Improving Virtual Machines Using String Deduplication and Internal Object Pools

by

Konstantin Nasartschuk

Master of Science, BRSU, 2013,
Master of Computer Science, UNB, 2013,
Bachelor of Science, BRSU, 2010

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Kenneth B. Kent, PhD, Computer Science
Examining Board:	David Bremner, PhD, Computer Science, Chair
	Michael W. Fleming, PhD, Computer Science,
	Mary Kaye, PhD, Electrical and Computer Engineering
External Examiner:	Karim Ali, PhD, Assistant Professor
	Dept. of Computing Science, University of Alberta

This dissertation is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

February, 2019

Abstract

The efficiency of memory management is one of the key metrics when researching virtual machines. In cases where deallocation of objects is performed automatically, garbage collection has become an important field of research. It aims at speeding up and optimizing the execution of applications written in languages such as Java, C#, Python and others. Even though garbage collection techniques have become more sophisticated, automatic memory management is still far from being optimal. Garbage collection techniques such as mark sweep, mark compact, copying collection, and generational garbage collection form the base of most virtual environments. These algorithms rely on a stop-the-world phase that is used to detect and free live objects.

The research presented in this dissertation aims at improving automatic memory management by investigating the optimization of memory layout as well as optimizing the allocation and deallocation processes of frequently created and freed objects. The first optimization aims at using the stop-the-world phase of the garbage collector in order to detect duplicate strings

and deduplicate them before copying them to a new region. The goal of this algorithm is to reduce multiple storage of the same data in memory, as well as copying of memory, in order to decrease the heap size and therefore the number of garbage collections required to execute the client application.

The second optimization aims at speeding up the allocation of frequently created and discarded objects by keeping a pool of empty objects. Instead of requesting new memory, the virtual machine requests an empty object of the class and initializes the values required. Object pools are a widely used software engineering pattern utilized by software developers to reuse object instances without the need of repeated allocation and instantiation. While the benefits of using object pool structures are still present when used in a garbage collected environment, it adds a memory management component to the development process. The dissertation investigates the feasibility of introducing automatically created and maintained object pools for predefined classes. Automatic object pools are implemented and discussed using the GenCon GC and Balanced GC policies of the IBM Java VM.

Acknowledgements

In this section, I would like to acknowledge the people around me who made this work possible. I would like to thank my parents, Wassili and Tatjana Nasartschuk as well as my sister Katharina Lambertz for their unwavering support, their substantial financial contribution to help me pursue my dreams and the never failing words of encouragement and trust in my abilities.

To my wife Krista Nasartschuk a great bow for the patience and support on all fronts conquering the big and the little foes. I am sure the experiences leading to this point will give us everything we need to enjoy more of our adventures that life might place in our way. You made this path not mine — you made it ours. You shared every worry and every smile with me and I still have to come up with a way to repay you for it. A great thank you also to my parents-in-law, Wilma and David Pike, who were always on our side and ready to share the difficulties and happy moments in our life. It is a great honour to be able to call you family.

The journey over the ocean to North America would never be possible without my friend Marcel Dombrowski. Thank you for always being there to

listen, complain about the world together and also just share a beer, when the moment "seems right".

I had the pleasure to be surrounded with the best lab mates one can imagine: Azden Bierbrauer, Devarghya "Dev" Bhattacharya, Taees Eimouri, Nicolas Neu, Federico Sogaro, Panagiotis "Panos" Patros, Maria Patrou and Vicky Wong. You all made this journey complete. Thank you for the discussions, the silly times in the lab and the camaraderie which you showed towards me throughout those years.

Thank you Stephen MacKay, for thousands of typos, grammar mistakes, and layout optimizations, that you found in my publication drafts, presentations, and disclosures. I learned a lot from every single one of them and maybe one day, I might even understand the difference between "that" and "which".

There are always people who are helping in the background. People without whom every signature, every application and every question about the administrative part of the university would take ten times longer to solve. In case of every UNB CS student, the names of those lifesavers are Jodi O'Neill, Jennifer Caissie, and Candace Currie. You saved me many times and helped me get through the jungle of papers and departments at UNB. Thank you!

I would also like to thank Active Fredericton and my ultimate frisbee team, the Ultimate Warriors, for giving me a way to balance the work with some exercise and social contact outside of university. A thank you to my friends Malte Borkenhagen and Maxim Egorov, who made the trip from Europe to come visit and support me. The stories of those visits are still being told and

are on their way to become legends.

I had the pleasure to be supervised and mentored by Kenneth B. Kent, who guided me through my academic career. Ken, you introduced me to Canadian culture and beer. You gave me advice and guidance not just in university matters but had an open ear to all the worries I could possibly have. Thank you for pushing me to do better when it was required and letting me have some space when I needed it most.

This research was conducted within the Centre for Advanced Studies-Atlantic, Faculty of Computer Science, University of New Brunswick. I am grateful for the colleagues and facilities of CAS Atlantic in supporting my research. The author would like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program. In addition, the author would like to thank the Natural Sciences and Engineering Research Council (NSERC) for funding support. Furthermore, we would also like to thank the New Brunswick Innovation Foundation (NBIF) for contributing to this project.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the last few decades, memory management of applications went through a number of iterations in order to automate the process and make it more efficient. Traditional languages such as C and C++ rely on developers to allocate objects on the heap when they are created and free objects that become no longer needed.

As stated by Jones et al. [32], 20% of the development time in a project is usually used for memory management. This amount of work led to attempts to automate the process to allow developers to focus their resources on the core application development and addition of functionality.

Garbage collection techniques, such as reference counting [9], mark sweep [44], mark compact [58] or copying [7, 20], are commonly used in automatic memory management. The most known and common languages that rely on garbage collection include Java, C#, Python, and Ruby. Languages such

as C++ allow developers the use of smart pointers, which wrap a pointer structure and call the destructor of a class according to the implementation of the wrapper. This can be based on reference count, exception handling or others [17].

One of the main disadvantages of automatic memory management is its inefficiency. Most garbage collection techniques rely on a stop-the-world phase in which the program is halted in order to clean up memory. This can be critical for the application's runtime. In addition, the heap size used by the application is almost never as small as it would be when managed by developers assuming no memory leaks are present. Attempts to close the gap between automatic and manual memory management techniques aim to improve metrics such as the number of garbage collections, stop-the-world time as well as to optimize the execution of the client application by improving caching, locking, concurrency, and other aspects of the environment.

This dissertation presents research performed on the heap structure and the allocation behaviour of virtual environments with the hypothesis that memory management in virtual environments, and more specifically the object structure and garbage collection, still leaves room for improvement. String deduplication aims at using the properties of string objects within the virtual machine in order to combine duplicates and reduce the overall memory footprint as well as the memory allocated by the client application. Strings and string related objects occupy a large portion of the Java heap and are therefore a large target for optimizations. A reduction in memory use can

potentially lead to fewer garbage collections and therefore a reduction in time used for memory management.

Further, the dissertation investigates automatically applying object pools, a frequently used software engineering pattern. The research applies the technique to the client application running in the virtual environment (mutator) without changing its structure to speed up allocation times and the overall application performance.

The dissertation is divided into seven chapters. Chapter 2 provides an introduction to automatic memory management techniques and discusses their advantages and disadvantages. Chapter 3 describes the investigation of a string deduplication approach aimed at saving memory by reusing existing objects. The approach is further enhanced and evaluated in Chapter 4. Chapter 5 includes a feasibility study of an automated object pool pattern, which reuses old objects instead of creating new ones, to speed up the allocation process. Contents of the three main Chapters 3, 4 and 5 were published at peer reviewed conferences as [48], [50], and [49] respectively. Finally, the dissertation is concluded in Chapters 6 and 7 with a short summary of the results and potential future work involving the work presented is listed.

Chapter 2

Background

Locating and freeing unused objects is one of the major challenges of automatic memory management systems. Five collection techniques have become the basics of garbage collection: reference counting [9], mark and sweep [44], mark and compact [58], and copying collection [7, 20]. This chapter introduces the techniques mentioned. Their advantages and drawbacks are discussed and an overview of how they are currently used in virtual machines is provided. The chapter concludes with an overview of how collection techniques are combined and objects located on the heap are structured and organized in virtual machines.

2.1 IBM J9 Virtual Machine (JVM)

The IBM J9 virtual machine (JVM) is a virtual environment initially created to execute bytecode created by compiling Smalltalk applications. When Java became popular, the virtual environment was changed to handle Java bytecode instead [60].

The virtual machine consists of a number of components including a JVM Application Programming Interface (API), a diagnostic component, memory management capabilities, a class loader, an interpreter and a platform port layer. The structure of the virtual machine is shown in Figure 2.1

While influencing all parts of the virtual machine, the main focus of changes and optimization in this dissertation is to the memory management component of the virtual environment. Memory management in a virtual environment can simplistically be divided into two main parts: the allocator and the garbage collector. The J9 virtual machine offers a number of garbage collection (GC) techniques such as the generational concurrent policy *GenCon*, the region based policy *Balanced*, the real time guarantee capable policy *metronome*, and the mark-compact capable policy *optthroughput* [30].

At the start of the research presented in this dissertation, the application source code was not publicly available. This changed with the release of Eclipse OpenJ9 in the year 2017 when most of the source code was published as an open source project managed by the Eclipse Foundation [29].

The virtual environment provides many state of the art optimization algo-

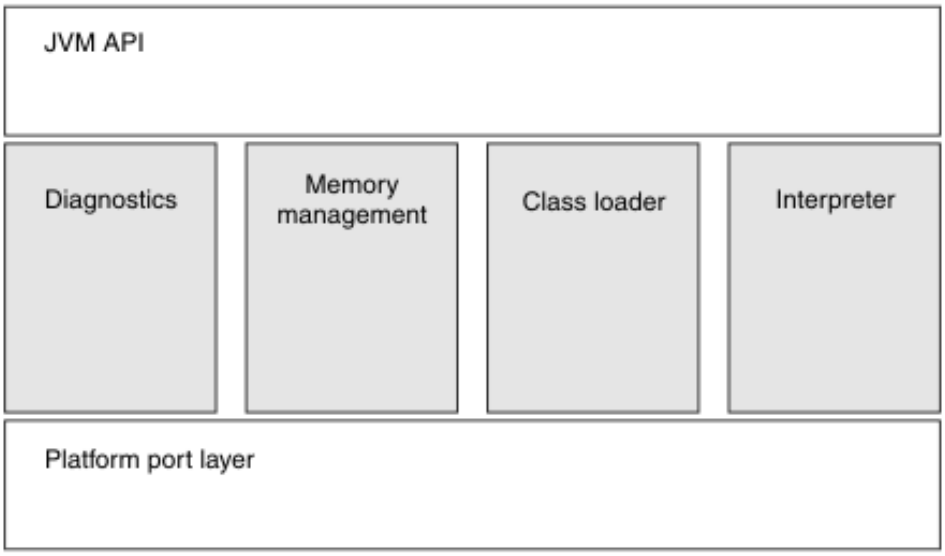

Figure 2.1: IBM J9 Virtual Machine (JVM) component structure [28].

rithms, parameters and structures. The relevant features are introduced in this chapter and are modified throughout the dissertation.

2.2 Reference Counting

Entities used by the client application (mutator) in the virtual environment for information storage and computation are referred to as objects. Objects have the capability to store information as well as reference other objects. The structure created by a running program is a connected graph of objects. When saved in memory, the graph is flattened and objects are arranged in memory. References are defined as memory locations of objects (Figure 2.2). The main idea of reference counting is to keep track of the incoming references

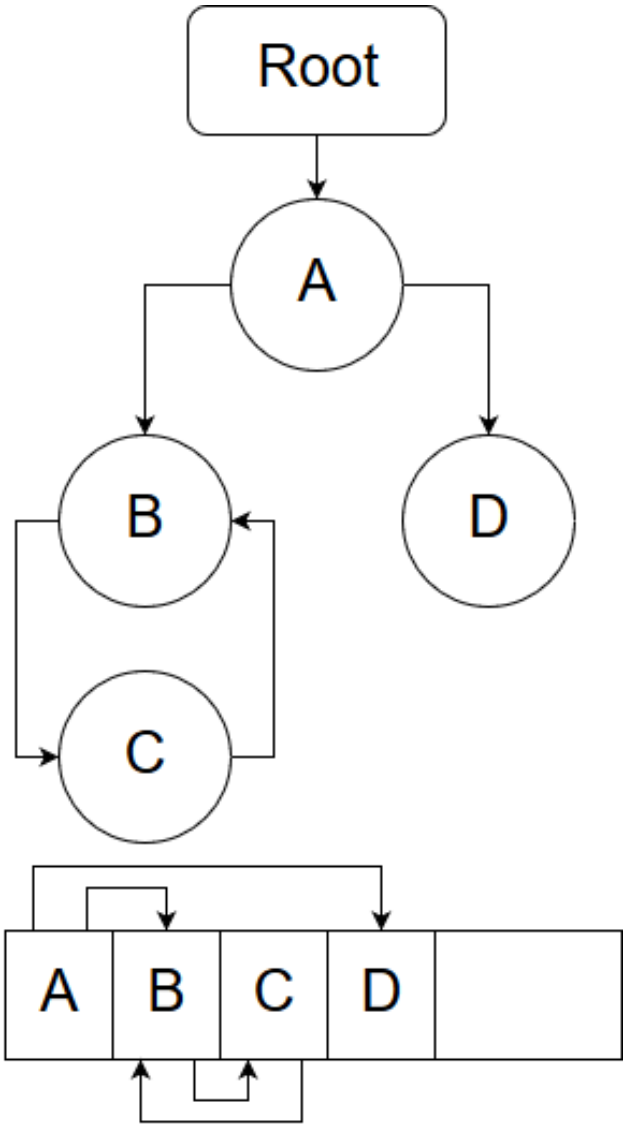

Figure 2.2: Object structure of a client application in a virtual environment. Objects reference each other creating a graph of objects. When stored in memory, the structure is flattened and references are memory locations stored in objects.

to each object [32]. Each object requested by the managed application is extended by a header, which contains a counter for the number of references pointing to the specific object. The count is changed whenever a reference is either removed from or added to an object.

