UNIVERSITY OF CALIFORNIA

Los Angeles

An Observational and Statistical Technique for Efficient Big Data Applications

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Christian Navasca

2023

ABSTRACT OF THE DISSERTATION

An Observational and Statistical Technique for Efficient Big Data Applications

by

Christian Navasca
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2023
Professor Harry Guoqing Xu, Chair

Big Data systems such as Hadoop and Spark enable companies and people to process huge data processing workloads that would otherwise be impossible to complete. Many thousands of applications run atop each of these systems, making it all the more important for them to be both performant and maintainable.

The development of these systems is no small feat. They handle issues such as cluster management and load balancing so that users of the systems can focus on the data processing problems instead. While this is good for the end user, it means that these systems' codebases are very complex, and often make some concessions (such as choosing a managed programming language) in order to mitigate this complexity in development.

This dissertation aims to improve these systems with the idea of *optimistic* optimizations. This means that we aggressively try to perform optimizations *even if* they may sometimes be unsound, or cause slowdowns, when they will be beneficial overall. We apply this idea in three works in this direction, to improve different aspects of Big Data systems.

First, we present a Java-based compiler and runtime system named Gerenuk, which transforms a Big Data program to use inlined native bytes, rather than objects, to achieve end-to-end speedups and improved memory usage. We show that while this transformation does not work in the general case, because of the typical behavior of objects in this context, most objects will see a benefit, and we show a recovery technique for objects that behave differently.

Second, we present a learning technique to predict profiling data of applications, to perform profile-guided optimization across a Big Data cluster, without profiling it in its entirety. Given a high enough model accuracy, we can predict how objects typically behave, based on their allocation sites. However, based on the confidence of the model, a system could choose to not perform the optimization.

Finally, we present a learning technique to predict context-sensitive points-to information from context-insensitive information. These predictions are much faster than calculating context-sensitive information, and can be useful in scaling these analyses to the size of these systems' codebases. While these predictions could be wrong, based on a user's needs, we can provide an insensitive solution, run the real (slow) analysis, or adjust the model based on required expected precision and recall.

The dissertation of Christian Navasca is approved.

University of California, Los Angeles

2023

To my family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I'd like to start by giving my deepest thanks to my advisor, Harry Xu. Without Harry, I would not be where I am today. Without his encouragement, I never would have started graduate school, and without his mentorship and support I never would have finished. Harry has always inspired me to be my best, and was always there to encourage me when I doubted myself. I'll always be grateful that I had him as my advisor.

I would also like to thank my other committee members: Michael Bond, Martin Maas, Todd Millstein and Jens Palsberg. Thanks for your valuable feedback on my work, which has been a tremendous help for improving and completing this dissertation.

I would like to give a huge thanks to my collaborators from my time at Google: Hyeontaek Lim, Petros Maniatis, and Martin Maas. I will always cherish my time working with them, be it debugging Apache Beam pipelines with Hyeontaek, discussing machine learning models with Petros, or JVM hacking sessions with Martin. In particular, I would like to give my sincerest thanks to Martin, for taking a chance on me with my first internship, providing countless research opportunities, and many hours of thought-provoking discussions when we finally met at in-person conferences. Thanks for everything.

I am very lucky to have been surrounded by such amazing people in the UCLA systems lab: Yifan Qiao, Haoran Ma, Shi Liu, Shan Yu, Jiyuan Wang, Chenxi Wang, Jonathan Eyolfson, John Thorpe, Arthi Padmanabhan, Pengzhan Zhao, Usama Hameed, and Khanh Nguyen. Thanks for countless research discussions, for the fun gatherings and great friendships, and for much-needed empathy and support on our shared journey. In particular, I would like to thank Khanh for his help and mentorship, on numerous research projects during my undergrad and early PhD work. Khanh is a great mentor who helped inspire me to pursue this PhD, and I'm grateful to have had the opportunity to work with him.

I'd also like to thank my close friends from my hometown: Anthony, Diego, Cousin Daniel, and Rosie, for all the laughs and good times throughout the years. You helped keep me sane, and reminded me that there is more to life than just my research career. Thanks for being there for me, even when I was so hyper-focused on my work. I'd like to thank Anthony in particular, who has listened to countless hours of me venting about my work, and for helping me to learn to manage my ever-growing stress.

I would also like to thank my family for their support as I pursued this PhD. In particular, I would like to thank my parents for being so supportive of me to chase my dreams and passions (no, this PhD doesn't make me a Professor). My brother, Jojo, also deserves a special mention, for always pushing me to be better and always being there to listen when I needed help making a big decision. I'm forever grateful that my family is there on all of my ups and downs.

Finally, I would like to give a very special thanks to my partner, Stephanie. She has always been a source of joy and support in my life, and was always by my side during this long journey. I would never have made it this far without her emotional support and understanding along the way.

This work contains modified versions of published and co-authored work. Chapter 2 is a version of [119] and chapter 3 is a version of [120]. (PI: Harry Guoqing Xu).

VITA

Research/Teaching Assistant University of California, Los Angeles, CA	Sept 2018 - Dec 2023
Student Researcher Google	Sept 2020 - April 2021
Research Intern Google	June 2020 - Sept 2020
B.S. Computer Science University of California, Irvine	Sept 2014 - June 2018

PUBLICATIONS

Christian Navasca, Martin Maas, Petros Maniatis, Hyeontaek Lim,and Guoqing Harry Xu. Predicting Dynamic Properties of Heap Allocations Using Neural Networks Trained on Static Code. ACM SIGPLAN International Symposium on Memory Management (ISMM'23), June 2023.

Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, **Christian Navasca**, Shan Lu, Guoqing Harry Xu, MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime, 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22), July 2022. **Won a Jay Lepreau Best Paper Award**

Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Harry Xu. Gerenuk: Thin Computation over Big Native Data Using Speculative Program Transformation, the 27th ACM Symposium on Operating Systems Principles (SOSP'19), Huntsville, Ontario, Canada, October 2019.

Khanh Nguyen, Lu Fang, **Christian Navasca**, Harry Xu, Brian Demsky, and Shan Lu. Skyway: Connecting Managed Heaps in Distributed Big Data Systems, the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18), Williamsburg, VA, USA, March 2018.

CHAPTER 1

Introduction

Big Data systems such as Hadoop and Spark enable companies and people to process huge data processing workloads that otherwise be impossible to complete. Each framework supports many applications, making these systems a good target for optimization tasks, as improving a single system can benefit thousands of downstream tasks.

The development of these systems is no small feat. They handle issues such as cluster management and load balancing so that users of the systems can focus on the data processing problems instead. Modern Big Data systems such as Hadoop [10], Spark [175], Flink [12], and Hive [11] are written in object-oriented managed languages, such as Scala and Java. These languages ease development and enable quicker deployment, as the managed runtimes handle tasks such as memory management automatically. This of course comes with a cost: there is an overhead to managed runtimes.

Managed runtimes incur heavy runtime costs [122, 125, 56, 30] which can greatly reduce the efficiency and processing capabilities of a cluster. A major source of this cost comes from the fundamental abstraction used by object-oriented programming – everything is an object. We address this problem first using *speculative program transformation* which *optimistically* optimizes programs based on the observation that most data objects are *immutable* and *confined* in this setting (chapter 2). While this observation does not hold for all cases, it is true for the vast majority of data objects.

More generally, if we know how certain objects will behave, we can make smarter optimization choices. This idea appears in the form of profile-guided optimization (PGO), and has been used to great success in a number of tasks [142, 69, 160], but as the name suggests, require *profiling* runs. However, it is difficult to gather *performance profiles* across an entire cluster to capture these object behaviors. This is due to added requirements to deployment, as PGO pipelines must be carefully constructed to produce good, representative, profiling data. Instead, we can use machine learning, specifically large language models for code, to predict object-level behavior (chapter 3). These object-level predictions could then be used to make optimizations without any profiling.

We can also look to make optimizations in a static context. Static analysis is incredibly useful for improving the correctness of a system, but it is difficult to handle the scale of Big Data system code bases. In Chapter 4, we investigate predicting context-sensitive points-to information using neural networks, augmented with statically available code. Instead of calculating a context-sensitive solution, we train a model to predict it from the context-insensitive solution, allowing us to find a context-sensitive solution in a fraction of the time. This predicted context-sensitive points-to information can be used as a basis to analysis clients such as virtual call resolution, null pointer dereference detection, and many other bug finding tasks.

Below is a brief overview of the works in this dissertation:

Chapter 2: Gerenuk, Thin Computation over Big Native Data Using Speculative Program Transformation. Big Data systems process billions of data items, which are represented as billions of objects. This object representation inflates memory usage due to meta-information required by the managed runtime, such as object headers. This memory inflation incurs exceedingly high garbage collection (GC) overhead and results in either increased computation costs to satisfy the memory need, or increased memory-disk round trips. Additionally, this sea of objects must be shuffled between machines, and each shuffle must *serialize* objects into native bytes before sending them over the network, and *deserialize* them back into objects at the receiving end.

One solution is to directly represent the data items as native bytes, rather than objects.

This representation would have a number of benefits. First, we would remove the memory overhead of objects (as there are no object headers). Second, we would remove the need to serialize and deserialize data (as the format on the network is the same). Finally, we would remove a significant amount of GC cost, as these data items are no longer objects and the GC needs to handle many fewer objects.

However, as discussed in chapter 2, this approach is only viable if objects are *immutable* and *confined*. One key insight of this work is to do this optimization *optimistically*, as most data objects will behave this way. In the cases where objects do not behave this way, we can simply fall back to the unoptimized, non-Gerenuk, code. While this means we cannot optimize every case, it enables Gerenuk to focus on the common case: the well-behaved data objects. By focusing our efforts on these data objects, we improved performance of Big Data applications by 1.6x, and reduced memory usage by 26%.

Chapter 3: Predicting Dynamic Properties of Heap Allocations using Neural Networks Trained on Static Code. Memory allocators and runtime systems can leverage dynamic properties of heap allocations – such as object lifetimes, hotness or access correlations – to improve performance and resource consumption. A significant amount of work has focused on approaches that collect this information in performance profiles and then use it in new memory allocator or runtime designs, both offline (e.g., in ahead-of-time compilers) and online (e.g., in JIT compilers). This is a special instance of profile-guided optimization.

This approach introduces significant challenges: 1) The profiling oftentimes introduces substantial overheads, which are prohibitive in many production scenarios, 2) Creating a representative profiling run adds significant engineering complexity and reduces deployment velocity, and 3) Profiles gathered ahead of time or during the warm-up phase of a server are often not representative of all workload behavior and may miss important corner cases.

In this paper, we investigate a fundamentally different approach. Instead of deriving heap allocation properties from profiles, we explore the ability of neural network models to predict them from the statically available code. We describe the trade-off space of this approach, investigate promising directions, and motivate these directions with data analysis and experiments, but we do not believe that this is a solved problem. We believe that future work in this space could be very impactful, especially given recent advancements in Large Language Models, especially LLM's for code.

Chapter 4: Code Embedding for Refinement-based Context-sensitive Points-to Analysis. Points-to analysis is a fundamental static analysis that answers the question "what can this variable point to?". While important on its own, it is an essential building block for nearly any program analysis tool. In order to produce sound results, these analyses are typically imprecise, meaning they report many false positives (which result in many user-facing warnings). Context-sensitivity has been one of the most successful ways of improving precision of points-to analyses, but they are incredibly slow – sometimes taking days to analyze even mid-sized codebases.

In this work, we aim to predict this context-sensitive information using GNNs. The benefit is performance: whole-graph GNN inference might take seconds or minutes, an incredible speed up potential over the slow analysis. We describe a way to create a GNN-based classification model from a CFL-reachability formulation of context-sensitive points-to analysis, and how we can address some challenges we encounter on the way, such as generalization and model mispredictions.

CHAPTER 2

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

Object-oriented managed languages are popular because they automate development tasks such as memory management. However, it comes at a performance cost. Object representation adds memory and computation overhead, but it is acceptable in the context of small applications or even applications that use thousands of objects. However, in this Big Data setting, when we process billions of objects, this overhead can become a problem. In this chapter we ask a question: *can we remove objects altogether*? Gerenuk shows that it is possible, due to a simple observation: *most* objects in these systems can be represented in a much simpler way.

2.1 Introduction

Modern Big Data systems, such as Hadoop [10], Spark [175], Flink [12], or Hive [11], were all implemented in object-oriented languages such as Scala and Java due to the high productivity enabled by these languages. However, managed runtime systems incur a heavy runtime cost [122, 125, 56, 30], leading to reduced efficiency and processing capabilities. A major source of this cost comes from the fundamental abstraction of object-orientation—everything is an *object*.

For data-intensive systems, each computational iteration needs to process billions of data items, which are represented as billions of objects. Previous work shows that this object-based representation can inflate the memory usage by almost four times [30] due to the extra meta-information required by the managed runtime such as object headers, padding, pointers, etc. Such a huge memory inflation incurs exceedingly high garbage collection (GC) overhead, up to 70% of the total execution time [123], and results in either increased computation cost to satisfy the memory need or increased memory-disk round trips.

Big Data systems often need to shuffle billions of objects, and representing each data item as an object dictates that each shuffle needs to *serialize* many objects into native bytes, transfer them over the network, and *deserialize* the bytes back into objects before they can be processed at a remote machine. Evidence shows that serialization and deserialization can take up to 30% of the execution [125]. This cost will only increase as systems become more heterogeneous by incorporating heterogeneous software/hardware components, such as UDFs in different languages or accelerators. All represent data in their unique formats, dictating even more frequent serialization and deserialization.

State of the Art. The cost of representing data as objects has been a known problem, with much recent effort attempting to reduce this cost. Some work tackles one aspect of the problem. Skyway [125] lowers the (de)serialization cost by directly transferring objects through the network, and hence, cannot help with memory inflation and GC problems (it actually brings some data inflation into network packages). Yak [123] lowers the GC cost by adapting the GC algorithm to fit lifetime patterns in data-intensive systems, and does not solve memory inflation or (de)serialization problems. Broom [64] provides APIs for Naiad [116] developers to manually allocate/deallocate data objects in regions. It works only for Naiad and does not provide automated support.

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

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

The good news is that prior work [30] observed that the majority, more than 95%, of runtime objects are created and used by a rather small and simple codebase that primarily conducts data manipulation like map, reduce, and relational operations, which are amenable to and can benefit greatly from such a transformation. Only less than 5% of runtime objects are created by a large and complex codebase for cluster management, scheduling, communication, and others, which are extremely difficult to, and fortunately need not to, be transformed. The bad news, however, is that there does not exist a *clear separation* between the former, referred to as *data path*, and the latter, referred to as *control path*—they are often heavily mixed inside one class and even one method.

A recent technique Facade [122] attempts to provide such a transformation as described above. However, since it aims to transform one whole class at a time, and turns every field in a class into a native representation and every statement in every method of the class into a native-byte operation, using Facade requires a huge amount of code refactoring to split many classes and methods to make sure that data and control code modules do not interfere, making it extremely difficult to use in practice (more details in §2.2).

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

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

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

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

The *first challenge* is how to identify which code statements to transform. Our approach is based on a key observation that data-processing logic is *task-based*. As illustrated in Figure 2.1, each task (*e.g.*, a stage in Spark or a Map/Reduce execution in Hadoop) starts at a shuffling phase where each compute node reads in a new set of data items; these data items get deserialized into objects, which then flow through a series of system-level and user-defined processing functions; at the end of the task is another shuffling phase that serializes each object into a sequence of bytes and then sends them to files or the network. In other words, data *flows* from the deserialization point at which heap objects are created from native bytes (*e.g.*, a call to method `readObject`) to the serialization point at which heap objects are converted back to bytes (*e.g.*, a call to method `writeObject`).

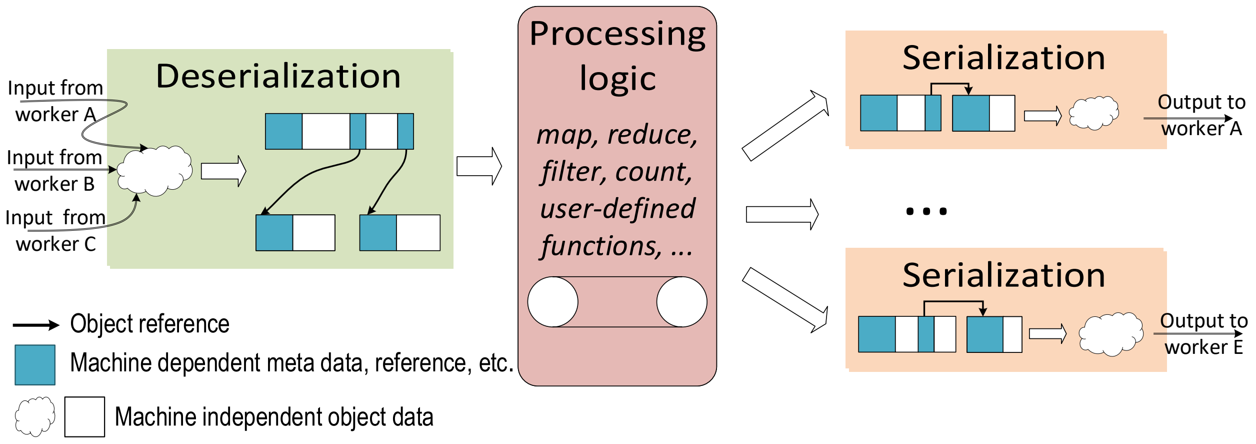


Figure 2.1: The flow of data objects.

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

Note that this is a *one-stone-multiple-bird* approach. First, no objects including their headers and pointers are created for data items, leading to great reduction in memory consumption and GC overhead. Second, Gerenuk eliminates almost all serialization/deserialization effort, which has been shown to be expensive [125, 121]. Third, since a native buffer stores only data items for one particular task, the buffer is naturally amenable to *region-based memory management* — we can safely release the buffer as a whole at the end of the task without even needing to scan the items. Essentially, Gerenuk removes *all* three types of overhead—memory usage inefficiency, serialization, and garbage collection—from the managed runtime for data processing whereas existing techniques could eliminate only one or a subset of them.

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

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

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

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

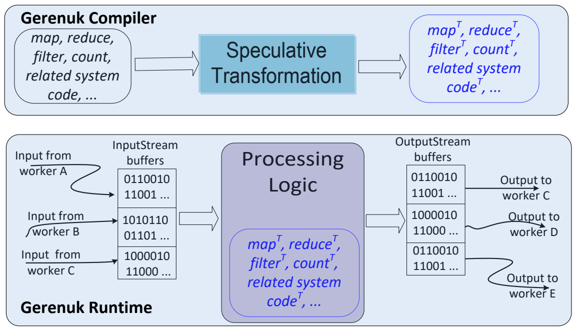

Gerenuk Compiler
map, reduce, filter, count, related system code, ...
Speculative Transformation
map^T, reduce^T, filter^T, count^T, related system code^T, ...
InputStream buffers
Input from worker A
Input from worker B
Input from worker C
Processing Logic
map^T, reduce^T, filter^T, count^T, related system code^T, ...
OutputStream buffers
Output to worker C
Output to worker D
Output to worker E
Gerenuk Runtime

Figure 2.2: Gerenuk's overview.

```
class DenseVector[V] (
    val data: Array[V],
    val stride: Int)
  ...
}
```

```
class LabeledPoint (
    val value: Double,
    val features:
    DenseVector[Double])
    ...
}
```

Figure 2.3: A user-defined data structure in Spark for Logistic Regression program.

execute the "slow path", which is the unmodified SER, with the original input buffers.

Note that this is possible only for data-parallel systems — there is no global state shared between multiple tasks (*e.g.*, a similar observation has also been used in [56]) — a SER either succeeds and produces results to be fed to another SER, or aborts, causing the system to instead execute the unmodified version of the same SER on the same input. The implementation of *abort* only needs to call a few methods to launch a new executor and terminate the current executor.

An overview of Gerenuk is illustrated in Figure 2.2. We have implemented the Gerenuk runtime in OpenJDK 8 and the Gerenuk compiler on the Soot Java compiler framework [148]. Using Gerenuk, we automatically transformed a set of applications on Apache Spark [175] and Apache Hadoop [10]. Gerenuk's transformation has improved the end-to-end performance for these two systems by an overall factor of **2**× and **1.4**×, respectively.

