TÉCNICO
LISBOA

Efficient Data Structures with GraalVM Native Image

Roman Babynyuk

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Rodrigo Fraga Barcelos Paulus Bruno

June 2023

Acknowledgments

First and foremost, my deepest gratitude goes to Rodrigo Bruno, my supervisor, whose patience in providing guidance, insights and teaching was quintessential to this thesis' completion, along with Professor Luís Veiga, whose pertinent questions and detailed evaluation of the project for this thesis were invaluable to fully define our scope, goals and methods.

I'd also like to thank my partner, Mariana, for the endless support and care I've received throughout the process, having made the process far easier on the mind.

Further acknowledgements go to my friends Patrícia Piedade and João Marques, for their crucial help in proofreading and pushing me onwards, along with my colleague Rafael Soares, whose rousing speech helped me and many of my colleagues push ahead in the final stretch.

Finally, to the Native Image and GraalVM teams and the developers in the GraalVM Slack, my deepest regards for the myriad of help given to me throughout the process. The insights provided were invaluable to development of both ideas and code.

Abstract

Data structures often constitute a large proportion of a program's total memory footprint and are frequently used inefficiently, where only a small part of the allocated memory is in use, with the rest being effectively wasted (by having been allocated but never actually used). Addressing this issue requires not only more careful selection of data structures by developers, but also automatic data structure optimization. Automatic optimization is a necessity, as many data structures are by their nature inefficient, and a developer cannot avoid using them entirely. While there are many compile-time optimization techniques, these mainly focus on increasing execution speed instead of reducing memory footprint.

The objective of this work is to focus on automatic memory optimization, specifically data structure memory footprint. We present a novel algorithm for compiler-driven analysis of application code and optimization of inefficient data structures to better match the specific execution profile of the application. This optimization is invisible to the developer, having no impact on developer productivity nor the application's correctness. It is implemented onto GraalVM Native Image, a Java ahead-of-time compiler, by injecting profiling code during compilation, and modifying data structure allocation calls where necessary.

Our results, obtained from the execution of synthetic benchmarks and popular microservices, show a measurable decrease in the program's memory footprint, reaching reductions of up to 30%.

Keywords

Data structure, Ahead-of-time compilation, Native Image, Static analysis, Closed-world assumption

Resumo

Estruturas de dados constituem uma grande porção da memória ocupada por software, sendo muitas vezes usadas de forma ineficiente, onde apenas uma pequena parte da memória alocada é efetivamente utilizada, com a restante porção desperdiçada. É crucial que os desenvolvedores minimizem o uso de estruturas de dados e algoritmos ineficientes no seu código, mas não é suficiente. Otimização automática é necessária, dado que muitas estruturas são ineficientes por natureza, e um desenvolvedor não consegue evitar usá-las por completo. Embora existam numerosos métodos de otimização em tempo de compilação, estes têm um foco maior no aumento da velocidade de execução.

O objetivo desta tese foca-se em vez disso na otimização automática de memória, nomeadamente estruturas de dados. Apresentamos um novo algoritmo, integrado na compilação, para análise de *bytecode* e transformação de estruturas de dados para versões mais apropriadas para o contexto de utilização. Esta substituição ocorre de forma invisível para o programador, não alterando a funcionalidade do programa. É implementada sobre GraalVM Native Image, um compilador Java *Ahead-of-time*, injetando código de *profiling* durante a compilação, e modificando a alocação de estruturas onde necessário.

Os nossos resultados, obtidos através da execução de *benchmarks* personalizadas e micro-serviços populares, mostram uma redução no espaço de memória ocupado pelo programa, atingindo reduções de até 30%.

Palavras Chave

Estrutura de dados, compilação ahead-of-time, Native Image, Análise estática, Closed-world assumption

Contents

List of Figures

List of Tables

Listings

1 Introduction

Contents

Data structures rank among the very first things a Computer Science student learns, and for good reason: they are a fundamental building block in almost every language and program. It is practically unfeasible to write any algorithm or program without using one. It then follows that out of a program's memory footprint, the majority is consumed by data (and the data structures it is stored in). While data optimization is by no means an abandoned topic, the increasing memory capacities of consumer machines, along with the small impact data structure inefficiencies have for small scale computing, mean that it is not as much of a priority for most developers as it was when a system's total capacity was measured in KB. That being the case in small-scale computing, it is however an extremely important topic in large-scale computing, such as server-side programs. In these use cases, where any program may handle thousands or millions of data structure entries, memory limitations again come into prominence.

As reported in the US Equities Historical Market Volume Data provided by Cboe Exchange, Inc. [1], across all U.S. equities exchanges and trade-reporting facilities, the number of shares handled on 21/10/2022 equals 12.3 *billion* entries. Assuming each trade occupies around 100KB of data, decreasing the space requirements per trade by 1KB, roughly 1%, decreases the day's memory footprint by 11.5 terabytes. In storing the year's trades, a decrease of 4.1 petabytes. It is in these scenarios where data optimization plays an extremely important role, as even individually miniscule optimizations (or lack thereof) make an order of magnitude of difference in how much data can be stored/processed, and hence the system's effectiveness.

1.1 Motivation

Data is stored within data structures. These can vary by language, but all of them are invariably developed and tuned towards the "average use case". When their usage deviates from this targeted usage, efficiency (be it in terms of time or space) takes a hit. For example, a standard `ArrayList` in Java is allocated with 10 empty slots, resizing itself if more are needed. If the developer only ever uses 4 of those slots, the remaining space goes untouched, though allocated. In addition, some data structures are developed to target a specific characteristic such as access speed, in detriment of their efficiency in other areas. A hashtable, for instance, may be allocated with just enough slots for the number of items it is expected to store. And yet, it is statistically unlikely that all of them will be filled evenly. It is expected that multiple entries of the table will remain empty throughout its entire lifespan, occupying significant memory space. Sparse data is another such example, where a majority of space goes unused though allocated.

Developers, too, play a role in memory efficiency. Custom data structures, designed and tuned for a specific use case by the developer are rare, with most developers preferring to simply use "off-the-shelf" structures, accepting the efficiency decrease in exchange for the reliability, ease of development and code simplicity that come with them. This “acceptable overhead” may be negligible in small consumer-facing applications, but prove costly for use cases such as big data or server-side applications.

1.2 Objectives

With that, we identified the following opportunity: optimize generalist data structure memory allocation to better match the specific use case the developer created. The objective of this work can be broken down into three main points, as follows.

First, the study of data structures, namely in terms of their prevalence, usage patterns, inefficiencies, and alternatives. Given all data structure design involves balancing characteristics such as access speed and memory usage, this will provide us with the information necessary to accurately strike a balance between the optimization of memory usage, development time or execution time. With this information, an optimization algorithm can be developed. The real-world data necessary for this study will be collected automatically by a profiler, injected into the code during compilation.

Second, the design and implementation of such an algorithm onto an existing compiler. In this step, a modification of the compiler is made such that during compilation, the algorithm scans the Intermediate Representation for optimization opportunities and, based on profiling data, decides if and how to modify the current data structure. This change must be transparent to the developer/user, and easy to use. Given it is a compiler integration, the developer does not need to perform any change to the source code, just build it with our modified compiler.

The third objective is to evaluate the results of our modification through a performance evaluation which takes advantage of several benchmarks representing real-world scenarios. The optimization must not impact the program’s correctness (*i.e.* the program’s logic or behavior must not be modified in any way) nor can it be allowed to diminish its throughput or responsiveness (latency). This is called a transparent (to the developer, who does not know modification of their work has taken place) optimization. Ideally, the added efficiency in memory usage will translate into better locality and thus a performance improvement. However, the main goal and evaluation objective is memory footprint reduction in realistic workloads.

1.3 Solution definition

The problem and its challenges as described in Section 1.1 provide us with a starting point to define a solution: automatic data structure modification at compile-time, informed by profiling.

It must be automatic, in order to not increase the developers’ workload. Additionally, it must target the commonly used default data structures, and work to optimize them for memory efficiency outside the “average use case”, without impacting performance.

Executing the tuning operations at compile-time allows for automatic optimization, while not introducing additional performance overhead from having them execute at run-time. The optimization itself consists of modifying the current structures’ memory allocation and growth to better fit the usage patterns of the application. Said patterns can be obtained through profiling. To achieve this, we implement two phases onto the GraalVM Native Image ahead-of-time compiler, to be executed separately. First, relevant data structures are found and profiling instructions are appended for an execution to ascertain the aforementioned usage patterns. Second, using the collected profiling information to modify said data structures to reduce their size to one that more closely approaches their actual usage.

1.4 Organization of the Document

In Chapter 2, we provide information relevant to the understanding of this thesis. This is done in two portions: Section 2.1 provides the reader with a high-level explanation of the concepts necessary to understand the topic and thesis; Section 2.2 then elaborates on related works: what has been done in the field of memory optimization, their results, contributions and drawbacks. This chapter also contains a deeper dive into GraalVM and the Native Image sub-component, providing relevant details of these systems.

The solution’s architecture is detailed in Chapter 3, providing a thorough explanation on the theory behind data structure optimization, the solution’s algorithms, along with theoretical details about data structures and how it influenced the work’s logic and choices.

Chapter 4 contains information on the implementation of our solution and its components. Besides justification for the choices of platform, language, targeted data structures and practical details of algorithms, we detail our process for search, decision and modification, its integration and effects, along with examples of these methods.

In Chapter 5 we outline our evaluation process, including what data will be needed, along with our goals for this project’s results. In addition, we detail our evaluation methods, the reasons behind their choosing, and how the data they generate provides meaningful results. Having defined our criteria and goals, we then detail the results obtained from said methods, and perform analysis on them.

Chapter 6 contains our main conclusions and information regarding potential future work to build upon our research and development.

2

Background

Contents

Data structures are a prevalent feature in most programming languages, whether explicitly or not. The first section of this Chapter provides the reader with the necessary background information to understand the concepts upon which this work is based, along with information on the challenges in terms of data structure optimization. Optimization is an equally wide and varied field, from attempts to improve the cross-language definitions of algorithms, data structures, paradigms and good practices, to highly-specific optimizations such as compressing pointers in linked lists [2]. Scope, objectives, methods and target languages vary immensely, and the second section of this Chapter will provide examples of works whose methods, scope or objectives are closest to our work, along with descriptions on their applicability and eventual contributions to this work.

2.1 Topic overview

The following subsections will provide the reader with important information regarding the particulars of data structures and their optimization as a whole, the challenges involved, and the tools we chose to use to help us tackle them. Subsection 2.1.1 presents a short introduction on data structures, their design and usage, along with an introduction on the theory of HashMaps, that will serve as a basis for the more in-depth analysis in Chapters 3 and 4. Following this, we present a brief overview of the particularities and challenges inherent to the optimization of dynamic languages such as Java. Subsections 2.1.3 and 2.1.4 introduce the tools we use to solve the aforementioned challenges, and thus enable our optimization techniques. The last Subsection of this topic overview focuses on profiler-guided optimization, with our work being part of this family of techniques. As such, we present a brief overview of the importance and methodology of such optimizations, their purpose, along with information on works in the area we consider relevant to understand profiler-guided optimization.

2.1.1 Data structures

A data structure is at its core a layout in which to store data, with the intention of increasing efficiency and providing additional tools in its use [3]. A data structure may take many forms, from simple arrays to more complex graphs and hash maps. Some are more specialized than others, allowing for fast lookup of data (such as binary trees) or reduced memory utilization, for example. They are an ubiquitous component of programming, and by design able to be used in a large number of scenarios. This adaptability, however, may lead to issues of inefficiency, as an “off-the-shelf” data structure may not be as efficient as one purpose-made for the scenario, with factors such as development time making it impractical to create the ”ideal” data structure for every use-case. Additionally, programming is not reserved to software engineers. Efforts to expand basic programming knowledge to a vast array of STEM (science, technology, engineering and mathematics) fields like Physics or Mathematics, code schools focused on practical coding knowledge that do not cover topics such as data structure theory, and low-code platforms mean that many developers may not have the necessary knowledge to identify why certain data structures may not be the most efficient in a given context. This work will focus on the aspect of data structures in the Java language, both due to its position as one of the most used languages, and how it allows ample and very flexible usage of non-optimal data structures. For the purposes of notation clarity, HashMap refers to the theoretical definition of the data structure, and `HashMap` refers to its specific implementation/object in Java.

It is entirely unfeasible for the designers of any programming language to develop an ideal data structure for every single possible use case or developer intention. As such, any data structure to be included in a language's development kit must be able to be used in a multitude of scenarios. A Map, for example, must be able to accept any type of object both as value and as key to said value, and support not only the standard map operations such as retrieving a value with a specific key, but also operations more commonly associated with other structures, such as iterating through all entries.

In addition to this ”swiss-knife” design, there are numerous performance characteristics that must be taken into account during design and usage of a data structure, such as access speed or memory efficiency. Attempting to increase one will inevitably lead to a decrease of another. As an example, let us take a simple linked list. To know how many entries it contains, one would have to iterate through all of them and count. This can be circumvented by keeping a variable containing the current number of entries, and updating it upon removals and insertions. To accelerate insertions at the end, one would need to store the last value's reference as well. In both of these situations, an improvement in temporal performance is obtained at the cost of additional memory footprint. These trade-offs may seem small in practice, but as demonstrated in works such as [4], an object's properties may, in some situations, take up more space than the data it is meant to contain. Some data structures, on the other hand, eschew attempting to balance these characteristics in favor of maximizing one or several of them. An interesting example of this approach is the HashMap (also known as a HashTable). This family of data structures prioritizes access time, trading off space efficiency to obtain average *O(1)* read and write speeds.

Given the large amount of existing data structures, there are equally many situations where a non-optimal structure may be used. This work, however, will focus on one specific situation: where the structure, in light of its usage, occupies more memory space than is necessary. Examples of this are the aforementioned HashMaps and sparse matrices. In a sparse matrix, most positions are empty, and will remain so throughout the entirety of the execution. These positions are allocated, however, which means that by definition over half of the memory used by the sparse matrix is being wasted. Thus, usage of structures such as these presents an opportunity for reducing their memory footprint by minimizing empty entries based on effective utilization.

