memory when we observe the first signs of pressure. Hence, ITask can quickly take the system back to the memory "safe zone" before much time is spent on garbage collection (GC) and way before an out-of-memory error occurs. As a result, ITask improves both scalability (because out-of-memory crashes are avoided) and performance (because GC time is reduced).

Second, ITask uses a *staged approach* to lower memory consumption for an interrupted task $t$. It consists of five steps, covering all components of a running task's user-level memory usage with varying cost-benefit tradeoffs: (1) heap objects referenced by local variables during $t$'s execution are all released; (2) the part of the input data already processed by $t$ is released; (3) final results generated by $t$ are pushed out; (4) intermediate results that need to be aggregated before being pushed out will be aggregated by a follow-up task in an out-of-core manner; and (5) other in-memory data are serialized (e.g., to disk). Not all of these steps will be performed at every interrupt: the handling is done *lazily* and it stops whenever memory pressure disappears.

Third, ITask consists of a new programming model and a runtime system that can be easily implemented in any existing data-parallel framework. The programming model provides interrupt handling abstractions for developers to reason about interrupts. The ITask runtime system (IRS) performs task scheduling and runtime adaptation. The amount of work needed to implement ITask is minimal: the user simply restructures code written for existing data-parallel tasks to follow new interfaces, and the framework needs to be slightly modified to delegate task scheduling to the IRS. The IRS sits *on top of* the framework's job scheduler,

providing complementary optimizations and safety guarantees.

# Chapter 6

# Related Work

## 6.1 Finding Performance Bugs

Jin *et al.* [52] study a number of performance bugs in real-world programs and develop a pattern-based approach to find bugs in the program source code. Song and Lu [93] propose a statistical approach to finding performance bugs in real-world software. Xu *et al.* propose static [126, 127] and dynamic [123] techniques to detect various kinds of performance problems. Recent work from [77] proposes a static analysis that can find redundant container traversals. Mitchell *et al.* propose heap analysis techniques [69, 68] that can correlate system metrics with program behaviors. Nistor *et al.* [75] propose a runtime technique to detect performance problems by looking for code loops that exhibit similar memory-access patterns. Han *et al.* [46] identify performance problems by mining large numbers of stack traces. Stewart *et al.* propose EntomoModel [98], a dynamic framework that uses decision tree classification and a design-driven performance model to identify and avoid performance anomaly. Caramel [73] detects loop-related performance bugs that can be easily fixed by adding conditional breaks. These existing works are postmortem debugging techniques,

focused on finding the root cause of a performance problem that already manifests in a user-provided test execution or actual production run, while our work provides a general framework to describe and amplify performance problems during testing.

Xiao *et al.* [118] develop a delta inference technique that predicts workload-dependence performance bugs by using models with respect to workload size to infer iteration counts. Yu *et al.* [132] propose a technique that collects large numbers of traces to measure performance impacts and identify their root causes. WuKong [140] is an automated technique that builds a feature-based behavioral model to predict bugs that manifest at large-system scale. WuKong falls into the category of *learning-based bug detection* where small-scale training runs are employed to build the model, while our approach uses developer-provided symptom specifications to amplify performance problems.

## 6.2 Test Amplification Techniques

Test amplification is a notion wherein a single test execution can be used to learn much more information about a program's behavior than is directly exhibited by that particular execution. Zhang *et al.* [139] proposes a test amplification technique that exhaustively explores the space of exceptional behaviors to validate exception handling code. Work from [60] uses a static information flow analysis to amplify the result of a single test execution over the set of all inputs and interleavings for verifying properties of data-parallel GPU programs. Dynamic test generation techniques such as [21, 40, 87] can also be considered as test amplification techniques which generate many tests from one test execution to achieve path coverage.

## 6.3 Domain-Specific Languages

The Broadway compiler project [45] contains an annotation language and a compiler that can customize a library implementation for specific application requirements. Other domain-specific languages and compilers attempt to either incorporate application semantics into compilation to improve performance (such as [38]) or generate application code from declarative specifications (such as [91]). Annotations such as DyC [43] have been used to direct dynamic optimization. Vandevoorde [112] proposes specifications based on Larch [117] to enable performance optimizations. Work closest to the proposed ISL language is [86] that develops a declarative language called DeAL to describe assertions checkable within one single GC run. Unlike DeAL that aims to inform the GC how to check assertions, ISL is an event-based instrumentation language that specifies (1) how a program should be instrumented, (2) what data need to be collected, and (3) how the mutator and the GC should collaborate.