The benefits of this procedure are the simple implementation of the technique and the distributed impact. Time needed for memory management is always applied when work is created: allocation, stack frame handling, reference change, and deletion. This technique is not widely used due to its down sides, such as difficulties in handling circular reference graphs. Figure 2.3 demonstrates that a circular structure, even if not reachable by the mutator, is never detected and thus never freed by the pure form of reference counting. The problem is usually addressed by introducing a stop-the-world phase where one of the different collection algorithms is applied to the heap structure in order to detect these "floating garbage" elements. Floating garbage is defined as objects, which cannot be reached by the client application any longer, but still remain on the heap occupying space. In a garbage collected environment, objects usually remain in this state starting with the loss of reference from the client application and until the next garbage collection of the memory region.

Further, in the worst case the number of objects pointing to a specific element can grow to become $n - 1$, where n equals the number of all objects on the heap. This requires a field to be added to each object for memory management purposes. As most objects are very small, this can pose a large

memory overhead. In the case of Java, where objects average from 20-64 bytes [12, 13, 5], the overhead is substantial.

While memory management work is performed at the same moment when objects are created and die, pause times can still occur during runtime. Once an object that connects a large subtree to the object graph is collected, it initiates the collection of the whole subtree. The collection time depends on the number of objects in the subtree. Research suggests that pauses created can exceed those of collection times when using garbage collection techniques [6], which are presented further. Long collection times can be dealt with using lazy reference counting [61], which uses "to-delete"-lists in order to divide large collection phases into several short ones. However, this simply masks the overall problem [6].

2.3 Mark and Sweep

The mark and sweep garbage collection policy treats the heap structure as a directed graph. Objects are connected using their references to each other [44]. The roots of the graph are represented by pointers in registers, global variables, class variables and variables on the stack.

The algorithm is divided into a marking stage and a sweeping stage. A stop-the-world period is required at least for the marking stage of the policy. Marking describes the traversal of the object graph and marking all objects reached as alive. Once the traversal is done, the sweeping stage can free the

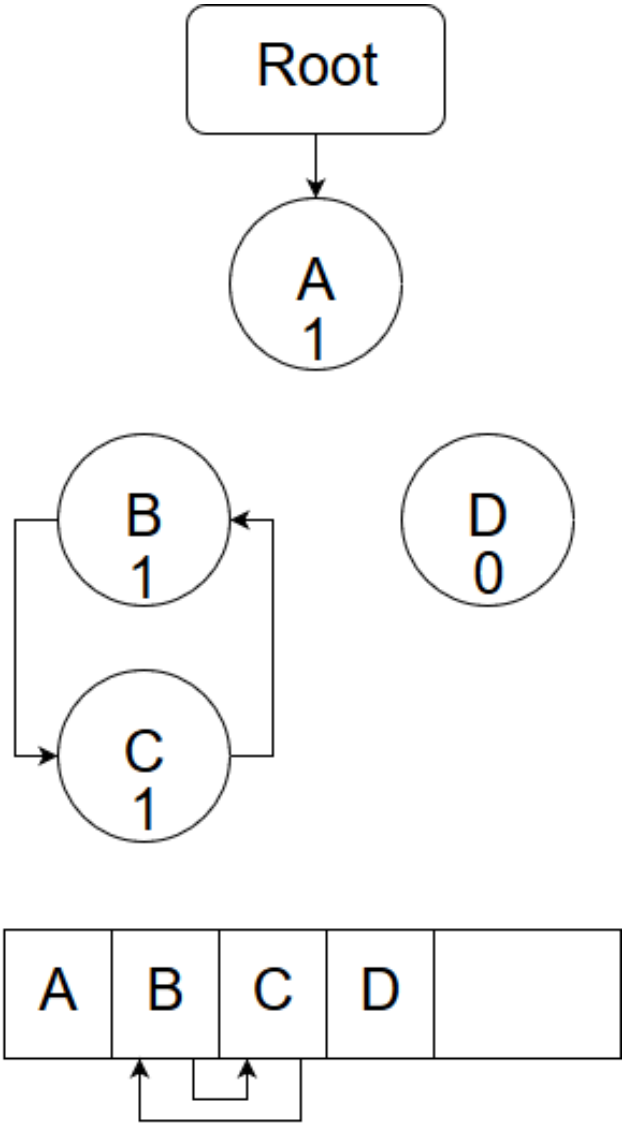

Figure 2.3: The main disadvantage of reference counting visualised. The number in the object indicates its reference count. An object is only detected as garbage once this number reaches 0. While object D reaches the reference count of 0 and can be deleted, objects B and C keep each other alive by referencing each other. All three objects A, B, and C cannot be reached by the application and are considered garbage.

space used by iterating over all objects and freeing those not marked as alive. One downside of the policy is the stop-the-world phase itself as the mutator is stopped completely while the time is used for memory management. The main problem of this policy, however, is fragmentation. As objects are created and freed, they leave gaps in the memory and over time, those gaps tend to become small and spread around the heap space. This becomes more problematic when the application uses a high percentage of the heap. Fragmentation causes allocations to fail even when there is enough free space available, but the space is not contiguous. The heap size in a system using mark and sweep compared to the amount of memory used by all objects used by the application is shown in Figure 2.4. It is worth noting that the same application flow represented by allocation and reference operations was used to present the heap size behaviour of this and all upcoming garbage collection techniques.

Improvements to the mark-sweep policy include lazy sweeping [27], which uses the fact that the locations of objects never change, to postpone freeing objects from the garbage collection stage to the allocation stage. The sweeping is performed by the allocator until a memory gap is large enough for the allocation to succeed. The fact that no objects are moved allows marking and sweeping algorithms to be performed concurrently with a very low cost. Algorithms to introduce concurrency to the mark-sweep policy can be applied during stop-the-world phase [37] or *on-the-fly* [15, 14] using thread-local heaps. Concurrent stop-the-world mark-sweep may utilize multiple markers

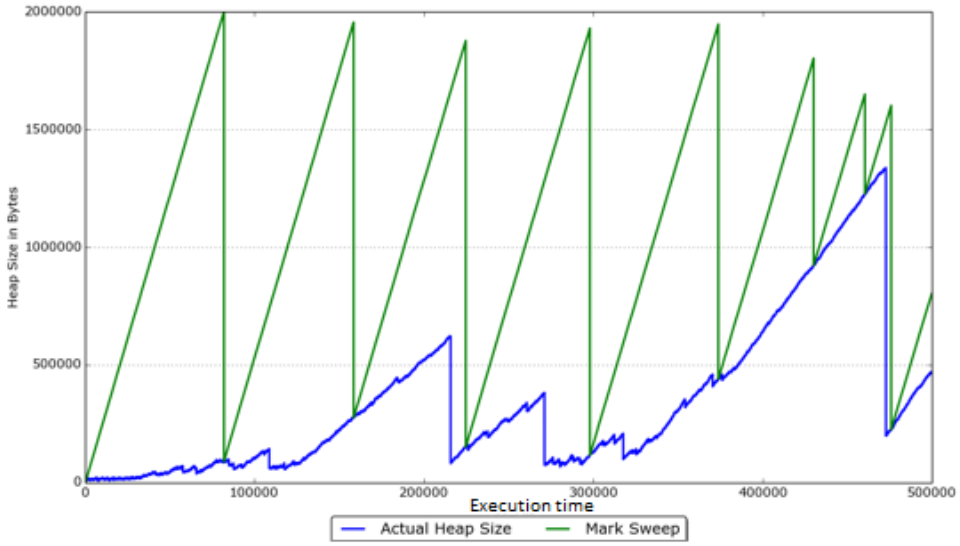

Figure 2.4: Mark and sweep policy heap space allocated by the virtual machine for the mutator. The heap size drops every time a garbage collection phase is initiated. The data was gathered using the GarCoSim simulator [47, 21]. Execution time of the application in this case is defined by the number of allocation and reference operations performed.

and sweepers, whereas on-the-fly marking attempts to never stop all mutator threads at once. Instead, one thread or thread group is stopped from further execution in order to use its root set as an initial marking point. This however introduces complexity of dealing with objects which are referenced by multiple threads (escaped objects).

2.4 Mark and Compact

In order to prevent fragmentation, a compaction phase can either be added as a periodically-appearing third stage of the mark and sweep policy, or as a complete replacement for the sweep stage [58]. Once all live objects are marked, the objects are compacted in order to remove all fragmentation from the heap. The space located after the last memory location used is marked as free. Compared to the sweep phase, compaction requires more time. However, the benefit of not having fragmentation has a significant impact on the number of garbage collections required in order to accommodate the mutator. The difference of mark and sweep to mark and compact is shown in Figure 2.5. The changes in heap size during execution using a mark and compact collection policy can be found in Figure 2.6. Comparing Figures 2.4 and 2.6 shows that mark-sweep requires eight collections whereas mark-compact can complete the task using seven collections.

2.5 Copying Collection

The idea behind copying collection is to remove the marking stage and to only use a compaction phase. In order to accomplish this, live objects have to be copied during traversal. As it is not certain how much space and which addresses will be free during the process, objects have to be copied into an entirely separate address space. This requires the heap space to be divided into two halves. One, where new objects are allocated and the second space

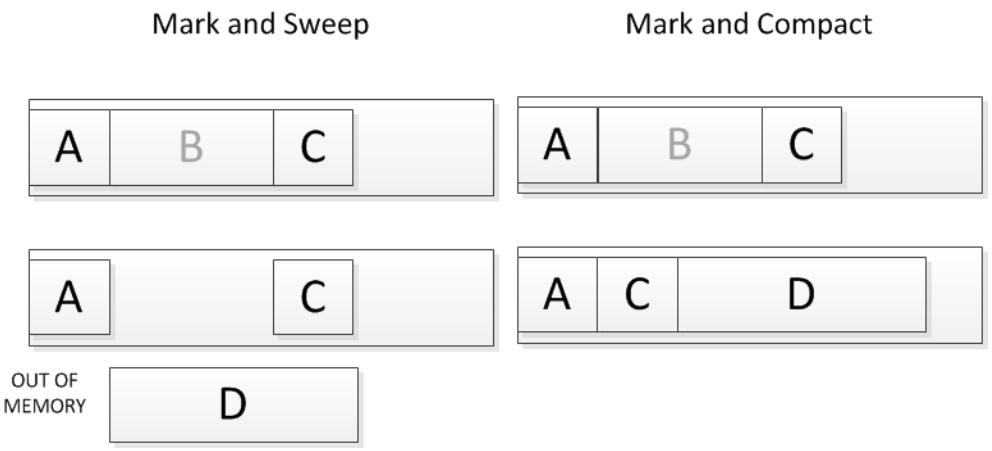

Figure 2.5: Difference between mark and sweep with mark and compact visualised. The initial heap structures are the same. Objects A and C are still alive while object B is no longer referenced and therefore can be freed. Once the application attempts to allocate object D, a GC is initialised in both cases. The allocation succeeds in mark and compact, but does not, due to fragmentation, in mark and sweep.

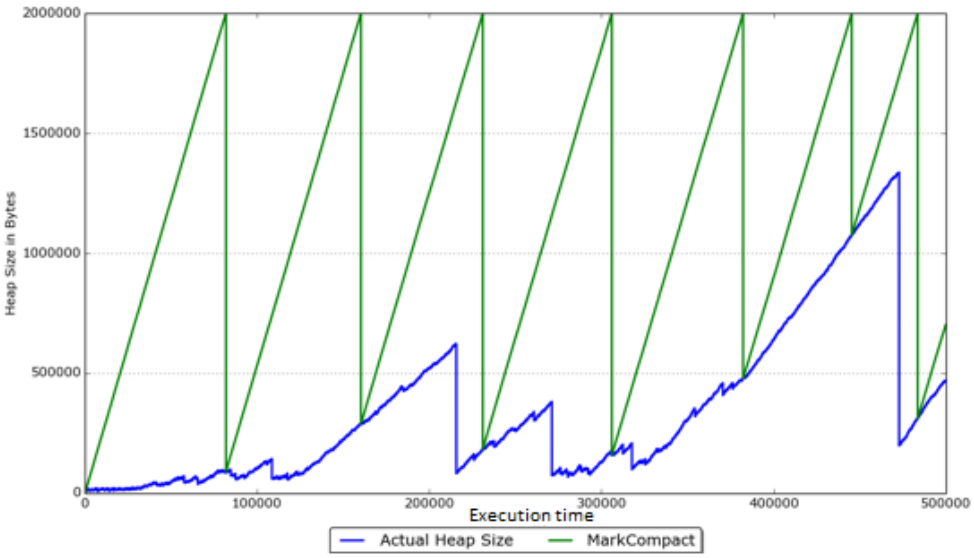

Figure 2.6: Heap size using a mark compact garbage collection policy. The execution time in the graph is measured in allocation and reference operations performed.

where live objects will be copied to during a garbage collection. The benefits are a very short stop-the-world time and the fact that no fragmentation is possible. The largest disadvantage is that the effective available heap space for mutator objects is half of the physical memory space [20, 7].

2.6 Generational Collection

A generational garbage collector does not use the heap as a monolithic structure. The collection policy leverages the observation that most objects are small and die off quickly. A larger number of survived garbage collection phases (object age) increases the probability of this object surviving for a longer period of time. The heap is divided into two or more heap regions based on object age. Objects are allocated in the youngest region and are promoted to older regions once they survive a certain amount of time [39].
One of the major advantages of the collection policy is the ability to collect only part of the heap. Remember sets are kept for each region and represent how many pointers have incoming references into a certain region. Using mono-directional remember sets, which represent references from an older generation to a younger one, only a collection of the oldest generation equals a global collection comparable with the previously mentioned policies.
A large number of parameters can be used in order to configure a generational collector. Most of them are trade offs and can be adjusted based on the mutator needs. The size of the nursery space determines how often young

objects are collected. Each collection causes a pause of the mutator. However a larger size also means a longer pause, which can be less desirable than a high number of stop-the-world phases. The promotion age of objects determines how long they stay in a specific region. This time can be measured either in time lived or garbage collections survived.