A HashMap is a form of *key-value store*, consisting of an Array *A* and a hashing function *h(k)* [3].

0	1	2	3	4	5	...	n
V_0		V_2		V_4	$V_{5,0}$; $V_{5,1}$	...	V_n

Figure 2.1: Diagram of a generic HashMap, showing a collision at *A*[5].

Given a key *k*, the hashing function uses *k* to obtain an index, denoting the value's location in the Array. Thus, the value can be accessed at *A*[*h(k)*]. In an ideal HashMap, the array's size is large enough to fit all elements in individual entries (called buckets) and the hashing function is *collision-free*, that is, no two keys exist that produce the same index as a result of the hashing function. However, such properties are not feasible in reality due to the sheer size of such an ideal HashMap. Thus, there will be situations where different keys produce identical indexes, or several values are stored with the same key. In such situations, the value may be placed in another algorithmically-chosen bucket, or share the bucket with other values, as shown in Figure 2.1. A common implementation is to have the values stored as a linked-list in each bucket.

In practice, HashMap implementations tend to be more complex than this theoretical example, due to the desire to support more operations, such as checking whether a given key is present (thus requiring keys to be stored). Additionally, the possibility of collisions requires us to differentiate inside a given bucket between values belonging to different keys. Figure 2.2 demonstrates a diagram with this added complexity.

0	1	2	3	4	5	...	n
K_0: V_0		K_2: [V_0, V_1]		K_4: V_0	K_{5a}: [V_0, V_1] K_{5b}: [V_0]	...	K_n: V_0

Figure 2.2: Diagram of a generic, but more realistic, HashMap, showing two values for the same key at *A*[2], and a collision of two keys, each with its own values, at *A*[5].

The objective of this thesis is to exploit this opportunity by creating and implementing a technique which automatically detects and optimizes data structures according to predetermined heuristics, along with data derived from the application's usage of said data structure.

2.1.2 Optimization of dynamic languages

This kind of optimization is, however, not always possible, especially in the context of Java. Java's somewhat dynamic structure provides a significant challenge: classes are not all loaded *a priori*. Java classes are compiled and loaded only when they are first called or referenced (i.e. when needed). This demonstrates the limitation mentioned in the previous paragraph: a class may be loaded initially and deemed fit for tuning. Later in the timeline, the JVM may load another class which interacts with the previously optimized one in ways that are incompatible with the optimization, or load a class that overwrites a method that was previously used for tuning purposes. The issue this raises is simple to describe: an assessment which holds true now (*“Is this data structure safe to modify?”*) may not be true in the future. We need, then, a way to know *a priori* which classes will be loaded, so that we can ensure that the objects we modify will indeed remain safe to modify throughout. In addition, ”de-optimization” (removing the optimization by returning to the previous state) is made hard with data structure tuning, as the memory’s layout at a later point post-tuning may hinder efforts to, for example, return the data structure to its original size.

Rarely do compilers directly convert source code into machine code. There are often multiple steps in the process, and one of them is Intermediate Representation (IR), which allows for many of the optimizations we see in compilation. IR is a data structure (frequently in form of a graph) obtained from an initial translation of the source code. This IR, if properly created from the source code, creates numerous opportunities, such as the ones we will take advantage of in this work, namely code analysis (through analyzing the graph’s shape and flow, for example) and optimization (with the information from analysis). We will expand upon the uses and research on Intermediate Representation in the following subsections.

After these challenges, which tend to affect to some degree all attempts at optimization, there are some challenges specific to our technique. Firstly, the aforementioned java capability for dynamic class loading may hamper our efforts to detect data structures viable for tuning, or introduce new methods or behaviors that affect the accuracy with which we are able to model the data structure’s usage patterns.

The most difficult issue is perhaps the act of profiling: for good optimization, the data obtained must be as accurate as possible. This entails correctly instrumenting all interactions which modify the structure’s contents or size, using accurate data for the profiled executions, and calculating with high accuracy the resulting statistics, which will inform the decision whether or not to optimize any given data structure.

2.1.3 Native Image

These notions then bring us to our choice of base upon which to build this project: GraalVM [5], a HotSpot-based Java Virtual Machine, and most importantly its Native Image component [6]. The nature and features of GraalVM and its components allow us to bypass some of the challenges outlined above, and otherwise simplify some of the work required to research and implement our optimization.

First, Native Image allows for something uncommon in Java: ahead-of-time compilation, compiling the whole program at once and creating an executable binary, much as other non-interpreted languages such as C do. Besides the faster startup and lower memory footprint this brings, it solves our challenge with class loading by searching for and integrating all classes that would be dynamically loaded throughout a ”classic” JVM-based execution. This approach allows us to apply the ”closed-world assumption” logic paradigm with regards to verifying whether an optimization will be safe for the entirety of the program’s runtime. Per Reiter [7], in a closed-world assumption, ”a negative fact is implicitly present provided its positive counterpart is not explicitly present.” [7], that is to say, we may only assume as true that which we can prove to be so. For the purposes of Native Image, and thus our work, if all classes the program uses are loaded at once during compilation, then the value of our assertion (*“Is this data structure safe to modify?”*) can be reliably calculated, and will remain unchanged throughout compilation/execution. To verify this, the Native Image compiler goes through the entire application classpath and loads all classes. Afterwards, with the main method as the entry point, it traverses all of the code and prunes any class or method that cannot be reached, that is, that will not be used. Therefore, from this static analysis, it is possible to prove types for class fields and, by extension, that the closed-world assumption holds true.

Second, GraalVM possesses a very complete Intermediate Representation, GraalIR [8], with an API and tools available to facilitate its viewing, interpretation and usage [9]. As such, this IR allows us to more easily analyse the code, and implement our optimization strategy, as we can integrate it into the remaining compiler optimizations performed onto the IR.

As a third benefit, there are also several profiling tools. By using these, directly or with modification, we can obtain relevant information on the program’s behavior and usage of data structures. These allow us, for example, to decide whether it is beneficial to alter a given data structure’s characteristics given its real-world usage in the program.

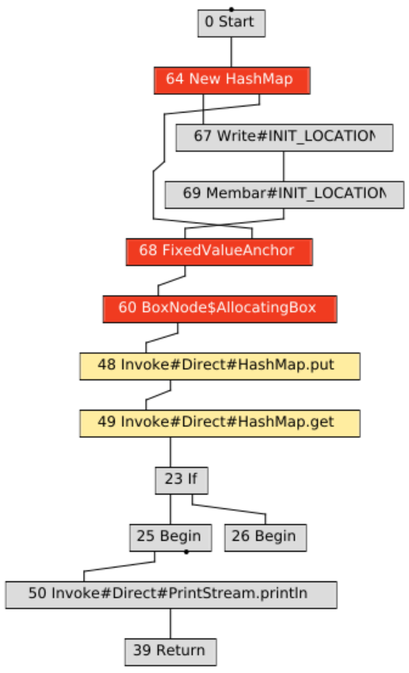
2.1.4 GraalVM Intermediate Representation

Intermediate Representation, specifically GraalIR, plays a major role in all aspects of this work. It is the base for all of our operations, from the discovery of potential optimization opportunities, to their implementation onto the program. The only step not directly based upon the IR is the execution of the profiled program, though the profiling instructions are also inserted at the IR level. It is not an exaggeration to say that were GraalIR any less feature-rich, our work would have been substantially more difficult. Thus, understanding its functionality is integral to understanding the details of our work.

GraalIR is ”a graph-based IR that models both control-flow and data-flow dependencies between nodes” [8]. This IR, generated from the source code during compilation, represents in a directed graph the program’s code, with nodes representing instructions, and separate edges for data and control flow. An example of source code can be seen in Figure 2.3, along with the corresponding (simplified for readability) IR. This graph contains not only the allocation and method calls, but also all of the routines inherent to the data structure’s usage, such as memory allocation, accesses, and the multitude of safety checks performed by Java when it is accessed. All of this information is very useful for optimization purposes. For example, node 64 in Figure 2.3 corresponds to the `new HashMap<>()` section of the

```
public class TestClass {
    public static void main(String[] args) {
        HashMap<Integer, String> exampleMap = new HashMap<>();
        exampleMap.put(8, "Goodbye, World!");
        System.out.println(exampleMap.get(8));
    }
}
```

(a) Source code



(b) IR Graph

Figure 2.3: Java source code where a HashMap is used, and the IR generated from it. Some elements from the IR such as exception flows were truncated from the graph shown here in the interest of readability.

code, allowing us to add the newly-declared data structure to our list of potentially optimizable instances whereas node 48, representing a call of the `put()` method, denotes one of the accesses to the structure. This access is one we seek to instrument in order to collect our profiling data. The ability to directly represent instructions as nodes makes it relatively simple to not only track structures and their interactions, but also to insert new instructions (to enable our runtime collection on the structure's usage) and modify existing ones (such as modifying the allocation method's arguments to change the structure's initial size or load factor).

Intermediate representation sits at the core of Native Image, enabling both the ahead-of-time compilation process as well as the numerous optimization procedures applied by the Native Image compiler. After the initial IR is generated from the input source code, Native Image performs several successive optimization "passes" (called phases), taking advantage of the traversal opportunities presented by the graph representation. Initially, this entails the traversal of the graph's control and data flow edges to uncover all of the classes accessed by the program throughout its execution. Accordingly, classes that will not be used during execution can be safely discarded (pruned), reducing the "bloat" common to standard JVM-based execution. Following phases perform analysis on the graph, identifying opportunities for further optimizations. At the end of the compilation, the graph's final version has been the target of numerous optimization passes, and represents a far more efficient program than the one it received as input. We will provide a deeper look into compiler and IR-based optimizations in Chapter 3.

Intermediate Representation has served as the basis for numerous works on optimization. The work of Lattner [10] provides very valuable knowledge on several central concepts for this thesis, such the analysis and modification of data structures at compile-time, the creation and analysis of intermediate representation graphs, and shape analysis, allowing for the recognition of patterns within these graphs, which can then be optimized. This work's main goals do not particularly align with our thesis, them being pointer analysis and locality improvement through "Automatic Pool Allocation" [10]. In addition, they introduce "Automatic Pointer Compression, which reduces the size of pointers on 64-bit targets to 32-bits or less, increasing effective cache capacity and memory bandwidth for pointer-intensive programs" [10]. However, many of the methods, insights and drawbacks are applicable.

2.1.5 Profiler-guided optimizations

Static analysis, which in the context of this work refers to analysis of the IR graph, can allow for some very impressive optimizations. Listings 2.1 and 2.2 demonstrate this potential: through static analysis, the compiler knows exactly how many times `total` will be incremented, and thus can assign to it the final value directly. Further, the compiler also knows said variable will never be used elsewhere in the program. Knowing this, there is no need to have a variable at all, and the compiler will simply use the final value as argument to the `println` method. Through static analysis optimizations, the compiler has saved memory (variables `total` and `i`) and execution time (the `for` loop).

However, static analysis is limited in what it can "know" about the program, and thus in its ability to optimize it. Taking the aforementioned example, imagine that '123456' (from Listing 2.1) was not a hard-coded value, but rather a variable whose value is assigned at runtime by the user (through the command line, for example). In such a situation, the compiler cannot perform any of the optimizations it did in the example, since it cannot know how many times the loop will be executed, even if the user inputs the same value in every single execution.

Profiling, in essence, entails executing the target program simulating real usage, while collecting data on how it is behaving. By profiling the example code, we are able to determine that the user always inputs '123456', for example. Passing this information to the compiler, it once again is able to perform the above-described optimizations. This is an example of a profiler-guided optimization, where profiling data is used to enable optimizations that would be unfeasible without it. The profiling information can be stored and passed to the compiler to be used for future compile-time optimizations [11], or used *during* said execution to speed it up [12–15]. The just-in-time compilation of the HotSpot JVM can be considered an example of the latter, as the JVM will interpret most bytecode, but will directly compile to

Listing 2.1: Example of source code as it enters the compiler.

```
int total = 1;
for (int i = 0; i < 123456; i++) {total++;}
System.out.println(total);
```

Listing 2.2: Example of source code optimized as a result of static analysis.

```
System.out.println(123456);
```

machine code bytecode sections that it recognizes as executing frequently, improving their performance. The decision whether it is worthwhile to spend execution time to compile a given bytecode section is informed by the JVM profiling how often each section is executed.

Data structures are especially resistant to static analysis-based optimizations, as they tend to be used to store information obtained at runtime, and thus we cannot know *a priori* just how many entries a given structure will have when the program terminates. Given our work focuses on improving the memory usage of data structures by tailoring their size to how much data they will contain, profiling is a central aspect of it.

The work of Gope and Lipasti [13] uses profiling information to dynamically optimize executions through the inlining of hash maps in scripting languages. According to the authors, in a compiled language, such as Java, the properties of an item would be stored in an object. However, in scripting languages hash maps are often used to this effect, storing a fixed set of keys as a substitute for value fields. As such, this work "describes a compiler algorithm that discovers common use cases for hash maps and inlines them so that keys are accessed with direct offsets" [13]. While the authors showed promising results, their work relates to scripting languages and thus does not transfer to Java.

Another excellent work of runtime optimization is that of Kessels [16]. Focusing primarily on trees, this work focuses on optimization as a collection of low-scale operations that are appended to the access function of the structure. As such, as the structure is accessed, the portions accessed are optimized. Over the time of the execution, this leads to the structure tending towards an optimized state, with the benefit that no functionality is lost and that the structure can be fully accessed *while* it is being optimized. However, that same integration with access functions means that there is no guarantee a structure will ever reach a state of optimality, as this is fully dependent on how it is accessed.