## 6.4 Optimizations of Data-Parallel Systems

While there exists a large body of work on optimizing data-parallel systems, most existing efforts focus on domain-specific optimizations, including, for example, data pipeline optimizations [24, 44, 141], query optimizations [32, 70], or shuffling optimizations [63, 138, 109]. Despite these optimizations, Big Data performance is still fundamentally limited by memory inefficiencies inherent in the underlying programming systems. ITask is the *first attempt* to help data-parallel tasks written in a managed language survive memory pressure and scale to large datasets by providing a programming model for developers to reason about interrupts and a runtime system that interrupts tasks and tunes performance.

Cascading [115] is a Java library built on top of Hadoop. It provides abstractions for devel-

opers to explicitly construct a dataflow graph to ease the challenge of programming data-parallel tasks. Similarly to Cascading, FlumeJava [24] 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 [47] 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. ITasks perform autotuning in orthogonal way: it is not bound to a specific framework nor is limited to a specific task semantics. Instead, it provides a generic way to reduce memory pressure for a variety of different frameworks and tasks.

Resilient Distributed Datasets (RDD) [136] provides a fault tolerant abstraction for managing datasets in a distributed environment. It is similar to ITask in that the physical location of a data structure is transparent to the developer. However, ITask scatters data between memory and disk on each machine while RDD distributes data in the cluster. Moreover, ITask focuses on enabling managed tasks to survive the presence of high memory pressure while RDD focuses on data recovery in the presence of node failures.

Spark [137] implements RDD and divides jobs into "stages". While resource contention can be avoided between stages, memory problems can still occur inside each stage. In Spark, RDDs can be spilled to disk; but the spilling mechanism is much less flexible than ITask: when spilling is triggered, all RDDs with the same key need to be spilled; partial spilling is not possible.

Mesos [48] and YARN [103] provide sophisticated resource management that can intelligently allocate resources among different compute nodes. Although these job schedulers have a global view of the resources on the cluster, their resource allocation is semantics-agnostic and based primarily on resource monitoring. However, the memory behavior of a program on each node is very complex and can be affected by many different factors. Hence, memory

pressure still occurs, impacting application performance and scalability. ITask is designed to bring the execution back to the safe zone of memory usage when pressure arrives.

PeriSCOPE [44] is a system that automatically optimizes programs running on the SCOPE data-parallel system. It applies compiler-like optimizations on the declarative encoding of a program's pipeline topology. FACADE [72] and Broom [41] optimize the managed runtime by allocating data items in regions. While ITask aims to solve a similar memory problem, it does so by allowing tasks to be interrupted and using a runtime system to automatically interrupt/resume tasks, rather than eliminating Java objects.

## 6.5 Performance autotuning

There is a large body of work on autotuning for parallelism adaptation, especially in the high performance computing (HPC) community [119, 130, 62, 39, 114, 108, 7, 25, 6]. Charm++ [54, 55] is a parallel language based on C++ that provides a high-level abstraction while delivering good performance on different hardware platforms. STAPL [108] is a C++ template library that supports adaptive algorithms and autotuning. ADAPT [113] augments compile-time optimizations with run-time optimizations based on dynamic information about architecture, inputs, and performance. SPL [119] is a language and compiler system for digital signal processing, and PetaBricks [7, 25, 6] is another language and compiler system that provides automated algorithm selection for parallel computing. Recent efforts on autotuning general-purpose programs include Varuna [95], Parcae [85], DoPE [84], work from Curtis-Maury *et al.* [33, 34], PD [94], TT [59], FDT [99], and CoCo [120]. Unlike these existing techniques that adapt either DoP or algorithmic choices or data structure choices to the execution, ITasks targets the problem high memory pressure in Big Data systems.

# Chapter 7

# Conclusions and Future Work

Performance problems widely exist in real-world applications, and managed languages further exacerbate these problems. Performance problems are notoriously difficult to find during testing, since the effects of these problems are almost invisible under small workloads. When these bugs manifest in production runs, they can lead to severe problems, such as badly hurting user experience and huge financial losses. Fixing performance bugs is equally difficult, because developing a fix may require understanding the complex logic of large software systems, and/or changing the code of the underlying systems.