2.2 Extended Motivation

How Data Path and Control Path Mix. Consider the class `StreamingContext`, which is the main entry point for streaming-related functionality in Apache Spark. Many meth-

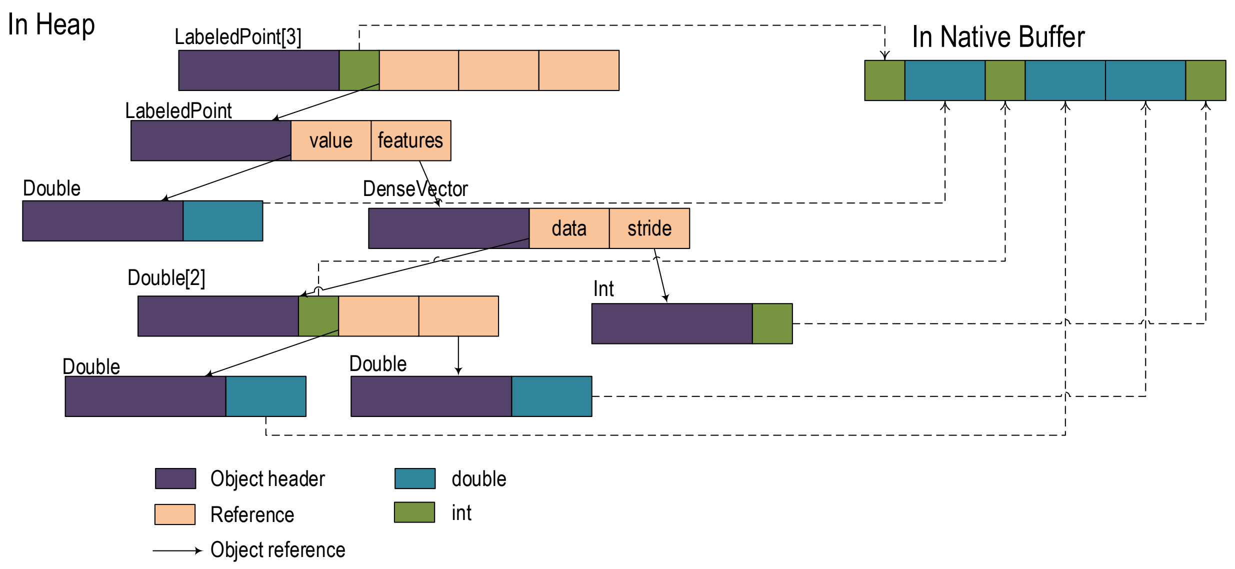


Figure 2.4: The difference in memory layout between the heap and native-buffer (inlined) representation of an array of three `LabeledPoint` objects; the space overhead incurred by the JVM is nearly 2× larger than the actual data size.

ods defined in this class are data-processing methods (*e.g.*, `textFileStream` which creates an RDD from a text file) while other methods contain control code that does not manipulate user data. All of these methods need to share certain global control state, which is maintained in instance fields of the class. For example, one of these fields stores the current file system directory information, which is used by both data manipulation methods and control methods.

If we follow the whole-class transformation philosophy in Facade [122], developers must manually refactor and split `StreamingContext` into a data component (*e.g.*, a new `DataContext` class) and a control component (*e.g.*, a new `ControlContext` class) that contain methods that process vs. do not process user data, respectively. Since the shared instance fields that used to be in the same class now get split into two copies (one in the heap and a second in native memory), the user must also guarantee consistency between these copies. For any real-world system, such manual refactoring effort is huge (*e.g.*, several weeks to months), creating significant obstacles for practical adoption.

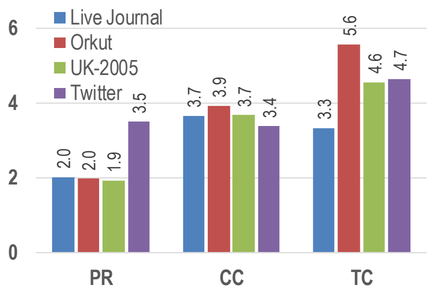


Figure 2.5: The ratios between the total bytes of data objects and the size of their actual payload.

Analytical Motivation. To understand the space overhead incurred by the object-based data representation, we studied a Spark application that computes logistic regressions and analyzed the number of bytes consumed by its data objects. In this application, the user defines a `LabeledPoint` class as the data type (*i.e.*, the type of the RDD elements). This definition is shown in Figure 2.3. Each `LabeledPoint` object references a variable-length vector of `double` values.

Figure 2.4 compares the object-based representation of an array of three `LabeledPoint` objects and its corresponding native, inlined representation. Under the native representation, each inlined `LabeledPoint` contains 3 `int` and 3 `double` values, taking 36 bytes. An array with three `LabeledPoint` records all inlined only needs $4+36\times3 = 112$ bytes, where 4 is the number of bytes needed to record the array size. However, the object-based representation, which connects these objects with pointers, requires, in addition to the 112-byte data payload, an overhead of 8 object headers and 9 object references. These headers and references take $8\times16+9\times8 = 200$ bytes, bringing the total bytes needed to 312 bytes. Hence, the object representation overhead is nearly twice the size of the actual data payload.

Empirical Motivation. To empirically quantify this space overhead and its related runtime overhead, we ran three graph analytics programs on Apache Spark. We used Spark version 2.1.0 on a cluster of 11 nodes, each with 2 Xeon(R) CPU E5-2640 v3 processors, 32GB memory, 1 SSD, running CentOS 6.9 and connected by a 1000Mb/s Ethernet. The three graph programs were PageRank (PR), ConnectedComponents (CC) and TriangleCounting

(TC) over four real-world graphs, LiveJournal [14], Orkut [67], UK-2005 [26], and Twitter-2010 [92]. Kryo [91], a high-performance serializer, was used.

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

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

Anticipated Benefits. These results highlight several potential benefits that can be achieved if the processing engine can work directly over native bytes. First, we expect that much of the 2.3× memory overhead can be eliminated as the program directly operates over the serialized bytes. Memory savings in Big Data systems can often translate to increased degrees of parallelism due to more available memory and thus the ability to run more executors. Second, we expect significant reductions in computation time due to reduced pointer chasing effort as well as eliminated runtime checks (*e.g.*, array bound checks and write barriers performed by the managed runtime). Third, we expect that the serialization/deserialization time, which could take up to a third of the execution time [125, 121], will be significantly reduced because the same (native) format is used when data is transferred over the network. Finally, we expect large savings in garbage collection since data items are no longer represented as objects, and hence, the GC only needs to scan a much smaller number of objects. In the initial experiment described earlier in this section, we did not see much GC overhead ($< 10\%$) from Spark's executions because the size of data per machine was small relative to the size of the heap (*i.e.*, 30GB) used. However, the GC overhead can grow to dominate the execution for large inputs, as shown in prior works [123, 30, 111, 64] as well as our experimental results (§2.4).

Applicability. Gerenuk is designed specifically for dataflow systems. There are two conditions under which a system is amenable to our transformations. First, data objects exhibit the data flow shown in Figure 2.1 (*e.g.*, there are sources and sinks). Second, data objects do not carry state and are immutable. While our current implementation does not work directly for other dataflow systems such as Scope [33], Hyracks [80], or Naiad [116], the aforementioned two conditions hold for these systems, and hence they can all benefit from the proposed transformations. Gerenuk cannot optimize systems for which the conditions do not hold. For example, if a system does not clearly define sources and sinks for data objects, we cannot identify the dataflow. As another example, if data objects are not immutable, the transformed program would always abort, resulting in large performance penalties.

2.3 Gerenuk Design and Implementation

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

2.3.1 User Effort

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

First, since a speculative execution region starts at a deserialization point and ends at a serialization point, Gerenuk relies on the user to identify these two points. For Hadoop, a call to `WritableDeserializer.deserialize()` in method `nextKeyValue` of class `ReduceContextImpl` is a deserialization point, while a call to `WritableSerializer.serialize()` in method `append` of `IFile` is a serialization point. There can be multiple deserialization and serialization points for a system. Gerenuk uses these (de)serialization points to automatically identify the statements through which data flows for transformation (see §2.3.5). Although specifying serialization APIs requires system-level knowledge, this task can be done by a Spark or Hadoop system developer once and for all and has only one-time cost, since Spark and Hadoop all have clear serialization modules and their APIs rarely change.

Second, the user needs to specify the types T of *top-level* data items processed in the program (*e.g.*, via annotations). For example, for a Spark program, these data types include the element types of the RDDs in the program (*e.g.*, `DenseVector`). For a Hadoop program, they include the types of keys and values that are read from or written into HDFS. The Gerenuk compiler will identify the data structure rooted at each such type to establish a mapping between its native (inlined) format and object-based format (see §2.3.3).

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

2.3.2 SER Code Analyzer

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

Our analysis is similar to a *static taint analysis*—it starts by tagging the variable defined at a deserialization point (*e.g.*, `v` in `v = readObject()`) and then propagates the taint mark from one variable to another by tracking the data flow from the source to the sink. In general, for each assignment `a = b`, we taint `a` if `b` is tainted. A static taint analysis is known to be imprecise due to its reliance on static modeling of heap accesses and static resolution of virtual method calls. To make the analysis report less irrelevant information, we adopt the following three approaches.

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

For an object write such as `b.f = a`, we do not taint the left hand side $o.f$ because if a references a data object, this write would indicate that the object escapes the current data structure to a different one. In this case, our compiler will insert an *abort* instruction at the write to abort the SER instead of tracking the object further. How *abort* instructions are inserted is discussed in detail in §2.3.4.

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

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

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

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

2.3.3 Data Structure Analyzer

The second component of Gerenuk compiler is a data structure analyzer. Given a top-level data type T specified by users (like an element type of an RDD as discussed in §2.3.1), the analyzer explores all classes referenced directly or transitively by T, and outputs a map that maps each primitive- or array-type field in these classes to its corresponding offset inside the native buffer-based representation of T. This offset information will later be used by Gerenuk to replace object-based field accesses in the original program with native buffer-based accesses in the transformed program.

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

Consider the following class that defines a data structure that does not have a statically decidable serialized size: `class C { int a; long[] b; double c; }`.

The offsets for field `a` and field `b` in a record of C are straightforward, 0 and 4, but the offset for `c` is non-trivial. The offset of field c consists of (1) the size of field a, which is 4, and the size of array-field b in its native representation, which further consists of (2) 4 bytes that stores the length of array b, denoted as $b.len$ and (3) the size of the content of array b ($8 \times b.len$). Since $b.len$ is stored in front of the data content of b and right after field a, it can be accessed at run time by an auxiliary function provided by our runtime `readNative`, which takes three parameters, the base address of the current record, the offset in the current record, and the number of bytes to read. Consequently, the length of b can be computed by `readNative` (BASE_C, 4, 4), with BASE_C representing the starting address of the current record of C in the inlined bytes; the offset of field c can be computed by $4 + 4 + 8\times$`readNative`(BASE_C, 4, 4); and the total size of class C is $16 + 8\times$`readNative`(BASE_C, 4, 4).

The above size expression of C will be used to compute offsets for the fields in classes that directly or transitively reference C. Specifically, when the DFS traversal finishes C and returns to an upper-level class C', $\mathtt{BASE}_C$ will be replaced with an expression containing a new symbolic value $\mathtt{BASE}_{C'}$ representing the starting address of a record of C'. Eventually, when the DFS exploration finishes back at the top-level type T, all the offset expressions are represented w.r.t. the starting address of a record of T. The concrete value of this address will be provided by the runtime during execution.

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

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

2.3.4 Computing Violation Conditions

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

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

- **Violation #1: Load-And-Escape.** Consider the following code snippet: `v = n.f; o = new O(); o.g = v;`

Suppose our analysis finds that variable `n` refers to an object in an inlined data record (rooted at a user-defined type T). This case violates our *confinement* assumption—a reference read from `n.f` *escapes* the data structure and gets assigned into another object `o`. In the inlined bytes, such a reference would not exist, and hence, this piece of code would cause an execution failure.

- **Violation #2: Disrupt-the-Native-Space.** If there exists a statement `n.f = o` that writes a reference of a regular heap object `o` into an object `n` that is part of an inlined data record T, the execution would fail because the inlined bytes cannot hold Java references. This is a violation of our *immutability* assumption.

- **Violation #3: Invoke-Native-Method.** A violation occurs if we encounter a call site `p = n.m()` where `n` is an object in a data structure T and `m` is a native method, because a native method may create external side effects. However, certain native methods are frequently invoked. While calls to native methods are generally considered as violations, Gerenuk provides customized implementations (that can work with the inlined bytes) for a set of frequently-used native methods, such as `clone`, `hashcode`, `toString`, and `arrayCopy`, to improve usefulness.

- **Violation #4: Use-Object-Metainfo.** A violation occurs if the metadata (such as the lock) of an object in an inlined data record T is explicitly used. An example code snippet is `v = n.f; synchronize(v){...}`. Here variable `n` refers to an object in an inlined data record and hence v also points to an object in the data record. Use of v as a lock would lead to a violation detected by our compiler because in the transformed program, v and n are no longer objects and hence no lock can be obtained from them.

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

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

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

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

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

—these *aborts* will be triggered, although such cases are extremely rare.

2.3.5 Speculative Transformation

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

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

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

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

In Case 2, each variable assignment is replaced with an address assignment. Case 3 deals with the transformation of parameter passing statements. Case 4 and Case 5 handle heap stores and loads, respectively. These heap accesses are replaced with calls to our methods `writeNative` and `readNative`, respectively, that access the inlined bytes. Note that for heap stores, we only allow writing into primitive-type fields because writing into reference-type fields are prohibited by the second violation condition.

Depending on whether the offset of the field is a static constant or an expression that contains symbolic variables, code generation is done in different ways. For example, if the offset is an expression, the expression will be resolved by the runtime via an auxiliary function `resolveOffset` (see §2.3.6), assigned to a temporary variable t, which is then used to invoke `writeNative` and `readNative`.

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

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

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

If a call is encountered (Case 9) and the call is made on an object whose type is in the class hierarchy C, we inline the method m into the caller and recursively transform m's body. We use a pointer analysis [101] to resolve virtual calls.

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

2.3.6 The Gerenuk Runtime

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

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

Determining Offsets. Recall that each object in a data structure rooted at T has an offset computed statically by our data structure analyzer (§2.3.3). At run time, the content of the object will be written into a native buffer at the location specified by the offset. However, if an object (say o) follows an array in the data structure, o's offset is represented as a symbolic expression that contains the length of the array as a variable. This expression will be resolved at run time by the function `resolveOffset`, shown in Algorithm 1.

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

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

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

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

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

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

2.4 Evaluation

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

To evaluate Gerenuk, we transformed Apache Spark [175], and Apache Hadoop [10], two most popular, widely-deployed Big Data systems. All of our experiments were run on an 11-node cluster, each with 2 Xeon(R) CPU E5-2640 v3 processors, 32GB memory, 1 200GB SSD, running CentOS 6.9 and connected via InfiniBand. All benchmarks were run for three times and the median value is reported. We also verified that no incorrect results were produced by our transformation.

2.4.1 Improving Apache Spark with Gerenuk

We evaluated Gerenuk using Spark 2.4.0, under Hadoop 2.9.2 and Scala 2.11.8. Overall, Gerenuk has transformed statements in 55 classes in Spark. The static analysis reports 126 violation points, none of which were triggered at run time. We used a set of five programs: PageRank (PR) and KMeans (KM) from the GraphX [66] libraries, as well as Logistic Regression (LR), Chi Square Selector (CS) and Gradient Boosting Classification (GB) from the MLlib libraries. These programs were selected due to their diverse computation and data types and can represent a large class of applications.

Note that we focus on iterative programs for our benchmark selection because iterative processing is the domain for which Spark was designed. One observation here is that the top-level data types T used in four of the five programs (except PageRank) have complex data structures that have 3 or 4 levels of objects connected by pointers. These data types cannot be stored directly in native memory by `DataFrame` and Tungsten. For PageRank that uses simple types, we compared our performance with that of Tungsten and our results are reported in §2.4.3.

Table 2.1 shows the details of the input of those programs. Each Spark worker was given all available cores (*i.e.*, 32) on each machine. The JVM on each node was evaluated under 3 different heap sizes: 10GB, 15GB, and 20GB. Kryo, the recommended high-performance serializer, was used.

Running Time. Figure 2.6(a) shows the runtime comparison between Spark and Gerenuk. A summary of different aspects of improvements is reported in Table 2.3. On average, Gerenuk makes Spark run 1.96× faster. The majority of the savings comes from the reduced computation time. Since all (billions of) data objects are represented as inlined native bytes, not only is data locality improved, Gerenuk eliminates many sources of runtime overhead, including pointer chasing, write barriers (*i.e.*, a piece of code executed per object write for GC purposes), and array bound checks.

Note that serialization and deserialization is not completely eliminated by Gerenuk, as shown in the purple and orange bars in Figure 2.6(a). Most of these costs are associated with sending *closures* (*i.e.*, lamdas) from the driver to worker nodes, *not* data objects.

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

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

Memory. Comparisons of peak memory usage are reported in Figure 2.7(a). Since Gerenuk uses native memory while the original Spark uses managed heap, to enable a fair comparison, we periodically ran `pmap` to measure the process-level memory consumption and reported the maximum consumption in Figure 2.7(b). Across all benchmarks under different heap configurations, Gerenuk saves up to 38% of memory, with an average of 18%. These savings are primarily from the elimination of object headers and references.

2.4.2 Improving Apache Hadoop with Gerenuk

We evaluated Gerenuk on Hadoop's latest version (2.9.2) using two real-world datasets: the full StackOverflow and Wikipedia data dumps. Table 2.2 shows the details of the seven programs we used. These programs are also real-world programs that were uploaded/discussed by developers on StackOverflow for technical inquiries. We converted them (*e.g.*, by adding some code to make them compilable) for use as our benchmarks.

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

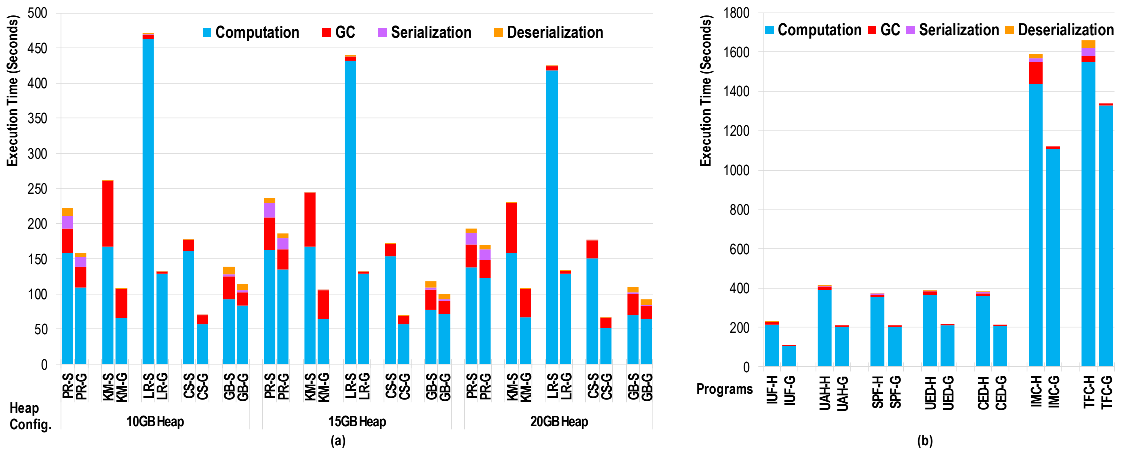


Figure 2.6: Runtime comparisons for Spark (a) and Hadoop (b); each group compares running time, for each program, between the baseline on the left and the **G**erenuk version on the right; each bar is further broken down into four components: computation (in blue), GC (in red), serialization (in purple), and deserialization (in orange).