2.2 Related works

As explored in Chapter 1, the wide availability of large quantities of memory, even in consumer machines, has driven software and hardware development and optimization efforts towards improving temporal performance, rather than memory efficiency. From the gaming enthusiast to the professional artist, users have demanded ever increasing frame-rates and ever decreasing render times. A cursory search at a local computer store's website [17] shows that at the time of writing 70% of both laptops and desktops on sale have 16GB or more of RAM. Thus, it is unsurprising that optimization research has followed suit, with the majority of the works we found relating to data structure optimization focusing on performance.

While less numerous, there have been several notable works whose focus is memory efficiency rather than performance. Few works, however, deal with tuning data structures for individual use cases, preferring to replace them with other structures that may be more adequate [18], or even removing them entirely through inlining [19]. We were not able to find works whose focus was to use profiling-guided optimization to tune data structures at compile-time to improve memory usage. That being the case, while the objectives and methods of the related works described in this section do not necessarily equate to ours, the results thereof can provide strong insight into the challenges and possibilities of this thesis.

Before any optimization can take place, the targets for it must be identified and analyzed. Though some runtime data structure detection works exist [18–20], data structure detection is more frequently done at compile-time [4, 21–23]. Besides their other contributions, Lattner [10] proposes several new methods for analysis of IR graphs to detect data structures through shape analysis. The work regarding a novel IR proposed by Duboscq et al. [24], besides allowing for more flexibility in dynamic optimization and deoptimization (which tends to be extremely costly in the context of a JIT compiler), reaches several interesting conclusions, especially with regards to creation, analysis and modification of IR graphs, which is a central component of the thesis.

With the data structures identified, it must now be decided how to optimize them. Often, especially in Java, the data for this decision is gathered through runtime profiling [18–20, 25]. The works of Changhee Jung [18, 20] closely align with our goals in this regard, given their objective is to identify not only which data structures are used, but to track how the program interacts with them, using runtime instrumentation to measure the data structure's memory usage in terms of space and organization, along with read and write operations. However, no optimizations are directly performed by the algorithms and tools presented in them. Rather, the data is used to provide the developer with detailed information about the data structures, allowing the developer to make an informed decision how to best optimize them, whether that entails using different parameters in their creation, or replacing them with another type. Our work seeks to go further, by using similar information to automatically perform optimizations, rather than leaving it to the developer, who may be limited in terms of time or knowledge to fully take advantage of the information provided.

Applying the optimizations is the third and final step in the process. Certain works implement their optimizations at runtime [16, 25], or just-in time compilation [14]. Particularly, Grimmer et al. [25] take advantage of the additional information available at runtime to dynamically modify data structures to, for example, store primitive data types rather than their full-fledged boxed counterparts, reducing each entry's memory footprint, and thus the data structure's as a whole. For some objects, the footprint of the headers and other non-value fields exceeds the footprint of the value itself [4]. Object inlining, which can be described as "embedding the payload of an object into another" [4], is a core technique of several works parallel to ours. A given *parent* object may possess as a field another object (*child*), which itself contains the *payload*. It then follows that replacing the *child* value field inside the *parent* with just the payload can lead to major memory savings. According to the authors, there are major performance benefits: up to 3x throughput and up to 40% less memory footprint, all the while requiring only a small intervention from developers [4]. This approach, however, carries a drawback: automatic inlining is often infeasible outside of very simple objects due to their structure and usage within the code, requiring the developer to manually and correctly flag all fields that can be inlined. While our optimization does not implement any variant of inlining, many of the processes are shared between our work and that of these authors. Particularly in the case of Bruno et al. [4], the techniques for detecting optimization candidates and applying optimizations in an *ahead-of-time* manner in the IR are remarkably similar, along with both of our works being built upon Native Image and taking advantage of GraalIR. Both the aforementioned work, and those of Dolby [21, 22], perform their optimizations at compile-time, with the compiler analyzing and automatically inlining objects where it deems possible. The latter work expands upon it by formalizing a model for inlining, including correctness conditions.

2.2.1 Observations

These works provide a good showcase of the difficulty in memory optimization, especially with regards to data structures. Firstly, most data structures can be optimized in some way, from simple vectors to trees [16]. Given this multitude of optimizable data structures, compounded by equally many ways of using them, it can be a daunting task to select which of them will be the focus of one's work and research.

None of the works, however, match in full our motivations, objectives and techniques. Instead, we find ourselves sharing only partial aspects, such as IR analysis [4, 10, 21–24], profiling data collection [18–20, 25], and AOT compile-time optimization [4, 10, 21, 22]. One of the major differentiators of our work is universality: rather than being restricted to a certain type of structure, usage pattern or architecture, our work seeks to be able to improve the memory footprint of almost every usage of a given data structure. While Java HashMaps were chosen as the target for the practical aspect of this work, the same principle applies to all data structures whose instantiation involves the allocation of a predefined number of ”slots” and usage-based resizing, such as Arrays. Additionally, our techniques can be applied to other statically-compiled languages such as C++, making this work's applicability rather substantial.

Most of our related works focus on memory optimization, with exceptions such as Lattner [10] and Duboscq et al. [24], which focus on performance improvement (with possible footprint improvements being a side effect of their respective techniques). Out of these memory-focused works, the a sizeable portion [4, 13, 14, 21, 22] implement or develop a variation of an inlining technique, eliminating the data structure rather than modifying it. While it is effective, it shows its limitations in terms of what can be inlined without loss of functionality, being rather limited to what Bruno et al. [4] describe as *value fields*, that is, objects whose only purpose is to store a specific set of values. Since the number, format and access pattern of values in, for example, a HashMap is variable by definition, inlining is usually not possible. The work of Gope and Lipasti [13] is an outlier, but in that work the inlined maps were used to store a specific set of values, the size of which did not change in execution. In other words, the Hash Maps were used as *value fields*. Table 2.1 details some of our related works and the aforementioned traits.

Table 2.1: Related works and their characteristics

Work	Timing	Technique	Contribution	Limitation
Jung and Clark [18], Jung et al. [20]	N/A	Data structure analysis	Analysis algorithms	Information only
Kessels [16]	Run-time	Optimization-on-access	On-the-fly optimization	Depends on structure usage
Pape et al. [14]	JIT	Data Structure Inlining	Value class optimization	Demo language only
Bruno et al. [4]	AOT	Object Inlining	Value fields	Developers must flag targets
Lattner [10]	AOT	Pointer analysis/compression	Macroscopic optimization	Performance focus
Duboscq et al. [24]	JIT	Speculative optimization	IR	Performance focus
Gope and Lipasti [13]	JIT	Object inlining	Hash Map inlining	Scripting languages focus
Dolby [21], Dolby and Chien [22]	AOT	Object inlining	*Automatic* object inlining	Finds half of inlining chances

Given these observations, we find our work fits a set of criteria which have not been fulfilled by previous research, namely:

- *Modification instead of inlining or replacement.* Data structure tuning allows for a wider set of optimization opportunities besides value fields, while being more universally applicable than replacement:

- *Java as the target language.* Most related works either target C/C++, a simple demo language, or none at all. We seek to target Java, it being a widely used language. In addition, we aim for our work to cleanly integrate into the Native Image compiler, which allows it to be ready for real-world usage without further adaptation necessary;

- *No developer action required.* Some of the analysed works require developers to make changes to their code. In the work of Bruno et al. [4], for example, the developer must explicitly flag in their code the objects which can be inlined. This requires (admittedly trivial) changes to existing code, and for the developer to have a modicum of knowledge about the topic in order to know which objects to flag. Our work aims to be completely developer-independent, whereby no changes to the code are required, and the developer may not even be aware of the optimizations the compiler runs, as there is no change to functionality;

- *Ahead-of-time compilation.* Many related works implement their optimization as a component of just-in-time (JIT) compilation (see table 2.1). While there are several advantages to this approach, the nature of JIT compilation inherently limits the reach of such algorithms, as they may impact the program's responsiveness. However, a more likely issue is the introduction of classes mid-execution, as detailed in Section 2.1. Through the usage of ahead-of-time (AOT) compilation, namely with Native Image, we can avoid this problem by relying on a closed-world system. All classes being loaded at compilation, and as such known, allows us to engage in more aggressive tuning, as we can be sure that there is no danger of them being made invalid/breaking by a mid-execution class loading.

3 Architecture

Contents

In languages such as C the developer must, in many cases, manually allocate a data structure with a fixed size based on what they predict the usage to be (or use a value large enough to where it is implausible the structure will overflow). This is a direct consequence of C's ”the developer has absolute control over everything” approach. Though powerful and efficient for those with the abilities to take advantage of this, this is not optional, and C developers without plenty of experience (such as first year CS students) quickly become familiar with the expression ”Segmentation Fault”.

Between the increasing memory capacities of current devices (as described in previous chapters) and the rise of higher-level languages like Java or Python whose memory management and garbage collection algorithms continually grow more efficient, this approach has fallen out of favor. The ability to simply declare a data structure and leave allocation, resize and de-allocation operations to the language's control is a popular one, providing far easier development and drastically decreasing the chances of problems like memory leaks occurring. To enable this, data structures in such languages automatically change their size as required, as it is generally impossible to determine just how many entries the structure will receive during execution. In the interest of reducing wasted memory, data structures are allocated with a predefined small size, and resize if they need to accommodate more data. Throughout this work we will refer to *initial capacity* as the starting size of a data structure.

Resizing every time an entry is added is unfeasibly inefficient in terms of CPU cycles, as resizing is a relatively costly operation. Thus, some sort of heuristic is needed to guide how, when and by how much the structure grows. Generally speaking, a resize is triggered when a structure reaches a certain utilization percentage. Given the performance overheads involved in a resize, it is of interest to perform them as seldom as possible by waiting until the structure is close to full before resizing, and having the resize create plenty of new empty spaces. While these heuristics are successful in minimizing the number of resizes, they also have the side effect of wasting memory. After all, the more new spaces a resize creates, the further away in time the next resize is, but the chance of at least some of these newly-allocated spaces going unused increases as well. In addition, many data structures never resize “down”, which means that any growth from resizes is permanent, with small spikes in usage leading to massive wastes of memory. When the *utilization* (proportion of total entries that are ”occupied”) equals or exceeds the *load factor* (proportion of how many entries must be ”occupied” before a resize is triggered), the data structure is resized in accordance with the heuristic that guides the process.

Section 2.1.1 goes into detail on how data structures are built to be most efficient for the ”average” use case, and in general attempt to strike a balance between memory footprint and performance. It is not surprising, then, that the values for initial capacities and load factors are also defined as a balance between the two metrics. Larger initial capacities mean less resizes, improving performance at the cost of a larger memory footprint. This focus on average values means that specific situations end up less efficient than they could theoretically be. An example would be a structure that stores a small fixed number of entries, that cannot be separated due to a need to be passed as an object (optimizing this specific situation is the focus of the related work by Gope and Lipasti [13]). In such a situation, the rest of the structure's entries are allocated, empty, and will always be so. This inefficiency is cause by the initial capacity, tailored for the average use case, being too high for this specific situation.

The data structure we chose as focus for this thesis is the HashMap. As a hash-type structure, it prioritizes fast access times (*O(1)*) over memory efficiency, being a wasteful data structure in even the most appropriate use cases. In Java, HashMaps are allocated (unless the allocation is manually modified by the developer, which is quite rare) with a default size of 16 and a load factor of 0.75. Since a resize is triggered once utilization is equal/higher than the load factor, 25% of the memory occupied by the HashMap is always inaccessible, and thus wasted. This 25% waste is actually a best-case scenario in terms of waste, occurring when utilization is just one element away from a resize. While 25% of 16 spaces is not a major memory waste, it is a constant proportion, and after several resizes becomes very significant, as can be observed in Table 3.1, wherein after 8 resizes over 1000 spaces are effectively unusable but allocated.

Table 3.1: Allocated and usable space of HashMaps as a function of resizes (Java default values)

Resizes	0	1	2	3	4	5	6	7	8
Allocated space	16	32	64	128	256	512	1024	2048	4096
Usable space	11	23	47	95	191	383	767	1535	3071

Resizing the HashMap is also a very costly operation in terms of performance. It involves the allocation of additional space (along with potentially having to move the entire structure to another memory address if the subsequent positions have already been occupied with other data) and, more importantly, *rehashing*. During accesses, the hashing function calculates an index where the given object should be according to its key, and this index is of course dependent on the map's size. When a map's size changes, the index generated by the hash function for a given key may not be the same as the one before the resize. Let us illustrate this with an example: during data structure introduction courses, the example often given for a basic HashMap implementation is one that uses prime numbers as a basis for both size and hashing function. In this example, a HashMap is an Array whose size is a sufficiently large prime number, such as 317. The hashing function then takes a numeric key, and performs a *modulo* operation ($h(x) = x \bmod 317$), the result of which is used as the index. If the array were to be resized to some other prime number, the new hashing function would return a different index value for the same key ($x \bmod 317 \neq x \bmod 419$). Thus, all of the existing entries are incorrectly placed, and must be moved to the new index. Performing this operation on every entry in the HashMap is both computationally expensive and unavoidable when resizing. While methods exist to reduce this impact, such as rehashing only part of the HashMap at a time, they only spread out the operation's cost over time, rather than eliminate it.

Thus, it is in the best interest of an implementation's designer to set the values of initial capacity and load factor such that, for the average use case, this occurs as infrequently as possible. This is one of the main factors why at first glance it might seem that a default initial size of 16 might be too wasteful (memory-wise) for the common use case, where the developer will store only a couple entries in the HashMap: there has been a choice to trade some wasted memory in exchange for the chance that the average use case will see at most one (and ideally none) resize.

In other use cases, however, this design choice becomes a downside. For example, microservice-style applications often use HashMaps to store caches to avoid fetching the same immutable data multiple times through expensive requests to other services. In such situations, the number of entries is expected to grow steadily to a large number throughout the application's runtime, especially in the beginning when the frequently-used data points haven't yet been cached. In this example, resizes (and consequently rehashes) are numerous, and the continuous doubling of the HashMap's allocated space can quickly lead to large amounts of wasted memory, especially given that such a caching approach is often applied to several data types at once, and in multiple services. Use cases like this are a prime target for our optimization, especially since cache size growth tends to stabilize as the most common data points get cached, allowing us to set the HashMap's size to one that better matches the stabilized cache size. Additionally, if the cache growth stabilization exists and is properly represented during profiling, resizes may be prevented altogether, improving the application's performance.