In this dissertation, we have proposed a set of techniques to help developers find and fix a wide class of performance problems. In Chapter 3, we first design a language, ISL, to describe the symptom and counter-evidence of a performance problem if the symptom and counter-evidence can be expressed by logical statements over a history of heap state updates. Using examples, we have demonstrated that ISL is expressive enough to describe various memory-related performance problems.

Based on ISL, we have built a general performance testing framework, PerfBlower, that helps developers capture almost invisible performance bugs during testing by amplifying the

problems. The framework contains a compiler and runtime system support, and design and implementation details are described in Chapter 4. PerfBlower compiles an ISL program, performs the instrumentation, amplifies the target problem when the symptom is observed at runtime, and provides test oracles and useful diagnostic information. Our testing framework is built on ANTLR and Jikes RVM, and we have used it to successfully amplify four types of heap data-related performance problems.

Next, in Chapter 5 we present interruptible tasks as a systematic approach to help data-parallel tasks survive memory pressure. ITask contains a novel programming model that can be used by developers to reason about interrupts as well as a runtime system that automatically performs interrupts and adaptation. Using real-world examples and experimental data, we demonstrate that (1) ITasks can be easily integrated into a distributed framework and interact seamlessly with the rest of the framework; and (2) the runtime is effective at reducing memory usage, thereby significantly improving the performance and scalability of a variety of data-parallel systems.

The techniques proposed in this dissertation may inspire new research on performance testing and debugging, and some potential directions are listed as follows:

- **Amplifying inefficient computation.** In current version of ISL, users can define the performance problems by specifying their symptoms on heap. Some inefficient computation cannot be expressed in ISL, since they have no observable heap symptoms. One potential improvement is adding the the support of inefficient computation specification to ISL so that computation-related problems can be directly captured in testing.

- **Detecting asymptotic inefficiencies in algorithms.** Quite a number of programs perform well under small workloads; but they cannot scale to large input due to the superlinear algorithms in their critical routines. To detect such asymptotic inefficiencies

in algorithmically-critical routines, one potential solution is building the relationship between input size and performance to predict the existence of performance problems under large workloads. It is exciting and challenging to explore how to build the relationship between input and performance and how to combine this model into current test oracles.

- **Non-intrusive performance testing tools.** PerfBlower is built on Jikes RVM, and its current design requires heavy modification of Java virtual machines. An alternative solution is a library-based approach: the runtime support and the instrumentation are implemented as a library, and they communicate with Java virtual machines through some interfaces (e.g., JVM TI [79]). This non-intrusive design makes it extremely easy to integrate the performance testing tool into any JVM as long as it supports the interfaces.

# Bibliography


[1] 0xhacker. Out of memory error in hash join using DistributedCache. `http://stackoverflow.com/questions/15316539/`, 2013. Accessed: 2017-06-01.

[2] Aditya. Out of memory error in mapper when processing customers' reviews. `http://stackoverflow.com/questions/20247185/`, 2017. Accessed: 2017-06-01.

[3] Ahmad, F., Chakradhar, S. T., Raghunathan, A., and Vijaykumar, T. N. Shufflewatcher: Shuffle-aware scheduling in multi-tenant MapReduce clusters. In *USENIX Annual Technical Conference (USENIX ATC)* (Berkeley, CA, USA, 2014), USENIX Association, pp. 1–12.

[4] Ahmedov. Out of memory error in processing a text file as a record. `http://stackoverflow.com/questions/12466527/`, 2012. Accessed: 2017-06-01.

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

[6] Ansel, J., ans Cy Chan, Y. L. W., Olszewski, M., Edelman, A., and Amarasinghe, S. Language and compiler support for auto-tuning variable-accuracy algorithms. In *International Symposium on Code Generation and Optimization (CGO)* (Washington, DC, USA, 2011), IEEE Computer Society, pp. 85–96.

[7] Ansel, J., Chan, C., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A., and Amarasinghe, S. PetaBricks: A language and compiler for algorithmic choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2009), ACM, pp. 38–49.