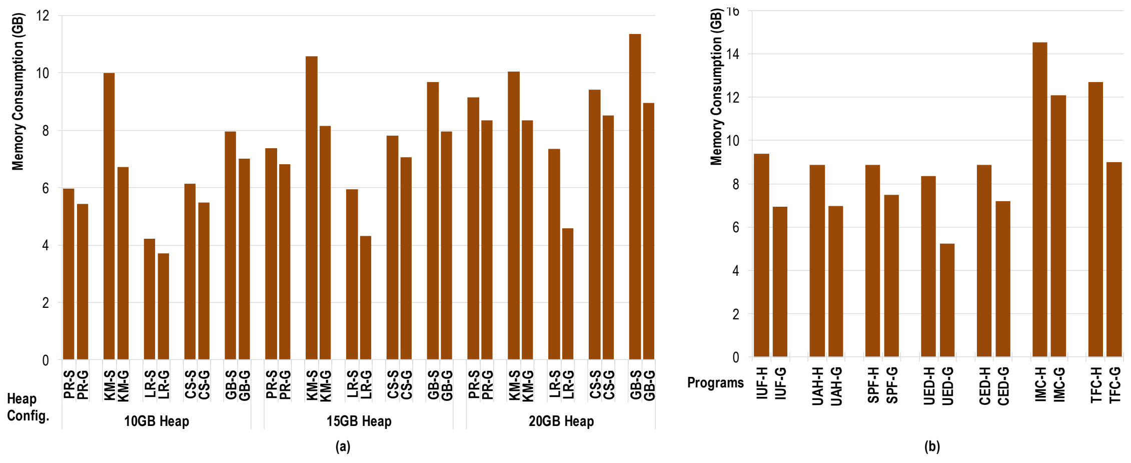


Figure 2.7: Peak memory comparisons between the baseline and the **G**erenuk version for Spark (a) and Hadoop (b); memory consumptions are collected by periodically running `pmap`.

Running Time. Figure 2.6(b) shows the time comparison between the original Hadoop programs and the Gerenuk-transformed programs. Table 2.3 also shows the performance of Gerenuk normalized to that of the baseline of across all benchmarks and all configurations.

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

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

Memory. Because our Hadoop programs created billions of small data objects (*e.g.*, of primitive and string types), inlining their data contents brings significant reductions in memory usage due to the elimination of *all* headers of these objects. As can be seen in Figure 2.7(b), the peak memory consumption on a worker node is reduced by up to 42% (with an average of 31%).

Algorithm 1: Our code transformation algorithm.

```
Input: (1) A set of statements $S$ returned by the SER code analyzer;
(2) a set of classes $C$ forming the class hierarchy of each inlined data structure;
(3) a map $Sizes$ between each class in $C$ and its (inlined) size;
(4) a map $Offsets$ between fields and their offsets in the class hierarchy;
(5) a set $V$ of violation points.
Output: A set of transformed statements $S'$.
foreach Statement $s \in S$ do
    /* Case 1: deserialization point */
    if $s$ is a deserialization "a = readObject()" then
        Replace("long  $addr_a$  =  getAddress()")
    /* Case 2: regular assignment */
    if $s$ is "a = b" and Type(b) $\in C$ then
        Replace("long  $addr_a$  =  $addr_b$")
    /* Case 3: parameter-passing */
    if $s$ is "a = p" and $p$ is a formal param then
        if Type(p)$\in C$ then
            Replace("long  $addr_a$  =  $addr_p$")
    /* Case 4: primitive-type field store on a data object*/
    if $s$ is "a.f = b" and Type(a) $\in C$ and $a.f$ is of a primitive type then
        $off \leftarrow 0$
        if Offsets[Type(a), f] is a static constant then
            /* Offset is statically known*/
            $off \leftarrow Offsets$[Type($a$), $f$]
            Replace("writeNative($addr_a$," + off + "," + SizeOf(f) + ", $b$)")
        else
            /* Offset is an expression*/
            Emit("t  =  resolveOffset(" + Offsets[Type($a$), $f$] + ")")
            Replace("writeNative($addr_a$, $t$, $b$)")
    /* Case 5: field load on a data object */
    if $s$ is "b = a.f" and Type(a) $\in C$ then
        $off \leftarrow 0$
        if Offsets[Type(a), f] is a static constant then
            /* Offset is statically known*/
            $off \leftarrow Offsets$[Type($a$), $f$]
            Replace("b  =  readNative($addr_a$," + off + ", " + SizeOf($f$) +")")
        else
            /* Offset is an expression*/
            Emit("t  =  resolveOffset(" + Offsets[Type($a$), $f$] + ")")
            Replace("b  =  readNative($addr_a$, $t$, "+ SizeOf($f$)) + ")")
    /* Case 6: allocation site */
    if $s$ is "a = newA()" and $A \in C$ then
        $size \leftarrow 0$
        /* The size of the structure is a constant*/
        if Sizes["A"] is a static constant then
            $size \leftarrow Sizes$["A"]
            Replace("appendToBuffer(" + size + ")")
        else
            /* The size is an expression */
            Emit("t  =  " + Sizes["A"])
            Replace("appendToBuffer($t$)")
    /* Case 7: violation handling */
    if $s \in V$ then
        Emit("Abort()")
    /* Case 8: serialization point */
    if $s$ is a serialization "writeObject(a)" then
        Replace("gWriteObject($addr_a$)")
    /* Case 9: method call */
    if $s$ is a call "n.m(a)" and Type(n) $\in C$ and $m$ is not native then
        InlineAndTransform($m$)
```

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

Table 2.1: Description of Spark programs.

Name	**Dataset (Size)**	**Description**
IUF [2]	StackOverflow (37GB)	Inactive Users Filtering
UAH [155]		Active User Activity Histogram
SPF [85]		Spam Posts Filtering
UED [46]		User Engagement Distribution
CED [84]		Community Expert Detection
IMC [151]	Wikipedia Data (49GB)	In-Map Combiner
TFC [54]		Term Frequency Calculation

Table 2.2: Our Hadoop programs and their descriptions.

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

Table 2.3: Summary of Gerenuk performance normalized to baseline in terms of **Overall** run time, **GC** time, **App**lication (non-GC) time, and **Mem**ory consumption across all settings. A lower value indicates better performance. Each cell shows a percentage range and its geometric mean.

2.4.3 Comparing with Existing Systems

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

In our benchmark suite, PageRank is the only program whose data type can be optimized under Tungsten. We rewrote PageRank by using the `DataFrame` API and ran it with Tungsten enabled. Note that to use `DataFrame` and Tungsten, Spark needs to dynamically generate query plans. This does not work well for iterative algorithms such as PageRank because the query plan can keep growing. This is a known and yet unresolved issue since the beginning of `DataFrame`[1]. During our experiment, the `DataFrame`-based implementation of PageRank failed to converge even after 15 hours (while the RDD-based implementation reached convergence in less than 250 seconds). To reduce the execution time, we had to fix the number of iterations to 10. The performance comparison between the original PageRank, Tungsten-enabled PageRank, and Gerenuk-transformed PageRank is shown in Figure 2.8(a).

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

To enable a fair comparison, we added a WordCount program that does not have itera-

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

tive logic. A comparison between the original, the Gerenuk-transformed, and the Tungsten-enabled WordCount is shown in Figure 2.8(b). In this case, Tungsten outperforms Gerenuk by 20% primarily due to Tungsten's string optimizations, which are not performed in Gerenuk. However, these optimizations are designed for simple structured data while Gerenuk targets general user types and data structures.

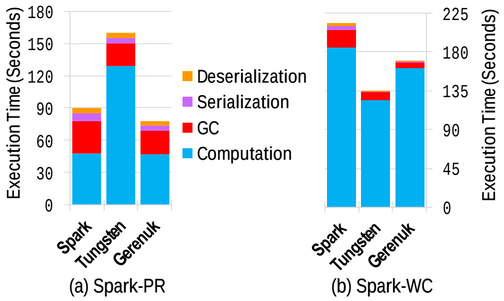


Figure 2.8: Comparison with Tungsten.

Yak [123]. Yak is a Big-Data-friendly GC that is designed to enable region-based memory management for data objects. To understand Gerenuk's ability in reducing the GC cost, we have compared Gerenuk extensively with Yak. To ensure Yak is at its best performance, we obtained Yak's source code from the authors of [123] and replicated the environment described in [123]. We used the StackOverflow dataset as input and annotated the program to enable the Yak optimizations (*i.e.*, by putting `epoch_start()` in the Hadoop method `setup()` and `epoch_end()` in the Hadoop method `cleanup()`). We also followed the heap configurations described in [123]: for each map and reduce worker, we used two different heap configurations: 3GB (map) + 2GB (reduce) and 2GB (map) + 1GB (reduce). The experiment was done on the same cluster as described earlier in this section.

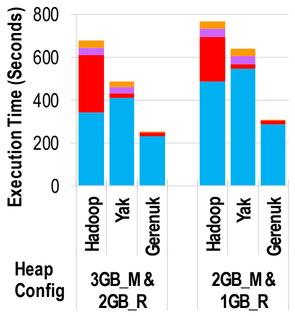


Figure 2.9: Comparison with Yak using Hadoop IMC.

Due to space constraints, we only report the result for the program IMC here. For the other programs, similar behaviors were observed. Figure 2.9 shows the execution time comparison for Hadoop between using the Parallel-Scavenge GC (*i.e.*, the default GC in OpenJDK 8), Yak, and Gerenuk. Clearly, the GC cost for the Gerenuk-transformed IMC is lower than that of both Parallel-Scavenge and Yak. Compared to Parallel-Scavenge, the amount of GC effort during the execution of the transformed program was reduced by a factor of 13.7×. Compared with Yak, Gerenuk further reduces the GC time by 19%. This is because we do not need to scan objects when performing deallocation — our compiler guarantees that control objects can never flow into native buffers while Yak does not have such guarantees.

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

Facade [122] and Flare [55]. Facade transforms programs at the class granularity and relies on the human developer to refactor the program to create a clear boundary between the control and data path. We could not successfully compile Spark and Hadoop using Facade due to Facade's inability to support modern language features such as lambdas and due to its heavy requirement to manually refactor code.

Flare [55] is a compiler-based optimization technique for Apache Spark, which can generate code that processes data directly from optimized file formats. Although it works only for a single-machine environment, we also wanted to compare Gerenuk with it to understand performance differences. However, Flare's source code is not publicly available. Based on the results reported in [55] (*e.g.*, from dozens to hundreds of times of speedup), we speculate that Flare would outperform Gerenuk on a single machine. However, Flare has limited generality because it does not work at all for a distributed environment.

2.4.4 Overhead of SER Aborts

With the programs we took directly from various libraries and online forums, we did not observe any SER aborts. To understand the impact of aborts, we found a Stack Overflow analytics application that uses more complex data types. This application powers a study on collaborative knowledge exchange over a variety of topics from socio-technical sites like StackOverflow [128]. The application has two phases: the first phase constructs a database of all posts grouped by user ID and the second phase uses NLP libraries for concept discovery. In terms of program transformation, we are interested only in the first phase. Internally, the application uses a complex user-defined type called `Account`, which represents a user and her posts. All of a user's posts are stored in a `Vector`. A violation is detected in its `resize` method — when its internal array overflows, a new array needs to be created to replace this array and this replacement involves a reference write, prohibited by our second violation condition. Hence, an *abort* instruction is inserted before writing the new array into the vector object.

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

Figure 2.10(a) reports the results of running time. Overall, about 10% of all `Vector` instances required resizing, triggering aborts. With these violations and re-executions, the Gerenuk-transformed version is 7% slower than its original counterpart. The increase was largely due to wasted computations. We did not observe large increase in other runtime components such as GC, serialization, and deserialization.

To further investigate the cost of aborting a SER, we manually forced SERs to abort at arbitrary points during the execution of PageRank. We varied the number of aborts from 1 all the way to 20 (*i.e.*, which is about 50% of the total number of iterations in PageRank) and measured various aspects of performance on the same cluster where each Spark worker was given a 20GB heap. Figure 2.10(b) shows the running time comparison. The leftmost bar represents the performance of the original program under vanilla Spark, followed by 8 bars showing the performance of the Gerenuk-transformed program with $0 - 20$ re-executions.

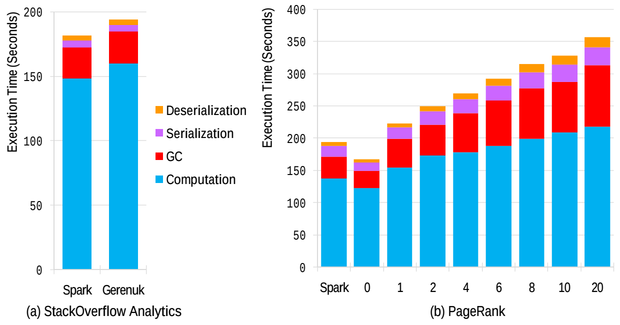


Figure 2.10: Overhead of re-executions in running time for StackOverflow Analytics (a) and PageRank (b).

Overall, each re-execution incurs a 9% overhead compared to a SER in the original Spark, and a 14% overhead compared to a SER in the Gerenuk-transformed Spark. This is primarily due to the round-trip format changing — each re-execution starts with a costly deserialization process that transforms all data in native memory into heap objects. As can be seen in Figure 2.10(b), the amounts of time spent on serialization and deserialization increase by 2× and 3×, respectively. We also observed significantly increased GC effort. In the original Spark, the GC takes about 19% of the execution time. However, with re-executions, the heap contains orders of magnitude more objects deserialized from the bytes, putting severe pressure on the GC. The GC time grows to 30.3% (with the average of 26.1%) of the execution time. The peak memory consumption also increases by 11%, compared to the original Spark run.

2.5 Related Work

Data-Parallel Systems. Numerous optimizations [81, 34, 27, 173, 32, 129, 177, 4, 70, 130, 117, 98, 45, 51, 126, 134, 171, 48, 152, 153, 3, 93, 5, 141, 47, 58] have been attempted from various research communities to improve the performance of Big Data systems. Since most Big Data systems were developed in managed languages, recent works [122, 123, 111, 112, 64, 56, 125] attempt to remove the penalty incurred by the managed runtime. Skyway [125] is a JVM-based technique that can improve the performance of serialization/deserialization by providing support to transfer objects *as is*. Closest to our work are the two compiler techniques Facade [122] and Deca [108]. As discussed earlier in §2.1, these they are both conservative transformation techniques that require heavyweight code refactoring from the user. Flare [55] also tries to transform Spark program into C program but it is limited to only the Spark system. Gerenuk, on the contrary, performs program transformation speculatively based on a transaction model, which significantly improves the transformation practicality.

Niijima [169] is an optimizing compiler that automatically consolidates C# computations on a SQL pipeline to reduce the cost of serialization and deserialization when data is passed from the native (.NET) to the .NET (native) runtime. While Gerenuk is also a compiler-based approach, we achieve soundness by inserting aborts at potential violation points instead of transforming the whole program. Since Gerenuk targets large Java systems, it is much more difficult to guarantee soundness for our transformations than for those performed by Niijima on SQL pipelines.

Compiler Optimizations and Static Analysis. Traditional optimization techniques [40, 24, 161, 62, 63, 71, 168] for object-oriented programs use various approaches to reduce the number of heap objects and their management costs. Free-Me [71] is a compiler-based technique that adds compiler-inserted frees to a GC-based system. Pool-based allocation proposed by Lattner et al. [95, 94, 96] uses a context-sensitive pointer analysis to identify objects that belong to the logical data structure and allocate them into the same pool to improve locality. Prolific types [144] is a static technique that splits objects into a prolific and a non-prolific region to reduce the GC cost. However, these compiler analyses are not designed for Big Data systems that exhibit strong iterational behaviors. Object inlining [52, 99] is a technique that statically inlines objects in a data structure into its root to reduce the number of pointers and headers. While object inlining offers significant performance benefit, existing inlining techniques are impractical as they give up on transforming a program upon finding a violation. By contrast, Gerenuk uses a speculative execution model that does optimistic transformation while aborting speculative executions when necessary for safety.

2.6 Conclusion

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

2.7 Acknowledgements

Chapter 2, in full, is a reprint of Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Harry Xu. Gerenuk: Thin Computation over Big Native Data Using Speculative Program Transformation, the 27th ACM Symposium on Operating Systems Principles (SOSP), Huntsville, Ontario, Canada, October 2019. The author was the primary investigator and author of this paper.

CHAPTER 3

Predicting Dynamic Properties of Heap Allocations using Neural Networks Trained on Static Code

In the previous chapter, we describe an optimization that was enabled by an observation about object behavior. More generally, if we know how objects behave, we can make more informed optimization choices. To do this systematically, we can collect *performance profiles* to do profile-guided optimization. However, in the context of large data centers, it is not enough to do PGO for a single application, and it is not easy to do this for every application on every machine of a cluster. In this chapter, we examine the use of machine learning, particularly code understanding models, to address this issue. These models can help us make with out *statistical* approach to optimistic optimizations: if we have expected statistics about object behavior, we can make informed decisions.

3.1 Introduction

Memory allocators and runtime systems often rely on predicted properties of heap allocations to maximize performance. For example, HALO [142] uses memory access profiling to identify related data accesses, which can be used for heap-layout optimizations. MaPHeA [127] places data based on allocations' memory access frequencies (*hotness*). The LLAMA C++ memory allocator [113] and the ROLP Java garbage collector [29] rely on predicted object lifetimes.

These approaches have parallels to profile-guided optimization (PGO), which in this paper, we take to refer to both offline (e.g., ahead-of-time) and online (e.g., JIT-compiled) approaches. For example, both static ahead-of-time compilers [36] and JIT compilers [60] can leverage branch profiles to optimize code. While branch profiles are cheap to collect [36], heap allocation properties such as object lifetimes, data hotness, or memory access correlations are often much more expensive to profile. For example, ROLP reports up to 6% runtime overheads for profiling even coarse-grained lifetimes, and DJXPerf [102] reports 8.5% overheads even with statistical sampling. While these overheads may not seem large, they are prohibitive in production deployments where even 1% performance degradation is substantial [88].

A common approach is to collect these profiles during a profiling phase: In ahead-of-time compiled languages such as C++, benchmark runs on an instrumented binary are used to collect a performance profile that is then used during the compilation of the final binary. Meanwhile, managed runtimes such as Java Virtual Machines spend a significant amount of time on warm-up [104] during which they collect initial performance profiles and use them to JIT-compile code. These approaches introduce significant challenges:

1. **Deployment Velocity**: Profile collection introduces a long delay into the deployment process. In ahead-of-time compilation, this results in a large number of additional steps before a final binary can be produced. In JIT compilation, it reduces elasticity by requiring services to warm up for longer.

2. **Deployment Complexity**: Setting up PGO pipelines can be very complex, requiring the development of representative benchmarks and flows to run workloads automatically with instrumentation. For JIT compilers, ensuring that the initial load that (e.g.,) a server sees is representative is challenging as well.

3. **Non-representative Profiles**: If the workload used to generate the profile is not representative of the real workload, results will be suboptimal. It is very difficult to ensure that benchmarks are fully representative.

4. **Incomplete Profiles**: Individual workloads often do not exercise every corner case in the code, which means that profiles will often be incomplete.

In this paper, we investigate an experimental and radically different approach to solving these problems: Can we train a machine learning model that can predict heap allocation properties such as lifetimes or object hotness for future workloads from the statically available program code alone, without running an instrumented build or a warm-up phase?

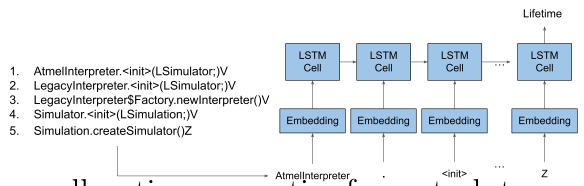
This paper does not offer a conclusion to this question but instead lays out the trade-off space of such an approach, investigates promising directions that suggest such an approach is feasible, and highlights the challenges that need to be collectively overcome to enable it. Our specific contributions are as follows:

- We introduce a framework to reason about the design space for predicting heap allocation properties using machine learning.
- We gather and analyze a data set derived from the DaCapo benchmarks [18] that combines static code with dynamic heap allocation properties.
- We introduce a range of model architectures to predict heap allocation properties from code, and characterize their trade-off space.
- We provide a detailed discussion of challenges to making this approach work in practice, and highlight future research direction.