This kind of usage pattern cannot be inferred at compile time, and must be identified using data obtained at runtime. Thus, this work's optimization process occurs in two steps. During a program's first Native Image compilation, profiling instructions are injected into the program's code, allowing for the collection of usage information. After the program has been executed with an expected load/use case, compilation is called again. In this second compilation, no profiling instructions are appended. Rather, the data generated by the previously appended profiling instructions is analyzed, and from it, data structures are checked whether optimization is viable, and it is applied if deemed positive. The final, optimized, executable of the program is now ready for usage/deployment. The following sections provide more details on the theory behind each of the aforementioned steps.

3.1 Tuning opportunity detection

Before any sort of optimization can be performed, we need to identify the relevant data structures to be able to insert the relevant profiling instructions. While there are methods to detect the types of data structures a program uses at runtime [18], doing so at compile-time allows for several advantages. Firstly, said runtime method involves instrumenting a sample execution [18]. Given the kind of program that stands to benefit most from our optimization are long-running, high-footprint applications such as

Listing 3.1: Java-style pseudocode for the tuning opportunity detection algorithm.

```
ArrayList<Integer> interestNodesIds = new ArrayList<>();

/*Verifies graph contains structures that can be optimized according to
our rules, finds allocations of data structures of the appropriate type,
and checks against custom values.*/
private void findOpportunities(StructuredGraph graph){
    if(matchTargetPackage(graph){
        for(Node n : graph.getNodes()){
            if(n instanceof NewInstanceNode){
                if(isHashMap(nin.className)){
                    if(verifyValues(nin))
                        interestNodesIds.add(nin.id);
                }
            }
        }
    }
}
```

microservices, this profiling run will most likely take longer than the extra compilation time caused by the analysis. The second benefit is accuracy: during compilation, the program's source code is fully available to us as an IR graph, allowing our identification of a data structure to be very accurate. While Java's dynamic class loading makes this approach quite difficult, this is negated by our use of Native Image. As a third benefit, we can restrict our analysis to specific packages and implementations. This allows us to, for example, exclude any HashMaps used in Java's standard libraries, ensuring we do not accidentally change their behavior or performance, maintaining optimization safety.

Detecting the relevant data structures without data flow or safety analysis is facilitated through the use of a graph-based IR. Our algorithm for this consists of a simple traversal of the graph (provided it matches our target package), where each `NewInstanceNode` (the node representing the allocation of a new object) is verified against a whitelist of viable data structure types. While our scope for this work only involves HashMaps, this allows for simple extension to other data structure types such as Arrays.

Valid tuning opportunities, then, consist of HashMaps allocated in developer-authored code, whose parameters were not manually set to a custom value by the developer. We assume the developer that sets such values has a good understanding of the logic behind them, and thus is targeting a specific use case or system behavior. In these cases, we defer to the developer and do not modify their design.

Algorithm 3.1 details a pseudocode implementation of the opportunity detection and analysis steps detailed above. It iterates *O(n)*, *n* being the total of nodes in the system. Given the average program contains comparatively few data structure allocations, and few of them are of the type considered for tuning, the loop is expected to run in full very few times. After verifying that the IR belongs to a method it is allowed to work on, and that the developer has not set custom values for the properties, the result of the algorithm is a list of IDs for tuning candidate nodes that were found to be of the type suitable for modification. This list is then fed to the next step: profiling.

3.2 Application Execution Profiling

With the unsuitable tuning candidates filtered out, it is now necessary to decide which data structures it is beneficial to modify, and with what values. Since the objective is to modify generalist-tuned data structures into ones more suitable for the program's specific use case, it is crucial to determine what that use case may be. However, this can hardly be done at compile-time, as program behavior (and thus data structure usage) usually depends on inputs given to it during its start-up or mid-execution. Profiling answers this conundrum, allowing us to attach our own code to the developer's and having it execute with the program. We can then use it to extract data on the execution, from execution time down to the utilization of individual data structures.

To determine the viability of a data structure's modification, we require knowledge on the following items:

- *Initial capacity, load factor, resizes performed during execution, and allocated size.* These are necessary to calculate the size the structure occupies, and how much of it is available to be used;

- *Number of elements contained, both at the end of execution and the maximum achieved.* When compared against the ones in the previous point, we can calculate the utilization of the structure, and thus the unused, but allocated, space. This will be used to calculate whether it is preferable to modify the initial size, load factor, or both, and what the new values must be to improve the structure's memory footprint;

- *Average utilization/number of elements per access.* With this metric, it is possible to determine the structure's usage patterns, both for use in more accurate optimization, and to avoid tuning structures based on utilization spikes. These may not be indicative of average usage, or indicate an unstable data structure, one where the number of elements fluctuates very strongly, leading to potentially harmful or low-benefit tuning.

The list of tuning opportunities created in the analysis step, as detailed in Section 3.1, is now fed to the algorithm that will attach the profiling instructions to the developer's code. This algorithm goes through the IR graph once, and attaches instructions when it encounters a relevant node. Relevant nodes are determined by cross-referencing the node's ID or, when the node represents a method, the ID of the method's target, against the list of tuning opportunity IDs. For allocations, the instructions entail the creation of a new entry in the database which holds all of the metrics detailed above, whereas for methods they execute updates on said metrics. After all profiling instructions have been appended to the

IR graph, compilation proceeds as usual. These additional instructions have no bearing on the original program's logic, and are designed to have as little of a performance impact as possible.

With the executable generated, it must now be executed in order to gather the profiling data. It is important that this execution mimics real-world usage as closely as possible, in order to provide maximum accuracy on the data generated. Failure to do so might compromise the post-tuning program's results, with the algorithm tuning data structures for smaller quantities of data, as guided by the profiling information it received, only for the real-world usage to be far higher, thus forcing resizes on the low-utilization-tuned data structures. The opposite case will also yield subpar results, as the optimization will leave space "on the table" which will not be used in real-world conditions. Ideally, the developer would record real-world usage metrics from a production version of the application, and use those for the profiling execution.

After the execution of the profiling-appended application, files containing the results from the execution are generated. When the compiler is once again called onto the source code, it will identify these files and use the information within to inform the decisions regarding which, if any, tuning to implement for each candidate data structure. Figure 3.1 demonstrates the entire process from the initial compilation that inserts profiling code, the profiling execution, and the second compilation that produces the final executable.

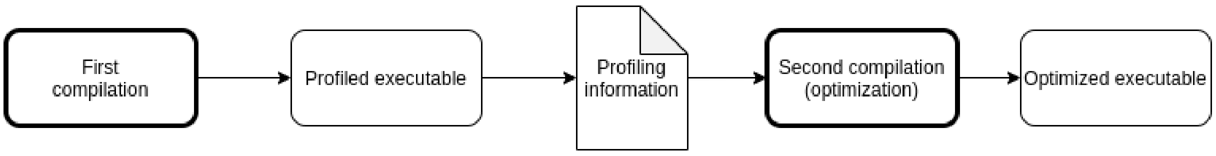


Figure 3.1: Flow diagram of the process of obtaining an optimized executable through profiler-guided optimization.

3.3 Data Structure Tuning

As the second compilation begins, the compiler will identify that profiling information exists for the target code, and use it as input for the tuning algorithm. It is important that the source code not be changed between the first and second compilations, both for reasons of safety (ensuring that all of the ids and methods identified in previous steps correspond to their intended targets) and results (changes to the code may result in different data structure utilization that invalidates previously obtained profiling results). It is up to the developer to ensure this is the case, though the compiler takes an additional step to protect against using outdated information by deleting the files containing the profiling results after the tuning phase is complete. This forces the initial compilation and profiling process to be executed again when tuning is once again necessary. Though our model requires two compilations to obtain optimization results (one for the profiling step and another for the optimized code), we believe this to not be a problem in real-world application, as often a new version of a program is prepared in development environments while a production version runs concurrently in a user-facing environment and can simply be replaced in the same way as any other service update.

As an optimization, data structure tuning offers some advantages when compared to other methods such as replacement. First and foremost, the scope of application is far larger. For replacement, the main concern relates to safety and correctness: there are a myriad methods, such as `InstanceOf`, whose return would be changed in the modified program, altering the program's correctness. In the same way, the replacement data structure must allow for the same type of usage with the same performance characteristics to not deteriorate performance compared to the program's "unoptimized" state, and creating a new implementation of a data structure that maintains functionality and performance while consuming less memory is a daunting task. Additionally, there are numerous situations where replacement is not feasible by design, one of them being service-type programs: if the data structure is in some way "packaged"/serialized to be sent as a message/payload to another service, it cannot be replaced as that would impact other services that we may not be able to modify. The same restriction applies to inlining, as "wrapper" objects (which are often the targets for inlining) are often used to wrap the contents of messages into a single object rather than sending a multitude of individual fields. Contrasting with these restrictions, tuning has, theoretically, a very wide scope: by not changing anything in terms of type, functionality or existence of a data structure, almost every self-scaling data structure with stable usage is a viable candidate for tuning, and in almost every such case there is a possibility for at least a small improvement in footprint.

Table 3.2: Utilization and used space as a function of entries. Load (l) refers to how much of the allocated space is being used, while utilization (u) denotes the proportion between entries and how much space is effectively usable (maximum number of entries before a resize is triggered). All values calculated using a default load factor of 0.75 and default doubling resizing.

Max entries (e)	1500	1900	2300	2700	3000
Allocated space (S_a)	2048	4096	4096	4096	4096
Usable space (S_u)	1535	3071	3071	3071	3071
Load (l)	73.24%	46.39%	56.15%	65.92%	73.24%
Utilization (u)	97.72%	61.87%	74.89%	87.92%	97.69%

Table 3.2 demonstrates an example of this "universality of opportunity" with regards to HashMaps, noting possible memory savings in multiple situations. With a load factor of 0.75, a resize will occur as soon as 75% of all slots are occupied. As such, at any given point, *at least* 25% of the HashMap's footprint is unoccupied.

There are 3 possible situations for a given size (S_a represents the *current* allocated space, while

S_{a-1} denotes allocated space before the resize):

- $e < S_u$ (column 1): when the number of entries is lower than that of usable spaces, no resize has yet happened. In this situation, however, at least 25% of allocated space is unused.
- $S_{u-1} \leq e \leq S_{a-1}$ (column 2): the number of entries is sufficient to trigger a resize. However, it is lower than the pre-resize allocated space, meaning that if the load factor had been larger, no resize would have needed to occur. This is the most wasteful situation, as over half of the resized HashMap now consists of memory that was allocated unnecessarily.
- $S_{a-1} < e$ (column 3): since the number of entries exceeds the allocated space, a resize is mandatory. After the resize, however, this becomes equivalent to the first situation.

The following subsection will detail how each situation can be tackled to obtain memory savings, and the challenges and requirements for successful optimization.

3.3.1 Tuning algorithm

Two things are required to successfully use our version of data structure tuning. Firstly, we must accurately know the number of entries in the HashMap in standard/expected usage. Without this, we cannot tune the data structure to its usage, and simply setting the load factor to 1 would only degrade HashMap performance without any significant memory savings. While probability tells us all buckets should have an equal chance to receive a key/value pair, that is often not the case. Setting it to 1 would fill up every available bucket, practically guaranteeing collisions to happen. The second aspect is knowing *how* this number evolves over the program's runtime, as irregular usage may make the optimizations have a negative, rather than positive, effect. We can illustrate this by describing two ways to use a HashMap in a real-world application:

- *As a cache.* In a cache scenario, information is expected to grow at a relatively linear and constant pace. At first, the number of entries grows rapidly during the cache's initial warmup phase, and after some time stabilizes at a given number when all of the frequently-accessed data points have been cached. Additionally, in a cache situation entries are seldom removed, but rather updated or replaced if a new version of a particular data point exists. For our purposes, this tendency for stability allows for very safe tuning as we can set, for example, a certain value as the new initial size, and be confident it is unlikely to be exceeded;
- *As a burstable resource tracker.* In, for example, a ride-sharing application where available vehicles are stored in a HashMap, with the vehicle's grid square number as a key, and an object containing their information (precise location, vehicle details, driver details) as the value. We can expect the number of entries to vary frequently and strongly, decreasing sharply in times of high demand

(morning and afternoon rush hour), and growing equally sharply as the high demand period ends and the service is able to finalize all of the requests. In such a situation, the high volatility can at best prevent us from making any changes, or at worst cause ”optimizations” to have the opposite effect. Column 3 of Table 3.2 can help us numerically illustrate such a situation: with an average entry number of 2300, we set the HashMap’s initial size to 3067 (maintaining the 0.75 load factor), thereby reducing footprint by 25%. However, a peak of 2400 entries triggers a resize, doubling our HashMap’s footprint to 6134, an increase of 50% compared to if we had made no changes to the data structure.

The above two situations demonstrate why profiling using real-world input data is necessary, and the importance of accurately calculating the values obtained from said profiling. During the second compilation, once the profiling data is loaded and interpreted, it is used by the tuning decision algorithm to formulate which changes are necessary to obtain maximum gains. The algorithm itself is quite straightforward, due to the optimization itself being relatively trivial to perform, and requiring very little in terms of safety analysis. The modifications performed by the algorithm affect two properties of the HashMap: load factor and initial size, changing one or both of the values to attempt to minimize unused space.