[8] Anuj. Out of memory error due to large spill buffer. `http://stackoverflow.com/questions/8464048/`, 2015. Accessed: 2017-06-01.

[9] Apache Software Foundation. Tuning Spark. `http://spark.apache.org/docs/latest/tuning.html`, 2017. Accessed: 2017-06-01.

[10] Behm, A., Borkar, V. R., Carey, M. J., Grover, R., Li, C., Onose, N., Vernica, R., Deutsch, A., Papakonstantinou, Y., and Tsotras, V. J. ASTERIX: Towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases (DAPD) 29* (2011), 185–216.

[11] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (New York, NY, USA, 2006), ACM, pp. 169–190.

[12] Bloch, J. *Effective Java, 2nd Edition*. Addison-Wesley Publishing Company, USA, 2008.

[13] Bloomberg News. How healthcare.gov botched $600 million worth of contracts. `https://www.bloomberg.com/news/articles/2015-09-15/how-healthcare-gov-botched-600-million-worth-of-contracts`, 2015. Accessed: 2017-06-01.

[14] Bond, M. D., and McKinley, K. S. Bell: Bit-encoding online memory leak detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (New York, NY, USA, 2006), ACM, pp. 61–72.

[15] Bond, M. D., and McKinley, K. S. Probabilistic calling context. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (New York, NY, USA, 2007), ACM, pp. 97–112.

[16] Bond, M. D., and McKinley, K. S. Leak pruning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (New York, NY, USA, 2009), ACM, pp. 277–288.

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

[18] Borkar, V. R., Carey, M. J., and Li, C. Inside "Big Data management": Ogres, onions, or parfaits? In *International Conference on Extending Database Technology (EDBT)* (New York, NY, USA, 2012), ACM, pp. 3–14.

[19] Bryant, R. E., and O'Hallaron, D. R. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2010.

[20] Bu, Y., Borkar, V. R., Jia, J., Carey, M. J., and Condie, T. Pregelix: Big(ger) graph analytics on a dataflow engine. *International Conference on Very Large Data Bases (VLDB) 8*, 2 (2014), 161–172.

[21] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R. EXE: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security (CCS)* (New York, NY, USA, 2006), ACM, pp. 322–335.

[22] Callan, J., and Yang, Y. Out of memory error in efficient sharded positional indexer. `http://www.cs.cmu.edu/~lezhao/TA/2010/HW2/`, 2010. Accessed: 2017-06-01.

[23] Chaiken, R., Jenkins, B., Larson, P., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. SCOPE: easy and efficient parallel processing of massive data sets. *International Conference on Very Large Data Bases (VLDB) 1*, 2 (2008), 1265–1276.

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

[25] Chan, C., Ansel, J., Wong, Y. L., Amarasinghe, S., and Edelman, A. Autotuning multigrid with PetaBricks. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (New York, NY, USA, 2009), ACM, pp. 5:1–5:12.

[26] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4:1–4:26.

[27] Chenab. Out of memory error in a web parser. `http://stackoverflow.com/questions/17707883/`, 2013. Accessed: 2017-06-01.

[28] chih Yang, H., Dasdan, A., Hsiao, R.-L., and Parker, D. S. Map-reduce-merge: Simplified relational data processing on large clusters. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (New York, NY, USA, 2007), ACM, pp. 1029–1040.

[29] Chu, C. T., Kim, S. K., Lin, Y. A., Yu, Y., Bradski, G. R., Ng, A. Y., and Olukotun, K. Map-Reduce for machine learning on multicore. In *Conference on Neural Information Processing Systems (NIPS)* (Cambridge, MA, USA, 2006), MIT Press, pp. 281–288.

[30] Clark, M. JUnitPerf: A collection of junit test extensions for performance and scalability testing. `https://github.com/clarkware/junitperf/`, 2017. Accessed: 2017-06-01.

[31] Cohen, N. 6 biggest web failures of 2016. `https://dzone.com/articles/6-biggest-web-failures-of-2016`, 2016. Accessed: 2017-06-01.

[32] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. MapReduce online. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, 2010), USENIX Association, pp. 313–328.

[33] Curtis-Maury, M., Dzierwa, J., Antonopoulos, C. D., and Nikolopoulos, D. S. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *International Conference on Supercomputing (ICS)* (New York, NY, USA, 2006), ACM, pp. 157–166.