Another important design decision for the generational collector is which collection policies are used for the different regions. A copying collector is often used for the youngest generation as it often fills up, has to work with an almost full heap most of the time and is collected frequently. Mark compact is usually used for the older generations due to its fragmentation handling and the fact that no heap space is lost as opposed to the copying collector. The optimal number of generations was found to be two [1][43] even though there was research performed to create hybrid approaches of two and three generations [33]. The size growth of a managed heap using a generational garbage collector is shown in Figure 2.7.

2.7 Heap Object Structure

Each object created by the mutator and allocated on the heap is supplemented by a header with information needed by the virtual machine. This information depends on the virtual machine, collection policy, and other parameters. Garbage collection policies add an object header to each object in order to store necessary information. Reference counting for example re-

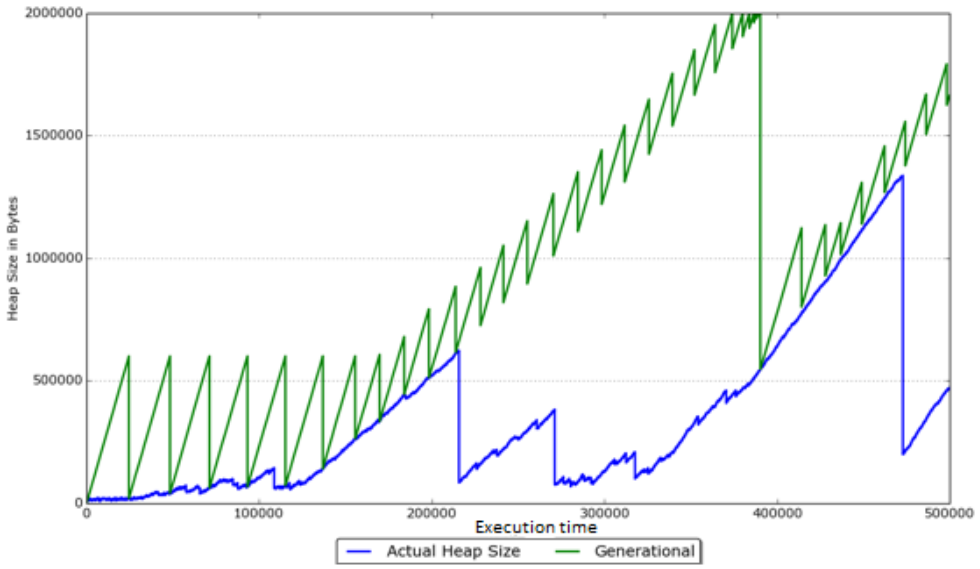

Figure 2.7: Heap size and garbage collections using a generational garbage collector.

quires at least the count of references pointing at a specific object in order to know when it is safe to deallocate it.

Research in automatic memory management aims to optimize the heap structure in order to perform fewer collections, shorten collection phases, and to result in less fragmentation. However the information needed to accomplish this increases the object size and is therefore an addition to the problem. This leads to the idea of using information about objects and their relationships in order to increase performance and decrease memory use by the mutator. One goal of this behaviour is to make up for the additional space needed by providing a better execution time as a result of fewer page faults and cache misses. Research directly working on improving object locality by

using different traversal algorithms [4] and object usage profiling [8] in order to improve caching showed advantages in certain cases.

In addition, the Java heap structure shows that a large percentage of objects stored on the heap are string objects. The number is 20% on average in most applications, but can be as high as 40% in special cases such as the J2EE application server [34] as shown in Figure 2.8. The research proposed aims to use the heap object structure in general and the string object structure specifically in order to improve the memory use of the Java VM.

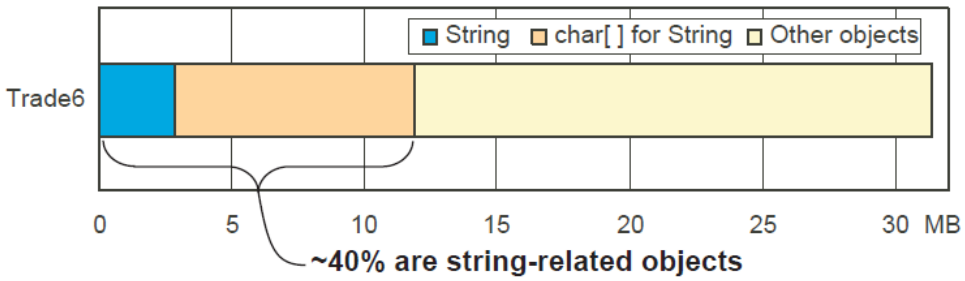

Figure 2.8: Heap memory structure. About 40% of the heap used by a J2EE application server are strings or string related objects [34].

2.8 String Structure in the Java VM

String objects in Java are immutable; they have the specific property that they can never change their content. If a change to the object is requested by the mutator, a new object with that content is created and a reference to the new object is written into the mutator reference. The creation of a new

object when changing is used to gain an advantage by multiple mechanisms in the Java VM such as the string table or the `substring()` command.

```
String a = "Hello World!";
String b = new String("Hello World!");
String c = b.substring(0, 6);
String d = "Hello World!";
```

Three different string objects are created by the mutator as strings `a` and `d` return the same reference. All four declarations are valid, but are being processed in three different ways. String `a` is created using the string table. The string table acts as a buffer containing all strings created in this way. Since the string is static, it is known at compile time that this particular string can be added to the string table. This is used for string deduplication during the allocation process. If the mutator attempts to create a string with the exact same char array (string `d`), a reference to the object header of `a` is returned.

String `b` is a request to create an object with a unique header and the provided content. This is not handled by the string table and a new object is allocated to comply with the Java specification that using the keyword `new` always results in the creation of a new object. String `c` represents a special case. The method uses the structure of the string object header in order to reuse the char array, but at the same time to provide a new header as the resulting reference. Figure 2.9 presents the string header structure. The `substring()`

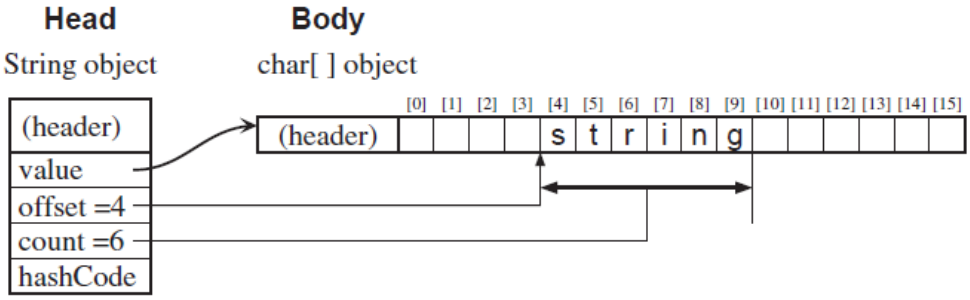

Figure 2.9: String header structure with the necessary fields [34].

method copies the character array reference field, but changes the offset and length according to the input values.

Even though the operation `.equals()` using any combination of strings `a`, `b` and `d` would return true, the `==` operation acts in a different way. The `.equals()` method performs a string compare in order to determine if the character array contents are equal to each other. Comparing the variables to each other using `==` compares the references stored in the fields on both sides of the operator. As shown below, this does not represent the equality of the string content.

```
if(a == b){...};//false
if(a == c){...};//false
if(a == d){...};//true
if(b == c){...};//false
if(b == d){...};//false
```

Research on strings and string deduplication has taken several different approaches. As strings and string related objects occupy more heap space than

any other single class, Kawachiya et al.[34] suggest collecting strings first in order to check if this would free enough space to continue execution. By doing this, the authors were able to reduce the garbage collection time as not all objects had to be processed. The approach utilizes the given properties to limit the number of objects participating in the marking and sweeping stage. However they do not use the content of the string objects in order to optimize the structure of live objects.

Horie et al. [26] suggest a deduplication based on offline profiling where they gather information about strings existing in the JVM. The main goal is to identify the most significant characters of all strings and use this to create an efficient hashing and deduplication algorithm. They demonstrate a considerable speedup once the profiling stage is completed and the JVM is trained on an application. This behaviour can be extended to share the deduplication information between multiple virtual machines. The approach is useful in cases where a single application is executed frequently in one or multiple virtual machines. However applications that change their execution flow frequently or require significant computation, but are not executed over and over again, tend to not benefit from using such an approach as the right amount of profiling can be difficult to determine. A more dynamic approach that is capable of deduplicating strings during runtime could potentially close this gap.

There are also more general attempts at deduplicating structures. This includes string objects as much as any other type of object. Marinov and

O'Callahan [42] present a full scale deduplication approach, where all objects are analyzed throughout a run and are compared in order to find duplicates. The results are later presented to the developer, who can use the information to eliminate those duplicates. The limitations this approach presents in regard to offline profiling are the same as with Horie et al. [26]. Both approaches can be used as guidelines and profiling tools for developers, but cannot optimize the memory structure dynamically.

Chapter 3

String Deduplication

The automatically managed heap and its structure in a virtual environment are at the same time the biggest advantage for developers and one of the most optimized areas when it comes to mutator execution. The inefficiency of automatically managing user space memory led to not only improving the collection strategy itself, but to also use the time spent collecting unreferenced objects to improve the memory layout in order to provide a better overall execution time for the mutator. Strings and string-related objects often occupy the largest amount of memory when compared to other classes. This chapter introduces optimizations for string related objects and presents an additional approach to improve the string structure further.

The chapter is divided into three parts. Section 3.1 discusses the motivation of the project based on the background and related work provided in Section 2.8. Section 3.2 explains the design and implementation before experiments

are presented and analyzed in Section 3.3.

3.1 Motivation

The string table approach currently used in virtual machines creates a large benefit for string objects during allocation time and decreases the memory usage of the application. However, it does not cover all of the strings created. Strings created using the `new` keyword as well as constructed strings remain unaffected. This thesis refers to strings pieced together at runtime, the content of which is unknown at compilation time, as constructed strings. An initial analysis of live objects on the heap was performed using DaCapo [5], SPECjbb2005 [10], and SPECjbb2013 [11] benchmarks. SPECjbb benchmarks aim to stress test specific parts of virtual environments while DaCapo is a collection of real life applications. The applications and their function can be found in Table 3.1. The initial analysis revealed that some applications have many duplicate strings even when considering the deduplication performed by the string table and `substring()`.

Figures 3.1, 3.2, and 3.3 show an extract of a complete analysis of all live strings after each garbage collection for the DaCapo benchmarks `jython`, `lusearch`, and `h2` as an example for the study results. The particular benchmarks were chosen to represent varying behaviour of the results. Other benchmarks resemble one of those variations. The graphs show the percentage of duplicate strings encountered as well as the possible heap space, which

Benchmark Name	Mutator Task
avrora	AVR microcontroller simulation of multiple programs
batik	Scalable Vector Graphics (SVG) rendering
fop	PDF creation from an input file
h2	JDBDbench-like in-memory benchmark
jython	Python interpretation using the pybench Python benchmark
luindex	Document indexing
lusearch	Text search in the works of Shakespeare and the King James Bible
pmd	Java source code analysis
sunflow	Image rendering using ray tracing
tomcat	Tomcat server queries
xalan	XML to HTML transformation

Table 3.1: Tasks performed by the DaCapo benchmarks chosen to evaluate the string deduplication approach [54].

could have been saved if all possible strings were deduplicated. Experiments were performed using a 2GB heap size and default execution parameters in the J9VM.

The capturing process is performed before and after each garbage collection. A higher number of duplicate strings indicates a higher amount of deduplication work required to merge strings and use the same character array. A higher duplicate heap value indicates increased potential in heap space saving due to deduplication. This presents a trade off, where the perfect candidate for deduplication would have a small number of long duplicate strings occupying a large portion of the heap.

Note that the results in the graphs for the benchmarks are quite different. Even though all three of the benchmarks show a large percentage of duplicate

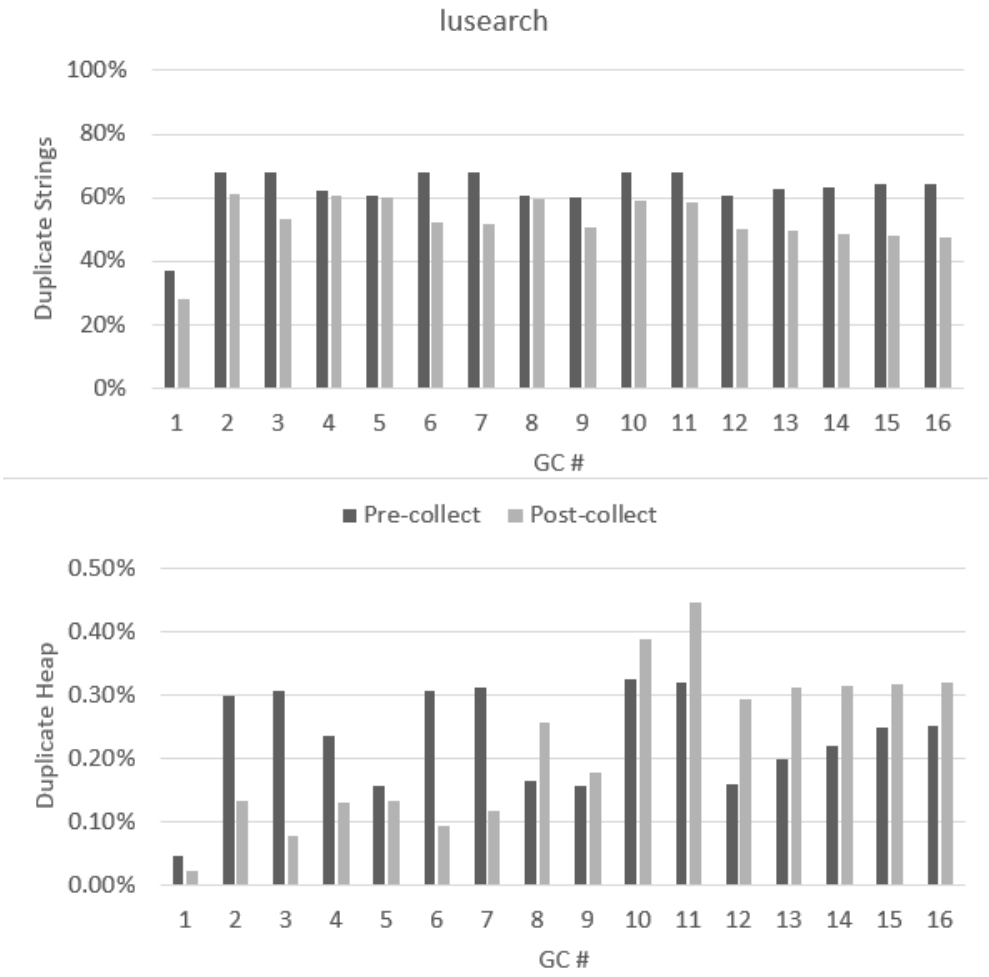

Figure 3.1: Lusearch: Number of duplicate string objects which do not share a character array (top) and the possible heap savings based on the string lengths and the current heap size (bottom).