Listing 3.2: Java-style pseudocode for the tuning algorithm.

```
1  public void structureTuning(graph, profiledMethods){
2
3      for(MapObject map: profiledMaps){
4          //Verifies map properties are default
5          if(map.maxEntries >= 12 && verifyDefaultValues()) {
6              //Ensures load is consistent and without major spikes
7              if(map.avgEntriesPerOp * 1.25 >= map.maxEntries){
8                  //Max load higher or equal to previous size, but within
9                  //load factor of current size
10                 if(map.maxEntries >= (map.allocatedSpace / 2)){
11                     map.initialCap = round(map.maxEntries / map.loadFactor);
12                     map.loadFactor = 0.8f;
13                     modifyMapProperties(graph, map);
14                 } else {
15                     //Max load higher than previous real capacity, but
16                     //lower than allocated size
17                     int newUsableSpace = min(
18                         max(Math.round(map.maxEntries * 1.05, map.maxEntries + 1)),
19                         map.allocatedSpace / 2);
20                     if (newUsableSpace == (map.allocatedSpace /2)){
21                         map.initialCap = round(newUsableSpace / map.loadFactor);
22                         map.loadFactor = 0.8f;
23                     } else {
24                     //New usable space is less than the previous allocated space.
25                         map.loadFactor = newUsableSpace / (map.allocatedSpace / 2);
26                     }
27                     modifyMapProperties(graph, map);
28                 }
29             }
30         }
31     }
32 }
```

3.3.2 Theoretical results

The theoretical footprint reduction for each method's best and worst case is not hard to calculate in a vacuum, and we have done so for each of the situations shown in Listing 3.2:

- $S_{a-1} \leq e < S_u$: The number of entries has exceeded the previous allocated space, and a resize was unavoidable (line 10 of Listing 3.2). However, since resizes double the allocated number of spaces, between 25 and 50% of currently allocated entries are unused. We can thus attempt to reduce the space occupied by the HashMap post-resize. While a profiling execution is ideally close to real usage, real-world usages will always vary by some degree. Simply setting the new initial capacity to *maxEntries + 1* and load factor to 1.0 would leave the optimized HashMap vulnerable to the slightest variation in entries, potentially leading to footprint increases when compared to the unoptimized version (Subsection 3.3.1's second example demonstrates how such a situation might happen). Thus, we calculate new values for initial capacity i_c and load factor l_f in the following manner (lines 11 and 12 of Listing 3.2):

$$i_c = \frac{\text{maxEntries}}{\text{default } l_f \text{ (0.75)}} = \text{maxEntries} \times \frac{4}{3}; l_f = 0.8$$

With these parameters, the HashMap will be allocated with a size already fitted to its profiled load (133% of the maximum number of entries). The slightly larger load factor gives our usable space an increase of just over 6.6% relative to maxEntries, providing a margin for variation between the profiled and real-world executions. Setting the initial capacity outright to its final value means a footprint that is larger at startup/declaration, but lower once usage stabilizes. We find this to be most fitting for applications such as the first example of Subsection 3.3.1, with fast initial growth followed by stabilization. Additionally, this removal of resizing operations can provide a small performance boost, ideally offsetting the slight performance deterioration from the increased load factor. The best result for this optimization occurs when the number of entries is closest to the pre-resize allocated space ($e \approx S_{a-1}$), and the worst when it is closest to the current usable space ($e \approx S_u$), with the following improvements:

$$\text{Best case } (e \approx S_{a-1})\text{: } \frac{e/l_f}{S_a} = \frac{e}{l_f \times S_a} = \frac{e}{0.75 \times 2 \times S_{a-1}} \approx \frac{S_{a-1}}{1.5 \times S_{a-1}} = \frac{2}{3}$$

$$\text{Worst case } (e \approx S_u)\text{: } \frac{e/l_f}{S_a} = \frac{e}{l_f \times S_a} \approx \frac{S_u}{l_f \times S_a} = \frac{S_a \times l_f}{l_f \times S_a} = 1$$

Thus, our best case scenario yields a footprint reduction of 33%, with the reduction asymptotically nearing zero as e approaches S_u.

- $S_{u-1} \leq e < S_{a-1}$: the number of entries is sufficient to trigger a resize. However, it is lower than the pre-resize allocated space, meaning that if the load factor had been larger, no resize would have needed to occur. Thus, we want to change the parameters so that no resize is triggered. For this, we calculate a new S_{u-1} with a 5% margin for variation (lines 17-19 of Listing 3.2):

  $$S_u = min(max(e \times 1.05, e + 1), S_{a-1})$$

  The *max* ensures there is always at least one slot for margin, while the *min* ensures the new S_{u-1} does not exceed S_{a-1}. If $S_{u-1} = S_{a-1}$, we would have to set the load factor to 1. This would greatly degrade HashMap performance, however, and as such we instead set the load factor and initial capacity in the same way as in the previous case, with the optimization result being the same as the best case of that situation (lines 20-22 of Listing 3.2). Outside of this edge case, $l_f = S_{u-1}/S_{a-1}$. As for the initial capacity, we can set it to S_{a-1} (eliminating resizes as described above) or leave it at default (allowing for smoother footprint growth over runtime). We chose to leave it at default to keep the footprint growth at early runtime closer to the unoptimized program. Since this optimization prevents a resize (which doubles the allocated space), the improvement is always (with exception to the aforementioned edge case) a footprint reduction of 50%.

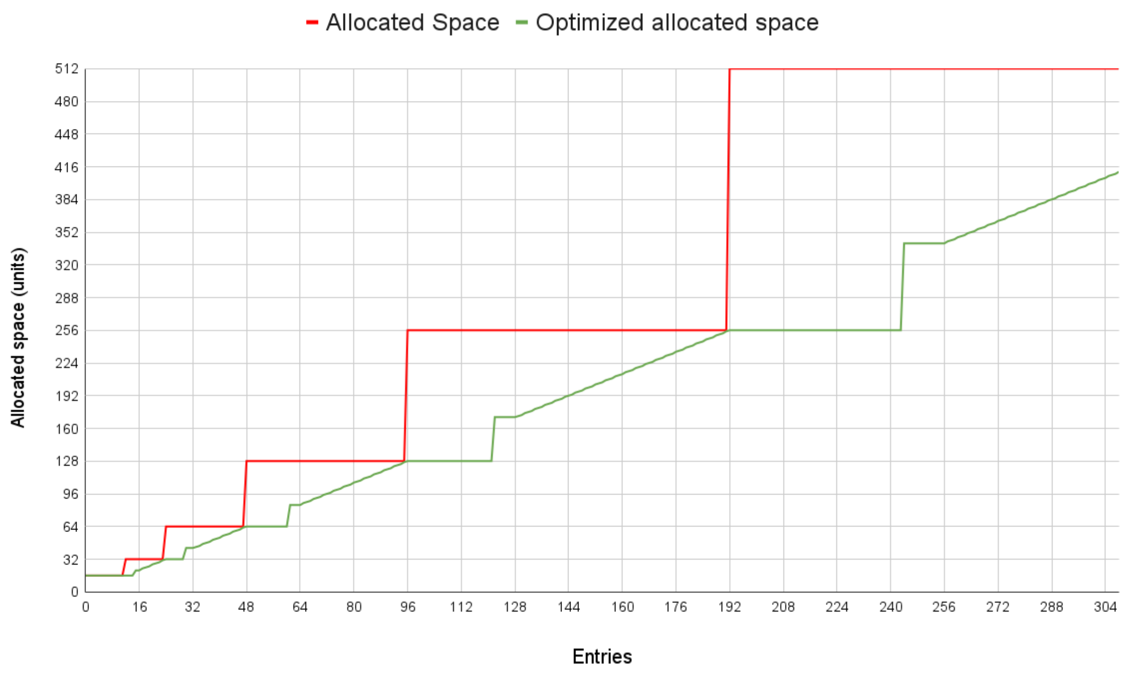


Figure 3.2: Evolution of allocated space (S_a), with (green) and without (red) our optimization as a function of the number of entries in a HashMap (e). The optimized line contains two repeating "sections": the first one (*e.g.* $e = [96..121]$) demonstrates our prevention of a resize while avoiding a load factor of 1 (average load factor for the entire graph is 0.827); while the second (*e.g.* $e = [122..191]$) "section" shows how we reduce the impact of an unavoidable resize. S_a has no unit, as this is a theoretical language-agnostic simulation.

We have shown that our approach can yield a theoretical maximum HashMap footprint reduction of 50%. Unfortunately, the large number of factors in any given program makes it extremely difficult to predict how our tuning will affect the program's overall footprint. Some of the factors are:

- *How much of the program's footprint is data, and how much of said data is stored in structures versus variables and other objects?* Depending on purpose and design, programs may or may not store large quantities of data. All else being equal, the amount of data stored in-memory by the program directly affects the gains obtained by our optimization. Distributed systems, for example, often exchange data through *Data Transfer Objects* (DTO). A microservice can receive a DTO, store it in a variable and perform actions on it, then send it to another service or client. Throughout the entirety of the exchange, all of the data contained in the object was not stored in any data structure, and thus excluded from our optimization.

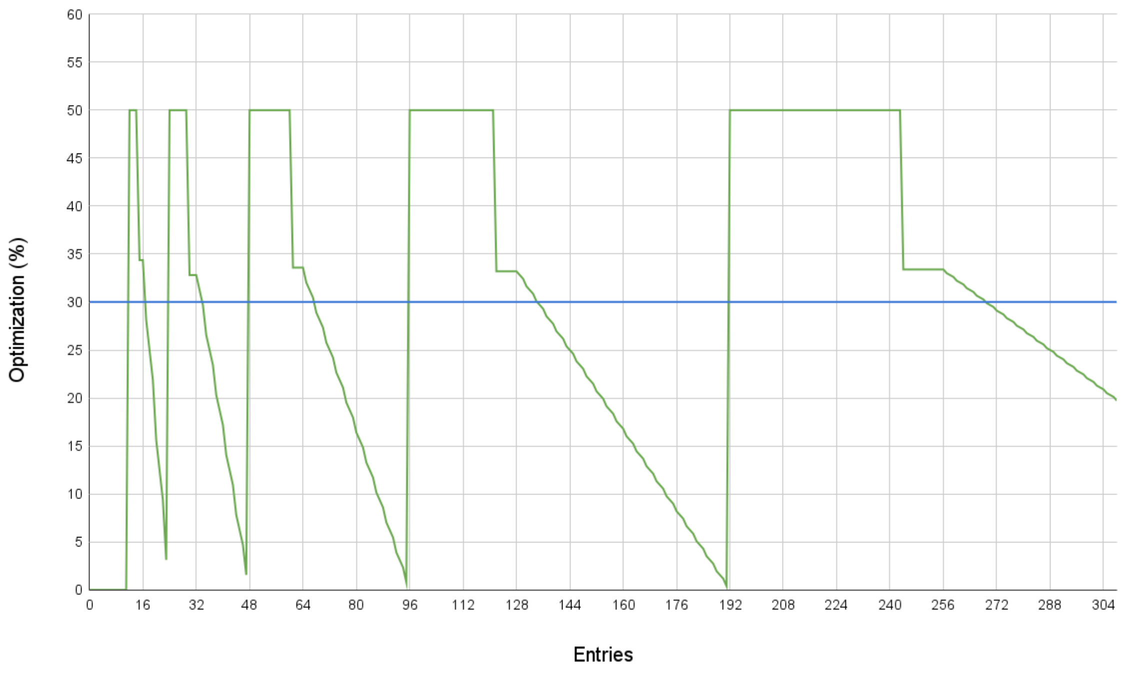


Figure 3.3: Theoretical HashMap footprint reduction as a function of the number of entries in a HashMap (e), represented by the green line. As explained above, when a resize is inevitable, the value of our footprint reduction linearly decreases as e grows towards S_u. On average, our optimization yields a theoretical footprint reduction of 30%, represented by the blue line.

- *Of the data structures, how many are of the type targeted by the algorithm?* Our optimization targets a small subset of all available data structure types. While these are very popular to use, they nonetheless represent only a fraction of all possible structures. However, this work's scope can be extended in future works to encompass many or even all data structures that follow a dynamic-resize approach to allocation.

- *What kind of data is stored in the targeted data structures, and in what quantities?* The amount and type of data stored in a data structure can heavily impact how much our optimization is ”felt” in the overall footprint. In Java, HashMaps do not store the keys and values themselves, but only references (pointers) to them. Thus, the space occupied by a HashMap of size 32 is only *32 (entries)* $\times$ *32/64(-bit)* (plus all of the headers and object information for the HashMap object), and our optimization reducing its size in half represents a saving of 16 pointer sizes, rather than 16 of the objects and keys it stores.

In practical terms, this means that we cannot make predictions for overall footprint reduction, as the results can vary drastically per the program's design and usage. The effect of this is shown in Chapter 5, where we provide a more detailed analysis of how our optimizations change the program's footprint.

4

Implementation

Contents

With regards to implementation, HashMaps in the Java language were the chosen data structures and language. Though some of the most complex data structures to tackle for optimization research, they are very relevant in today's programming. Java is an industry standard language, and as such allows for our work to achieve real world results that can be directly applied to current industry practices and use cases, rather than using a demo language with no direct real-world application. In addition, HashMaps are among the most common data structures, ensuring our optimizations apply to a wide range of programs, in contrast with those which target specific applications or workflows.

Oracle's GraalVM greatly facilitates both the development and execution of our algorithms, as it possesses a very complete Intermediate Representation, with an API and tools available to facilitate its viewing, interpretation and usage. As such, this IR allows us to more easily analyse the code and implement our tuning strategy, as we can integrate it into the remaining compiler optimizations performed by the compiler.

Additionally, the use of Native Image allows for something uncommon in Java: ahead-of-time compilation, compiling the whole program at once and creating an executable binary, much as other non-interpreted languages such as C do. Besides the faster startup and lower memory footprint this brings, it solves our challenge with class loading. Since the program is compiled ahead-of-time, all classes it uses are also loaded at once. This ensures that we will not have to deal with conflicts between the Java HashMap implementation and any other classes that may try to override it after we have performed our optimization. In addition, Native Image contains a multitude of tools for IR analysis and traversal, thus removing the need to develop a custom static analysis tool. During Native Image compilation, after the initial IR is generated, it is then modified through a number of phases. These phases serve to further the compilation process, through actions such as lowering (separating a given node, such as a method call, into several more low-level nodes, representing the contents of the called function), and to enact the numerous optimizations the compiler performs, such as constant propagation, loop peeling and unrolling and function inlining.