Our goal is to open up a new research direction for the ISMM community, combining research on ML models for code with research on data-driven memory allocators.

3.2 Background & Related Work

We provide an overview of approaches for predicting heap allocation properties and related work in this area.



```
1.  AtmelInterpreter.<init>(LSimulator;)V
2.  LegacyInterpreter.<init>(LSimulator;)V
3.  LegacyInterpreter$Factory.newInterpreter()V
4.  Simulator.<init>(LSimulation;)V
5.  Simulation.createSimulator()Z
```

Figure 3.1: Predicting heap allocation properties from stack traces using the LLAMA [113] approach.

3.2.1 Predicting Heap Allocation Properties

Leveraging heap allocation properties to improve memory allocation has been a long-standing topic of interest. A classic example is pre-tenuring in managed runtimes such as JVMs, which relies on predicted object lifetimes [20]. Other examples include the prediction of object affinity [142], hotness [127, 102], and container sizes for presizing optimizations [43].

What these approaches have in common is that they need to make predictions at the time a heap allocation is performed. Such predictions are usually performed based on the *allocation context*, namely the program counter of the *allocation site* and the current *stack trace* (Figure 3.1). Note that the allocation site alone is often insufficient to uniquely identify an allocation context [17]. For example, an allocation performed in a string constructor does not provide much information about the allocation's lifetime, but the frame on the stack where the string is allocated might.

Once an allocation can be attributed to a particular allocation context, the second question is how to profile the relevant property. There are a range of different methods. For example, Harris [73] introduced an approach that samples objects and keeps information in a separate data structure. V8 uses "memento" objects that contain an additional pointer back to the allocation site [43], and when these objects are recycled by the garbage collector, it attributes any accumulated profiling information back to the allocation site. TCMalloc [79] samples a small fraction of allocations and collects profiling information just for these objects. This profiling may not always be active.

A significant amount of work in this space has focused on effective ways to summarize the stack trace at the time of allocation to enable such methods, including keeping track of the current calling context in a bit vector [142] or using stack hashing strategies [118]. There has also been work on trying to leverage data on the stack to make these predictions, rather than program counters [44]. However, such approaches are rare and not widely used in practice.

3.2.2 Profile Guided Optimization

Collecting profiles for each allocation context and using them for optimizing memory allocation is a special instance of profile-guided optimization (PGO). Here, we use this term to describe two different types of setup[1]: 1) Collecting profile data from previous, specially instrumented, runs of an application and using this data to improve performance in future builds of this application. 2) The analogous approach in JIT compilers, where profile data may either stem from an instrumented run [22] or the same run [60] of the application. In the latter case, most JIT profiling data is collected at the beginning of the execution, resulting in a period when the JVM runs more slowly, known as *warmup* [104].

PGO has been used to great success in tasks such as cache miss reduction [142], receiver class prediction [69], and I/O partitioning [160]. However, collecting the profiling data required by PGO is often expensive and complex. Generating this profiling data typically requires running an instrumented (and hence, slower) version of the application. For example, techniques such as hot datastream profiling [39] or value profiling [31] can achieve speedups as high as 20%, but require profiling runs that can be tens to hundreds of times slower than the original program.

Our proposed approach takes the place of a PGO optimization pass, but instead of relying on profiles collected in a previous run, it tries to predict these properties from program code. This approach has similarities to LLAMA [113], a recently introduced approach for lifetime

[1]The terminology differs and sometimes only refers to the first as PGO.

prediction in C++.

3.2.3 LLAMA

LLAMA is a memory allocator that uses a neural network to predict lifetime classes of objects using a symbolized allocation context. It assumes a scenario where a *partial* profile is available – e.g., from a previous version of the same program or where only a subset of allocation contexts was observed due to sampling. The authors show that profiling data (in their case, lifetimes of objects) transfers between similar binaries. For instance, LLAMA could accurately predict lifetimes of unseen stack traces even after a number of code changes or when changing compiler settings.

LLAMA performs these predictions by converting each stack trace into a list of symbols and treating this representation as natural language (Figure 3.1). It subdivides the stack trace into tokens which are then passed into a *Long Short-term Memory Network* (LSTM) [78]. This model is trained against known stack traces and associated object lifetimes. A caching mechanism is used to run this model only the first time a particular stack trace is encountered.

The key idea of LLAMA is to use ML to extract programmer intent by treating the symbols of the program as language. While this work takes a step towards generality, it is rather limited. For example, LLAMA cannot generalize to completely different programs, primarily because the calling contexts were represented only by function names. While LLAMA performs well on a single program, it is not applicable to a new program full of different function names, as the function names in the training set might not appear in this new program. Furthermore, the approach is not amenable to even the same application with all of the function names changed (e.g., due to refactoring or obfuscation), even if program behavior has not changed.

3.2.4 ML for Code and Programming Languages

There has been a large amount of work on ML for code in recent years. Allamanis et al. [6] provide a survey. A significant portion of this work looks at the problem from a software engineering perspective — such as neural code completion [131, 49] and finding bugs in code [135, 157].

There have been uses of ML for compiler optimizations [97], but they are less common. MLGO trains models to make inlining and register allocation decisions within LLVM [154] (the latter is a problem that sometimes uses PGO). Autophase [72] learns an ML policy for ordering compiler passes that generalizes to unseen programs, but makes decisions at the granularity of entire compiler passes, rather than individual objects or their profiles. Rotem and Cummins show that instead of relying on PGO profiles for branches, they can be predicted with learned decision trees [139], which resemble a complex, learned compiler heuristic for branches. None of this work looks at properties of heap allocations, which are more complex and difficult to predict. Source code could provide the necessary signal for these predictions to the model, even without profiling data for a calling context.

3.3 High-Level Overview

We now provide an overview of our approach. We start with a conceptual framework how to reason about the general problem of predicting heap allocation properties. We then describe the intuition behind our proposed ML approach.

3.3.1 Conceptual Framework

At a high level, prediction of heap allocation properties can be performed using a broad range of methods, including profile-guided optimizations, program analysis and heuristics. The lines between these methods can be blurry. We therefore start by introducing a conceptual framework to reason about these problems and the associated challenges.

We classify approaches addressing this problem along three dimensions: Data, Model, and Application. *Data* describes the input used to drive predictions (such as the instruction pointer of the allocation site, symbolized stack traces, or an abstract syntax tree of the code). *Model* describes the mechanism by which the data is used to determine a label, such as object lifetime – this could be any function and does not have to be an ML model. *Application* describes how the predictions made by the model are used (e.g., how code is compiled differently based on this prediction, or how a memory allocator may leverage it). Many different strategies map to this conceptual framework:

Profile-guided optimization Depending on the specific optimization, the *data* the prediction is based on is code locations of a particular allocation site or on the call stack. The *model* is a lookup table that maps these call stacks to previously measured values. This approach may be *applied* offline or online, at compile time or at run time.

Static analysis The *data* is typically some form or intermediate representation (e.g., LLVM IR). The *model* is an algorithm that processes this IR symbolically (e.g., by applying escape analysis) to make a prediction. The *application* of this prediction occurs usually within the compiler.

LLAMA The *data* LLAMA uses are symbolized stack traces rather than the instruction pointers. The *model* is an LSTM neural network that maps these stack traces to lifetime classes. The *application* is to make these predictions online in the context of a custom lifetime-aware memory allocator.

The part of the design space explored in this paper is one where the data spans all code that is statically available without running the application, models capture a wide range of different machine learning methods, and applications contain a range of offline and online methods. We now motivate why we believe this is a promising design space to explore.

3.3.2 Our Approach

Instead of collecting profiling data through expensive instrumentation like PGO, we could predict it using machine learning. A strawman approach would be to train a model on profiling data from a set of binaries, and use this model to predict the profiling data for new, unseen binaries. If the accuracy is high enough, we could use these predictions in the same way as profiles, but without the overheads.

LLAMA took a first step in this direction, by training a model on a subset of allocation contexts in one binary and using this model to predict unseen allocation contexts in a potentially different version of the same program. LLAMA showed that symbolized calling contexts are sufficient to perform these predictions with high accuracy.

However, we find that the same approach does not work well *across* programs. We recreated a LLAMA model and attempted to predict object lifetimes on DaCapo [18] benchmarks. When predicting across benchmarks, we find that this model performs only one percentage point better than random prediction (we discuss this further in Section 3.5).

Intuitively, the function names contained in the symbolized calling context capture the behavior of the functions. A model can learn that when certain names appear in a certain order in a calling context, the corresponding object will be long or short-lived. However, since we want to predict on unseen binaries, it is likely that the model will encounter many unknown names, or a new and unusual combination of names, in unseen calling contexts.

To address these problems, we propose to move beyond just function names and instead consider the whole source code, which could capture program behavior from code structure, variable names, and even comments (although we do not currently explore the latter in this work).

3.3.3 Intuition

Source code defines the program behavior and could thus be used to predict heap allocation properties, which are determined by this behavior. There have been a number of works that predict many different properties of code, such as security vulnerabilities [37], performance [35], bugs [135, 157], types [74], summarization [57], and other static program properties [138]. There is also work on representing code in other ways, such as graph representations [7]. However, prediction of dynamic properties has seen less attention.

Listing 3.1a shows a strawman example of the intuition why code structure is predictive of heap allocation properties, such as object lifetimes. We can see two allocation sites, one defining a variable x, and another defining y. When looking at the source code, we can see that y will outlive x, as it exists throughout the execution of the *main* function. On the other hand, the variable x will live only as long as one iteration of the inner `for` loop. In this way, we can compare the lifetimes of the two variables, based on the source code, and conclude that y will have a longer lifetime than x. A model could learn a general pattern that variables defined in the inner-most part of a nested loop are likely short-lived, and variables defined in the main function are likely long-lived.

Additionally, the variable names themselves can provide useful information [8, 87]. For example, variable names may contain short descriptive atoms such as *tmp*, or suggest a relative lifetime ordering between variables. Consider, for instance, *requestBatch*, which hints at an object that encompasses multiple requests, versus *requestStatus*, which hints at an object with a lifetime shorter than that of a request.

A more concrete example is shown in Listing 3.1b. In this example, taken from the FOP benchmark in DaCapo, we can see an array named *tmp* that is used only for constructing the String and nothing else. Here, the variable name signals intent that it is not used in other places.

Another useful variable naming pattern is that short-lived objects often have short names.

```
void doWork() {
    for (int i = 0; i < ... ; ++i) {
        for (int j = 0; j < ... ; ++j) {
            Bar x = new Bar();
            doTask();
        }
    }
}

void main() {
     Foo y = new Foo();
     doWork();
}
```

(a) Strawman: Code structure provides hints about object lifetime.

```
public final String readTTFString()
    throws IOException {
    int i = current;
    while (file[i++] != 0) {
        ...
    }
    byte[] tmp = new byte[...];
    System.arraycopy(file, current,
                     tmp, 0,...);
    return new String(tmp, ...);
}
```

(b) Apache FOP: Variable names provide hints about lifetime.

```
public Grammar getRootGrammar(...)
    throws IOException {
    ...
    File f = null;

    if (haveInputDir)
        f = new File(...);
    else
        f = new File(...);

    fr = new FileReader(f);
    br = new BufferedReader(fr);
    grammar.parseAndBuildAST(br)
    ...
    br.close();
    fr.close();
    return grammar;
}
```

(c) ANTLR: Short-named variables have short lifetime.

Listing 3.1: Motivating code examples.

An example is shown in Listing 3.1c. This method is found in the source code for ANTLR[2], a parser generator written in Java. In this example, we can see a number of variables with short names: *f, fr, br.* Each of them is used locally and does not outlive the method call.

Finally, when available, code comments, literals, and logging statements can provide hints about lifetimes of objects.

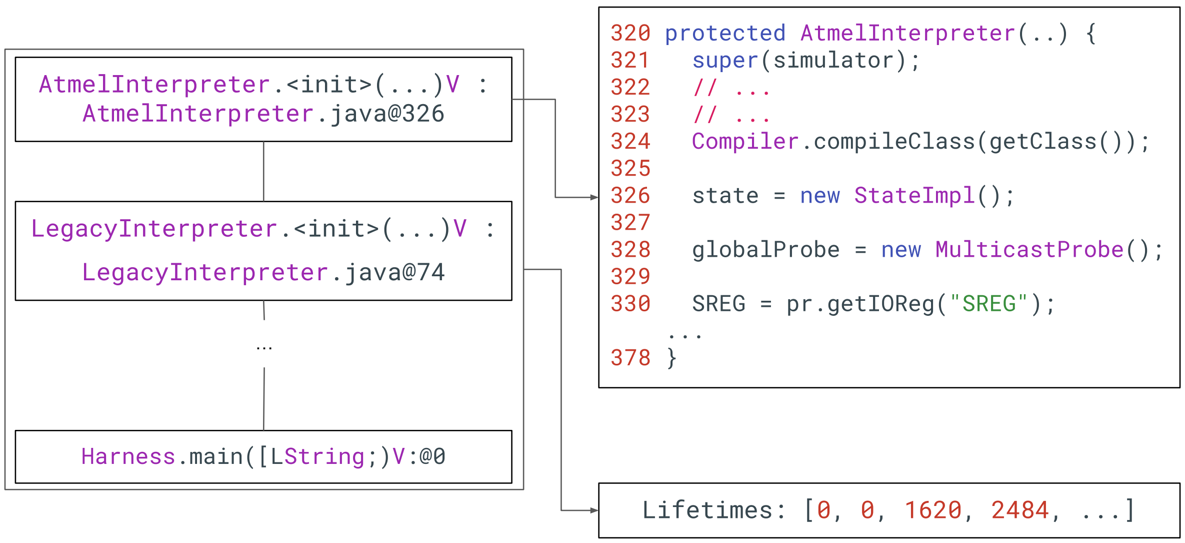
Some of these properties could be exploited without learning. In fact, we could manually construct a very large number of such rules, which is similar to how compiler heuristics are often designed. However, such rules would be brittle and shift over time. Machine learning provides a way to "generate these rules automatically", by having a model learn them instead of deriving and encoding them by hand.

In the following sections, we discuss the potential design space, challenges, and potential solutions associated with such a machine-learning based approach. To support our exploration, we developed an end-to-end implementation of the approach for object lifetimes. While our experiments show evidence that an ML approach is able to learn heap allocation properties, we also find that its current accuracy is limited. In each of the following sections, we describe the trade-offs and challenges that contribute to these limitations, and the research problems that we believe need to be solved to further increase accuracy. Following our framework, we will discuss the underlying *Data* (including training data collection), *Model* (including a range of different designs) and *Application.* We focus on Java, but our approach is generally applicable to other languages such as C++.

3.4 Part 1: Data

Training a heap allocation property model requires two types of information: static information and dynamic information. The static information can be found in code, while the dynamic information must be found through profiling. Related work has mostly looked at

[2]https://github.com/antlr/antlr4


AtmelInterpreter.<init>(...)V : AtmelInterpreter.java@326
LegacyInterpreter.<init>(...)V : LegacyInterpreter.java@74
...
Harness.main([LString;)V:@0
Lifetimes: [0, 0, 1620, 2484, ...]

```
320 protected AtmelInterpreter(..) {
321   super(simulator);
322   // ...
323   // ...
324   Compiler.compileClass(getClass());
325
326   state = new StateImpl();
327
328   globalProbe = new MulticastProbe();
329
330   SREG = pr.getIOReg("SREG");
  ...
378 }
```

Figure 3.2: An example of a stack trace with associated profiling data and source code

either one or the other. Code models look at large amounts of code, but do not connect them to object lifetimes or other dynamic properties. PGO collects such dynamic properties, but only minimally connects them to code, usually via stack traces.

Similar to LLAMA, we collect stack traces that represent *allocation contexts*, the calling context under which particular objects are allocated. Each stack frame of the stack trace represents a certain function, centered around a callsite or an allocation site (in the case of the topmost frame). Each stack trace is associated with the distribution of object lifetimes. In contrast to LLAMA, we also collect the appropriate source code corresponding to each stack frame. An example of a single stack trace is shown in Figure 3.2. We collected this stack trace from the avrora benchmark in the DaCapo benchmark suite [18]. On the left side, we can see a sequence of function calls. Each stack frame is qualified with the full classname of the function (although some are omitted here for the sake of clarity). The stack frames are also associated with a particular file and line number. For example, the first stack frame, representing an allocation site, corresponds to line 326 of the file AtmelInterpreter.java. Note that some stack frames are missing source code locations. We discuss this problem in Section

3.4.4. Lastly, the entire stack trace is associated with our object lifetime profiling data. We now describe one way of collecting this dataset, and challenges that we encounter.

3.4.1 Collecting Dynamic Properties of Objects

While lifetime predictions are a language-agnostic problem, in this work we collected a Java dataset. To this end, we modify OpenJDK11 to perform fine-grained profiling of object lifetimes. While OpenJDK does not have an easy way of collecting fine-grained object lifetimes, there are several approaches in the literature on how to collect object lifetimes in Java. Naively, it would be possible to measure lifetime as the number of garbage collections that an object survived. However, object lifetime granularity would be determined by the frequency of GC passes. Since a full garbage collection can take milliseconds or even minutes, this is much more coarse-grained than the lifetimes that we wish to capture.

Alternative approaches include algorithms such as Merlin [76] and Resurrector [166]. The general approach in these cases is to track incoming references to objects and determine the time at which an object dies, either retroactively during GCs (Merlin) or during execution (Resurrector). We add a simpler version of the latter approach to OpenJDK, by adding reference counts to each object, to detect when it becomes unreachable. We modify the object header to include a counter and modify relevant instructions (aload, astore, areturn, new, athrow) to update these reference counts. Additionally, when an object is allocated, we walk the stack to find the object's allocation context. Like Resurrector, our approach cannot detect cycles and ignores them.

For each object, our instrumentation must collect a lifetime. We defer the definition of our units of lifetime to Section 3.4.4.2 but for clarity in this section, we track lifetime as a *logical duration*, i.e., a difference between values of *logical time*. We maintain a global, monotonically increasing logical clock. Each object tracks its initial allocation time (*i.e.*, the logical clock at the time of its allocation). When an object's reference count reaches zero, we find its logical lifetime by subtracting the current logical clock by the object's logical allocation time. Because we care about allocation context (rather than individual objects), we aggregate this lifetime information by allocation stack trace.

It would be prohibitively expensive to store the individual lifetimes of each object at full granularity. Instead, we classify each object into lifetime classes separated by orders of magnitude: < 10 time units, 100 time units, 1,000 time units, etc. This mirrors LLAMA's bucketing of object lifetime, however in our case the bucketing is based on logical lifetime, rather than wall clock time. For each observed allocation site, we then store only ~ 10 integers: a histogram of the number of observed objects with each lifetime class. While this does lose some specific per-object information, it is a good tradeoff between collecting lifetimes and saving memory. Because we eventually classify stack traces based on this bucketing scheme, this granularity of data collection is sufficient.

3.4.2 Collecting Source Code

In order to use source code as input for our models, we need to collect the source code of our target application and any third party libraries it uses. We need all of this code, as some stack frames will be calls to these third party libraries.

Depending on the code base, this may be difficult to do automatically. We target the DaCapo benchmarks, which are built using tools such as Ant and Maven. We manually inspect the Ant build file and record the necessary third party dependencies that are downloaded. It is important to point out that not every dependency has a readily available source code download, even if many are hosted on the central Maven repository. This is particularly challenging as we must also use the exact same version of the source code that the compiled dependency uses, as using the wrong version of the source code could degrade the quality of the dataset.

We then manually create a source code repository containing all of the application code, Java standard library code, and third party dependency code. We organize the files by package, as Java code is typically organized. For example, the source code for the class `java.lang.String` will be in the file `java/lang/String.java`. This approach works well for Java code, as each source file will contain a single outer-level class and and we can find the source file for `java.lang.String` within the folder `java/lang`.