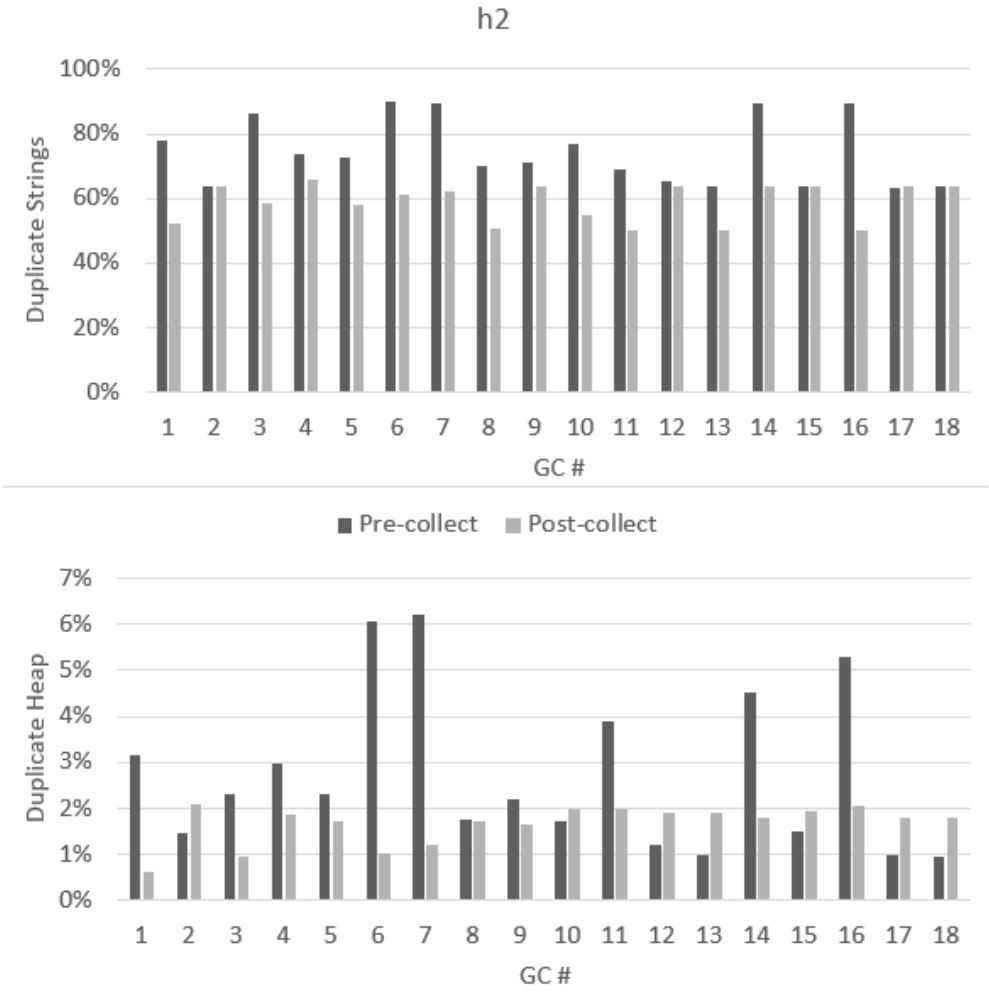

Figure 3.2: H2: Number of duplicate string objects which do not share a character array (top) and the possible heap savings based on the string lengths and heap size (bottom).

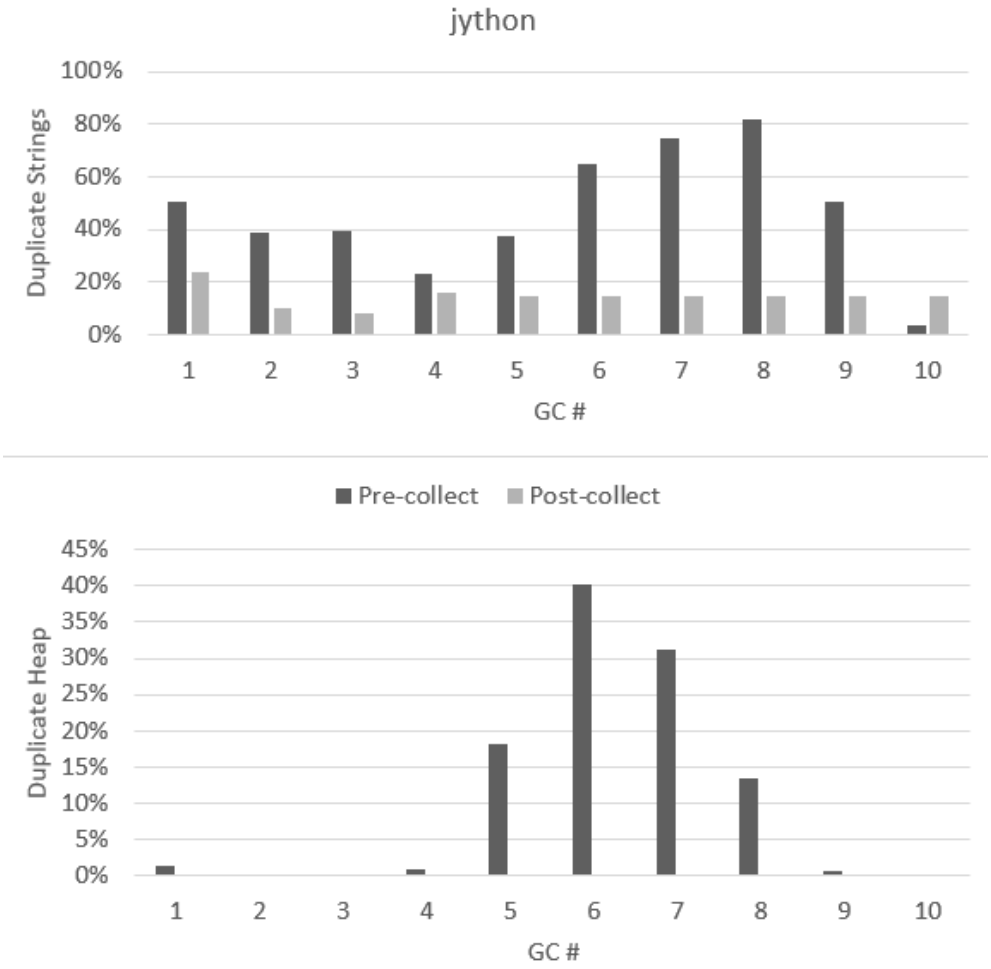

Figure 3.3: Jython: Number of duplicate string objects which do not share a character array (top) and the possible heap savings based on the string lengths and heap size (bottom). Most of the strings die during a garbage collection, which explains the seemingly missing post-collection character numbers for the bottom graph

strings, the heap impact of the strings varies by a considerable margin. While most duplicate strings are collected in `jython`, the heap impact for `lusearch` and `h2` after each garbage collection averages at 0.2% - 1%.

It is important to underline that, depending on the garbage collection policy, a deduplication in the garbage collector does not traverse all live objects on the heap. This process is usually too time consuming and is reduced by the creation of generations or heap regions. The graphs in Figures 3.1, 3.2, and 3.3 merely show what could be achieved in the optimal case for those specific applications. A preliminary test was conducted in the IBM JVM generational concurrent garbage collector. All strings encountered during each garbage collection in combination with their addresses were saved.

The evaluation of the approach requires deterministic benchmarks. The performance of the benchmark utilizing string deduplication is compared to an unchanged Java virtual machine in order to determine and quantify benefits and disadvantages of the algorithms presented.

3.2 String Deduplication Design

The approach presented in this research is to add an additional string deduplication stage during the stop-the-world phase used for garbage collection. All live objects in a collected region are traversed and processed by the garbage collector. In the case of a copying collection policy, all objects are evacuated to a new allocation space before the region is marked as empty. A mark-

compact collector has to mark all live objects as such before the compaction phase discards all unused memory gaps.

The research proposes an additional step during traversal of the objects. The idea is to store and compare all string objects encountered. In cases where duplicate strings are found, they can be deduplicated. The flow of garbage collection will change as shown in Figure 3.4.

The deduplication has to maintain conformity with the Java language specification. In order to achieve this, the duplicate object must keep its object header. The presented deduplication behaviour treats a deduplication similar to a `.substring()` call on a string object.

Creating a substring using the `.substring()` function creates a new object header and keeps the address to the character array of the source string. Only the offset, marking the beginning of the substring in the character array, and the length value, marking the end of the substring, are different.

The described approach takes two full objects consisting of two object headers and two identical character arrays and deduplicates the character arrays. Both object headers are changed to point to the same array keeping all other parameters as they were. This procedure relies on the presumption that the comparison stage does not compare substrings, but always compares the content of the complete character arrays.

In order to reduce the time overhead needed for storage and comparison, the approach utilizes a hash table. The size of the hash table remains unchanged. Collisions are handled by using a linked list structure chaining col-

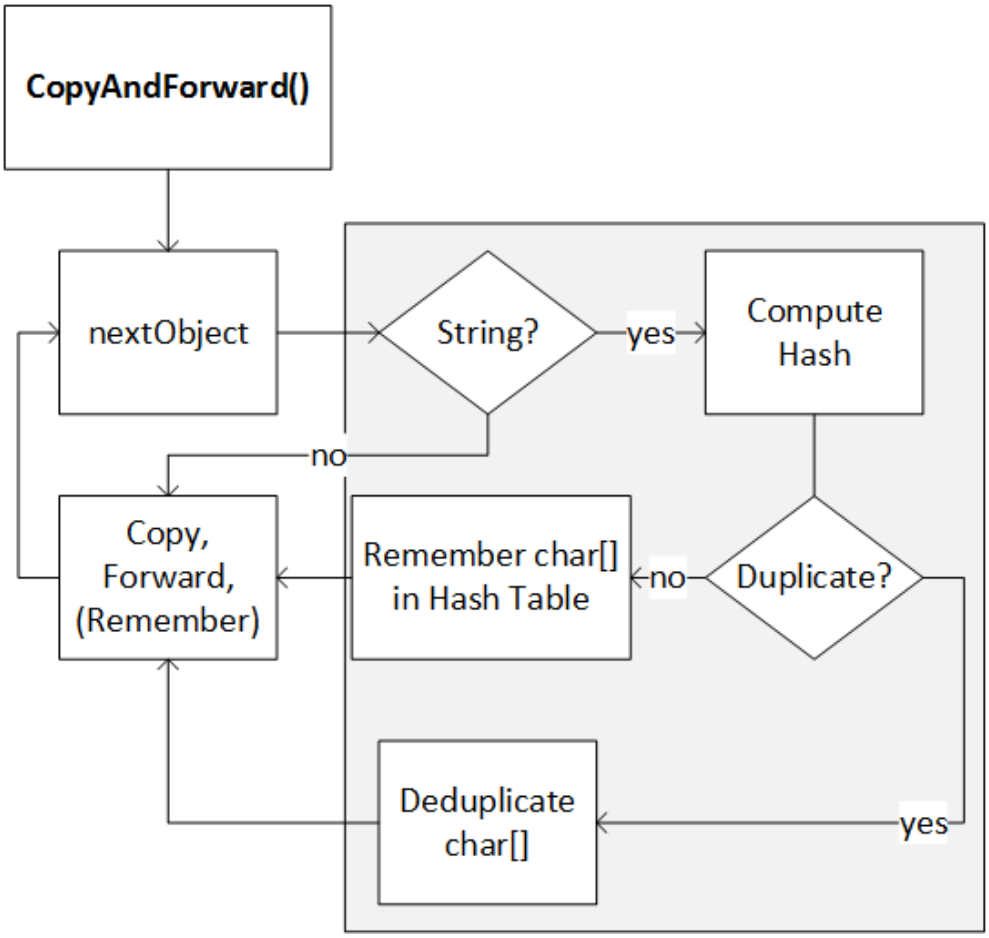

Figure 3.4: The flow of the string deduplication process in the garbage collector. Proposed changes to the flow are highlighted by a grey background.

liding strings into the same slot. The slot used for a string is determined by using it as a circular structure and therefore computing $slot = hash \ \% \ size$ on it. Each node in the hash table contains the real hash value before being trimmed down to represent a hash table slot, the address of the string object found and the next node in the list. Most of the time added to the stop-the-world stage is spent comparing strings to each other. A reduction of this time is achieved by comparing characters, until the first mismatched character is found.