Our optimization was implemented as a new phase in this process rather than as an addition to an existing one. The main advantage of this is increased control and freedom for changes, since we don't have to ensure whatever operations we perform on the IR interfere with other changes in the same phase. In addition, this allows us to easily turn on or off individual features of our process, or even the entire optimization, using command-line options on the compiler executable. Our phase in inserted right after the first compiler phase, which generates the IR graph from the bytecode. At this point, the IR most closely represents the source code, as no operations such as inlining have yet to occur. Not only does this increase the accuracy of our analysis (calls that may be redundant in the unmodified code might be important for us to detect), but also the performance of the profiled execution. Placing our algorithm at this stage has several advantages:

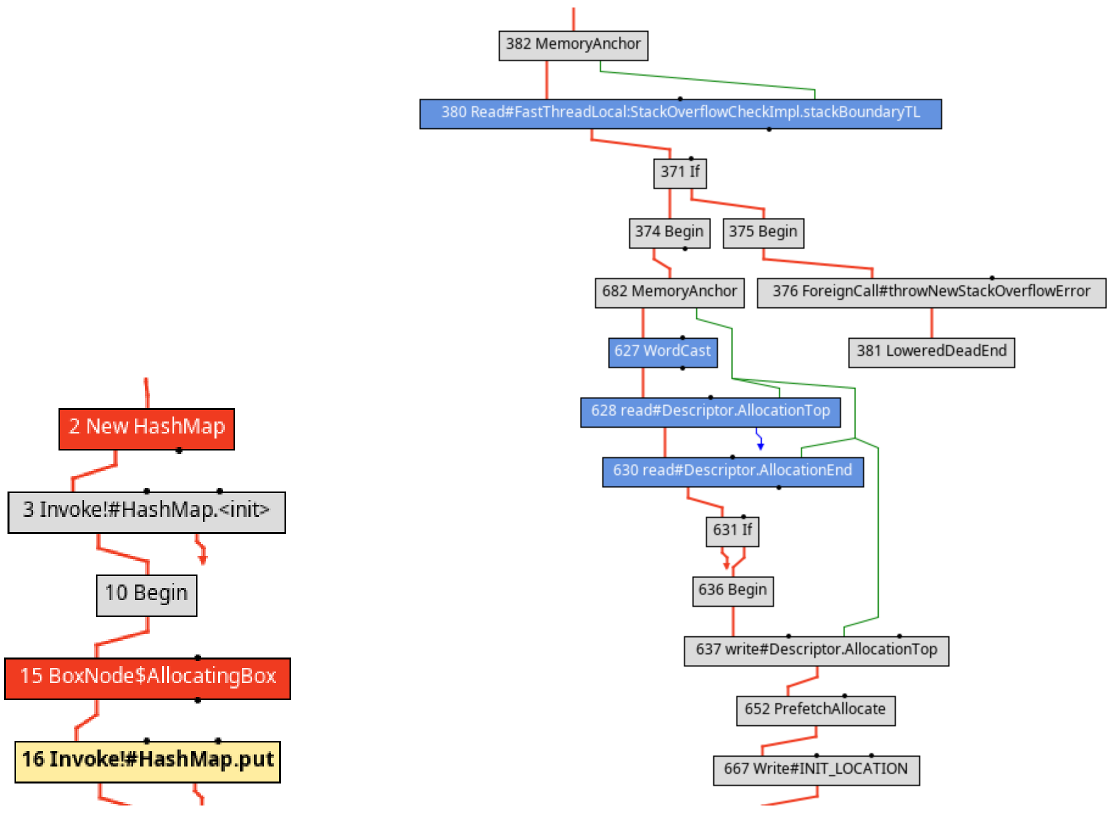
- The lowering level of the profiling code, inserted as foreign call nodes (representing calls to our methods), matches that of the other nodes in the graph, thus allowing us to insert only a single call node rather than having to create a sub-graph of nodes to match the level of the graph at a later phase.

- Tuning is greatly facilitated as well, as replacing a given call entails changing only one node rather than a group of nodes.

- After our operations, compilation proceeds as standard, allowing all of the Native Image optimizations to take place. Thus, not only does the source code receive the same optimizations it would have without our intervention, but our code is also subject to them, potentially increasing its performance compared to if it were introduced at a later phase.

A unique and independent IR is generated for each method of the target program. While this approach has some downsides (none of them of relevant to our work), it also has a significant advantage for us: since each method gets its own IR graph, we can easily restrict our optimization to apply to some methods and not others. In practice, this means we can modify the code written by the developer while leaving the standard Java implementation, along with any external libraries the developer chooses to use, untouched by our algorithm. While this means we leave some improvement ”on the table”, it also ensures we don’t interfere with the performance and functionality ”advertised” by these packages. In practice, we refuse to modify any code that isn’t provided by the developer.

Placing our work at a later phase would also make analysis far more difficult, as demonstrated in Figure 4.1. The IR graph in the early compilation stages closely resembles source code. Here, the `new HashMap<>(16, 0.75f)` call is represented as two nodes: a `NewInstanceNode`, containing information on the assignment and type of the data structure, and an `InvokeWithExceptionNode`, the information of which contains the setters for the `HashMap`’s initial capacity, load factor, and other information for the structure’s creation and allocation. To identify a new data structure has been declared, we need only look for occurrences of `NewInstanceNode`, and should we want to change this `HashMap`’s initial capacity and load factor values, we could just change the values stored in the `InvokeWithExceptionNode`’s arguments. Comparing this to the same code at the time of phase 45 demonstrates the impact phase placement has on our work’s complexity. At this phase, to identify a new structure was we would have to attempt to fully or partially match the graph to corresponding patterns. Besides being a far more computationally expensive process, it would introduce inaccuracy to our analysis, as the optimizations performed by previous phases could lead us to not find all relevant data structures, or ”detect” more data structures than there really are. The complexity of all other operations and their detection/modification increases proportionally as well. In essence, the difference in representation complexity between these two phases can be compared to the difference between Java source code and bytecode.



(a) Phase 0 **(b)** Phase 45

Figure 4.1: The same operation (instantiating a `new HashMap` and initializing the property fields) as represented after phase 0, and after phase 45. Note that some edges and floating nodes were removed for readability, and that the `init` method's representation in phase 45 is not fully shown in the image in the interest of space on the page.

4.1 Analysis and opportunity detection

Our detection algorithm is a direct implementation of the one shown in Listing 3.1. The first step is to verify whether the method is one we want to attempt to modify. Since each method gets its own IR graph, and the `StructuredGraph` object contains information such as the fully qualified name of the method it refers to, we can simply match it against a whitelist of acceptable methods/packages by using regular expressions. A whitelisting approach was chosen over a blacklisting one to prioritize guaranteeing compatibility/immutability of libraries and other implementations not authored by the developer. The next step is to verify the existence of a file containing information from a previous profiling run. This file is created by our profiling code, injected into the target program, and is deleted after the information it contains is used for the second compilation. Through deleting the file, we introduce some measure of protection against profiling data of previous code versions being used to inform optimizations of new versions. While a more robust safety "net" could have been implemented, we did not prioritize doing so for this work. If no profiling data is found, we can move on to detecting data structures compatible with our optimization, and inserting the respective profiling instructions. Given the direct representation of the IR at our phase, the analysis process is very straightforward: iterate through all nodes of the graph (which is relatively compact due to not having yet been unraveled (lowered) like the ones present in the latter phases), verify which `NewInstanceNode` nodes represent a desired data structure type (once again through a whitelist approach). The IDs of all such nodes are collected, and passed on to the profiling section.

Listing 4.1: Phase declaration and main entry point for our optimization. Some variable declarations were omitted for readability.

```java
public class DataStructReplPhase extends Phase {

    @Override
    protected void run(StructuredGraph graph){
        String dataPath = "ProfilingData/DataStructProfilingInfo.txt";
        File instancesFile = new File(dataPath);

        if (Pattern.matches(fileHandler.generateRegexPackageName(packageName),
                graph.method().getDeclaringClass().getName())) {
            //Check whether profiling data exists.
            if (!instancesFile.exists()) {
                //No profiling data. Will try to find tuning candidates
                //and append profiling instructions
                ArrayList<String> safeNodesIds =
                    analysisHandler.findCandidateNodes(graph, regexPackageName);
                profilingHandler.appendProfilerInfo(graph, safeNodesIds,
                        regexPackageName);
            } else {
                Map<Integer, ArrayList<MapObject>> profiledMethods =
                    fileHandler.readProfilingData(instancesFile);
                //Tuning main algorithm
                tuningHandler.structureTuning(graph, profiledMethods);
            }
        }
    }
}
```

Our work targets the HashMap family of data structures, namely the following implementations and interfaces: `Map, AbstractMap, HashMap, ConcurrentMap, ConcurrentHashMap`. While these are distinct data structures `NewInstanceNode` nodes represent all object allocation operations, regardless of the object's type, which is stored in the `NewInstanceNode`'s properties. Methods that interact with the data structure are stored as `InvokeWithExcceptionNode`, and also contain information on the type of object the interaction with which they represent. This allows for straightforward enlargement of the scope of our work to encompass many different types of data structures, should we or other future researches desire to do so. In theory, our work's general principles are compatible with all data structure that follow similar resize-style behaviors for dynamic sizing. On the implementation side, the work is extensive but reasonably simple: modify detection to include these new structures, obtain information on their workings and resizing policy, then add profiling methods to obtain data which will be used to inform changes to the structure's scaling factors (ones that play similar roles to the load factor and initial capacity in `HashMap`s.

4.2 Profiling

The first sweep of the IR graph detects all nodes representing `HashMap` declarations. Once this list has been created, a second iteration over the graph is performed, this time to insert profiling instructions. Placing these two steps in separate sweeps allows us to ensure all new instances have been accounted for and can be matched against to verify the profiling instructions are inserted in the correct place. The algorithm responsible iterates a single time through the graph's nodes, searching for specific nodes. When such a node is found, and the ID of the node or the node's target (if a method call) corresponds to one of the IDs received from the analysis step, the appropriate instructions are inserted. A `ForeignCallNode` is inserted into the IR graph, containing a reference to the profiling function to be executed, and arguments to it if necessary. These arguments contain the structure's ID, method and package (in a hashed format), and may also include the `HashMap`'s load factor and initial capacity values. Since we append entire functions and data structures into the compiled code, they behave exactly as if they were in the source code since the beginning. In practice, this means we can call methods directly onto the profiled data structures, such as `HashMap.size()`, which is very useful for accurate profiling, as we discuss further in this section.

IR graphs generated by Native Image are deterministic (two separate compilations of the same source code will generate equal graphs). This determinism isn't restricted only to shape, but also to the IDs of the nodes generated. The same `NewInstanceNode` will always have the same ID in all compilations (provided the source code does not change) While this consistency is greatly appreciated, it also creates some possibilities for problems: since node IDs are unique only in the context of their respective graph (two graphs representing distinct methods will have nodes (which may be entirely different) with the same ID). For the purposes of this work, this necessitates tracking not only the IDs of nodes we interact with, but also distinguishing which graph they belong to.

Two node types are considered: `NewInstanceNode` (representing the structure's declaration) and `InvokeWithExceptionNode` (containing calls to the structure's methods, such as `put`). Initialization instructions are appended to the `NewInstanceNode` that create a new data entry for the structure in the profiling database, which takes the form of two instances of `ConcurrentHashMap` in nested form. This is necessary for one reason: unlike the IR graphs during compilation, the profiling database is not unique for each class or method that contains a `HashMap`. Rather, the database is added and declared at the application's entry point, and shared between all profiling methods. In practice, we effectively insert the following call into the application's `main` class:

Listing 4.2: Declaration and allocation of the profiling information "database". Allocation of the nested `Map` entries occurs on an as-needed basis whenever the target `HashMap` is first accessed.

```java
Map<Integer, Map<Integer, MapCalculator>> methodProfilingMap
    = new ConcurrentHashMap<>();
```

This database, implemented as a `ConcurrentHashMap`, maps a set of `MapCalculator` objects (one for each `HashMap` being profiled, paired to its IR node ID as key) to a `HashCode` generated from the names of the package and method the `HashMap` was declared in. It is important that the IR node ID is kept throughout profiling, as it will later inform the tuning algorithm (the input of which is the same IR graph as was used for analysis and detection) as to which structure is to be modified. The `MapCalculator` object (shown in Listing 4.3) contains the values of all the metrics (as enumerated in Section 3.2) gathered throughout profiling, and performs the necessary calculations to ready the data for analysis by the tuning decision algorithm.

`InvokeWithExceptionNode` is used in the IR to represent all method calls onto the `HashMap`, and thus is the only other node type we append profiling instructions to. This node's properties contain the method target's ID, the method name, and the arguments passed to it. Depending on the method, we append different instructions immediately before or after the node in the IR control flow. Some methods, namely `put`, `remove`, `clear` and `init`, have custom profiling instructions that update the relevant metrics in the `MapCalculator` object. All other metrics use the same profiling function, which accesses the structure's `size` method at runtime, and updates the metrics accordingly. The necessity to use the live structure's size property rather than manually alter the metric values on each method call is precipitated by the existence of methods such as `putIfAbsent(key, value)`, which inserts a value into the map if an entry with the key doesn't already exist. Given this method's outcome depends on runtime data, intercepting the call and predicting its outcome would prove too computationally expensive and complex to program, and thus it is easier to retrieve the structure's `size` after the method completes. While there may be a loss of accuracy in the profiling data, this deviation will not be statistically significant given how frequently the metrics are updated with the structure's true size.