3.4.3 Contextualizing Stack Traces

After collecting the necessary source code files, we need to connect them to our stack traces. We call this process *contextualizing* the stack traces, as it connects the source code context to each stack frame of the stack trace. Each stack frame contains a line number (representing a call or allocation site) as well as a fully qualified method name. Using the fully qualified method name, we can find the appropriate file in the aforementioned source code repository, as the method name (with periods replaced with slashes) will point to a particular file. We then associate every stack frame of every stack trace with the entire source code of the appropriate method. This is a very large number of tokens for a model to handle. We discuss this challenge in Section 3.5.

3.4.4 Challenges

We now describe important research questions that need to be addressed to improve the approach.

3.4.4.1 Finding All Source Code

In many scenarios, not all source code is available. At least in Java code bases, it is rare for all third party library source code to be in the same repository as the application code. Additionally, because of the multitude of Java build tools (*e.g.*, Gradle, Maven, Ant), it would be difficult to gather code on a large scale (*e.g.*, on all of GitHub) since the process would be different for every repository.

In this work, we give a special "..." token to such stack frames (instead of source code) as a way to inform the model that the source code was missing. Although missing source code of a stack frame robs the data representation of code-specific nuance, at least a partial stack trace is visible to the model, and missing frames are represented the same way (albeit with a different function signature), akin to the way that ML vocabularies represent an "unknown" token.

Future work could look at more sophisticated techniques, such as imputing unknown code, decompiling byte code, or approximately adapting prior code versions to fill gaps.

3.4.4.2 The Right Logical Clock

Traditionally, allocation lifetimes in memory managers are measured in terms of a logical clock, usually the total number of *allocated bytes* [17]. The reason is that such a clock is more stable than wall clock time in light of performance variations (e.g,. due to sharing a machine between different workloads). Given that instrumentation is often expensive, logical time may sometimes be the only option – e.g., we observed more than two orders of magnitude slow-down in our instrumentation, and Merlin reports up to 300$\times$ slowdown [76]. Wall clock times would be meaningless in such a scenario and we therefore use *allocated bytes* as logical time.

However, a logical time base is specific to a particular workload. For example, an image processing workload with MB-sized allocations may have entirely different allocation sizes than a text processor with KB-sized allocations, and the same library code behaving in the exact same way in both applications may result in orders of magnitude of difference in logical lifetimes. When used in the context of training a model across workloads, this means that these lifetimes may not be comparable and thus not learnable – an effect we observed in our own experiments.

This creates a dilemma: While wall clock time may be more stable during learning and enables better transfer across workloads, it is more sensitive to local performance variations and instrumentation effects. Meanwhile, logical time is more stable within an individual workload, but does not transfer well across workloads.

We believe there are several directions of addressing this problem. For example, LLAMA enables the use of wall clock time by introducing a cheap, sampling-based approach for C++, at the cost of not capturing all allocation contexts and sensitivity to performance variations due to compiler settings [113]. Another approach may be to develop new logical time bases that are consistent across workloads.

3.4.4.3 Missing Context

Even with wall clock time and in the absence of performance variations, lifetimes may not be stable across different binaries – or even the same binary with different inputs. For example, imagine two server workloads that use the same server framework to process requests, and where the lifetime of an object is identical to that of its request. If the timescale of work within each request is very different (e.g., microseconds vs. seconds), the lifetime cannot be statically predicted without analyzing the unrelated code that performs the actual operation within the request. This code may not be anywhere near the allocation site and is thus not included in our contextualization approach.

As in the previous section, logical time bases that are less dependent on such variations may alleviate the problem. Another approach may be to define a *context* and express lifetimes with respect to this context rather than in absolute terms (e.g., that the lifetime of an object does not exceed a partcular request or subportion of a program). A third option may be to expand the scope of the data set to include code that is not involved in the allocation itself.

3.4.4.4 Data-Dependent Lifetimes

Objects with the same allocation site may have different profiling data, but are represented by the same stack trace. For example, the lifetime of an allocation may be determined by a dynamic input parameter and is thus different for every input. Because we want to assign a single label to each allocation context, we must choose a single value from the distribution of profiling data. In our dataset, 72% of the stack traces observed objects with more than one lifetime class.

Choosing the right label in such cases is an important challenge. We currently assign a label to each input example based on the most common lifetime class found among objects allocated at the stack trace. For example, if a stack trace observed objects of every lifetime class, but had mostly lifetimes in the range 0 to 9 bytes, then it is assigned a label of 0. However, in some cases it may be preferable to pick the extreme labels (*i.e.*, the max observed or min observed lifetime), but this is best determined by the downstream task.

Additionally, different inputs might result in entirely different stack traces, so it is important to collect a representative dataset during profiling.

3.4.4.5 Ambiguous Allocation Sites

Each allocation context represents a different number of objects and thus lifetimes. Some represent a single object, while others may represent millions of objects. In our dataset, on average, an allocation site represents about 2,000 objects. This may affect the best labelling strategy, as the most common label may be incorrect for thousands of objects, even if it is the most common label for the stack trace.

There are a number of potential strategies to address this issue. One approach would be to introduce additional features into the model to facilitate differentiation between allocation sites. Another approach is to define lifetime classes in such a way that allocation sites are more likely to only have one label. Finally, Zhou and Maas investigated a similar problem in storage systems and proposed predicting a distribution of lifetimes rather than a single value [176].

3.5 Part 2: Model

We now turn to our exploration of learning the lifetime of objects. In all cases, we attempt to learn a function f that predicts a lifetime label: $predicted\_label = f(stack\_trace)$.

In this exploration, we are probing the ability to generalize a lifetime model beyond the binary it was trained on. We compare LLAMA as a baseline (Section 3.5.2), to techniques that enhance generalization of function-signature models (Section 3.5.3), as well as techniques that utilize more of a stack frame, such as code tokens (Section 3.5.4) and code structure (Section 3.5.5) or their combination (Section 3.5.6). We see that model generalization improves, but our results are only a hint that more work in this space is desirable. Section 3.5.7 suggests future directions.

3.5.1 Training Details

We convert our lifetime class profiling data into a binary classification task, where logical lifetimes $< 100K$ Bytes are "short" and higher logical lifetimes are "long".

We split our DaCapo dataset by benchmark, and arbitrarily choose three sets of benchmarks: 1 benchmark for validation (fop), 1 for testing (h2), and the rest for training. We perform the split this way as it most closely aligns with our goal of predicting across binaries. The alternative is to mix all stack traces into one dataset, then create splits. However, this would be more akin to a single-binary prediction task (with incomplete profiles) rather than cross-binary prediction.

We focus on classification accuracy as our target metric. Our datasets have high skew between the short and long lifetime classes. Roughly 90% of stack traces have the short label, while the remaining 10% have the long label. So, if a model simply predicted "short" for every example, it would achieve a vacuous 90% accuracy. We combat this skew in two ways: (a) we subsample the majority class to have a similar size to the minority class during training, and (b) we use Mean Per-Class Accuracy (MPCA) on the (non-subsampled) validation and test datasets. In the pathological example above, if a model predicted "short" for every stack trace, it would only achieve 50% MPCA (100% accuracy on "short" and 0% accuracy on "long"). We do not subsample the test split, because we wish to control for a task with skewed labels in practice, although during training it is important to teach the model with enough emphasis on both labels.

We implement these models using TensorFlow and Keras [1, 42]. We train our models using TPUv2s and TPUv3s on Google Cloud Platform. For the Transformer-based models, we use the Transformer implementation found in the BERT repository [50]. We perform a hyperparameter search for each model. For the simple LSTM models, we vary learning rate, sequence length, embedding size, and LSTM cell size. During training, we utilize recurrent dropout in the LSTM cell. For the Transformer-based models, we vary learning rate, hidden size, and number of layers. Due to memory constraints, we use only the top 32 frames of a stack trace in the Transformer models. A single configuration for the LLAMA-like models takes roughly an hour to train, while a single configuration for our Transformer-based models finishes training in roughly 22 hours.

We train by minimizing the binary cross-entropy loss on *predicted_label* compared to the ground truth we collected (Section 3.4). Specifically, we try to minimize function $L = -\frac{1}{N}\sum_{i=1}^{N}[t_i \log(p_i) + (1-t_i)\log(1-p_i)]$, where N is the number of examples, t_i is the ground-truth of the i-th example (0 for "short" and 1 for "long"), and p_i is the predicted probability that example i is "long". For each hyperparameter configuration, we select the checkpoint that achieves the highest MPCA. For each model type, we report the MPCA of the best checkpoint of the best configuration.

3.5.2 Baseline: LLAMA

LLAMA treats each stack frame in the stack trace as a string and tokenizes function signatures on special characters such as `,` and `::`. It then separates the tokenized stack frames with a special `@` token. These tokens are then encoded using a vocabulary to map each token to a specific vocabulary ID. The tokens of the entire stack trace are then fed into an embedding layer, then this sequence of embeddings is fed into an LSTM recurrent neural network, resulting in an embedding of the entire stack trace. The final embedding is then used to predict a lifetime label for the stack trace. This relatively straightforward model performs very well in their task of predictions on a similar binary to the training data.

LLAMA used only function signature tokens in its representation. However, this information is not enough to generalize. To demonstrate this experimentally, we recreate a LLAMA-like model and try to predict object lifetimes on our DaCapo dataset. Since our dataset is in Java, we tokenize our stack traces in similar ways to LLAMA's C++ tokenization.

We train this model in two different scenarios. First, we train on the entirety of our DaCapo dataset and test on the entirety of the dataset. This scenario parallels perfect coverage in LLAMA's data collection (LLAMA needs to use predictions, as not all stack traces can be covered by their sampling-based data collection). In this experiment, this LLAMA-like model for Java achieves 92% MPCA, meaning that this LSTM model could mostly lookup this previous profiling data.

The more interesting case is when we attempt this across benchmarks. When we train on only the training set, the model is not able to predict well on the test set. It achieves an MPCA of 51% on the held out test dataset (recall that random prediction would be 50%).

There are a few potential explanations for this result. First, it could be that out-of-vocabulary words can have a big impact. While our vocabulary of 5,000 tokens covers more than 99% of tokens (*i.e.*, less than 1% of tokens must be encoded as a special out-of-vocabulary token), it might be the case that these tokens are very important for generalization. Second, function names may just not be representative of object lifetime across benchmarks. One potential solution is subword tokenization [25], which we address next.

3.5.3 Subtokenization of Function Names

Subword tokenization is a middle ground between word-based and character-based tokenization. Common words will be included in the subword vocabulary, while rare words can be losslessly encoded using subword tokens. This removes the out-of-vocabulary problem, as in the worst case, even unseen function names can be encoded (as single characters).

Another potential benefit is that some subword tokens in class or method names might be helpful for the learning task. Simpler token encoding would consider an entire name at once. This means that three method names called *get*, *getBestPlanItem*, and *getErrorListener* would each receive an unrelated, different ID. However, their names all contain *get*. With subtokenization, these methods could all share a *get* subtoken, followed by a *BestPlanItem* or *ErrorListener* subtoken if needed. Stack traces that share a descriptive subtoken might behave in a similar way. For example, in our DaCapo dataset, stack traces that contain a *get* subtoken in the top-most frame observe long-lived objects 39% of the time. Stack traces without such a subtoken observe long lived objects 8% of the time, which is much closer to the distribution of lifetimes as a whole.

However, this still is not enough signal for the model. We again create a LLAMA-like model, but this time we use subword tokenization. We use the CuBERT [86] Java tokenizer to tokenize our function signatures, ignoring whitespace tokens. We then encode these tokens using CuBERT's Java subword vocabulary and the Tensor2Tensor library [158] to produce a sequence of IDs. Using the same model architecture as in section 3.5.2, we embed the IDs and produce a prediction. On the same DaCapo train/test split, this model achieves an MPCA of 53%, which is only 2 percentage points higher than the non-subtokenization model.

One possible explanation could be that subtokenization adds more tokens, but perhaps not enough extra signal. While subword tokens might be useful information, since we can only process a limited number of tokens, adding more tokens might reduce useful signal as

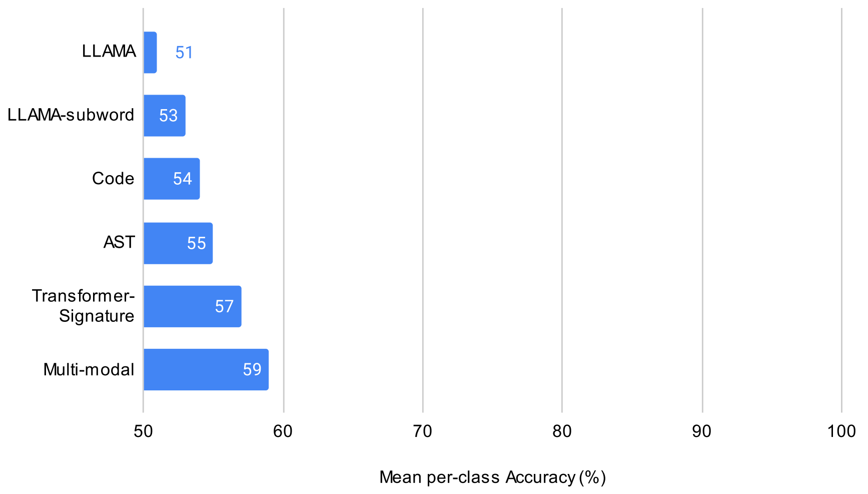


Figure 3.3: Mean per-class accuracy on the DaCapo holdout test set. Note that random predictions are 50% MPCA, so the difference between 51 and 59% is larger than it may appear

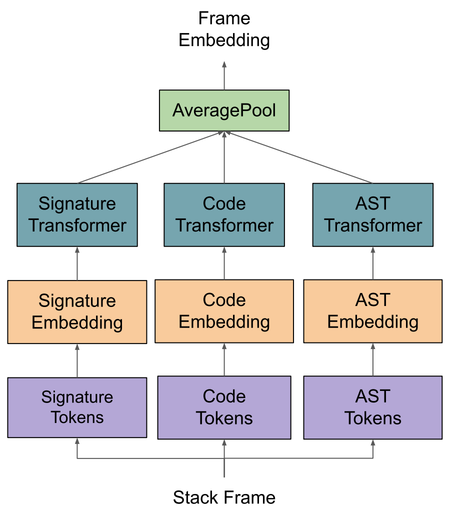
some tokens must be pushed out. It could also be the case that these "helpful" subword tokens are simply not enough signal.

From these experiments, it is possible to conclude that function names, even when encoded in different ways, do not seem to provide much signal to the model for this lifetime prediction. We therefore turn to a different feature: code.

3.5.4 Featurizing Stack Traces with Code

Code may offer the signal that we need for this prediction. As mentioned in Section 3.3, code defines program behavior, and should be more useful for generalization than function names across binaries. We thus apply code embedding models.

As described in Section 3.4, we associate every stack frame with the source code of the appropriate function. Additionally, each stack frame has a line number, representing an allocation site (in the case of the topmost frame) or a call site. One major challenge is selecting what code to embed for each stack frame. It is difficult to train long sequence lengths on traditional Transformer-based language models. For example, pre-trained BERT



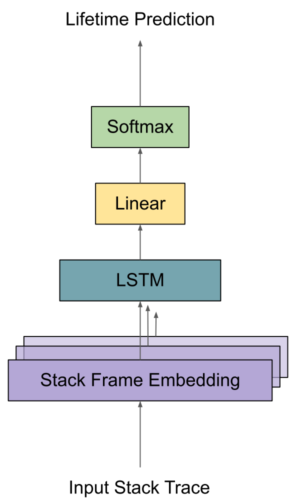


Figure 3.4: Multi-modal stack frame embedding (left). Lifetime prediction model with several frame embeddings (right).

models are often limited to sequence lengths of 512 tokens. However, even a single function in our dataset could have thousands of tokens.

When combined with the fact that our stack traces have tens or even hundreds of stack frames, we can easily run out of token budget. This problem is exacerbated by the fact that many code-embedding models focus on embedding a single code snippet or context. However, in our case, we need to look at many snippets at once, which may even be from separate code bases when considering third-party libraries. Given these uniquely difficult code embedding challenges, we believe that there is room to improve code featurization.

We must select a subset of code tokens for each frame in order to satisfy our token budget. In this work, we take a simple approach: in each stack frame, take a window of tokens around line number of the call or allocation site. Starting at the center token of the given line, we simply add one token from the left of the current window, then one from the right, and so forth until the per-frame budget is reached. If one side reaches the beginning or end of the function, we gather tokens from the non-exhausted side until we reach the per-frame budget, or the entirety of the function is selected. We are then left with a (smaller) sequence of code tokens for each stack frame. The tokens are then encoded using a subword vocabulary, leaving us with a sequence of subword token IDs for each frame.

In addition to limiting budget per-frame, we must also pick certain frames to include. These are related budgets: increasing one causes us to decrease the other to maintain a memory budget. In our experiments, we choose to keep the top 32 frames, and set a per-frame token budget of 256 tokens. We then embed these tokens using a Transformer-based model. Using a Transformer (of max sequence length 256 tokens), we embed each stack frame's code tokens to produce 32 Transformer embeddings, one for each frame. Next, we pass these Transformer embeddings through an LSTM to produce a single embedding of the entire stack trace. Finally, we apply a last Dense layer and a softmax to produce a lifetime label of the entire stack trace.

We train and test this code-only model with the same DaCapo training/test split. It slightly outperforms our simple LLAMA-like models, and achieved 54% MPCA on the test dataset, only 3 percentage points higher than the LLAMA-like model and 1 percentage point higher than the subtokenized LLAMA-like model. Note that this representation does not cleanly supersede that of Section 3.5.3: the function signature does not always fit in the per-frame token budget.

One potential validity sanity-check is that it is not the code that is causing improvement, but the model size. This is a valid concern, as the 32-frame Transformer model (188M parameters) is much larger than a small embedding layer and LSTM (4M parameters). So, we create a comparable signature-only 32 frame Transformer model. Given the same token and frame budgets (*i.e.*, 32 frames, 256 tokens per frame), we subtokenize and embed the signatures of each frame, rather than code. Interestingly, this Transformer-based signature-only model outperforms the code model by achieving 57% MPCA on the holdout dataset, 3 percentage points higher than the code version of the model.

A major problem is the number of code tokens. While a window of code around the call site is straightforward to collect, it may not be the right set of tokens. Tokens that are lexically far away from the callsite (*e.g.*, control flow such as *for* or *while*) may have a large impact on the prediction, but will not be captured by the window. This has been observed before [9, 7, 133, 75] and motivates the next approach.

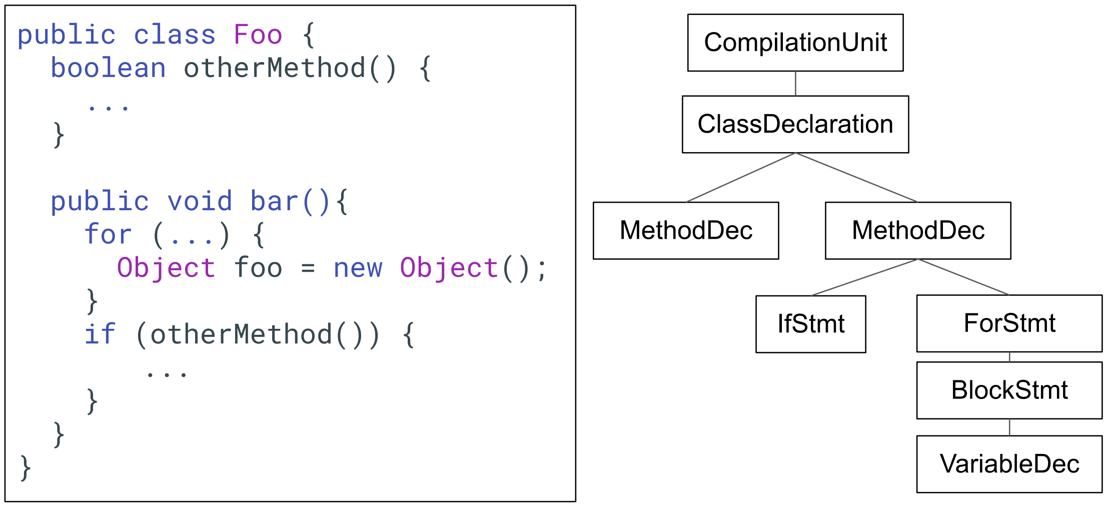
3.5.5 Representing Code with Abstract Syntax Trees

Another potential direction is to represent code using Abstract Syntax Trees (ASTs). ASTs are a tree representation of source code structure. As opposed to concrete syntax trees, or parse trees, ASTs do not capture every detail of the source code, but do capture important structural details. For example, an AST might represent an `if` statement with a handful of nodes: a node for the `if`-statement itself, and three child nodes, representing the condition, `then` branch, and `else` branch. We parse our source code using javalang[3], an open source Python library providing a lexer and parser for Java.