The distribution of strings within the hash table relies heavily on the hash function used. Experiments have shown good results with `sdbm` hashing [38] (source shown below). Other data structures were considered, but were found to be inferior as the look up time for strings is the most important metric for the approach. Adding elements to a hash table is relatively slow. In this particular case, the process depends on the length of the string and the number of elements already included into the slot of the hash table resulting in a worst case complexity of O(n+m) where n is the length of the string and m is the overall number of elements in the table. The average complexity of the insertion process is O(n) as the hashing function was chosen to spread strings throughout the table evenly. The structure provides a fast look up time for elements already included in the table, which offers the same complexity as the insertion process, but benefits from the 3-tiered comparison process using the content-based hashing algorithm. The first comparison checks the least significant bits of the hash value before comparing the complete hash value.

Once both of those checks are successful, the algorithm moves on to compare the actual contents of the two strings compared.

```
static unsigned long
    sdbm(str)
    unsigned char *str;
    {
        unsigned long hash = 0;
        int c;

        while (c = *str++)
            hash = c + (hash << 6) + (hash << 16) - hash;

        return hash;
    }
```

3.3 Experimental Results

The described approach was evaluated using a set of DaCapo benchmarks and a 2GB heap size. The runs were repeated 10 times and the runtime was averaged in order to reduce the impact of noise during the testing stage. VM warm up did not matter in this particular experimental setup as the VM was destroyed and recreated after each run. Performing the same test with an already running VM did not yield significantly different results as the dedu-

plication table is not reused for different clients of the virtual environment. Each of the benchmarks was executed in three configurations:

(A) Unchanged VM

(B) VM with duplicate detection, but no deduplication

(C) VM with string deduplication

The first configuration of the experiments represents the baseline. It is used to compare the execution time in order to determine if the cost of the additional stage during traversal can be amortised by performing fewer copying procedures and freeing more memory during the stop-the-world stage.
The second configuration is performed in order to compare the string structure on the heap. The configuration does not deduplicate strings, but maintains the string hash table in order to quantify the duplicates that could potentially be saved. The reason to run this configuration is to be able to compare the number of duplicate strings to that of configuration A. In theory, duplicates could be very short-lived strings, which would not survive the GC stage and would decrease the optimization stage. However, if string deduplication provides a benefit, fewer duplicate strings will be encountered with each GC.
The third configuration creates a hash table of strings encountered; it detects duplicate strings and deduplicates them. The execution time of this configuration is compared to the first run. In addition, the number of garbage

Benchmark	A	B	C
avrora	137.723s	136.253s	137.600s
batik	97.773s	98.186s	98.881s
fop	48.035s	48.945s	47.500s
h2	2239.564s	2290.825s	2235.637s
jython	765.020s	784.355s	780.386s
luindex	124.723s	124.672s	124.930s
lusearch	50.091s	54.390s	48.410s
pmd	61.215s	62.483s	61.035s
sunflow	151.774s	186.85s	181.658s
tomcat	93.845s	95.695s	96.385s
xalan	26.776s	28.803s	27.909s

Table 3.2: Benchmark runtime for the three configurations in GenCon. The time represents the complete VM execution time including startup and shut down. **A**: Unchanged VM, **B**: Duplicate string detection, but no deduplication, **C**: Duplicate detection and deduplication.

collections and the amount of memory freed during each GC is captured and compared to the first configuration.

The runtimes of each configuration for the benchmarks are shown in Table 3.2. The first column of the table shows the runtime of the program using an unchanged VM. The second column represents the second configuration and shows the impact of creating the duplicate detection infrastructure, such as the additional flow during garbage collection and string hash table, without the benefit of a deduplication. It represents the complete overhead of the approach and can be considered a worst case scenario. The last column in the table shows the overall impact of the approach presented. Table 3.3 shows the relative change of the configurations compared to the unchanged VM results.

Benchmark	B	C
avrora	-1.07%	-0.09%
batik	+0.42%	+1.13%
fop	+1.89%	-1.12%
h2	+2.29%	-0.18%
jython	+2.53%	+2.01%
luindex	-0.04%	+0.17%
lusearch	+8.58%	-3.36%
pmd	+2.07%	-0.29%
sunflow	+23.11%	+19.69%
tomcat	+1.97%	+2.71%
xalan	+7.57%	+4.23%

Table 3.3: The relative impact of the optimization on the runtime compared to running the benchmarks in an unchanged VM.

The runtimes show that the impact of the approach is not the same for all applications. In order to understand how some applications benefit while others perform worse or remain the same, more metrics have to be considered. As stated in previous sections, performing a string deduplication is a very specific optimization. The largest benefit is achieved for applications that handle a large number of strings. In addition, the approach changed the garbage collection process but does not interfere with any other part of the program.

In order to explain the results in Tables 3.2 and 3.3, two main factors have to be considered: the application structure and the number of unforced GCs. Even though the `avrora` benchmark has a runtime of over 2 minutes, only one GC is performed. This global garbage collection happens before the mutator application is started and the DaCapo internal time measurement

Benchmark	GCs
avrora	1
batik	1
fop	1
h2	18
jython	10
luindex	1
lusearch	16
pmd	2
sunflow	10
tomcat	4
xalan	5

Table 3.4: Number of garbage collections for each benchmark at 2GB of heap space using a GenCon collection policy.

is started. The same behaviour is observed for `batik`, `fop`, and `luindex`. The full list of GCs is shown in Table 3.4. All but one of the collections were partial for each of the runs.

Negative relative impact in column B in Table 3.3 occured for two benchmarks, which had one GC during the runtime. The impact is -0.04% for `luindex` and -1.07% for avrora. Both benchmarks only have a single GC in the startup time of the virtual machine. The number of strings found during this collection is minimal which reduces the impact of the maintenance and upkeep to almost none. The change in runtimes can be explained by noise, which can occur even after running each benchmark for 10 times and averaging the runtimes.

The benchmarks with the most GCs are `h2`, `jython`, `lusearch`, and `sunflow`. Those benchmarks are affected in different ways by the approach presented.

The `h2` benchmark adds a large overhead for the maintenance of the string deduplication data structures. This overhead in runtime is nullified by the benefit of string deduplication. Even though an overall benefit was not achieved, the cost of the data structures was balanced by the benefit. The reasons for this behaviour is a large number of strings that are used for the database communication in the application, which is a typical behaviour for database applications. The benefit was limited by the fact that most of the strings used were part of the internal string table and did not have to be deduplicated.

Benchmarks that performed worse when string deduplication was applied are `xalan`, `tomcat`, and `jython`. The `xalan` benchmark had only four GCs and a runtime of approximately 26 seconds, which is not enough for the approach to reclaim the cost for data structure maintenance and initialization. `Jython` and `tomcat` did not contain enough duplicate strings to create a benefit which caused the data structures to add runtime between 4% (1.1s) for `xalan` and 2.3% (15.3s) for `jython`.

Applications that benefited from string deduplication the most, while performing more than five garbage collections, were `lusearch` and `sunflow`. A relative benefit of about 3.36% was achieved for `lusearch`. Considering the variation of results of about 1.5% between test runs, it indicates that the approach does create a benefit for this particular application. Comparing the unchanged VM with the second experiment configuration shows that not deduplicating while comparing strings adds 8.5% to the runtime. The char-

acteristics of the application are very beneficial for the string deduplication approach as it repeatedly performs text searches.

The benefits achieved for the `lusearch` benchmark as well as the ability to maintain the `h2` execution time while maintaining the string deduplication data structures was indicated by the analysis shown in Figures 3.1 and 3.2. This fact suggests that in cases where developers find that there is a high number of duplicates that survive GC phases, the approach presented can be applied in order to save heap space and potentially delay GCs.

The particular set of benchmarks was used to present the overall impact of the garbage collection modification to a general set of applications. Since DaCapo is a collection of real life applications which include computational heavy, string heavy and other applications, the research aims at investigating the impact of creating and maintaining the string hash table during the collection phase on as many classes of applications as possible to provide a more general view of the impact and the challenges when it comes to creating a more general approach. Tests with artificially designed benchmarks creating a high number of duplicate strings were indicating potential benefits for the approach. The `lusearch` benchmark was found to be of the same class of applications that handle a high number of duplicate characters on the heap, which are not deduplicated by the internal VM string table.

In addition to the runtime analysis, the times when garbage collections occur during mutator execution are compared in order to visualise the impact of the string deduplication. To analyze this metric, the fact that GC phases

are longer due to a changed flow has to be considered. Figures 3.5 and 3.6 show the time of the GC in relation to memory freed during this particular GC. This is done for the unchanged VM as well as for the VM with string deduplication. The time indicated is not the mutator time and not the execution time overall. In order to obtain the number, the time of all previous garbage collection times was subtracted from the current execution time. The formula for this computation is shown in Equation 3.1.

$$T_n = OverallTime - \sum_{k=0}^{n-1} GCDuration_k \tag{3.1}$$

Figures 3.5 and 3.6 show that an application that benefits from string deduplications delays the GC phases as more heap space is freed during previous garbage collections. This can potentially lead to fewer necessary garbage collection phases, which would have a higher positive impact on the application.

3.4 Conclusions and Future Work

This chapter presented an approach of detecting and handling duplicate strings on the heap in virtual machines. It introduced optimization potential of a large percentage of immutable string objects in Java and presented the design and evaluation of string deduplication during GC stop-the-world time. The experiments showed that benefits of the approach are very application specific.

Especially string-heavy applications showed promising results and revealed

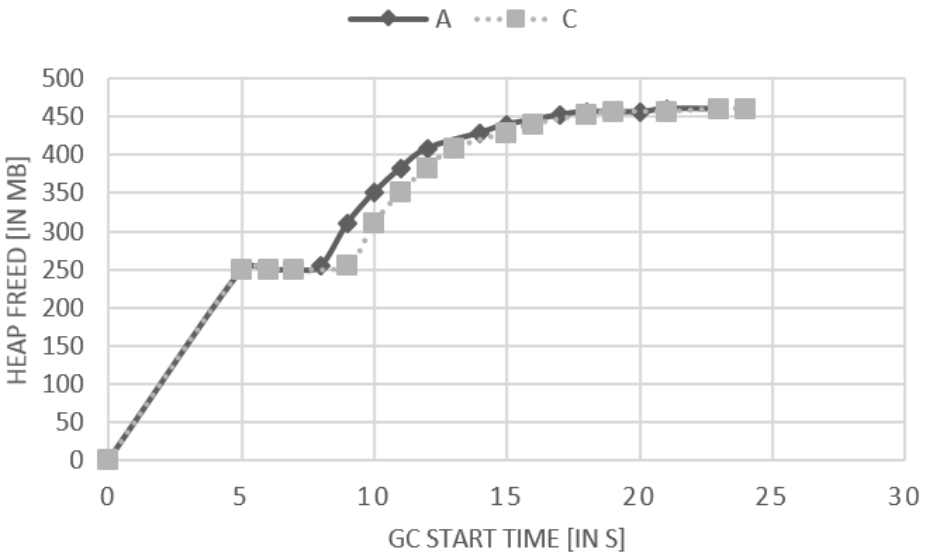

Figure 3.5: Start time and memory freed for each GC when running the lusearch benchmark.

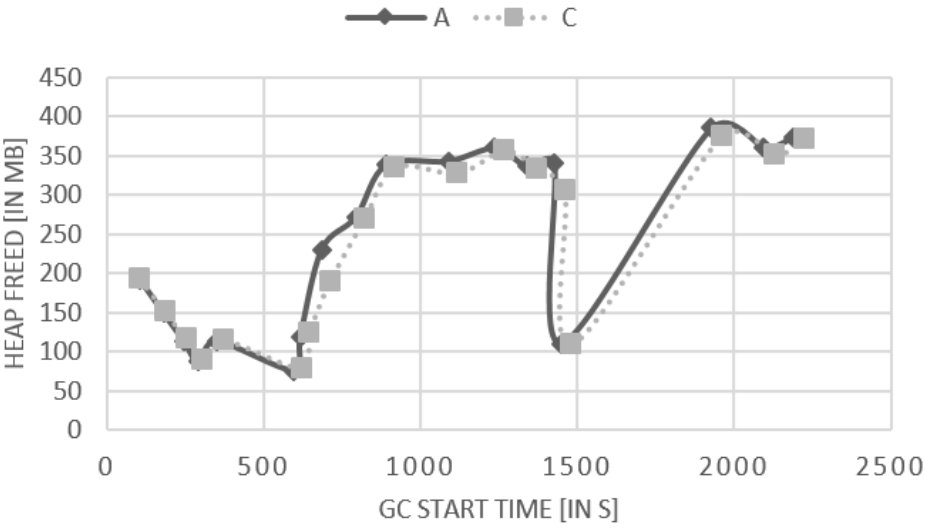

Figure 3.6: Start time and memory freed for each GC when running the h2 benchmark.

possible improvements that will be investigated further in the future. At the current time, the main point for optimization is the data structure needed for string deduplication. If the time taken to manage the data structures can be reduced, the performance benefit of the approach can be increased. The disadvantages and a number of improvements are presented and discussed in Chapter 4.

Deduplicating strings changes the overall object graph and object relations. This can also result in different locality numbers. This can have a positive impact as more objects point to the same data and fewer page faults occur, but also negative ones when the cache locality of deduplicated strings decreases because the char array of a different object is shared.

Chapter 4

Dynamic String Deduplication

A common memory reduction algorithm on the heap is data deduplication. One approach is to use the traversal phase of a Java VM garbage collector to deduplicate strings as described in Chapter 3 and [48]. While live objects are passed, marked and moved, duplicate string objects are analyzed. Information gathered is stored in a hash table for faster look up in a two-stage comparison procedure. The string object content is matched and duplicates encountered are shared between String instances without breaking the Java language specification [23].

This chapter investigates possible improvements to the approach aimed at decreasing the time needed for duplicate detection. The time needed for hash table creation, maintenance and actual deduplication is reduced and an increased benefit is achieved.