[34] Curtis-Maury, M., Shah, A., Blagojevic, F., Nikolopoulos, D. S., de Supinski, B. R., and Schulz, M. Prediction models for multi-dimensional power-performance optimization on many cores. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)* (New York, NY, USA, 2008), ACM, pp. 250–259.

[35] Data parallelism. `https://en.wikipedia.org/wiki/Data_parallelism`, 2017. Accessed: 2017-06-01.

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

[37] Edmunds, D. Apache JIRA issue tracker. `https://issues.apache.org/jira/browse/XALANJ-796`, 2015. Accessed: 2017-06-01.

[38] Engler, D. R. Incorporating application semantics and control into compilation. In *Conference on Domain-specific Languages (DSL)* (Berkeley, CA, USA, 1997), USENIX Association, pp. 9–9.

[39] Frigo, M., and Johnson, S. G. The design and implementation of FFTW3. *Proceedings of the IEEE (IEEE Proceedings) 93* (Feb. 2005), 216–231.

[40] Godefroid, P., Klarlund, N., and Sen, K. DART: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2005), ACM, pp. 213–223.

[41] Gog, I., Giceva, J., Schwarzkopf, M., Vaswani, K., Vytiniotis, D., Ramalingam, G., Costa, M., Murray, D. G., Hand, S., and Isard, M. Broom: Sweeping out garbage collection from Big Data systems. In *Workshop on Hot Topics in Operating Systems (HotOS)* (Berkeley, CA, USA, 2015), USENIX Association.

[42] Google. Caliper: Microbenchmarking framework for Java. `https://github.com/google/caliper/`, 2013. Accessed: 2017-06-01.

[43] Grant, B., Mock, M., Philipose, M., Chambers, C., and Eggers, S. J. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science (TCS) 248*, 1-2 (Oct. 2000), 147–199.

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

[45] Guyer, S. Z., and Lin, C. An annotation language for optimizing software libraries. In *Conference on Domain-specific Languages (DSL)* (Berkeley, CA, USA, 1999), USENIX Association, pp. 39–52.

[46] Han, S., Dang, Y., Ge, S., Zhang, D., and Xie, T. Performance debugging in the large via mining millions of stack traces. In *International Conference on Software Engineering (ICSE)* (Piscataway, NJ, USA, 2012), IEEE Press, pp. 145–155.

[47] Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F. B., and Babu, S. Starfish: A self-tuning system for Big Data analytics. In *Conference on Innovative Data Systems Research (CIDR)* (New York, NY, USA, 2011), ACM, pp. 261–272.

[48] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, 2011), USENIX Association, pp. 295–308.

[49] Hower, R. Load test tools. `http://www.softwareqatest.com/qatweb1.html`, 2016. Accessed: 2017-06-01.

[50] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)* (New York, NY, USA, 2007), ACM, pp. 59–72.

[51] Jikes RVM Research Virtual Machine. `http://www.jikesrvm.org/`, 2017. Accessed: 2017-06-01.

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

[53] Jump, M., and McKinley, K. S. Cork: Dynamic memory leak detection for garbage-collected languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (New York, NY, USA, 2007), ACM, pp. 31–38.

[54] Kale, L. V., and Krishnan, S. Charm++: A portable concurrent object oriented system based on C++. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (New York, NY, USA, 1993), ACM, pp. 91–108.

[55] Kale, L. V., and Krishnan, S. Charm++: Parallel programming with message-driven objects. In *Parallel Programming using C++*, G. V. Wilson and P. Lu, Eds. MIT Press, 1996, pp. 175–213.

[56] Kryo: A fast and efficient object graph serialization framework for Java. `https://github.com/EsotericSoftware/kryo/`, 2017. Accessed: 2017-06-01.

[57] Kwon, Y., Ren, K., Balazinska, M., and Howe, B. Managing skew in Hadoop. *IEEE Data(base) Engineering Bulletin (DEBU) 36*, 1 (2013), 24–33.

[58] Kyrola, A., Blelloch, G., and Guestrin, C. GraphChi: Large-scale graph computation on just a pc. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Berkeley, CA, USA, 2012), USENIX Association, pp. 31–46.