Listing 4.3: `MapCalculator` object, which stores all of our profiling information about a given `HashMap`. In the interest of space, the methods are presented in interface form, and getters/setters removed.

```java
public static class MapCalculator {

        private int entries;
        private int maxEntries;
        private int initialCap;
        private int resizes=0;
        private float loadFactor;
        private int average = 0;
        private Queue<Integer> history = new LinkedList<>();

        public void update(boolean addition);
        public void update(int size);
        public void updateAverage();
        public void mapClear();
        public void checkResize();
        public double getAllocatedSpace();
        public double getUtilization()
        public String export(int number, int mapId, int methodHash);
}
```

Throughout execution, our profiling methods collect a multitude of metrics about each `HashMap`, which are used to determine how the data structure is used, whether we should modify it, and what values to use in the modification. As shown in Listing 4.3, we keep track of the number of entries the `HashMap` holds at any given time, along with the all-time maximum number of entries. In conjunction with the average number of entries, we are able to track not only the number of entries, but also its evolution throughout the execution. By comparing the average, final and peak values, we can ascertain whether the `HashMap`'s usage profile is stable and without harsh entry fluctuations. In addition, a short-term first-in-first-out (FIFO) queue holds the last 10 values of entries, enabling a more accurate calculation of averages and evolution for small HashMaps where normal averages would not have time to stabilize and present a representative value.

In addition to the previously defined profiling methods, another instruction set is appended as a shutdown hook to the compiled program. These instructions will run whenever the program exits (gracefully or not), and are responsible for aggregating all the data collected during profiling, formatting it and exporting to a file, which will be read upon the next time the compiler is executed, supplying the tuning algorithm with all of the collected profiling data. Passing this information to the compiler in this indirect way is necessary due to the compiler not actively running during profiling. Additionally, this ensures the metrics passed are up-to-date and final. Since Java is a cross-platform language, we chose an equally cross-platform file type: the humble `.txt`, since it is a universally recognized file type independent of operating system or encoding. There are other data objects we could have chosen, such as JSON, but the universality of `.txt`, along with the added complexity of introducing/importing JSON parsing libraries into the source code, led us to reject those more advanced options.

The information contained in the file is exported through simple string concatenation and standard Java file writing, and loaded by the compiler using regular expressions and simple manually-implemented parsing. While overall simple, this approach is more than sufficient to store and recover the data we need while maintaining universal OS compatibility.

4.3 Tuning

Upon the start of our phase in the compilation process, we verify whether the file containing profiling data for the program exists. If it does not, safety analysis and profiling injection occurs as described previously. However, if such a file exists, the algorithm assumes it to be the correct one, reads it and uses the information for tuning. The data imported from the file is regenerated into a collection of `MapObject`s, which besides containing all of the gathered metrics corresponding to a given data structure, later also represents and stores the optimized load factor and initial capacity values. This *profile-or-tune* approach was implemented both to avoid simultaneous injection of profiling and structure modification, and to place both operations at the same phase of the compilation process, to take advantage of the possibilities enumerated in Section 4.1.

After all of the data contained in the file is read and processed, it is stored in a data structure, the layout of which is almost equal to the one presented in Listing 4.2, with the difference being it uses the previously explained `MapObject` objects, rather than `MapCalculator`s from profiling. The difference between the two centers around `MapObject`s not having to store all of the fields that `MapCalculator` does, and not needing all of the calculation methods the latter possesses. By design, each Native Image phase is executed independently and sequentially per method graph, and as such we cannot communicate between different instances of our phase which methods were already evaluated for tuning. While we could implement modifying the information file to remove the data of already optimized methods, this would have introduced a large quantity of complexity and additional failure points while providing a very small boost in terms of time and memory taken up by the optimization phase. As such, every time the phase runs it loads all (though this dataset is small for even real-world microservices such as Micronaut) profiling data, even that pertaining to methods other than the one currently represented by the graph.

Having finished recovering the data, we iterate over the graph in search of the relevant nodes, in particular `InvokeWithExceptionNode`s whose method call is the `HashMap.<init>` method. There are multiple variants of this method, differing by their number and type of arguments. Of interest to us is only one of them, being the same one we look for during the profiling portion, `HashMap.<init>(int initialCapacity, float loadFactor)`. This method, unlike something like `put`, is not explicitly called by the developer. Rather, it is implicitly called whenever the developer specifies these values during `HashMap` allocation.

Listing 4.4: Implicit calling of the `HashMap.<init>` method.

```
HashMap<String, Integer> cityPopulations = new HashMap<>();
/*K and V types are inferred from the ones set during declaration of the
cityPopulations variable, and the default values for initial capacity
and load factor are used. <init> is not implicitly called, as there is
no need to do so.*/

HashMap<String, Integer> cityPopulations = new HashMap<>(16, 0.75f);
/*K and V types are inferred from the ones set during declaration of the
cityPopulations variable, but values for initial capacity and load factor
have been specified by the developer. Even if they are the default falues,
<init> is implicitly called due to them having been specified.*/
```

`InvokeWithExceptionNode`s representing calls to the `HashMap.<init>` method are our main interest for tuning purposes. They contain the initial capacity and load factor values to be set in the `HashMap`, and additionally store the ID of the `NewInstanceNode` that represents the `HashMap`'s variable declaration and assignment. The information contained in this node is all we need to change to tune the data structure with our optimized values. However, creating such a node and inserting it into the graph in the right place if it doesn't already exist is a relatively complex operation to implement. Due to time restrictions, we were not able to do so. As such, the source code *must* specify the default values in the constructor (as exemplified in the second line of Listing 4.4) for our optimization to work. While this may seem at first to necessitate the developer changing their code to take advantage of our optimization (in a similar manner to the developer annotations described by Bruno et al. [4], this need not be the case. Using a basic text replacer on the source code (for cases when that is available), or using a bytecode manipulation tool such as Javassist [26] is sufficient to eliminate the need for the developer to change anything in their code.

Having found an `InvokeWithExceptionNode` of the correct method, ID, values and package/method name, we use the corresponding values stored in the profiling information to calculate which tuning approach we will take, and the values we need to change to implement it. In this regard, nothing has changed from the theoretical implementation detailed in Section 3.3, apart from the aforementioned safety validation to ensure we are modifying the correct node from the correct graph. Listing 4.5 demonstrates how our approach performs minimal changes to the developer's code, only changing the arguments for the implicit `HashMap.<init>` call. In this example, we have determined the maximum number of entries to be 200. Given the previous size for the `HashMap` was 256, we can prevent a resize from occurring, yielding a footprint reduction of 50%. As stated in Section 3.3, we elect to not change the initial capacity in this scenario, allowing the HashMap to grow throughout the execution rather than having it start already with its final size.

Listing 4.5: Original and tuned calls for the allocation of a new `HashMap` object.

```java
/*Code as written by the developer*/
HashMap<String, Integer> cityPopulations = new HashMap<>();


/*Allocation with tuned load factor as calculated by our algorithm.
Both arguments must be included in the call, even if one of them is
unchanged from the default value.*/
HashMap<String, Integer> cityPopulations = new HashMap<>(16, 0.8203125f);
```

No further changes to the node are required in terms of properties, as the control flow and remaining references remain unchanged and valid. This continuity both helps with debugging, and minimizes the risk of unwanted behavior or bugs being introduced into the IR or compilation process.

After the compilation process finishes, the file containing the profiling metrics is deleted. This was implemented as a security measure to provide a certain amount of protection against situations where the program's source code was modified and the metrics from a previous version used to perform tuning without correct profiling being done. The file's deletion forces a repeat of the compilation and profiling steps. Other security measures such as node type and id matching were also implemented. This is, however, not a perfect solution, as it only prevents the *reuse* of profiling data, rather than *first use* upon a modified source code. A proper code versioning/graph hash comparison functionality was designed, but unable to be implemented due to time restraints. Thus, it falls upon the developer to ensure correct utilization of the tool to avoid code mismatches.

5 Evaluation

Contents

This Chapter is dedicated to evaluating the system, and determining under which scenarios we obtain footprint and performance benefits. Besides presenting data points demonstrating our optimization's impact, we will perform some analysis on said data points and outline the conclusions that can be drawn from them.

5.1 Methodology

All testing was performed on a system running Manjaro Linux, kernel 5.15.85-1, equipped with an i5-8600K CPU, overclocked to 5.0GHz all-core and 16Gb of 2400MHz DDR4 RAM. Background and foreground applications were limited to a minimum, and each test execution was performed a minimum of three times, discarding extreme outliers. For each benchmark, the program was compiled in its standard form (using `javac` or `mvn` as defined by the authors), then an executable was compiled once using Native Image with all features related to this work disabled (referred to as vanilla). This represents our base (unoptimized) program. Memory usage was measured using `tstime` [27], a C-based non-polling tool that uses the `taskstructs` API of the Linux kernel to obtain the high-water RSS (Resident Set Size) usage of the program. The same tool also reports process time with values for the real, user and system time, and the last value was used to monitor the performance impact of our optimizations, as it effectively measures the CPU time taken up by the program and is as such the most consistent. The process for executing a benchmark or series of benchmarks is as follows:

1. Remove all compilation artifacts and profiling data from the previous run.
2. Run the vanilla Native Image executable under `tstime`, obtaining the unoptimized RSS. Arguments/usage of the program is the same as the profiling and optimized executions of the executable.
3. Perform the first compilation of the program with our modified Native Image, appending profiling instructions.
4. Run the profiled Native Image executable to generate profiling data.
5. Perform the second compilation of the program with our modified Native Image, using the profiling data generated in the previous step to perform data structure tuning.
6. Run the optimized Native Image executable under `tstime`, obtaining the optimized RSS.
7. Compare values to previous runs if applicable to filter outliers, and record results.

The main focus of our testing was a custom micro-benchmark, containing a single `HashMap<Integer, Integer>`, inserting a certain number of entries (set by a command line argument) with a sequential key and randomly calculated value, printing afterwards the number of entries in the `HashMap`. The number of entries, random values and printing were implemented to prevent the compiler from optimizing the program further though methods such as inlining or loop unraveling. Two sequences of values were used with this micro-benchmark, based on the theoretical best ($e = S_{a-1} \times 0.8$) and worst ($e = S_{a-1} + 1$) case scenarios detailed in Subsection 3.3.2.

In addition, we used the Shopcart Micronaut-based microservice, to test the compatibility and capability of our optimization to work on programs representative of real-world use cases. Like our micro-benchmark, the measurements were taken with `tstime`, while using the `wrk2` [28] tool to send several thousand HTTP requests to the service.

5.2 Results

On average, our profiling-step Native Image compilation of our custom micro-benchmark took 17.180 seconds and the optimization-step 17.468 seconds, as compared to the average of 17.168 seconds for the vanilla Native Image. These results demonstrate that our added phase has very minimal effect on individual compilation time. However, our process requires two Native Image compilations and one profiling run. Taking this into account, our total time until an optimized executable is produced represents an increase of, at least, 100% (plus the variable profiling time) in source-to-executable time when compared to the vanilla compilation. However, we do not believe this to be a significant downside of our optimization due to it being an increase in ahead-of-time compilation time. User-facing applications are not generally compiled on the user's computer, but rather distributed as pre-packaged binaries where the user downloads the one matching their system's architecture. Similarly, cloud applications are not usually deployed in such a way that the application's current version is unavailable while the new version is being compiled/built. Additionally, all profiler-guided optimizations (PGO) require two compilation rounds, and many are already in use in deployment pipelines. Our optimization can be integrated into existing PGO compilation rounds, leading to a minimal overhead from adding it to the process. As such, we believe the increased compilation time to not be a significant factor in the vast majority of cases.

As previously stated, for our micro-benchmark we tested two cases, representing best and worse case scenarios. In the best case scenario we observed a large spike in footprint reduction for the last two test values, with the next increment (not present in the table) returning to a more "normal" value of 1.6%. Though we find ourselves unable to explain the reason for this spike, it has been consistent through numerous executions, and as such was included as a valid result. Assuming a more "fitting" reduction of 3%, the average footprint reduction would be 2.091%, 40.8% better than the reduction obtained for the worst case scenario. This proportion conforms to our theoretical results, where the best case results in a 50% reduction, and a value of 33% for the worst case.

Table 5.1: Best case results, $e = S_{a-1} \times 0.8$. All memory values are in kb. Average RSS reduction: 3,950%.

Entries (e)	Vanilla RSS	Tuned RSS	Optimization (%)	S_{a-1}	S_{a-1}	Resizes
26	7871	7735	1.727	32	24	1
51	7941	7727	2,694	64	48	2
102	7869	7616	3,215	128	96	3
205	7853	7768	1,082	256	192	4
410	7819	7863	-0,562	512	384	5
819	8057	7844	2,643	1024	768	6
1638	8292	8129	1,965	2048	1536	7
3277	8649	8556	1,075	4096	3072	8
6554	9165	9119	0,501	8192	6144	9
13107	10379	10063	3,044	16384	12288	10
26214	12876	12669	1,607	32768	24576	11
52429	18159	17660	2,747	65536	49152	12
104858	28916	27760	3,997	131072	98304	13
209715	50201	48096	4,193	262144	196608	14
419430	97736	97712	0,024	524288	393216	15
838861	180079	180156	-0,042	1048576	786432	16
1677722	366256	352192	3,839	2097152	1572864	17
3355443	698740	488826	30,041	4194304	3145728	18
6710886	1982016	1758793	11,262	8388608	6291456	19

As predicted in our conclusions regarding the theoretical results discussed in Subsection 3.3.2, the real values measured are an order of magnitude lower than the ones predicted. While most of the data in a program resides in data structures, most of their footprint comes from the data they contain, not the structure itself (which is mostly composed of 32 or 64-bit pointers). In Java 8, a single empty string takes up 32 bytes. A single key and value pair of empty strings take up 64 bytes, or the same as 8 64-bit pointers, 25% of the 32 pointers allocated for entries on a default map of 16. As such, while we can measurably reduce the space occupied by the `HashMap` itself, the improvement in terms of overall program footprint may prove small. This can be exemplified by comparing two lines from the best case run of the micro-benchmark: at 51 entries, the entire program occupied 7941 kb, growing to 8057 kb at 819 entries. Despite the number of entries being around 16 times larger, the program's footprint only grew by 1.46%. Only at higher numbers of entries do we see the program's footprint growing (almost doubling) at a similar rate to the number of entries, as in those cases the sheer quantity of data is significant enough in proportion to the non-data portion of the footprint. Despite this, we were still able to obtain measurable footprint improvements in most test cases, and appending profiling instructions to the optimized program shows the tuning algorithm was successful in improving footprint of individual structures and preventing resizes where possible.