The chapter is divided into five sections. Section 4.1 presents the motivation

of the work and outlines the experiments performed based on the findings in Chapter 3. Sections 4.2 to 4.4 present changes to the algorithm and discuss their effects in detail. An application specific analysis is presented in Section 4.5. The chapter concludes in Section 4.6 and presents possibilities for future work.

4.1 Motivation

The benefit of GC-time string deduplication is application specific. Figures 3.5 and 3.6 show the impact of the string deduplication on garbage collection start times and the amount of memory freed for each garbage collection for two benchmarks. The time of the garbage collection start, shown in the figures, is mutator execution time. It does not take into consideration the time required by the garbage collections itself. Taking the overhead into account, the maintenance of the string deduplication data structures can increase runtimes by up to 20% as discussed in Section 4.2.

As presented in Chapter 3, the main downside of the deduplication flow is the time it takes to process each string object. The content has to be read in, a hash value has to be computed, and the hash value has to be checked and handled. Each part of this process adds overhead. If the benefit is not large enough, the process increases the application runtime instead of providing a benefit. Maintenance overhead is significant and as such a benefit is only achieved if an application provides a large amount of string data on the heap

and a large enough number of garbage collections.

This chapter describes three possible optimization areas for the string deduplication approach. This includes an analysis of the hash table maintenance flow and the reduction of expensive operations as described in Section 4.2. A new metric is introduced and evaluated in Section 4.3. Section 4.4 discusses the feasibility of attempts to perform a deduplication in order to compute potential benefit for the currently running application.

Experiments performed for the optimizations utilized part of the DaCapo benchmark suite [5], chosen because it provides a collection of deterministic real life applications. Short benchmarks, which include one or fewer garbage collections were not part of the benchmark set as they did not result in deduplication taking place. The underlying tasks performed by each benchmark used are presented in Table 3.1.

The research uses an IBM Java VM with disabled just-in-time (JIT) compilation as the baseline. The JIT was disabled to reduce the number of optimizations occuring in order to accomplish the proposed work. Each benchmark was executed at least 10 times to reduce noise effects. The heap size for all tests was set to be 2GB. The times reported are measured by collecting the system time pre- and post-benchmark. Runtimes reported by DaCapo itself are not considered in order to take into consideration the startup of the VM and its shut down process. As structures impacting those phases were introduced to the VM, it was considered more beneficial to take those impacts into account.

4.2 Flow Optimization

One problem with the algorithm flow in Figure 3.4 is the time required to create and maintain the structures. To reduce the time, the flow was redesigned and unnecessary costly operations were eliminated.

The largest impact was created by restructuring the section of the algorithm that traverses every object. Profiling revealed that the check to determine if the object is of the appropriate class (the "String?" test in Figure 3.4) represents the greatest percentage of the maintenance time. This is caused by the fact, that every single object has to be looked at, their class pointer has to be followed and then a string comparison performed to determine if the class is of class "java.lang.String". In comparison, an unchanged flow only considers whether or not the object is referenced. The process can be represented using the following pseudo code:

```
class = object.getClass();
className = class.getName();
if(className.equals("java.lang.String"){
   execute_deduplication_flow(object);
}
```

Reading the class name of each object and comparing it using a string comparison is very slow. Additionally, performing the test for every single object passed during traversal inflates the process to become a very significant factor. To reduce execution time, the algorithm was changed to perform class

checking through string comparison only once and to remember the reference of the string class as soon as it is found. Since class objects are not moved during execution time, the algorithm can use two simple address comparisons after the class pointer is stored. The algorithm was changed to the following:

```
class = object.getClass();
if(sdClass == NULL){
 classname = class.getName();
 if(classname.equals("java.lang.String"){
   sdClass = class;
 }
}
if(sdClass == class){
   execute_deduplication_flow(object);
}
```

This change and minor source code restructuring were tested. Both results were compared to an unchanged Java VM running without JIT compilation. Table 4.1 shows a comparison of two GC-time string deduplications to an unchanged JVM. The first column shows the impact of the string deduplication when using a string comparison for class analysis and without the code restructuring performed based on profiling. It is shown that with the exception of the `lusearch` benchmark, the applications show a medium to large overhead. The second column provides the relative impact of the updated

	Old	New	Change
jython	+7.25%	+5.13%	-2.12%
lusearch	-5.63%	-7.90%	-2.27%
pmd	+16.26%	+12.56%	-3.70%
sunflow	+19.96%	+17.08%	-2.56%
xalan	+28.92%	+27.33%	-1.59%
Geometrical Mean	+13.35%	+10.84%	-2.45%

Table 4.1: The relative impact of the optimization on the runtime compared to running the benchmarks in an unchanged VM. Results are relative changes to an unchanged VM. The comparison is between the old version of the string deduplication approach compared to the updated version as described in Section 4.2.

string deduplication flow on the baseline configuration.

The aim of the improvements described above was to reduce the overall overhead of the approach on the applications that do not provide enough string objects to create a benefit by deduplicating those objects. As seen in the difference column of Table 4.1, the overhead was decreased for every benchmark tested from a geometrical mean of the overall impact of +13.35% to -2.45%. The largest change was achieved for `pmd`, where the overhead was reduced by 3.7%. The execution time cost by `lusearch` was reduced by 2.27%. The change of the updated deduplication flow averages at a reduction of 2.45%.

Even though an overhead reduction was achieved, the slow down of the application caused by the creation and maintenance of the string deduplication hash table is still too high. In order to gain a benefit, the application's heap has to have a very large number of strings. Those strings have to include a

large number of duplicates. A potential solution for distinguishing when to utilize and when to disable string deduplication during runtime is discussed in Section 4.4.

4.3 Object Age Metric

A property often used in virtual machine memory management is that objects often die at a young age. However, the longer they stay alive, the higher the probability that they will survive even longer [32]. Using this property, the string deduplication approach can be modified to consider the object age while analyzing string objects. Object age in this case is defined as the number of garbage collections survived.

A configurable object age threshold is added to the deduplication flow to reduce the number of objects passing through the stages of hashing, deduplication and hash table storage. The idea behind this is to ignore temporary and short lived strings, which are analyzed, but die shortly after the garbage collection. According to Wilson and Moher [62] (Figure 4.1), only a small proportion of objects survive the first and the second garbage collection. This leads to the idea that many string objects can be skipped during the process when only considering objects older than ages one, two and three.

Figures 4.2 and 4.3 show one experimental result of this investigation (DaCapo `lusearch`). Figure 4.2 depicts the number of characters processed for each garbage collection during the execution on a logarithmic scale. The

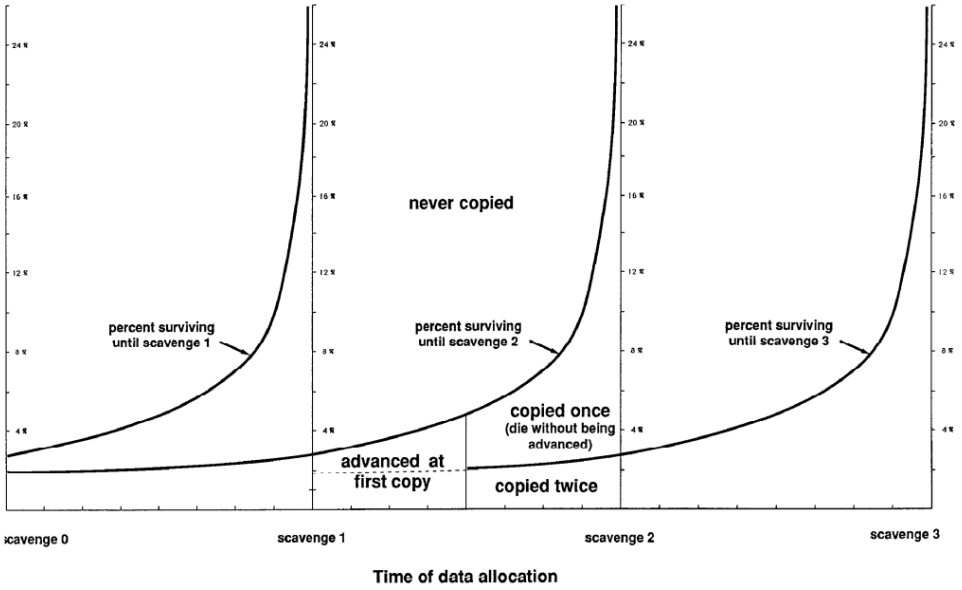

Figure 4.1: The survival of objects at ages one and two. The figure shows that in general, only a small fraction of objects survive the first two garbage collections [62].

numbers were gathered during the traversal of objects. Once determined that the current object is a string, the content of the string has to be loaded to create the hash value and to compare the content to a potential duplicate. The number of characters relates to the amount of time needed to complete this process for each garbage collection. While objects in Java provide a hash value for each object, the research chose to create a separate hash value based on the actual content of the string to use it as a preliminary comparison value during duplicate detection. The default hash value of a string object within the JVM does not necessarily have to be based on the content of the char array attached to it.

Figure 4.3 shows a similar graph. However the number shown for each age threshold and garbage collection is the number of characters that did not have to be moved for the particular garbage collection as they were duplicated. The numbers gathered while running the `lusearch` benchmark were chosen since it had the most garbage collections and the benchmarks created the most duplicates.

While a larger number in Figure 4.2 signals an increase in time spent for string deduplication, a larger number in Figure 4.3 shows how much time was saved by not copying the string object content. The benefit potentially extends to successive garbage collections, as surviving objects now occupy less memory that has to be evacuated.

It was found interesting that most duplicate strings seem to be created during the virtual machine startup. As seen for `lusearch` in Figures 4.2 and 4.3,

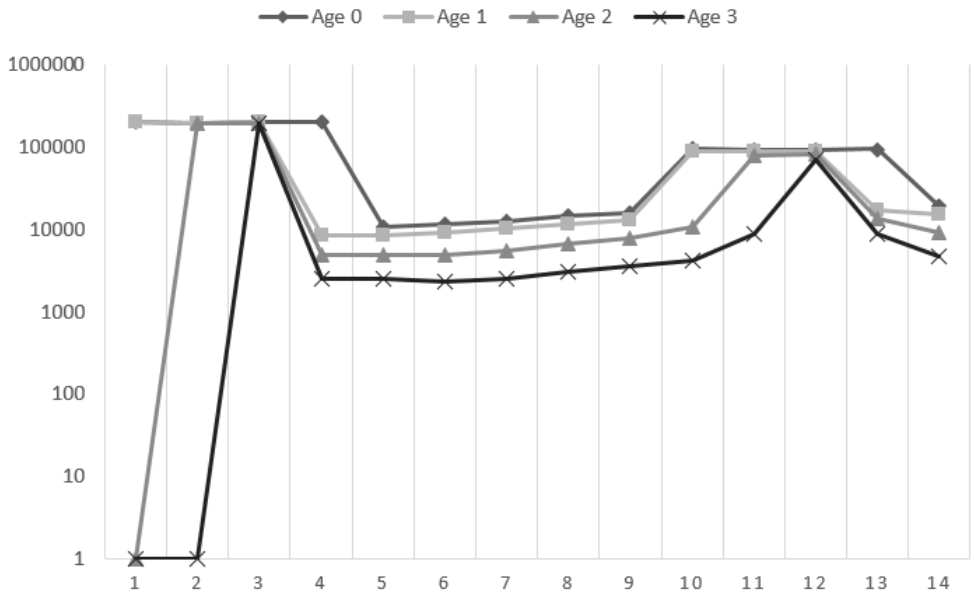

Figure 4.2: The number of characters processed during the deduplication process. The object age thresholds between one and three for the lusearch benchmark was used to consider only objects that survived at least a certain number of garbage collections. The Y axis shows the number of characters encountered during the traversal on a logarithmic scale. The number includes those characters deduplicated and not deduplicated encountered during the traversal process. The X axis represents the garbage collection number. The number of characters instead of strings was chosen to take into consideration the length of the strings and therefore the potential benefit.

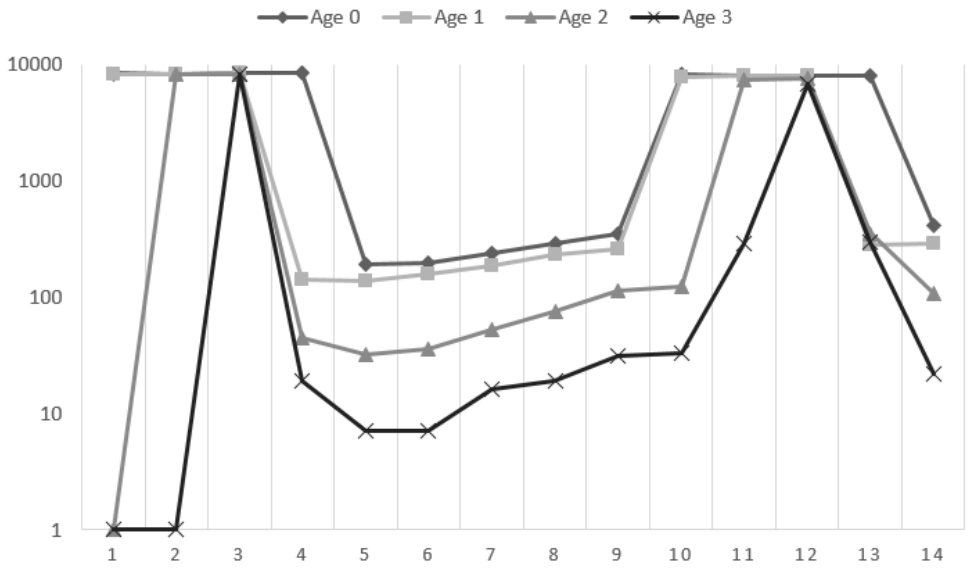

Figure 4.3: The number of characters deduplicated for each GC when using different object age thresholds. The object age is determined by the number of garbage collection phases an object survived. A higher threshold reduces the number of objects processed while executing the lusearch benchmark. The X axis represents the garbage collection number while the Y axis shows the number of characters deduplicated on a logarithmic scale. The number of characters instead of strings was chosen to take into consideration the length of the strings and therefore the potential benefit.

it was observed that multiple benchmarks had garbage collections where the graphs would either meet or reach an unusually high value in quick succession. In the included figures, one can observe this behaviour for garbage collection numbers 3 and 12. It appears as those are the points where many strings are created and that the percentage of duplicates at those points is larger than in other phases of the application flow. The observation led to the idea to investigate an approach for targeting those execution stages specifically as described in Section 4.4.