An example of this representation is shown in Figure 3.5. Note that some details are omitted from the figure for clarity. To represent an allocation site, we find the path to the allocation site on the AST. For stack frames that represent a call site, we instead find the appropriate method invocation node. To generate a token sequence, we simply use the names of the nodes on the AST. So, the *foo* allocation site would be represented by the tokens: CompilationUnit, ClassDeclaration, MethodDec, ForStmt, BlockStmt, VariableDec.

While this approach might lose some finer-grained information about the code, namely the variable names, the AST nodes may be useful for capturing high-level structural information, such as loop keywords. We train and test the same 32 frame model on the same DaCapo dataset as before, but using AST tokens. It achieves 55% MPCA, which is in between the code and signature performance.

[3]https://github.com/c2nes/javalang



```
public class Foo {
  boolean otherMethod() {
    ...
  }

  public void bar(){
    for (...) {
      Object foo = new Object();
    }
    if (otherMethod()) {
        ...
    }
  }
}
```

Figure 3.5: A code snippet and associated AST

3.5.6 Multi-modal Features

LLAMA showed that function names are sufficient in some cases. Code precisely defines program behavior and captures programmer intent (with variable names), but is verbose. ASTs are concise but lose fine-grained information.

Instead of using a single representation, we can try to combine them in an attempt to capture the best properties of each representation. This model is shown in Figure 3.4. We instantiate 3 Transformers, with sequence lengths 176, 16, and 64 for code, AST, and signature tokens, respectively. At each frame, the three Transformers produce one embedding each, representing their specific token-type embedding for the frame. The 3 embeddings are then max-pooled to produce a single embedding of the frame. We repeat this for each of the 32 frames, and use an LSTM to produce a single embedding, followed by a Dense layer and a softmax to produce a prediction. While this multi-modal model processes the same number of tokens per frame (256) as the single type model, it has about 190M trainable parameters, compared to the 70M parameters of the previous single-type models.

We train and test this combined-embeddings model on our DaCapo dataset, and find that it achieves 59% MPCA on the holdout dataset. While it is not by much, it was the best performing model by a couple of percentage points.

3.5.7 Challenges

Despite the extra signal we provide the model, and despite out-performing signature-only methods, this prediction accuracy is still not high enough to be usable. The challenges in Section 3.4 are also relevant here, as a dataset can strongly affect a model's performance. However, there are also a number of challenges specific to this family of models.

A major open question is how to solve the context selection process. Due to memory constraints in large Transformer models, selecting the right tokens is paramount. Even with TPUv3s, we found it difficult to train on more than 32 frames, even with tiny batch sizes, and had to ignore the excess frames. However, almost 93% of the stack traces we collect have more than 32 frames, meaning that almost every stack trace has frames missing in its representation. Additionally, some individual frames are very large and must be trimmed down. Of the stack frames that have source code (78% of frames), 18% have too many tokens, and must lose some of their tokens before being presented to the model.

We choose to use the top of the stack as these stack frames are "closer" to the allocation site, and may be more relevant. However, it could be the case that other frames (or even auxiliary features like its height) might be predictive as well. For example, objects allocated with a certain library call on the bottom of the stack are possibly longer-lived.

Code selection within individual frames is important as well. While ASTs might help alleviate the problem of lexically-far tokens, it is not perfect. Carefully selecting the "important" code tokens could greatly improve model performance, as the "non-important" tokens can almost be considered noise that hurts the model. We are considering techniques that prioritize tokens the model is likely to consider "important" (in Transformer parlance, have high *attention* scores), based on an Expectation-Maximization formulation akin to Code-Trek [132].

3.6 Part 3: Application

Our proposed approach could be used for a number of optimization tasks. While we show how it can be used to predict object lifetimes, it could also be used for predicting properties such as object hotness. In general, the predictions are not the end goal: the predictions themselves are used by some downstream task. In the case of our lifetime predictions, this could be deciding to pre-tenure [20, 22] or stack allocate an object [41]. LLAMA used object lifetime predictions to create a memory manager that organized its heap into lifetime classes, rather than size classes. Analogously, object hotness predictions could be used for learned remote-memory prefetching. While there are works that improve prefetching in this setting [114], there may be good opportunities for ML-based approaches because accurate predictions could reduce very expensive remote memory fetches.

An important consideration is the required accuracy of the downstream task. Before modifying an existing system, it is useful to first quantify potential speedups and required model accuracy. As a thought experiment, we can take the example of learned remote-memory prefetching. Before modifying the prefetching system, we can simulate the effect of a model. First, for some application, we could collect a representative sequence of memory accesses, using a tool such as Intel's Pin [109]. Given the sequence of memory accesses, we can compare the page fault rate of the existing system and a model. We might hypothetically observe that the existing prefetching system addresses 65% of page faults, *i.e.*, 35% of requests must fetch remote memory, while the rest avoids a page fault because of the prefetching. If a model could accurately predict 50% of pages to prefetch, the ML-based prefetching would perform worse than the existing system, as it would have a higher page fault rate. If a model could accurately predict 65% of pages to prefetch, it still may perform worse than the existing system because of the cost of running the model. We might then find that, given the cost of a remote-memory page fault, and the cost of running the model online, that the ML-based system breaks even when the accuracy is 75%. A user might then decide that the ML-based system is only worth implementing if they can train a model that achieves 85% accuracy.

Assuming a model achieves sufficient accuracy, we can integrate its predictions into a system in a number of ways, each making a tradeoff between richness of input features and the effect of model overhead.

3.6.1 Online Prediction

In this approach, we run the model at the time a decision is made. For example, at the time an object is allocated, a runtime system could run a model to make the online decision to pre-tenure an object. While this approach was suitable to LLAMA, it may not be possible with our larger code-embedding models. Object allocations are latency-sensitive and must finish in nanoseconds, which is not enough time to run a large model. LLAMA proposes amortizing the cost of running the model by caching model predictions and only running the prediction if the result is not already cached. However, Transformer-based models, such as the ones we use, can easily take milliseconds to run. Even with caching, this may be too expensive to run online in some applications.

One important benefit is that it may be possible to include other live state (*e.g.* the value of certain variables, cache-line state) as a feature to the model. These features would not be available to non-online predictions.

3.6.2 Prediction in JIT Compilers

Another possible use case could be during JIT compilation. For example, OpenJDK runs interpreted bytecode, but when a method is executed enough times, will profile it at run time and compile it with increasing amounts of optimizations. Since a JITed method is hot, any performance optimizations will have a large effect. Because the JIT compilation typically occurs in the background, running a model for some milliseconds could have less of an overhead than running it on the critical path, as an object is allocated. Similar to online prediction, live program state could be used as a feature in the prediction. However, managed runtimes have been tuned for decades and might already do a "good enough" job with their online-profiling and optimizations, even if they are simpler than stack trace predictions we could produce.

3.6.3 Offline Prediction

On the other end of the spectrum is moving the prediction completely offline. The benefit in this case is that there is no runtime overhead to run an expensive model. One way to use this type of prediction is to generate annotations that can be used by the runtime system [22, 124] or optimizations like ThinLTO [83]. However, these predictions can only use source code features, as dynamic state is not available.

3.7 Discussion

While the multi-modal representation of stack traces seems like a promising direction, the accuracy that these models achieve is not yet usable. We see a number of opportunities for improvement.

3.7.1 Token selection

One major open problem is selecting code tokens. There are far too many tokens in a stack trace to include them all. We could try to solve this problem at a function granularity, or by selecting only certain stack frames, or combining the two.

Within a single function, the most valuable tokens might be lexically far from the allocation site, and are missed by our simple window of code selection. One potential direction is to augment the stack traces using static analysis. For example, the code that defines and uses the object, may be more important than lexically-nearby code. These def-use chains may point to the most useful source code statements: the ones that actually affect the object. However, def-use (especially inter-procedural) chains may be difficult to gather on a large scale. This is a trade-off: using an analysis has a time cost, but may produce better predictions.

We select a fixed number of stack frames from the top of the stack, but other stack frames may be more important. Per-frame token budgets also do not need to be fixed. If could rank every token's importance, and we see that a certain function contains many useful tokens, we could increase the frame's token budget (at the expense of another frame).

Finally, we currently ignore comments because of token budget constraints. However, they may provide useful natural language hints to the model.

3.7.2 Modalities

Token-based representations are not the only option. Given recent success in graph neural networks (GNNs) and GNN-based code embeddings, including a graph embedding of the code might prove very useful. Additionally, there might be better ways of handling the different modalities. For example, we used separate Transformers for each embedding type, then a maxpool operation to combine the embeddings per-frame. However, an attention model might be better than a maxpool, or it might be better to keep all of the embeddings rather than aggregating them per-frame. There is a large design space still to be explored.

3.7.3 Labelling Examples

Stack trace-based representation can be ambiguous. Many objects can be allocated at the same site, and will be represented by the same stack trace even if they behave differently. While some tasks might be able to tolerate this label ambiguity by choosing one label (*e.g.* most common or max), others might not. One idea is to augment the data with some dynamic features, for example, GC-related data, current CPU load, or current memory load.

While not available to offline-only predictions, this would be a useful way to disambiguate object behavior, even if they come from the same stack trace.

3.8 Conclusion

In this paper, we present a framework for reasoning about the design space of predicting heap allocation properties with machine learning. We believe that our paper provides evidence that this is a promising approach, but a number of challenges need to be solved to make it practical. We hope that this intellectual abstract opens up a new research direction for the ISMM community and that our discussions of challenges and trade-offs in the design space of this problem will lead to more work that takes advantage of advancements in code embedding models within memory managers.

3.9 Acknowledgments

Chapter 3, in full, is a reprint of Christian Navasca, Martin Maas, Petros Maniatis, Hyeontaek Lim,and Guoqing Harry Xu. Predicting Dynamic Properties of Heap Allocations Using Neural Networks Trained on Static Code. ACM SIGPLAN International Symposium on Memory Management (ISMM'23), June 2023. The author was the primary investigator and author of this paper.

CHAPTER 4

Learned Refinement-based Points-to Analysis

In the previous chapter, we saw how we could use source code embeddings to make predictions about objects' runtime behaviors in order to enable optimizations. Static analysis is a more formal way of analyzing program behavior, based solely on the source source code of a program. In this work, we can take ideas of multi-modality and source code embeddings, as shown in the previous chapter, to make a different kind of optimization: speeding up a very important static analysis. We can again see the idea of *optimistic* optimizations on display: while we hope that our predictions are correct, we have ways of getting around mispredictions in the form of threshold tuning and falling back onto an easier analysis. This threshold tuning and fallback method is another *statistical* approach of optimization: based on needs of the user, and expected performance of our predictions, we can make an informed optimization.

4.1 Introduction

Points-to analysis is a fundamental static analysis which finds the memory locations that a pointer may point to at runtime. It is useful on its own for tasks such as detecting null pointer dereferences, but more importantly, it is a building block for all kinds of tools, such as bug detectors [106, 23], security analyses [13], program repair [61].

One of the most successful techniques for improving precision is context-sensitivity, which distinguishes a function's variables across different *contexts*. This allows the same function to be analyzed under these different contexts, which helps to reduce spurious object flows.

Context-sensitivity has been shown to be key to computing precise points-to information [100] and has been studied extensively. Some summaries of this area can be found in [77, 68]. Many different kinds of context-sensitivity have been introduced such as object-sensitivity [115], callsite-sensitivity [145], and type-sensitivity [146].

While it has been shown to have great precision benefits, context-sensitivity comes with heavy efficiency costs [100, 167]. We can see this experimentally as well. When performing a context-insensitive pointer analysis with Soot [148], on a relatively small DaCapo benchmark [19], it takes roughly 15 seconds. However, when adding context-sensitivity, it takes nearly 40 minutes to complete. There is a large body of work that aims to improve the scalability, using approaches such as selective context-sensitivity [82, 103, 107, 147], summarization [170, 38, 140], or refinement [149].

These problems are often posed as graph problems, such as reachability between two nodes. Given recent advancements in Graph Neural Networks (GNNs), it could be possible to predict this information in order to speedup points-to analyses. Inference time for these GNNs is often on the order of hundreds of milliseconds, which could offer large time-savings over traditional context-sensitive computations, which can take hours or days on even mid-sized codebases.

In this work, we introduce a GNN-based model to predict context-sensitive information of a *context-free language reachability* formulation of points-to analyses. We show how a pointer graph collected by the Soot framework can be used as input to a GNN to predict context-sensitive edge information, given context-insensitive edge information. Inspired by our previous work in chapter 3, we also show how we can augment the graph with multi-modal information, in the form of code embeddings, to improve generalization. Finally, we describe how such a model could be tuned to meet the precision needs of a downstream client tool.

4.2 Background

We provide an overview of some traditional points-to analysis approaches and some related work in the machine learning space.

4.2.1 CFL-Reachability for Points-to Analysis

In this work, we consider the *context-free language reachability* formulation of points-to analyses. Previous works have shown that context-sensitivity in Java can be formulated as a *balanced parentheses* problem [149, 150]. This CFL-reachability formulation is an extension of graph reachability that enables filtering of infeasible paths.

Given a graph with edge labels, and L, a language over the same edge label alphabet, each path in the graph is labeled with a string obtained by concatenating all of the edge labels in order. The path is an L-path if the the string $\in L$. Given nodes s and t, this formulation asks if there is an L-path from s to t. If so, then t is L-reachable from s. As we can see, this is a harder problem than standard graph reachability: while there may be paths on the graph, their edges may not form valid L-paths. For example, we can model field-sensitivity information using L-paths, by considering a balanced parentheses problem, where the language, L, is strings with balanced braces ([and]).

In the context of the points-to analysis, G represents the program. The nodes of the graph are variables and abstract locations. The edges of the graph are various types of assignments between them, such as regular assignments, field assignments, and so on. Edge labels capture sensitivity information. For example, a field write "x.f = y" might be captured by an edge between x and y,(representing an assignment) with a label $[_f$ (representing the write to field f). A field read "w = z.f" is similarly correspondingly represented by $]_f$. In a program, when we write y into a specific field, f, we can only retrieve it from x again with some $z.f$, as long as z points to x. Given a path on the pointer graph, we can collect its edge labels to form a string. If we saw a string "$[_f[_g]_g]_f$", we could conclude that it is a valid L-path, meaning that this is a valid assignment chain that could occur in a real program execution (at least in terms of field assignments). In this case, we captured field-sensitivity by matching the braces edge labels. We can extend this language to also capture callsite-sensitivity in the same way.

We can add parentheses edge labels to our language to also capture callsite sensitivity. Given a method `int foo (int x)` and method call `z = foo(y)`, we draw an edge (representing an assignment) from y to x, where x represents the formal parameter to the `foo` method and y represents some variable. The edge has a label $(_1$, meaning that it is a call at a particular *callsite*, numbered 1. If the function returned a variable r, we can draw an edge from r to z, with an edge label $)_1$, representing a return to a particular callsite. Now, similar to before, we can find L-paths by checking if a path's labels have matching call and return labels: $(_1(_2)_2)_1$ is an L-path but $(_1(_2)_3)_1$ is not. We can see how this becomes very computationally expensive: the string matching problem grows with the number of different callsites in the program. Additionally, we will re-examine the same method many times, if it is used in many contexts.

An concrete example is shown in figure 4.1. The corresponding source code is shown in figure 4.2. In this example, we create two Bar objects which are assigned to a_1 and a_2. Both are passed as parameters into the `foo` function. On the graph, this is captured by an assignment to the formal parameter p of the function. The parameter p is then assigned to some local variables, before the variable y is returned. We capture this return value as an assignment from y to another variable: here it may return to b or to c. We can see that the parameter p is return value. In the main function, we can see two calls to `foo`, one with a_1 and one with a_2. From the programmer's point of view, we can tell that variables b and c can *never* point to the same value: in a particular *context*, the foo function has a particular return value (the parameter). On the graph, if we did not have the edge labels (context-sensitivity), it would appear that either Bar object could flow to b and c (and hence, b may point to c and vice versa). But with the edge labels, we can consider the calling context,

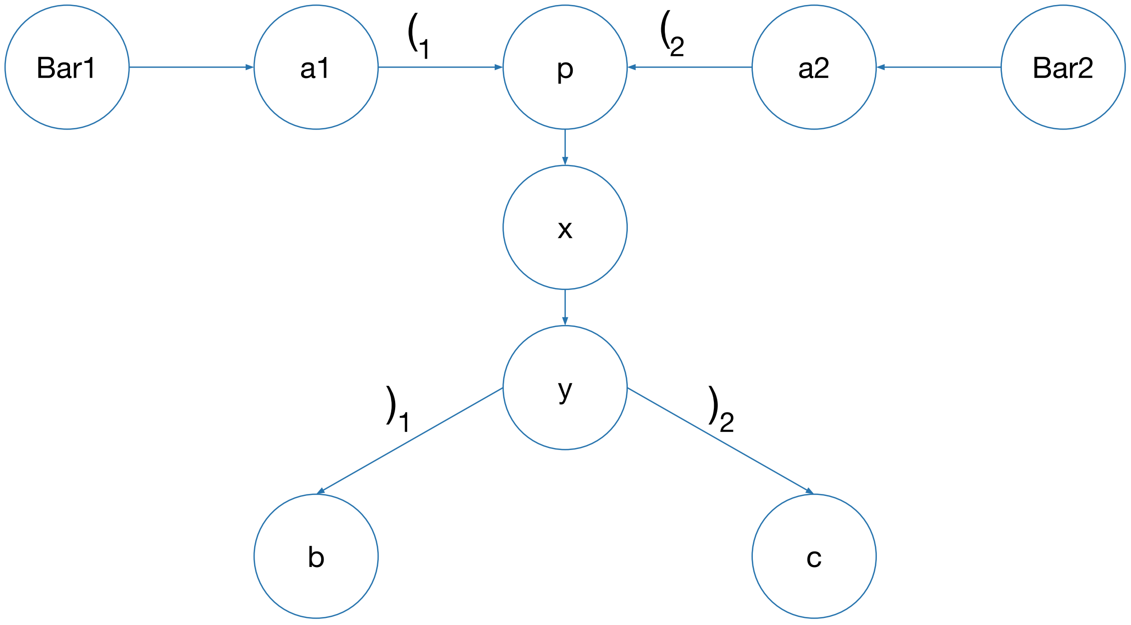


Figure 4.1: A context-sensitive Pointer Assignment Graph

and be more precise: b and c cannot point to the same objects.

It's important to point out that every context-sensitive path is also a context-*insensitive* path, but not vice versa. In this work, given an insensitive path, we would like to find if it is a context-sensitive path or not.

```
Bar foo(Bar p) {
    Bar x = p;
    Bar y = x;
    return y;
}

void main() {
    Bar a1 = new Bar();
    Bar a2 = new Bar();
    Bar b = foo(a1);
    Bar c = foo(a2);
}
```

Figure 4.2: Source code corresponding to PAG in figure 4.1

4.2.2 Graph Neural Networks

Graph Neural Networks (GNN) [90, 110, 143, 105] are a family of neural networks designed for machine learning over graph data [165]. Within this family are many different models, such as graph convolutional networks [90], graph attention networks [28], graph transformer networks [174], and many more. They have been used across many disciplines including recommendation systems [164, 172], chemistry [53], and code understanding [7, 15]. Given the recent successes of these models, we would like to predict context-sensitive information using GNNs, but first, we must structure our graphs and learning problem in the right way.