In addition to memory footprint, we also measured execution time for both micro-benchmark scenarios, through the system time value reported by *tstime*. For low numbers of entries, the micro-benchmark

Table 5.2: Worst case results, $e = S_{a-1} + 1$. All memory values are in kb. Average RSS reduction: 1,486%.

Entries (e)	Vanilla RSS	Tuned RSS	Optimization (%)	S_{a-1}	S_{a-1}	Resizes
17	7260	7240	0,275	16	12	0
33	7943	7788	1,951	32	24	1
65	7752	7632	1,547	64	48	2
129	8056	7788	3,326	128	96	3
257	7636	7480	2,042	256	192	4
513	7480	7344	1,818	512	384	5
1025	8036	7736	3,733	1024	768	6
2049	8456	8068	4,588	2048	1536	7
4097	7988	7852	1,702	4096	3072	8
8193	9496	9396	1,053	8192	6144	9
16385	9312	9200	1,202	16384	12288	10
32769	11676	11312	3,117	32768	24576	11
65537	21528	21008	2,415	65536	49152	12
131073	36156	35968	0,519	131072	98304	13
262145	64796	64664	0,203	262144	196608	14
524289	99268	99200	0,068	524288	393216	15
1048577	191252	191176	0,039	1048576	786432	16
2097153	435272	434944	0,075	2097152	1572864	17
4194305	862532	862280	0,029	4194304	3145728	18
8388609	1653788	1653492	0,017	8388608	6291456	19

ran so quickly that `tstime` reported a value of either 0 or 3 milliseconds for both optimized and unoptimized executables. As such, we deem the results for low numbers of entries to be inconclusive as the variance is extremely small. However, execution time starts to grow exponentially once the number of entries reaches 6 digits, giving us measurably different and consistent times. Figure 5.1 shows both optimized and unoptimized execution times for the worst case micro-benchmark series, as this series reaches higher numbers of entries and as such produces more consistent results. Given that only at 65537 entries did we consistently stop obtaining 0 milliseconds on either one of the executions, we chose to calculate our improvements only from this value forwards, as the previous ones were within reporting margin of error. As such, we obtained a median time reduction of 46.178%. While we did not explicitly target performance improvements, our optimization either eliminates or heavily reduces the number of resize operations, which are not only costly but also increase in cost in the same way that footprint increases (since each resize doubles the `HashMap`'s size, there are twice as many allocation and rehashing operations to perform). However, this performance improvement scales heavily with the number of resizes, and as such only obtains significant improvements for applications with large amounts of resize operations (be it through many smaller `HashMap`s, or a singular massive one). These results demonstrate that, at the very least, our optimizations incur no noticeable performance penalties as a trade-off for the footprint reduction in the use cases we tested.

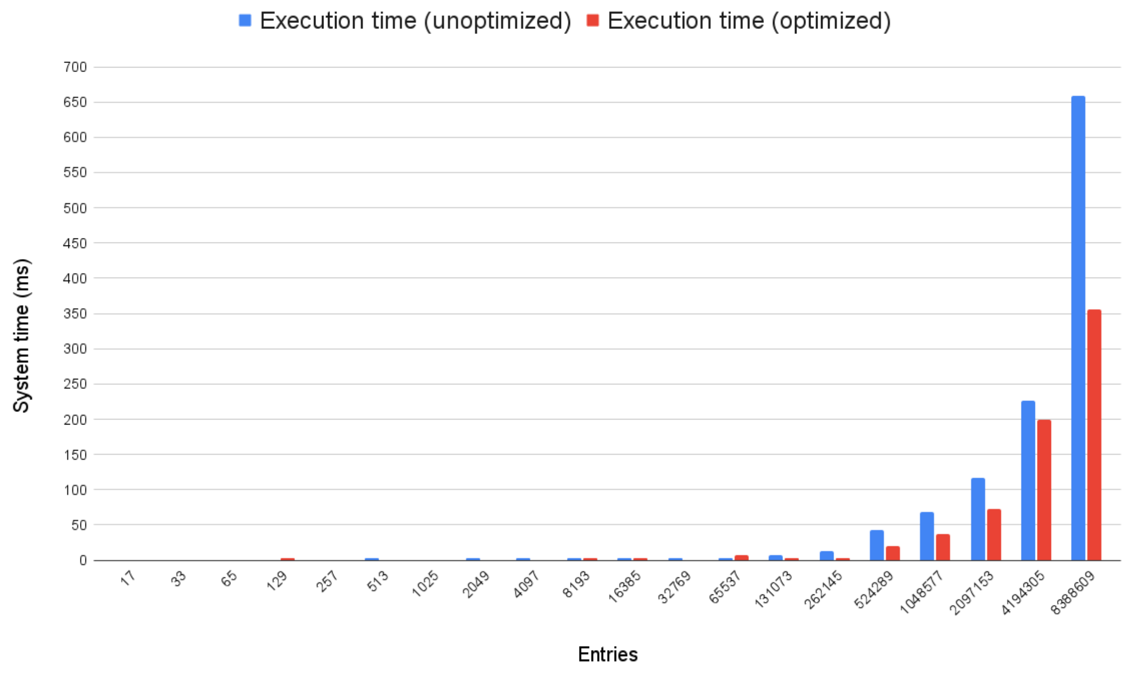


Figure 5.1: Comparison of optimized and unoptimized execution time for our micro-benchmark.

Additional testing was performed on the Micronaut-based Shopcart application, where we observed average RSS values of 162701 kb for the base version, and 159028 kb for the tuned execution, translating to an average footprint reduction of 2.258%. These results are consistent with the ones we obtained with the custom micro-benchmark, and confirm our solution's applicability outside of it. Micronaut generates a large amount of code during building, code over which the developer has no control over. By successfully analysing, profiling and tuning this application's automatically-generated data structures, we prove our optimization's ability to be applied onto real-world programs with large quantities of user and software-generated classes and data structures.

As such, we consider the results obtained positive, and the topic worthy of further research, especially for the enterprise world, given how a 2-4% footprint reduction per program/instance can quickly compound into significant memory savings in the context of distributed or serverless applications.

6 Conclusion

Contents

6.1 Conclusions

In this work we propose a new data structure optimization algorithm based on profiling-guided tuning for use with GraalVM Native Image, capable of memory footprint savings of up to 30%. Along with the algorithm itself, we believe our major contribution to be its universality through integration with Native Image. Given it makes no changes to the structure, class, methods or any other sensitive characteristics of HashMaps, our work is safe to use for almost all Java applications. Additionally, this universality in approach and focus on a ”never-worse” heuristic in regards to optimization enable it to be integrated into standard compilation workflows without fear of potentially compromising performance or increasing footprint. While Native Image is not as popular as most JIT JVMs, reducing our reach, we believe our work has great synergy with AOT compilation, as Native Image's mission to extract as much performance out of Java as possible matches our approach. In addition, growing adoption of Native Image and pursuit of performance gains allow us a positive outlook on the impact of this work on Java optimization and use.

6.2 Future Work

While our work shows very promising results, throughout the development and evaluation process we have identified several possibilities for improvement of our existing methods, and expansion towards tuning of other data structures. Among these improvement opportunities, enhanced profiling stands out as the most important one. Currently, we do not have custom profiling code for all `HashMap` methods, but only `<init>`, `put`, `remove` and `clear`. All other methods that access the `HashMap` are profiled with the same instructions that take advantage of how well Native Image allows us to integrate our profiling code into the developer's, making ample use of the `HashMap.size` method to effectively ”poll” the HashMap about how many entries it possesses after the profiled method finishes. While this is an accurate approach, especially for long-runtime applications where eventual profiling miscalculations get ”smoothed out” by the number of accesses, we feel more accurate profiling could help us perform tuning that is even closer to the application's usage patterns, further reducing footprint. Additionally, all of these calls to `size` incur a performance overhead on the profiled execution, ever slightly increasing the time it takes to obtain an optimized executable. As a future improvement, we would like to increase the granularity of our profiling, creating custom profiling code for more methods. The second improvement opportunity relates to performance: increasing the load factor may lead to a decrease in performance by increasing the chance of collisions. Given how HashMaps prioritize access speed over memory efficiency by design, we feel it is not worthwhile to sacrifice performance in favor of footprint on a performance-focused data structure. Here, too, can more accurate profiling be useful: with higher profiling precision, we could identify which data structures are more likely to suffer from high rates of collision, and implement our tuning algorithm in a way that either attempts to alleviate this collision problem, or avoids performing tuning on such data structures so as not to deteriorate the performance even further.

However, our future work also opens new possibilities for optimizations that are uniquely suited to our approach: by only modifying certain initialization arguments, our solution is applicable to a wide variety of HashMap use cases where solutions such as inlining would perhaps be too complex to implement. HashMaps are not the only data structures to take advantage of a dynamic resizing system, and future work could focus on expanding our technique to apply to all data structures that follow this kind of dynamic resizing ”paradigm”. While we were able to obtain footprint reductions, a HashMap is still one of many data structures in any large application. By extending our optimization techniques towards other data structures, we could potentially achieve a situation where every data structure in an application is tuned to some degree, combining into large footprint savings across the board.

In addition, at an early point in our development of this work, we investigated another technique for optimizing HashMaps and other data structures: data structure replacement. Rather than tuning the existing data structure to ”behave” better, this optimization method entails completely replacing the inefficient data structure with one that’s better suited for the predicted workload, and greatly synergizes with ours: a HashMap with very few entries (that will never trigger a resize) may be one to ignore from our side, but a perfect candidate to be replaced by an `EconomicHashMap`, a structure that is far better suited to storing small numbers of entries. While the standard HashMap takes a ”one-size-fits-all” approach, with fixed sizes, resize factors and hashing functions and structures, `EconomicHashMap` tends to take a more ”fitted” approach, by changing the hashing function (if all the hash positions can be stored on 32-bit shorts, why use full 64-bit integers?) depending on the number of elements (or not having one at all when their number is low enough) and varying resizes by the current structure size, among other techniques.

Since each approach addresses the shortcomings of the other, these techniques may prove to be a very powerful combination, and as such we believe data structure replacement is a theme that should be studied with great attention.

Bibliography

[1] Cboe Exchange, Inc. Historical Market Volume Data, October 2022. URL https://www.cboe.com/us/equities/market_statistics/historical_market_volume/. Accessed 23-October-2022.

[2] Chris Lattner and Vikram S Adve. Transparent pointer compression for linked data structures. In *Proceedings of the 2005 workshop on Memory system performance*, pages 24–35, 2005.

[3] Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed. *Fundamentals of data structures*, volume 20. Computer science press Potomac, MD, 1976.

[4] Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. Compiler-assisted object inlining with value fields. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 128–141, 2021.

[5] Oracle. GraalVM, April 2023. URL https://www.graalvm.org/. Accessed 24-April-2023.

[6] Oracle. Native Image - GraalVM reference manual, April 2023. URL https://www.graalvm.org/22.0/reference-manual/native-image/. Accessed 24-April-2023.

[7] Raymond Reiter. On closed world data bases. In *Readings in artificial intelligence*, pages 119–140. Elsevier, 1981.

[8] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal ir: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, pages 1–9, 2013.

[9] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Visualization of program dependence graphs. In *Compiler Construction: 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 17*, pages 193–196. Springer, 2008.

[10] Christopher Arthur Lattner. *Macroscopic data structure analysis and optimization*. University of Illinois at Urbana-Champaign, 2005.

[11] 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*, pages 187–196, 1993.

[12] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. *ACM Sigplan Notices*, 44(6):465–478, 2009.

[13] Dibakar Gope and Mikko H Lipasti. Hash map inlining. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 235–246, 2016.

[14] Tobias Pape, Carl Friedrich Bolz, and Robert Hirschfeld. Adaptive just-in-time value class optimization: transparent data structure inlining for fast execution. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1970–1977, 2015.

[15] Michael G Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J Serrano, Vugranam C Sreedhar, Harini Srinivasan, and John Whaley. The jalapeno dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, 1999.

[16] Joep L. W. Kessels. On-the-fly optimization of data structures. *Communications of the ACM*, 26 (11):895–901, 1983.

[17] PCDiga. PCDiga, April 2023. URL `https://www.pcdiga.com/`. Portuguese computer store, Accessed 24-April-2023.

[18] Changhee Jung and Nathan Clark. Ddt: design and evaluation of a dynamic program analysis for optimizing data structure usage. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 56–66, 2009.

[19] Christian Wimmer and Hanspeter Mössenböck. Automatic array inlining in java virtual machines. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 14–23, 2008.

[20] Changhee Jung, Silvius Rus, Brian P Railing, Nathan Clark, and Santosh Pande. Brainy: Effective selection of data structures. *ACM SIGPLAN Notices*, 46(6):86–97, 2011.

[21] Julian Dolby. Automatic inline allocation of objects. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming language design and implementation*, pages 7–17, 1997.

[22] Julian Dolby and Andrew Chien. An automatic object inlining optimization and its evaluation. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 345–357, 2000.

[23] Youtao Zhang and Rajiv Gupta. Data compression transformations for dynamically allocated data structures. *CC*, 2:14–28, 2002.

[24] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, pages 1–10, 2013.

[25] Matthias Grimmer, Stefan Marr, Mario Kahlhofer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. Applying optimizations for dynamically-typed languages to java. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, pages 12–22, 2017.

[26] Chiba, Shigeru. Javassist, April 2003. URL https://www.javassist.org/. Accessed 26-May-2023.

[27] Sauthoff, Georg. Tstime, January 2009. URL https://github.com/gsauthof/tstime. Measure highwater RSS memory usage of processes using the taskstats API, Accessed 26-May-2023.

[28] Tene, Gil. wrk2, March 2012. URL https://github.com/giltene/wrk2. a HTTP benchmarking tool based mostly on wrk.