In general, increasing the age threshold reduces both the processed and deduplicated metric as it omits very young strings. It is observed that the difference between threshold ages zero and one is smaller than the differences between the ages one and two, as well as two and three. The difference between ages three and higher values was observed to be much smaller, except for the very first cycles where no object had time to age.

Table 4.2 shows the difference of execution times of JVMs utilizing different age thresholds relative to not considering the object age for string deduplication as objects are traversed during garbage collection. The numbers indicate that a check for object age can impact the overall execution time of a benchmark by plus or minus 3.55%. As indicated in Jones et al. [32] when talking about excluding young objects from potential optimizations in general, the largest positive impact on the average execution time was found to be between one and three.

Choosing a threshold age represents a trade off. The larger the threshold,

	Age 1	Age 2	Age 3	Age 4	Age 5
lusearch	-2.00%	-2.53%	-0.12%	-1.29%	+0.29%
pmd	+0.05%	-0.10%	+0.28%	-0.07%	-1.20%
sunflow	-0.58%	-2.28%	-2.84%	-0.92%	+0.75%
xalan	-0.18%	-0.08%	-0.29%	+3.55%	+2.89%
Average	-0.68%	-1.25%	-0.74%	+0.32%	+0.68%

Table 4.2: Relative change of the execution time when increasing the age threshold. The configurations are compared to not utilizing an age threshold. The last row shows the average of each column. It can be seen that starting with the object age threshold of 4, the average execution time increases.

the fewer objects included in the string deduplication process and the deduplication hash table. While this reduces the overhead of the process itself, it limits the opportunities for deduplication.

As seen when comparing Figures 4.2 and 4.3 to each other, the shape of the graphs is quite similar. This indicates that a larger number of characters processed results in more characters deduplicated. The processing cost has to be invested in order to create a benefit for the currently running and successive garbage collections. The cost and benefit have to be balanced. Based on the average runtime change of -1.25% shown in the last row of Table 4.2, the threshold age of two was found to be the most beneficial.

The main observation of this experiment is that checking for an age threshold slightly reduces the previously found benefits and disadvantages. Applications that gained a benefit without the age threshold perform slightly better at ages two and three, while benchmarks with a less duplicated heap structure tend to perform slightly worse compared to not using string deduplication or remain practically unchanged. The overhead to applications that do not

benefit from string deduplication overall is reduced. As with the initial approach of string deduplication as part of the garbage collection phase, the best set of parameters can vary drastically based on the mutator's execution flow and heap structure.

4.4 String Deduplication Sampling

Deciding whether or not to use string deduplication is not trivial and potential benefits can be easily missed. An algorithm that performs string deduplication tests and enables or disables string deduplication is more likely to find potentially beneficial mutators.

One method is to choose certain garbage collection cycles and perform string deduplication while gathering statistics about the number of characters processed and the number of characters deduplicated. Based on the ratio of those two metrics, a potential benefit can be identified and used.

The approach investigated and shown in Figure 4.4 proposes to enable string deduplication for the first garbage collection that would deduplicate potentially surviving strings. The actual first GC where string deduplication is performed depends on the age threshold chosen for the configuration. Performing a deduplication during the first garbage collection while strings have to survive at least two GCs to be considered for string deduplication, would simply result in zero strings processed.

Once the first potentially viable garbage collection is processed, the ratio of

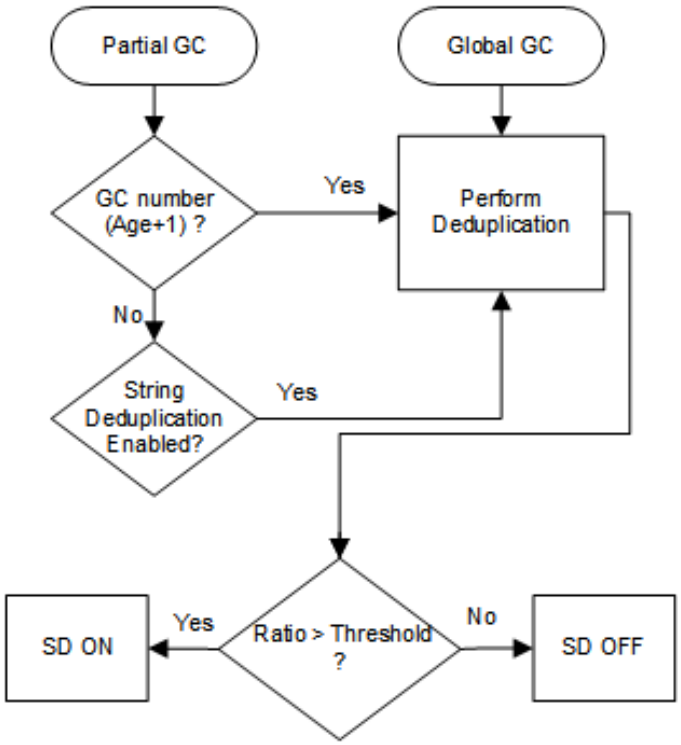

Figure 4.4: The steps involved in deciding whether or not string deduplication should be enabled for each garbage collection. The first potentially viable garbage collection and global GCs always perform a string deduplication in order to identify application phases with many string duplicates. Based on the ratio of characters processed and deduplicated, a string deduplication for the next garbage collection can be enabled or disabled. Connections leading to the end of the process were omitted for readability purposes.

characters deduplicated versus the number of characters processed can be utilized to decide whether or not string deduplication should be enabled. If the ratio is high enough, string deduplication can be performed for each successive collection until the ratio falls below the threshold chosen.

Should the ratio be below the threshold, string deduplication can be disabled, which eliminates the overhead of creating and maintaining the string hash table. This procedure reduces the runtime overhead and aims to provide a more dynamic optimization. As the mutator continues its execution, the heap structure can change.

Applications may change from not string heavy to string reliant, making a deduplication phase more viable in later stages of the execution. In order to detect such changes, global collections can be used as sampling points. There are multiple reasons for using global garbage collections. The first one is the fact that all strings on the heap are processed. A deduplication is more likely to succeed as each string content will be a potential subject for deduplication. Another reason is the length and frequency of global collections. They usually happen once the heap is filled to a high degree and have the goal of freeing as much memory as possible. Deduplicating strings would increase the potentially freed heap. Once the global collection is completed, the deduplication ratio can be computed in order to decide whether partial collections should now perform string deduplication or not.

The approach described was implemented in IBM's J9 Java VM and tested using the benchmark set described in Section 4.1. The relative change in

	Baseline	Dynamic SD	Change
lusearch	55.80s	51.36s	-7.96%
pmd	64.14s	62.87s	-1.98%
sunflow	155.80s	157.33s	+0.98%
xalan	28.11s	26.88s	-4.37%
Average	75.96s	74.61s	-3.33%

Table 4.3: Benchmark execution times in seconds. The dynamic SD approach represents the collection of all optimizations presented in this research combined with the deduplication sampling flow. The third column shows the relative change between the experimental results.

runtime compared to a baseline virtual machine is shown in Table 4.3.

The experiments were performed using a deduplication ratio threshold of 15%. Using the sampling flow presented, the overhead was reduced by computing a deduplication ratio and disabling the string deduplication if the maintenance seemed less likely to create a benefit.

The chosen threshold was discovered using a set of experiments starting from 0% to 30% in increments of 5%. The threshold aims to detect when keeping the hash table of all strings is too costly for the benefit gained from deduplication. The higher the threshold is set, the less likely it is that string deduplication is enabled. Setting the threshold to 0% means that no characters have to be deduplicated for the process to remain active.

Using a threshold of 0%, the preliminary experiment produced the same numbers as the numbers presented in configuration C of Table 3.2. The deduplication procedure remained enabled throughout the whole execution time for all benchmarks. Increasing the threshold to 5% changed the run-

times for `pmd` and `sunflow` as those benchmarks did not discover enough deduplications to justify leaving string deduplication active after the first garbage collection.

At a threshold of 15%, `xalan` changed the behaviour to disable string deduplication after the first garbage collection. At a threshold of 20%, `lusearch` stopped reaching the required threshold and the results changed to represent those in configuration A of Figure 3.2.

Thus, 15% was chosen as a compromise of keeping the speedup of the string heavy application in the benchmark set, while minimizing the impact of the optimization on the rest of the applications. This value is application set dependent.

The cost for string deduplication is keeping track of all strings. This means traversing them, indexing their content, looking for duplicates and performing the deduplication. If the percentage of duplicates compared to the overall character number is too low, maintaining the metadata costs more than deduplication creates in benefits. This leads to a prolonged overall execution time.

According to the experimental logs, `lusearch` was the only benchmark that continued deduplicating strings throughout the execution time. The number of strings and duplicates throughout execution time is presented in Figures 4.2 and 4.3.

All other benchmarks disabled the deduplication after one or two garbage collections. While an overhead of 0.98% was still present for `sunflow`, it is

considerably smaller than the 17.08% shown in Table 4.1.

Benchmarks `pmd` and `xalan` show a decrease in runtime by 1.98% and 4.37% respectively. The performance can be explained by the fact that strings created during the startup of the application are deduplicated. The benefits of this deduplication can be explained by the overall time the application ran after the initial deduplication was performed. Since the deduplication happened at the start of the application the time taken for garbage collection throughout the application was slightly reduced as fewer objects had to be copied or compacted. This carried through the whole application runtime and accumulated this slight improvement.

There are multiple explanations for the behaviour shown in Table 4.3. It indicates that many of the duplicates are created during the JVM startup as discussed previously. Another point would be that duplicate strings are less likely to be recreated. They do not seem to be objects which die off after a short period of time. This supports the process of utilizing the global garbage collections for string deduplication.

4.5 Application Specific Analysis

In order to understand what makes an application beneficial for string deduplication and why lusearch gained a benefit while other benchmarks did not, two applications were written to demonstrate the behaviour. The applications process the output file of Odin II [31, 41] saved using the Berkeley

Logic Interchange Format (BLIF) [2]. Files saved using this format describe a hardware circuit including inputs, outputs, nodes, connections between the nodes and their functionality. In order to reduce the size of the output file, the names of the wires connecting nodes are detected, shortened and replaced throughout the file.

This procedure is done using two algorithms. The first one, shown in Figure 4.5, processes the file line-by-line, detects wire names and adds them to a String-to-String map table. The structure maps the long version of the name to a newly created short one. Once done, all future occurrences of this name are replaced as the application continues. The file is processed once while the map structure continues to grow. The output is written in a line-by-line manner. The algorithm is further referred to as quickshort.

The second algorithm achieving the same result is shown in Figure 4.6. The algorithm reads in the contents of the whole file. The content is then processed and wire names are detected. As soon as a name is found, a short version is created and all occurrences replaced throughout the file. The contents of the file remain in memory and the contents are processed multiple times in order to locate and replace all instances of the name processed. Due to the nature of the algorithm, it is further referred to as slowshort.

While quickshort is the more elegant way of achieving the goal of this specific application, the two approaches attempt to demonstrate two varieties of string heavy applications. Both use String objects almost exclusively. While quickshort mostly uses small objects with a short lifespan while performing

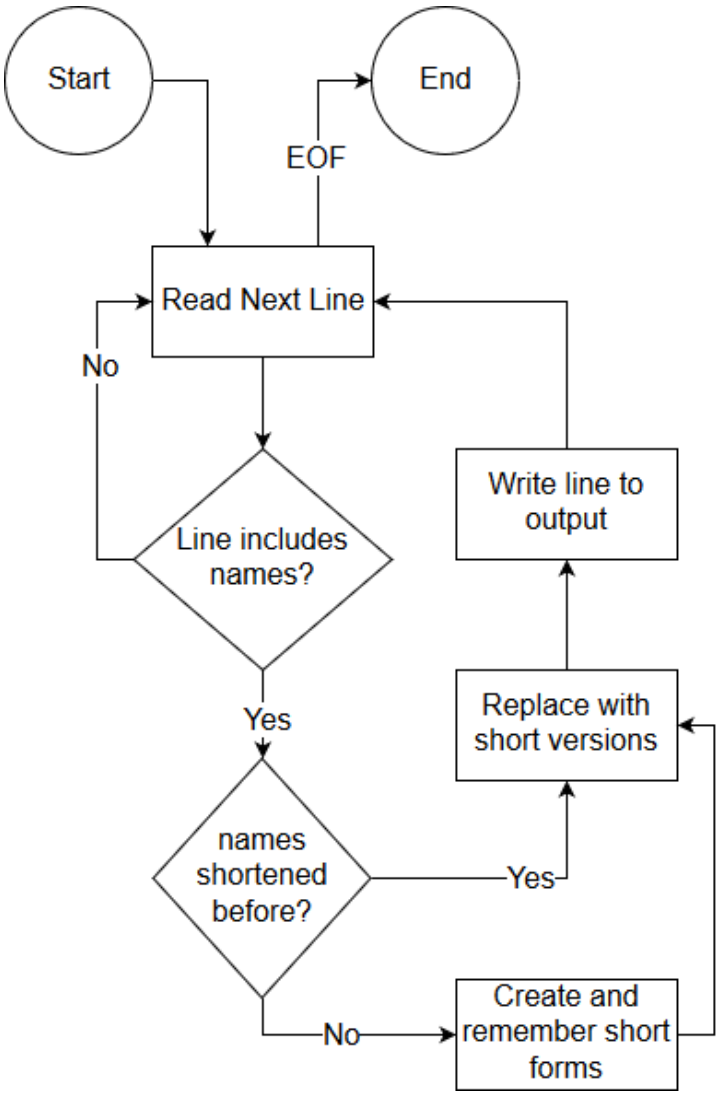

Figure 4.5: Algorithm flow of the quickshort application processing a BLIF file to shorten wire names.