4.3 High-Level Overview

Instead of running the expensive context-sensitive analysis, we could predict it using machine learning. In this work we use a GNN-based model to predict context-sensitive information of the previously described context-free language reachability formulation of the points-to analysis problem.

4.3.1 Expected Benefits

Whole-program context-sensitive points-to analysis can take hours or days to run, even on relatively small benchmarks, such as the DaCapo benchmarks. However, whole-graph GNN-inference can finish in minutes on graphs of the size of these benchmarks, orders of magnitude faster. This performance improvement can speed up certain downstream tools to run in near-realtime. This is of particular importance in tools that are embedded in tools such as IDEs, such as bug detection tools, which need to provide feedback to developers in seconds.

4.3.2 Dataset

We collect pointer assignment graphs of the DaCapo Benchmarks (version 9.12-bach), using the Soot Framework. These benchmarks are all primarily Java code. We perform both context-insensitive and context-sensitive points-to analyses over these graphs, using the built-in demand-driven analysis based on Sridharan et al. [149], to find the points-to set of every variable in each program.

We collect each graph separately. This means that to collect the Fop graph, we analyze only the code used for the Fop graph, and the dependencies used. When we collect multi-benchmark graphs, such as Fop+Batik, we analyze the entire graph at once. This may include points-to queries across benchmarks, i.e, the source node is in Fop, and the destination node is in Batik.

For each pointer assignment graph, we collect a set of nodes, numbered 1 through n, where n is the number of nodes in the graph. The nodes in the graph have different types. In this work we keep track of *allocation nodes* and *variable nodes. Allocation* nodes represent so-called *abstract locations.* Each abstract location represents a particular *new* invocation in the program. We label every other kind of node to be a *variable* node.

For each pointer assignment graph, we also collect a set of edges, which each connect two nodes, and reflexive edges are allowed. While the whole point of the graph is to capture assignments, there are multiple different types of assignments, and hence, each assignment edge has a type. For example, there can be regular assignments (`x = y`), field assignments (`x.f = y`), field reads (`y = x.f`), function calls (`foo(x)`), function returns (`y = foo()`), and so on. In total, we capture 9 different types of assignment edges.

Certain edges also have a particular label, to capture sensitivity information. For field reads, and field writes, we capture the field name. For example, for an assignment `x.f = y`, we would have an edge from x to y, of type field assignment, with label $[_f$. A corresponding assignment `z = x.f` would have an edge from x to z, of type field read, with label $]_f$. Similar to section 4.2.1, this enables field-sensitivity. For function calls and returns, we capture a callsite number. For example, given a function call `y = foo(x)`, we would have an edge from x to the formal parameter of the foo function, of type function call, with edge label $(_1$ (if the callsite is numbered 1). If there are multiple callsites which call `foo`, we represent it by creating multiple edges: for each argument variable, we draw an edge from the a to p (where p is the formal parameter of `foo`). We would also have an edge from the return variable to y, of type function return, with edge label $)_y$ (if the call site is numbered 1). If there are multiple returns within a function, an edge is created from each return variable in the function to each callsite that the function is called from. The callsites are numbered starting from 1 and are unique across a single analysis, *i.e.*, each callsite in the program has a particular callsite ID, but these IDs carry no meaning across pointer assignment graphs.

We then run the built-in demand-driven points-to analysis and find the points-to set of every variable in the program. For every variable, we create an edge to each context-*insensitively* reachable variable. For example, on figure 4.1, there would be *insensitive* edges between variables b, c, p, x, and y. For edges in the training dataset, we run the slow analysis to find, for each of these edges, if it represents a context-sensitive edge (*i.e.*, a *feasible* path) or not.

We also collect all of the methods encountered by the analysis. For each method, we collect its corresponding Jimple code and method name. Jimple is an intermediate representation of Java designed to be easier to optimize than Java bytecode, and can be constructed by Soot given only the Java bytecode. We collect Jimple rather than source Java code as it is more readily available during the Soot analysis, but future work could instead collect the Java code. It should be noted that not every method has associated Java code. For example, third-party libraries are often shipped as pre-compiled JARs, which contain bytecode and only optionally contain source code. Furthermore, unless the pre-compiled JAR contains all source code of all its dependencies, it can sometimes be a difficult task to find the right versions of all the source code in a project. This is a similar problem as found in the "stack trace contextualization" described in chapter 3.

For each method, we also collect the which nodes were initially defined in which methods. That is, if a variable is initially declared in a particular method, we associate it with that method. We also collect the set of abstract location nodes for each method.

We augment the pointer assignment graph with these methods. We create a method node for each method that we collected. We then draw edges from each method node to the nodes that were definied inside of that method. For example, if a variable v is initially defined in a function foo, then there is a method node, which represents `foo`, and an edge from the `foo` node to the v node. When available, we label this edge with a number, representing the source-code-level line number within the file that the node was declared on. It should be noted that line number information is not always available, in which case we assign an edge label of 0 to indicate that the data was missing (as line numbers start with 1).

4.4 Modelling

The goal of our GNN-based model is to predict if a particular *insensitive* edge represents a a context-sensitive solution or not. Specifically, as input we are given a set of $< x_i, y_i >$, where x_i is a particular context-insensitive edge with a particular source node $src_x i$ and particular destination node $dst_x i$, and y_i is a binary label representing if the edge is context-sensitively *feasible* or not.

Our model contains two modules, a pre-trained BERT [50] model which is used to embed method nodes, and a graph neural network model which learns the relationships between variable nodes to create embeddings of nodes and edges and classify insensitive edges with a binary label.

Since we use a standard GNN architecture, we keep the description relatively high-level and discuss how we map our particular problem to the GNN. For finer details about how GNNs work, we refer the reader to citations in section 4.3. For all nodes except method nodes, we create an initial embedding based on its ID, which is unique. To produce an embedding of a method node, we first gather all the Jimple tokens (including the method and parameter declaration) from that method. We then use an off the shelf pre-trained BertTokenizer and BertModel (from Huggingface Transformers [162]) to tokenize the Jimple statements and embed them. After passing the entire Jimple token sequence through the BERT model, we take the last hidden state. This last hidden state is projected to our node embedding size (if needed), and is used as the initial embedding of each method node. Method nodes are connected to non-method nodes with a special `contains-def` edge, which represents that a particular node was defined in that method.

We then apply a learnable embedding matrix to each node and each edge, based on its type. Here we have two node types (variable or abstract location) and 11 edge types (9 assignment types, insensitive edge, contains-def edge). We then use a GCN implemented with Deep Graph Library [159] to update the node and edge embeddings. To produce a prediction, for each insensitive edge in the graph, we concatenate the source and destination node embeddings, and use a linear layer to produce the final logits. Afterwards, we use a softmax to find the highest probability label, and compare this to the ground truth. We train the GNN by minimizing the binary cross-entropy of the predicted labels and the ground truth labels.

We learn parameters for the GNN itsel, the per-type node and edge embeddings, as well as the initial embedding matrix used for non-method nodes. It should be noted that we do not update (*i.e.*, fine-tune) the BERT weights for embedding of the method nodes.

4.5 Implementation

PAG Collection and Points-to Analysis We use Soot to collect pointer assignment graphs of the DaCapo benchmarks. We use `DemandCSPointsTo` to compute the points-to information for all variable pairs. We increase the traversal limitation for this PTA to

75000000, and filter out edges in which the analysis times out (*i.e.*, exceeds this limit) from our dataset. In total we wrote approximately 1000 Java LoC to produce this dataset. We wrote approximately 500 Bash and Python LoC to convert this raw data into a format suitable for our DGL-based model. Our graphs contain on the order of hundreds of thousands of nodes and edges. For example, the Fop graph we collect has 115643 nodes and 537227 edges.

Graph Neural Network The method node embedding module is the “bert-base-uncased” model found on Huggingface. We embed the Jimple tokens ahead-of-time, meaning that by the time the nodes are loaded by the GNN, each node is already associated with an embedding. However, based on the method nodes’ embedding size (a hyperparameter), we project the initial BERT-based embedding to the appropriate size. The GNN model is implemented in roughly 2000 Python LoC.

Model training We perform a hyperparameter sweep and vary the learning rate, node embedding size, edge embedding size, depth, and layer type of the GNN model. We then train on the specified training dataset (which differs for different experiments), and select the checkpoint, among all hyperparameter configurations, which has the highest accuracy on the validation dataset. We train our models on Google Cloud Platform, using 10 VM instances with configuration `n1-standard-4` with a single NVIDIA K80 GPU each. Our training takes roughly 3 GPU-weeks, and single benchmark inference time takes roughly 1-2 minutes. We then report the performance of this model checkpoint on the specified test dataset in section 4.6

4.6 Evaluation

This section evaluates the effectiveness of this GNN model and its ability to predict context-sensitive information. We seek to answer the 3 following questions:

RQ1. How effective is our model at predicting on a single benchmark? (Section 4.6.1)

RQ2. To what extent can our model generalize to unseen benchmarks? (Section 4.6.2)

RQ3. To what extent do method nodes improve our generalization? (Section 4.6.3)

4.6.1 Prediction on a single benchmark

Graph	MPCA
Batik	83.46%
Eclipse	93.74%
Fop	90.77%

Table 4.1: GNN Mean-per-class-accuracy (MPCA) performance on hold-out edges

Table 4.1 shows the mean-per-class-accuracy of our GNN model on 3 different DaCapo benchmarks. In this experiment, we trained on each different benchmark's pointer assignment graph separately. We train with 80% of the insensitive edges, and reserve 10% for validation and another 10% for testing. The model is trained until the accuracy on the validation set no longer improves. We select the checkpoint, among all hyperparameter configurations, which has the highest accuracy on the validation set. We report the MPCA of the test split in the table. We use mean-per-class-accuracy because the dataset is heavily skewed: there are far more edges labeled "infeasible" than there are "feasible" edges.

These MPCA values are quite high, and are a promising first step, but they are not directly useful. The problem is that this requires training over the same graph that will be used for inference (in this case, evaluating the test split). This means, that we would need to spend GPU-weeks training every graph before it could be used for downstream tasks, if we wanted it to perform this well. Furthermore, these GPU-weeks would take many more compute resources and/or time than just running the slow analysis, so this is not a good tradeoff. Instead, similar to chapter 3, we should look at how well the model can generalize.

Training Graph	Test Graph	MPCA
fop+batik	eclipse	47.25%
batik+eclipse	fop	43.303%
fop+eclipse	batik	50.93%

Table 4.2: GNN Mean-per-class-accuracy (MPCA) performance on unseen benchmarks

4.6.2 Generalization across benchmarks

Table 4.2 reports our model's MPCA when predicting across benchmarks. We again use mean-per-class-accuracy as the target metric because of label skew, as a better way to measure the performance of the model. Because we treat this as a binary classification task, simply predicting the more common label would result in a very high raw accuracy. For example, predicting "infeasible" on every edge would result in 88% accuracy on our fop dataset.

In this experiment, we train on the combined graph of two benchmarks (in their entirety), and test on a third benchmark (in its entirety). We use a portion of the combined graph as a hold-out set, used for validation and checkpoint selection. We provide the same hyperparameter space as the previous experiment, and provide an entire extra benchmark for extra training data, but we see that the MPCA is much lower.

This MPCA result is not strong. While a downstream tool may not need formal soundness guarantees, it is still not great when the context-sensitivity is correct only 50% of the time. This result helps to motivate the need for improving the model.

One potential problem is our representation of variable nodes. Each variable node is initially embedded with a unique ID, but this ID has no meaning across benchmarks as variables are numbered differently. Furthermore, the numbers may not even be stable across analysis runs of the same benchmark. Because this representation doesn't tell the model much about the graph node, it may be useful to augment the representation. The graph structure still provides rich information, but similar to our work in chapter 3, we can try to improve the model with *multi-modality*, in this case by adding code.

4.6.3 Addition of method nodes

Graph Nodes	MPCA
Variable / Allocation	37.83%
With method nodes	44%

Table 4.3: Comparison of performance with and without Method nodes

As described in sections 4.3 and 4.4, we augment the graph with method nodes. In this experiment, we compare the performance of our GNN model with and without these method nodes. The results are shown in table 4.3. Here, we train the specified model on the Fop benchmark, and test its accuracy on the Avrora benchmark, and report the MPCA. We again provide the same hyperparameter space to these models as in previous sections.

We can see that the model with the extra method nodes performs better, by 6 percentage points. While this is a sizeable improvement (because we are considering MPCA, not pure accuracy), this still leaves room for improvement.

4.7 Discussion

While the multi-modal code embeddings seem to be a promising direction, the accuracy needs to be improved before being practically useful. We see a number of ways to improve this work.

4.7.1 Potential Improvements

Improvement 1: Model size. The model that we train is not very large by modern standards. Other work has shown that GNN depth could be the most important hyperparameter to adjust for GNNs solving similar problems [65]. This is because depth corresponds to how far away a GNN can "look" from each node. For example, to make an accurate prediction about a node relationship 5 hops away, we need a GNN with 5 layers (or to use other tricks, such as skip edges). For reference, the previously cited Snowcat paper trains GNNs of depth of nearly 50, but in our experiments, the deepest GNN we train is 5 layers. But in our datasets, we observe paths of length 30 or more, most of the time. We use smaller GNNs mostly due to resource constraints – our K80 machines do not have as much memory as beefy machines with 8+ powerful GPUs, such as A100s.

Improvement 2: More sophisticated Code Embedding. We use a Transformer that is pre-trained on text, which is meant to be used for NLP problems. Because we use Jimple tokens, it is very possible that the pre-training did not help. We may find that the code embeddings work better when the Transformer is trained from scratch, or at least fine-tuned to perform better with Jimple. We also could fine-tune the embedding with the rest of the GNN, rather than do it ahead of time, however this would put additional pressure on our compute and memory resources and may not be worth it.

We could also change the code that we use to embed the methods. While we currently use Jimple, that is mostly out of convenience, as Soot directly provided this. But patterns in higher level source code such as Java may be more useful, and there has been more literature about training models on them.

Improvement 3: Threshold tuning. After we improve the performance of the model in the previous two ways, a very important addition is to add *threshold tuning*. A classifier is usually trained to predict the probability of a particular class (here, if an insensitive points-to relationship is also context-sensitive). A tunable threshold determines when we will report this result. If a model predicts below this threshold, no report is shown, but if it predicts with confidence above the threshold, we can report it to the user. This allows us to report fewer (but higher-quality) results, and allow us to make tradeoffs about the expected precision and recall of the predictions.

This threshold idea is shown in a number of works [65, 156], and is incredibly important for our use case. An example precision/recall graph from [156] is shown in figure 4.3. Based on the need of the downstream tool, we can choose a point on such a graph, to optimize for precision or recall. Developers often care more about receiving less warnings, even if their

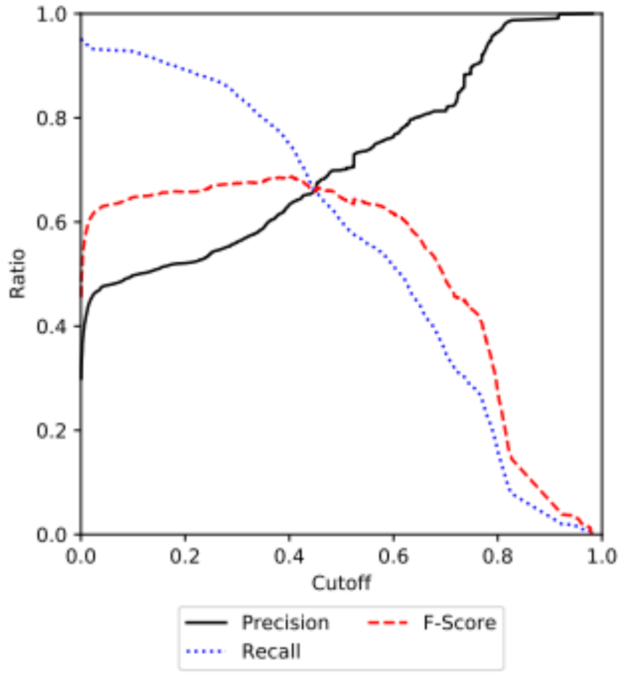


Figure 4.3: An Example Precision/Recall Curve from [156]

bug detection tools miss some bugs. Such a threshold could help us tune precision based on requirements of the downstream task (such as a bug detector). If a prediction is below the threshold, we also have the option of re-running the slow analysis, if a downstream task really did need the precise answer.

Future evaluation could then show how different threshold choices would affect different downstream client tools or analyses.

4.7.2 Downstream Clients

It is important to point out that points-to analysis is a fundamental building block for many program analysis tools. In addition to the clients mentioned in section 4.1, points-to analysis has also been used for program verifiers [59], symbolic execution [89], de-obfuscation [137], null-pointer detection [16], code completion [136], program slicing [163], and many more.

After we have improved our machine learning model, as described previously, we can improve this work by evaluating the performance of such downstream tools when the pointer analysis is replaced by our predictions. Many of these clients do not require 100% soundness of the underlying pointer analysis, and would benefit from the speedups that our technique would provide.

4.8 Conclusion

In conclusion, we present a way to predict CFL-reachability-based context-sensitive points-to analysis using a GNN. We notice that the straight-forward implementation of the GNN does not generalize well across benchmarks, and propose a way to improve generality, using code embeddings for the methods in the program. In the future, we plan to improve the work by improving the hyperparameter search, and adding threshold tuning.

CHAPTER 5

Conclusion and Future Directions

Machine learning models are constantly evolving to solve larger and more complex problems. This is especially noticeable now, with recent advancements and excitement relating to Large Language Models. There are countless new techniques and models that appear and create a new state-of-the-art every day. However, in systems research, adoption of these large models is often far behind development of these techniques. Often times, this is due to 1) the ever increasing resource requirements for these models, and 2) the cost of handling mispredictions.

In time-sensitive operations in a system (such as making a scheduling decision, or making a memory placement decision) "small" models such as an LSTM are often already too expensive to use at runtime. This is because the milliseconds needed to run an inference are more expensive than the potential time savings from a smarter placement decision. Furthermore, any mispredictions would undermine any performance improvements that would be seen.

This dissertation takes a step towards fixing these problems. In chapter 2, we show that if we could accurately capture the behavior of most objects, we can find performance optimizations in these systems, even if we are sometimes wrong. This idea appears in the rest of the dissertation as well – if we design a system to be "mostly" right (with fallbacks when it is not), we can find more opportunities. In chapter 3, we show a framework for making system optimization predictions using code embedding models. While we present a particular instantiation of this framework, newer and more powerful models could be swapped in. In chapter 4, we show a way to make predictions about a system, other than performance optimizations, in an offline context. Here, we show an example of using these models offline, and another way to make progress, in spite of mispredictions.

Moving forward, as models continue to become larger and more complex, systems should be built to accept and *enable* these ever-improving models to be used within them. We discuss potential future directions related to this idea.

Offline Predictions for Online Optimizations While chapter 3 touches on this idea, it has potential to be expanded. As models grow larger and more resource-intensive, it becomes harder and harder to run them online, *i.e.*, on the critical path during execution. Even simple linear models or lightweight neural networks are troublesome to use online, and larger models have too much runtime overhead. However, modern systems face highly diverse workloads and are thus in need of constant adaption to handle different (seen and unseen) situations. In order to do this, we need the large and powerful models of today – but these are too slow to run online.

One way to solve this dilemma is by moving the prediction to be fully offline. However this poses some key challenges, namely: how can we make such predictions offline, and how can the predictions be used online? As we saw in chapter 3 and related works, source code enables predictions about online behavior, but is accessible offline. This enables offline predictions – so long as the predictions are accurate enough. However, the large explosion of Large Language Models offer a great way to expand this research direction. While much of the publicity is around natural language problems, the same techniques can be applied to source code understanding, and achieve even better code predictions. Second, one way to use these predictions is in the form of annotations. We could, for example, generate annotations based on offline predictions about which objects are used together. Then, online, the system could allocate them together. We could take this one step further and construct a *knowledge graph* about our source code predictions, and embed this entire graph *into* a program, in such a way that the information could be quickly accessed. For example, if we take this code path, it corresponds to this node or edge in the knowledge graph.