[59] Lee, J., Wu, H., Ravichandran, M., and Clark, N. Thread Tailor: Dynamically weaving threads together for efficient, adaptive parallel applications. In *International Symposium on Computer Architecture (ISCA)* (New York, NY, USA, 2010), ACM, pp. 270–279.

[60] Leung, A., Gupta, M., Agarwal, Y., Gupta, R., Jhala, R., and Lerner, S. Verifying GPU kernels by test amplification. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2012), ACM, pp. 383–394.

[61] Lewis, J. Pokmon Go is losing millions of dollars. `https://www.pythian.com/blog/pokemon-go-losing-millions-dollars/`, 2016. Accessed: 2017-06-01.

[62] Li, X., Garzarán, M. J., and Padua, D. A dynamically tuned sorting library. In *International Symposium on Code Generation and Optimization (CGO)* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 111–122.

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

[64] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. Distributed GraphLab: A framework for machine learning in the cloud. *International Conference on Very Large Data Bases (VLDB) 5*, 8 (2012), 716–727.

[65] major1233, and D, S. Performance issues with Internet Explorer 9. `https://answers.microsoft.com/en-us/ie/forum/ie9-windows_vista/performance-issues-with-internet-explorer-9/440b00da-3d61-e011-8dfc-68b599b31bf5/`, 2011. Accessed: 2017-06-01.

[66] Marco. Massive performance problems in IE. `https://connect.microsoft.com/IE/Feedback/Details/1271727/`, 2015. Accessed: 2017-06-01.

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

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

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

[70] Murray, D. G., Isard, M., and Yu, Y. Steno: Automatic optimization of declarative queries. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2011), ACM, pp. 121–131.

[71] Naresh. Out of memory error in map-side aggregation. `http://stackoverflow.com/questions/16684712/`, 2013. Accessed: 2017-06-01.

[72] Nguyen, K., Wang, K., Bu, Y., Fang, L., Hu, J., and Xu, G. Facade: A compiler and runtime for (almost) object-bounded Big Data applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (New York, NY, USA, 2015), ACM, pp. 675–690.

[73] Nistor, A., Chang, P.-C., Radoi, C., and Lu, S. CARAMEL: Detecting and fixing performance problems that have non-intrusive fixes. In *International Conference on Software Engineering (ICSE)* (Piscataway, NJ, USA, 2015), IEEE Press.

[74] Nistor, A., Jiang, T., and Tan, L. Discovering, reporting, and fixing performance bugs. In *International Conference on Mining Software Repositories (MSR)* (Piscataway, NJ, USA, 2013), IEEE Press, pp. 237–246.

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

[76] Null-Hypothesis. Out of memory error in building inverted index. `http://stackoverflow.com/questions/17980491/`, 2013. Accessed: 2017-06-01.

[77] Olivo, O., Dillig, I., and Lin, C. Static detection of asymptotic performance bugs in collection traversals. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2015), ACM.

[78] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig latin: A not-so-foreign language for data processing. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (New York, NY, USA, 2008), ACM, pp. 1099–1110.

[79] Oracle. JVM™ Tool Interface (JVM TI). https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/, 2017. Accessed: 2017-06-01.

[80] Parr, T. ANTLR: ANother Tool for Language Recogntion. https://www.antlr.org/, 2017. Accessed: 2017-06-01.

[81] Pike, R., Dorward, S., Griesemer, R., and Quinlan, S. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming 13*, 4 (2005), 277–298.

[82] Raghava. Out of memory error due to appending values to stringbuilder. http://stackoverflow.com/questions/12831076/, 2012. Accessed: 2017-06-01.

[83] Raghava. Out of memory error in word cooccurrence matrix stripes builder. http://stackoverflow.com/questions/12831076/, 2012. Accessed: 2017-06-01.

[84] Raman, A., Kim, H., Oh, T., Lee, J. W., and August, D. I. Parallelism orchestration using DoPE: The degree of parallelism executive. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2011), ACM, pp. 26–37.

[85] Raman, A., Zaks, A., Lee, J. W., and August, D. I. Parcae: A system for flexible parallel execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2012), pp. 133–144.

[86] Reichenbach, C., Immerman, N., Smaragdakis, Y., Aftandilian, E., and Guyer, S. Z. What can the GC compute efficiently? A language for heap assertions at GC time. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (New York, NY, USA, 2010), ACM, pp. 256–269.