One other practical challenge in this area is collecting the data needed to train the model.

In our ISMM work, we experienced first-hand how difficult it is to instrument a production JVM and collect the data. Some frameworks or APIs should be developed to more easily enable collection of this training data. Projects such as MMTk [21] could be a great start for trying this for machine learning garbage collector optimizations.

AI Governance and Compliance Considerations While it has always been a concern, recent attention on Large Language Models helps to show that we should also be cognizant of how these models are being trained and how they are being used. For example, when creating datasets over our Big Data systems workloads, are we carefully considering what we sensitive customer data we are working with and training the model on? Importantly, are our large machine learning pipelines all compliant with the regulations in our municipality or state? It has become increasingly important, for both individuals and companies, to be mindful of these problems in this space. As we continue to improve our machine learning models, we should also develop new methods and tools for governance and compliance.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[2] Active/Inactive Users. Active/inactive users. https://stackoverflow.com/questions/49442420, 2015.

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

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

[5] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. ShuffleWatcher: Shuffle-aware scheduling in multi-tenant MapReduce clusters. In *USENIX ATC*, pages 1–13. USENIX Association, June 2014.

[6] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

[7] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to repre-

sent programs with graphs. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=BJOFETxR-.

[8] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code, 2018. URL https://arxiv.org/abs/1803.09473.

[9] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code, 2019.

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

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

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

[13] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, jun 2014. ISSN 0362-1340. doi: 10.1145/2666356.2594299. URL https://doi.org/10.1145/2666356.2594299.

[14] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: Membership, growth, and evolution. In *KDD*, pages 44–54, 2006.

[15] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs, 2017.

[16] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. Nullaway: Practical type-based null safety for java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 740–750, New York, NY, USA, 2019. Association for

Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3338919. URL https://doi.org/10.1145/3338906.3338919.

[17] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, page 187–196, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915984. doi: 10.1145/155090.155108. URL https://doi.org/10.1145/155090.155108.

[18] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press. doi: http://doi.acm.org/10.1145/1167473.1167488.

[19] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[20] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinely, and J. Eliot B. Moss. Pretenuring for java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, page 342–352, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581133359. doi: 10.1145/504282.504307. URL https://doi.org/10.1145/504282.504307.

[21] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in java with mmtk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, page 137–146, USA, 2004. IEEE Computer Society. ISBN 0769521630.

[22] Stephen M. Blackburn, Matthew Hertz, Kathryn S. Mckinley, J. Eliot B. Moss, and Ting Yang. Profile-based pretenuring. *ACM Trans. Program. Lang. Syst.*, 29(1):2–es, jan 2007. ISSN 0164-0925. doi: 10.1145/1180475.1180477. URL https://doi.org/10.1145/1180475.1180477.

[23] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Selective control-flow abstraction via jumping. *SIGPLAN Not.*, 50(10):163–182, oct 2015. ISSN 0362-1340. doi: 10.1145/2858965.2814293. URL https://doi.org/10.1145/2858965.2814293.

[24] B. Blanchet. Escape analysis for object-oriented languages. Applications to Java. In *OOPSLA*, pages 20–34, 1999.

[25] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. Enriching word vectors with subword information. *CoRR*, abs/1607.04606, 2016. URL http://arxiv.org/abs/1607.04606.

[26] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *WWW*, pages 595–601, 2004.

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

[28] Xavier Bresson and Thomas Laurent. Residual gated graph convnets, 2018.

[29] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga, and Paulo Ferreira. Runtime object lifetime profiler for latency sensitive big data applications. In *Proceedings of*

the Fourteenth EuroSys Conference 2019, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362818. doi: 10.1145/3302424.3303988. URL https://doi.org/10.1145/3302424.3303988.

[30] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J. Carey. A bloat-aware design for big data applications. In *ISMM*, pages 119–130, 2013.

[31] Brad Calder, Peter Feller, and Alan Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1, 1999.

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

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

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

[35] Binghong Chen, Daniel Tarlow, Kevin Swersky, Martin Maas, Pablo Heiber, Ashish Naik, Milad Hashemi, and Parthasarathy Ranganathan. Learning to improve code efficiency, 2022.

[36] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *CGO 2016 Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 12–23, New York, NY, USA, 2016.

[37] Zimin Chen, Steve Kommrusch, and Martin Monperrus. Neural transfer learning for repairing security vulnerabilities in c code, 2021.

[38] Sigmund Cherem and Radu Rugina. A practical escape and effect analysis for building lightweight method summaries. In *Proceedings of the 16th International Conference on Compiler Construction*, CC'07, page 172–186, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 9783540712282.

[39] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, page 199–209, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134630. doi: 10.1145/512529.512554. URL https://doi.org/10.1145/512529.512554.

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

[41] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, nov 2003. ISSN 0164-0925. doi: 10.1145/945885.945892. URL https://doi.org/10.1145/945885.945892.

[42] François Chollet et al. Keras. https://keras.io, 2015.

[43] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. Memento mori: Dynamic allocation-site-based optimizations. *SIGPLAN Not.*, 50(11):105–117, jun 2015. ISSN 0362-1340. doi: 10.1145/2887746.2754181. URL https://doi.org/10.1145/2887746.2754181.

[44] David Cohn and Satinder Singh. Predicting lifetimes in dynamically allocated memory. In M.C. Mozer, M. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9. MIT Press, 1996. URL https://proceedings.neurips.cc/paper/1996/file/a9078e8653368c9c291ae2f8b74012e7-Paper.pdf.

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

[46] Count Number of Posts. Count number of posts. https://stackoverflow.com/questions/39030644, 2015.

[47] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC*, pages 37–48, 2014.

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

[49] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural program learning under noisy I/O. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 990–998. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/devlin17a.html.

[50] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

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

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

[53] David Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Gómez-Bombarelli, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P. Adams. Convolutional networks on graphs for learning molecular fingerprints, 2015.

[54] Error in Computing Frequencies. Error in computing frequencies. http://stackoverflow.com/questions/23042829, 2015.

[55] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *OSDI*, pages 799–815, 2018.

[56] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *SOSP*, pages 394–409, 2015.

[57] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. Structured neural summarization, 2021.

[58] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Making state explicit for imperative big data processing. In *USENIX ATC*, pages 49–60, 2014.

[59] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), may 2008. ISSN 1049-331X. doi: 10.1145/1348250.1348255. URL https://doi.org/10.1145/1348250.1348255.

[60] Andreas Gal, Christian W. Probst, and Michael Franz. Hotpathvm: An effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, page 144–153, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933328. doi: 10.1145/1134760.1134780. URL https://doi.org/10.1145/1134760.1134780.

[61] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for c programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 459–470. IEEE Press, 2015. ISBN 9781479919345.

[62] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *CC*, pages 82–93, 2000.

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

[64] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G. Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *HotOS*, 2015.

[65] Sishuai Gong, Dinglan Peng, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. Snowcat: Efficient kernel concurrency testing using a learned coverage predictor. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 35–51, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613148. URL https://doi.org/10.1145/3600006.3613148.

[66] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

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

[68] David Grove and Craig Chambers. A framework for call graph construction algorithms.

ACM Trans. Program. Lang. Syst., 23(6):685–746, nov 2001. ISSN 0164-0925. doi: 10.1145/506315.506316. URL https://doi.org/10.1145/506315.506316.

[69] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. *SIGPLAN Not.*, 30(10):108–123, October 1995. ISSN 0362-1340. doi: 10.1145/217839.217848. URL https://doi.org/10.1145/217839.217848.

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

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

[72] Ameer Haj-Ali, Qijing Jenny Huang, John Xiang, William Moses, Krste Asanovic, John Wawrzynek, and Ion Stoica. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. *Proceedings of Machine Learning and Systems*, 2:70–81, 2020.

[73] Timothy L. Harris. Dynamic adaptive pre-tenuring. In *Proceedings of the 2nd International Symposium on Memory Management*, ISMM '00, page 127–136, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132638. doi: 10.1145/362422.362476. URL https://doi.org/10.1145/362422.362476.

[74] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 152–162, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236051. URL https://doi.org/10.1145/3236024.3236051.

[75] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=B1lnbRNtwr.

[76] Matthew Hertz, Stephen M Blackburn, J Eliot B Moss, Kathryn S. McKinley, and Darko Stefanović. Error-free garbage collection traces: How to cheat and not get caught. *SIGMETRICS Perform. Eval. Rev.*, 30(1):140–151, jun 2002. ISSN 0163-5999. doi: 10.1145/511399.511352. URL https://doi.org/10.1145/511399.511352.

[77] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, page 54–61, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581134134. doi: 10.1145/379605.379665. URL https://doi.org/10.1145/379605.379665.

[78] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[79] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 257–273. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/hunter.

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

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

[82] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3133924. URL https://doi.org/10.1145/3133924.

[83] Teresa Johnson, Mehdi Amini, and Xinliang David Li, editors. *ThinLTO: Scalable and incremental LTO*, 2017.

[84] Join operation with MapReduce. Join operation with mapreduce. https://stackoverflow.com/questions/4053857, 2012.

[85] Join to Filer Spams. Join to filer spams. https://stackoverflow.com/questions/29622750, 2015.

[86] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 12-18 July 2020*, Proceedings of Machine Learning Research. PMLR, 2020.

[87] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code, 2020. URL https://arxiv.org/abs/2001.00059.

[88] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 158–169, 2015. doi: 10.1145/2749469.2750392.

[89] Timotej Kapus and Cristian Cadar. A segmented memory model for symbolic execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 774–784, New York, NY, USA, 2019. Association for

Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3338936. URL https://doi.org/10.1145/3338906.3338936.

[90] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

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

[92] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

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

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

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

[96] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, pages 278–289, 2007.

[97] Hugh Leather and Chris Cummins. Machine learning in compilers: Past, present and future. In *2020 Forum for Specification and Design Languages (FDL)*, pages 1–8, 2020. doi: 10.1109/FDL50818.2020.9232934.

[98] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. Ysmart: Yet another SQL-to-MapReduce translator. In *ICDCS*, pages 25–36, 2011.

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

[100] Ondrej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: Is it worth it? In *CC*, pages 47–64, 2006.

[101] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In *CC*, pages 153–169, 2003.

[102] Bolun Li, Pengfei Su, Milind Chabbi, Shuyin Jiao, and Xu Liu. Djxperf: Identifying memory inefficiencies via object-centric profiling for java. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2023, page 81–94, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701016. doi: 10.1145/3579990.3580010. URL https://doi.org/10.1145/3579990.3580010.

[103] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. doi: 10.1145/3276511. URL https://doi.org/10.1145/3276511.

[104] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in Data-Parallel systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 383–400, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/lion.

[105] Qi Liu, Maximilian Nickel, and Douwe Kiela. Hyperbolic graph neural networks, 2019.

[106] V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. *SIGSOFT Softw. Eng. Notes*, 28 (5):317–326, sep 2003. ISSN 0163-5948. doi: 10.1145/949952.940114. URL https://doi.org/10.1145/949952.940114.

[107] Jingbo Lu and Jingling Xue. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. doi: 10.1145/3360574. URL https://doi.org/10.1145/3360574.

[108] Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. Lifetime-based memory management for distributed data processing systems. *Proc. VLDB Endow.*, 9(12):936–947, 2016.

[109] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930566. doi: 10.1145/1065010.1065034. URL https://doi.org/10.1145/1065010.1065034.

[110] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. NeuGraph: Parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 443–458, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/ma.

[111] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. Trash Day: Coordinating garbage collection in distributed systems. In *HotOS*, 2015.

[112] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *ASPLOS*, pages 457–471, 2016.

[113] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based memory allocation for c++

server workloads. In *25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[114] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857, 2020.

[115] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. *SIGSOFT Softw. Eng. Notes*, 27(4):1–11, jul 2002. ISSN 0163-5948. doi: 10.1145/566171.566174. URL https://doi.org/10.1145/566171.566174.

[116] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A timely dataflow system. In *SOSP*, pages 439–455, 2013.

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

[118] Todd Mytkowicz, Devin Coughlin, and Amer Diwan. Inferred call path profiling. *SIGPLAN Not.*, 44(10):175–190, oct 2009. ISSN 0362-1340. doi: 10.1145/1639949.1640102. URL https://doi.org/10.1145/1639949.1640102.

[119] Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu. Gerenuk: Thin computation over big native data using speculative program transformation. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 538–553, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359643. URL https://doi.org/10.1145/3341301.3359643.

[120] Christian Navasca, Martin Maas, Petros Maniatis, Hyeontaek Lim, and Guoqing Harry Xu. Predicting dynamic properties of heap allocations using neural networks trained on static code: An intellectual abstract. In *Proceedings of the 2023 ACM SIGPLAN*

International Symposium on Memory Management, ISMM 2023, page 43–57, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701795. doi: 10.1145/3591195.3595275. URL https://doi.org/10.1145/3591195.3595275.

[121] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, pages 291–305, 2015.

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

[123] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *OSDI*, pages 349–365, 2016.

[124] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 349–365, USA, 2016. USENIX Association. ISBN 9781931971331.

[125] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. Skyway: Connecting managed heaps in distributed big data systems. In *ASPLOS*, pages 56–69, 2018.

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

[127] Deok-Jae Oh, Yaebin Moon, Eojin Lee, Tae Jun Ham, Yongjun Park, Jae W. Lee, and Jung Ho Ahn. Maphea: A lightweight memory hierarchy-aware profile-guided

heap allocation framework. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2021, page 24–36, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384728. doi: 10.1145/3461648.3463844. URL https://doi.org/10.1145/3461648.3463844.

[128] Nigini Oliveira, Michael Muller, Nazareno Andrade, and Katharina Reinecke. The exchange in stackexchange: Divergences between stackoverflow and its culturally diverse participants. *Proc. ACM Hum.-Comput. Interact.*, 2(CSCW):130:1–130:22, November 2018. ISSN 2573-0142.

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

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

[131] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis, 2016. URL https://arxiv.org/abs/1611.01855.

[132] Pardis Pashakhanloo, Aaditya Naik, Hanjun Dai, Petros Maniatis, and Mayur Naik. Learning to walk over relational graphs of source code. In *Deep Learning for Code (DL4C) Workshop at the International Conference on Learning Representations (ICLR)*, 2022. URL https://openreview.net/forum?id=SubGAoOWJWc.

[133] Pardis Pashakhanloo, Aaditya Naik, Yuepeng Wang, Hanjun Dai, Petros Maniatis, and Mayur Naik. Codetrek: Flexible modeling of code using an extensible relational

representation. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=WQc075jmBmf.

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

[135] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection, 2018.

[136] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. *SIGPLAN Not.*, 49(6):419–428, jun 2014. ISSN 0362-1340. doi: 10.1145/2666356.2594321. URL https://doi.org/10.1145/2666356.2594321.

[137] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". *SIGPLAN Not.*, 50(1):111–124, jan 2015. ISSN 0362-1340. doi: 10.1145/2775051.2677009. URL https://doi.org/10.1145/2775051.2677009.

[138] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from 'big code'. *Commun. ACM*, 62(3):99–107, February 2019. ISSN 0001-0782. doi: 10.1145/3306204. URL https://doi.org/10.1145/3306204.

[139] Nadav Rotem and Chris Cummins. Profile guided optimization without profiles: A machine learning approach, 2021. URL https://arxiv.org/abs/2112.14679.

[140] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In Reinhard Wilhelm, editor, *Compiler Construction*, pages 20–36, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45306-2.

[141] Douglas Santry and Kaladhar Voruganti. Violet: A storage stack for IOPS/capacity bifurcated storage environments. In *USENIX ATC*, pages 13–24, 2014.

[142] Joe Savage and Timothy M. Jones. Halo: Post-link heap-layout optimisation. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 94–106, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370479. doi: 10.1145/3368826.3377914. URL https://doi.org/10.1145/3368826.3377914.

[143] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.

[144] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting prolific types for memory management and optimizations. In *POPL*, pages 295–306, 2002.

[145] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69, 2015. ISSN 2325-1107. doi: 10.1561/2500000014. URL http://dx.doi.org/10.1561/2500000014.

[146] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 17–30, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304900. doi: 10.1145/1926385.1926390. URL https://doi.org/10.1145/1926385.1926390.

[147] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. *SIGPLAN Not.*, 49(6):485–495, jun 2014. ISSN 0362-1340. doi: 10.1145/2666356.2594320. URL https://doi.org/10.1145/2666356.2594320.

[148] Soot Framework. Soot framework. http://www.sable.mcgill.ca/soot, 2006.

[149] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *PLDI*, pages 387–400, 2006.

[150] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 59–76, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930310. doi: 10.1145/1094811.1094817. URL https://doi.org/10.1145/1094811.1094817.

[151] The Performance Comparison between In-Mapper Combiner and Regular Combiner. The performance comparison between in-mapper combiner and regular combiner. http://stackoverflow.com/questions/10925840, 2015.

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

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

[154] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. Mlgo: a machine learning guided compiler optimizations framework, 2021. URL https://arxiv.org/abs/2101.04808.

[155] User Activity. User activity. https://stackoverflow.com/questions/33411920, 2015.

[156] Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. Striking a balance: Pruning false-positives from static call graphs. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2043–2055, New

York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510166. URL https://doi.org/10.1145/3510003.3510166.

[157] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair, 2019.

[158] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416, 2018. URL http://arxiv.org/abs/1803.07416.

[159] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks, 2020.

[160] Yijian Wang and David Kaeli. Profile-guided i/o partitioning. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, page 252–260, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137338. doi: 10.1145/782814.782850. URL https://doi.org/10.1145/782814.782850.

[161] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA*, pages 187–206, 1999.

[162] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2020.

[163] Jingyue Wu, Yang Tang, Gang Hu, Heming Cui, and Junfeng Yang. Sound and precise analysis of parallel programs through schedule specialization. *SIGPLAN Not.*, 47(6):

205–216, jun 2012. ISSN 0362-1340. doi: 10.1145/2345156.2254090. URL https://doi.org/10.1145/2345156.2254090.

[164] Shu Wu, Yuyuan Tang, Yanqiao Zhu, Liang Wang, Xing Xie, and Tieniu Tan. Session-based recommendation with graph neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):346–353, Jul. 2019. doi: 10.1609/aaai.v33i01.3301346. URL https://ojs.aaai.org/index.php/AAAI/article/view/3804.

[165] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, January 2021. ISSN 2162-2388. doi: 10.1109/tnnls.2020.2978386. URL http://dx.doi.org/10.1109/TNNLS.2020.2978386.

[166] Guoqing Xu. Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &amp; Applications*, OOPSLA '13, page 111–130, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323741. doi: 10.1145/2509136.2509512. URL https://doi.org/10.1145/2509136.2509512.

[167] Guoqing Xu and Atanas Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, page 225–236, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580500. doi: 10.1145/1390630.1390658. URL https://doi.org/10.1145/1390630.1390658.

[168] Guoqing Xu, Dacong Yan, and Atanas Rountev. Static detection of loop-invariant data structures. In *ECOOP*, pages 738–763, 2012.

[169] Guoqing Harry Xu, Margus Veanes, Michael Barnett, Madan Musuvathi, Todd Mytkowicz, Ben Zorn, Huan He, and Haibo Lin. Niijima: Sound and automated

computation consolidation for efficient multilingual data-parallel pipelines. In *SOSP*, 2019.

[170] Dacong Yan, Guoqing Xu, and Atanas Rountev. Rethinking soot for summary-based whole-program analysis. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, page 9–14, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450314909. doi: 10.1145/2259051.2259053. URL https://doi.org/10.1145/2259051.2259053.

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

[172] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '18. ACM, July 2018. doi: 10.1145/3219819.3219890. URL http://dx.doi.org/10.1145/3219819.3219890.

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

[174] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J Kim. Graph transformer networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/9d63484abb477c97640154d40595a3bb-Paper.pdf.

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

[176] Giulio Zhou and Martin Maas. Learning on distributed traces for data center storage systems. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 350–364, 2021. URL https://proceedings.mlsys.org/paper/2021/file/82161242827b703e6acf9c726942a1e4-Paper.pdf.

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