[87] Sen, K., Marinov, D., and Agha, G. CUTE: A concolic unit testing engine for C. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)* (New York, NY, USA, 2005), ACM, pp. 263–272.

[88] Sen, M. Out of memory error in computing frequencies of attribute values. http://stackoverflow.com/questions/23042829/, 2014. Accessed: 2017-06-01.

[89] Shacham, O., Vechev, M., and Yahav, E. Chameleon: Adaptive selection of collections. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2009), ACM, pp. 408–418.

[90] Shivkumar, A. Reducer hange at the merge step. http://stackoverflow.com/questions/15541900/, 2013. Accessed: 2017-06-01.

[91] Smaragdakis, Y., and Batory, D. DiSTiL: A transformation library for data structures. In *Conference on Domain-specific Languages (DSL)* (Berkeley, CA, USA, 1997), USENIX Association, pp. 20–20.

[92] SmartBear. The SmartBear distributed testing framework. http://support.smartbear.com/articles/testcomplete/distributed-testing-tutorial/, 2017. Accessed: 2017-06-01.

[93] Song, L., and Lu, S. Statistical debugging for real-world performance problems. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (New York, NY, USA, 2014), ACM, pp. 561–578.

[94] Sridharan, S., Gupta, G., and Sohi, G. S. Holistic run-time parallelism management for time and energy efficiency. In *International Conference on Supercomputing (ICS)* (New York, NY, USA, 2013), ACM, pp. 337–348.

[95] Sridharan, S., Gupta, G., and Sohi, G. S. Adaptive, efficient, parallel execution of parallel programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2014), ACM, pp. 169–180.

[96] Stack Exchange, Inc. Stack Exchange data dump. https://archive.org/details/stackexchange/, 2015. Accessed: 2017-06-01.

[97] Stanford CoreNLP Core natural language software. https://stanfordnlp.github.io/CoreNLP/, 2017. Accessed: 2017-06-01.

[98] Stewart, C., Shen, K., Iyengar, A., and Yin, J. Entomomodel: Understanding and avoiding performance anomaly manifestations. In *IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (Piscataway, NJ, USA, 2010), IEEE Press, pp. 3–13.

[99] Suleman, M. A., Qureshi, M. K., and Patt, Y. N. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (New York, NY, USA, 2008), ACM, pp. 277–286.

[100] The Apache™ AsterixDB Project. https://asterixdb.apache.org/, 2017. Accessed: 2017-06-01.

[101] The Apache™ Cassandra Project. https://cassandra.apache.org/, 2017. Accessed: 2017-06-01.

[102] The Apache™ Hadoop Project. https://hadoop.apache.org/, 2017. Accessed: 2017-06-01.

[103] The Apache™ Hadoop YARN. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2017. Accessed: 2017-06-01.

[104] The Apache™ Hive Project. https://hive.apache.org/, 2017. Accessed: 2017-06-01.

[105] The Apache™ Hyracks Project. http://github.com/apache/incubator-asterixdb-hyracks/, 2017. Accessed: 2017-06-01.

[106] The Apache™ Pig Project. https://pig.apache.org/, 2017. Accessed: 2017-06-01.

[107] The Grinder: A Java load testing framework. http://grinder.sourceforge.net/, 2015. Accessed: 2017-06-01.

[108] Thomas, N., Tanase, G., Tkachyshyn, O., Perdue, J., Amato, N. M., and Rauchwerger, L. A framework for adaptive algorithm selection in STAPL. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (New York, NY, USA, 2005), ACM, pp. 277–288.

[109] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. Hive - A warehousing solution over a Map-Reduce framework. *International Conference on Very Large Data Bases (VLDB) 2*, 2 (2009), 1626–1629.

[110] TIOBE Software BV. TIOBE Programming Community Index. https://www.tiobe.com/tiobe-index/, 2017. Accessed: 2017-06-01.

[111] TPC. The TPC Benchmark(TM)H (TPC-H). http://www.tpc.org/tpch/, 2015. Accessed: 2017-06-01.

[112] Vandevoorde, M. *Exploiting Specifications to Improve Program Performance*. PhD thesis, Massachusetts Institute of Technology, 1994.

[113] Voss, M. J., and Eigenmann, R. High-level adaptive program optimization with ADAPT. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (New York, NY, USA, 2001), ACM, pp. 93–102.

[114] Vuduc, R., Demmel, J. W., and Yelick, K. A. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series (JPCS) 16*, 1 (2005), 521+.

[115] Wensel, C. Cascading. http://www.cascading.org/, 2017. Accessed: 2017-06-01.

[116] Wikimedia. Wikimedia downloads. https://dumps.wikimedia.org/, 2015. Accessed: 2017-06-01.

[117] Wing, J. M. Writing larch interface language specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS) 9*, 1 (Jan. 1987), 1–24.

[118] Xiao, X., Han, S., Zhang, D., and Xie, T. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (New York, NY, USA, 2013), ACM, pp. 90–100.

[119] Xiong, J., Johnson, J., Johnson, R., and Padua, D. SPL: A language and compiler for DSP algorithms. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2001), ACM, pp. 298–308.

[120] Xu, G. CoCo: Sound and adaptive replacement of Java collections. In *European Conference on Object-Oriented Programming (ECOOP)* (Berlin, Heidelberg, 2013), Springer-Verlag, pp. 1–26.

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

[122] Xu, G., Bond, M. D., Qin, F., and Rountev, A. LeakChaser: Helping programmers narrow down causes of memory leaks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2011), ACM, pp. 270–282.

[123] Xu, G., Mitchel, N., Arnold, M., Rountev, A., Schonberg, E., and Sevitsky, G. Finding low-utility data structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2010), ACM, pp. 174–186.

[124] Xu, G., Mitchell, N., Arnold, M., Rountev, A., and Sevitsky, G. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Workshop on Future of Software Engineering Research (FoSER)* (New York, NY, USA, 2010), ACM, pp. 421–426.

[125] Xu, G., and Rountev, A. Precise memory leak detection for Java software using container profiling. In *International Conference on Software Engineering (ICSE)* (Piscataway, NJ, USA, 2008), IEEE Press, pp. 151–160.

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

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

[128] Xu, G. H., Mitchell, N., Arnold, M., Rountev, A., Schonberg, E., and Sevitsky, G. Scalable runtime bloat detection using abstract dynamic slicing. *ACM Transactions on Software Engineering and Methodology (TOSEM) 23*, 3 (2014), 23.

[129] Yahoo! Yahoo! Webscope Program. `http://webscope.sandbox.yahoo.com/`, 2017. Accessed: 2017-06-01.

[130] Yotov, K., Li, X., Ren, G., Cibulskis, M., DeJong, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P., and Wu, P. A comparison of empirical and model-driven optimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2003), ACM, pp. 63–76.

[131] YourKit Java Profiler. https://www.yourkit.com/, 2015. Accessed: 2017-06-01.

[132] Yu, X., Han, S., Zhang, D., and Xie, T. Comprehending performance from real-world execution traces: A device-driver case. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (New York, NY, USA, 2014), ACM, pp. 193–206.

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

[134] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Úlfar Erlingsson, Gunda, P. K., and Currey, J. 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)* (Berkeley, CA, USA, 2008), USENIX Association, pp. 1–14.

[135] yura. The performance comparison between in-mapper combiner and regular combiner. http://stackoverflow.com/questions/10925840/, 2013. Accessed: 2017-06-01.

[136] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, 2012), USENIX Association, pp. 15–28.

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

[138] Zhang, J., Zhou, H., Chen, R., Fan, X., Guo, Z., Lin, H., Li, J. Y., Lin, W., Zhou, J., and Zhou, L. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, 2012), USENIX Association, pp. 22–22.

[139] Zhang, P., and Elbaum, S. Amplifying tests to validate exception handling code. In *International Conference on Software Engineering (ICSE)* (Piscataway, NJ, USA, 2012), IEEE Press, pp. 595–605.

[140] Zhou, B., Too, J., Kulkarni, M., and Bagchi, S. WuKong: automatically detecting and localizing bugs that manifest at large system scales. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (New York, NY, USA, 2013), ACM, pp. 131–142.

[141] Zhou, J., Larson, P.-Å., and Chaiken, R. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *International Conference on Data Engineering (ICDE)* (Washington, DC, USA, 2010), IEEE Computer Society, pp. 1060–1071.