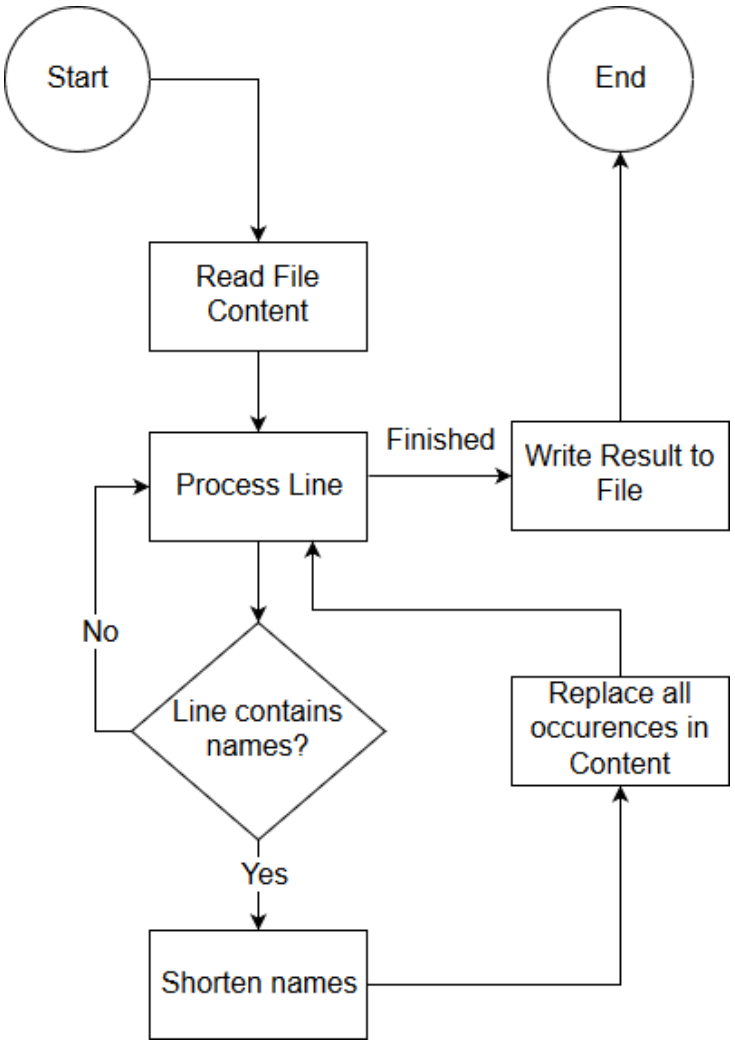

Figure 4.6: Algorithm flow of the slowshort application processing a BLIF file to shorten wire names.

	Baseline	SD enabled	Relative Change
slowshort	24:15 min	22:09 min	-8.7%
quickshort	1:37 min	1:39 min	+2%

Table 4.4: Experiment results for the quickshort and slowshort algorithms. A comparison of execution times and the relative impact of enabling string deduplication is presented.

IO operations throughout its execution time, the second flow has very long lived string objects and two IO phases.

The applications were executed using a BLIF file with 7.5 million lines and an overall size of 1.5GB. The virtual machine utilized was limited at 4GB of heap space with the just-in-time (JIT) compiler disabled. Both applications used an unmodified VM as a baseline and a configuration with string deduplication enabled. The ratio chosen for enabling string deduplication for partial collection was left at 15% as used in Section 4.4. The object age required to be part of the string deduplication process remained at two survived garbage collection phases as described in Section 4.3. Each execution was repeated 10 times in order to reduce noise factors.

Both approaches used the same input file and produced the same output file. When running in the baseline VM, the slowshort required about 15 times longer to perform the task due to the nature of the algorithm. When looking at the results with string deduplication enabled, it was found that the quickshort approach did not enable the string deduplication process as not enough duplicate strings survived long enough. Further analysis showed that a large number of duplicates was present at the precollection phase, but

almost none of them survived the collection. This can be explained by the fact that newly created strings are only used to process a single line of the input file before being discarded. The overall runtime of the application remained almost unchanged with a slight increase averaging at about 2% (Table 4.4) due to slightly prolonged global collections and string deduplication sampling checks.

The slowshort approach took a much longer time to process the file as it worked in memory and processed the content of the file many times. As strings survived longer while occurrences of each wire name were located in the content, the string deduplication approach was left enabled starting with the first global garbage collection. The impact of the deduplication flow relative to the baseline averaged at approximately 8.7% (Table 4.4). While this is an artificially constructed comparison, it can provide indications of application structures, which can lead to an improvement when using string deduplication. In addition, it provides insights of how even string heavy applications such as database servers (`tomcat`) and text indexers (`luindex`) can still have a structure not containing enough duplicates to justify using garbage collection time string deduplication.

4.6 Conclusions

This chapter presented multiple potential improvements to the GC-time string deduplication approach. Expensive operations performed on every

object traversed during garbage collection were reduced and the source base refactored in order to speed up the process. An object age metric was introduced in order to only process objects older than a threshold. Multiple threshold numbers were investigated and experiments showed that object age thresholds of two and three offer the most benefit.

Finally the string deduplication sampling algorithm was discussed. The aim of the algorithm is to use string deduplication for applications, or certain runtime phases of applications, which could potentially benefit from it. Should the sampling flow discover that the overhead of the execution time appears to be too high, no deduplication is performed and the overhead is minimized. Experiments combining all optimizations show an improvement in the overall approach. String heavy applications perform slightly better, while the overhead for other applications was reduced by a large margin. Applications that have duplicate strings at the start of the application but fewer during the execution itself also show an improvement. Additionally two artificially generated algorithms were written and presented to provide insights into the application structure necessary to benefit from enabled string deduplication. Future work in the field could include an investigation of a moving threshold and the timing of garbage collections to determine whether or not to enable string deduplication. The metrics gathered and used during runtime could be stored and used for future JVM executions.

Chapter 5

VM Internal Object Pools

While Chapters 3 and 4 focused mainly on utilizing the immutability of string objects in Java to improve their structure, they left out the rest of the objects on the heap. While memory managed environments offer the comfort of not performing manual memory management and analysis, the performance gap between manually managed applications and those executed in a virtual environment is still large. This is caused mainly by the fact, that garbage collection phases halt the application in order to clean up the heap space. Using the necessary memory management phases of a virtual environment to optimize application flow and execution is often a way to mitigate the cost of memory management itself.

As resource management is an established problem in computer science, there are additional software engineering patterns that are used to handle the problem of creating and destroying objects while reducing most of the creation

cost. One of those patterns is an object pool, which is generally created and maintained by a developer. Instead of allocating and freeing objects, they are kept in a buffer and requested when an instance is needed. Once the object is no longer used, it is not freed, but returned to the pool so another part of the application can reuse it without going through the allocation process. The main drawback of this approach when used in virtual machines is that developers have to perform manual resource management, which is contrary to the usual paradigm of applications running in a memory managed environment where memory handling is performed automatically. This chapter investigates the feasibility of an approach to perform the optimizations offered by object pools as part of the virtual machine.

The chapter is divided into five sections. Section 5.1 provides an introduction into the field of research, different memory management techniques and existing object pool approaches. Section 5.2 summarizes the benefits and disadvantages of the existing approaches and underlines the motivation of the approach presented. A case study investigating potential gains using this approach is presented in Section 5.3 while Section 5.4 describes the design, implementation and results of the object pools using two different garbage collection policies. Section 5.5 concludes and suggests some opportunities for future work.

5.1 Related Work

The object pool pattern [24] is a software engineering technique frequently used in environments with shared, but limited resources. The idea is to keep a constant pool of resources, which are created only once and to distribute those between users if requested. The most common uses of the pattern are thread pools [55] and connection pools, such as database connection pools [55, 22].

The general idea of the software pattern is to mimic an allocation and deallocation command. The basic class structure of the pattern is shown in Figure 5.1. All object instances are created by the pool structure during initialization time. Each instance is then requested by different mutator parts using the `acquire()` function. The function can be implemented in a blocking or non blocking fashion. Once the resource is no longer being used, the mutator can return the resource to the pool using the `release()` function.

The reason for the popularity of object pools, especially in areas such as database connections and threads, is the time it takes to instantiate such an object. The creation of a thread takes time, but the communication between threads is fast once the threads exist. This applies to databases as well. Establishing a connection to a database can take one if not multiple seconds. However, once the database connection exists, communication between the program and the database is very fast.

In addition, most applications do not benefit from more resources beyond a

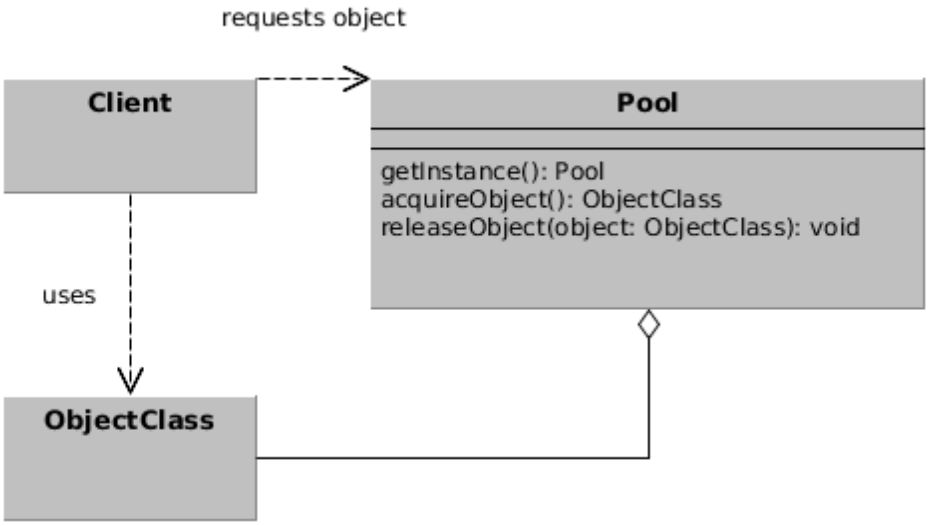

Figure 5.1: Object Pool pattern in UML. The Pool class creates objects that are then requested and returned by the client class.

certain point. Once a certain number of threads is reached, an increase would slow down execution as processor time is becoming the bottleneck resource. The same applies for database connections. If too many users attempt query commands, the database response time may become slower. This allows the runtime to predict and manage a suitable number of connections for the application balancing the number of users issuing queries while limiting the number of concurrent requests to the database to minimize response time.

This can be translated into other applications. The pattern can be found in many computer and cell phone games [56], often used in order to keep a pool of objects displayed to the user, such as bullets, enemies, etc. This limits the number of objects on the screen, ensuring good performance, fast response

times, minimizing garbage collections and therefore providing a better experience for the user.

Development using this pattern changed from implementing the pool structures and all security measures necessary to using prepackaged APIs such as Apparatus [52] or the Data Base Connection Pool API offered by Apache [22]. APIs providing this functionality are implemented in the application and are part of the mutator. Applying the pattern from the virtual machine side would reduce implementation time and could save memory on the heap as the structures required would not have to be part of the mutator. In addition, garbage collection work would be decreased as fewer floating garbage objects would be created.

5.2 Motivation

Both threads and database connections take a long time to be created, which makes it more profitable to keep the memory overhead down in order to save a significant amount of time [40, 25]. Even though implementations such as `java.util.concurrent` and Apache's DBCP component [22] are well known beneficiaries of the pattern, it appears that there are many other applications that could gain a significant benefit using object pools.

As described in Section 5.1, different APIs aim to provide predefined structures in order to provide the functionality of object pools without the requirement of a complete implementation of all structures needed [35]. The

problem with the object pattern, however, is that developers are responsible for the management of objects and memory they are utilizing. This offers freedom for the developer as resources can be utilized where they are being used without the need of freeing memory and reallocating it again. As soon as the developer knows that an object is not part of the execution any longer, the object is released to be part of the pool again instead of simply forgetting about it.

There are multiple disadvantages in the use of object pools [36]. The first is development time. Even using available APIs, keeping track of object references, their accessibility and usefulness costs development time, which is usually avoided by using an environment with automatic memory management.

The second disadvantage of the manual management of object pools is mistakes. Mistakes in pool sizing can either mean more allocations than necessary or, in the worst case, a deadlock which cannot be resolved. This might happen if the object pool is limited to a certain number of objects and is not allowed to grow [36]. If pooled objects are enforced during development, a scenario can occur where all resources are lost because a part of the mutator does not return unused objects. In the worst case, references to those objects are lost and freed by the garbage collector. In this case, the program would not be able to continue since no free instances are available. Another disadvantage is that retroactive addition of this optimization is very hard to achieve, as all resource flows have to be examined very carefully in order to

avoid the problems mentioned above.

Those reasons led to the idea of investigating an approach of providing the optimization offered by the object pool pattern within the managed environment. Its tasks already include managing the mutator's memory directly. Instead of implementing APIs and managing the references to the objects, the virtual machine could be configured to perform this while it manages all other objects, references and memory for the mutator.

5.3 Preliminary Case Study

A preliminary best case test application was developed in Java to measure the potential of object pools. The application simulates a self playing computer game without a graphical interface, where entities spawn pseudo-randomly and have to be removed by the player. The player component was performed by the application as well. In this way, entities of the `Enemy` class were created and removed at pseudo-random. The maximum number of existing enemies was predefined. The application flow is shown in Figure 5.2.

The application was configured to perform a varying number of steps and the wall time (execution time measured by the operating system) of the program run was measured as the baseline. Each run was performed 25 times in order to reduce the noise factor of the resulting times. A heap size of 2GB was used and the garbage collector performed a concurrent generational collection.

Once completed, the use of the `Enemy` class was modified to utilize the object

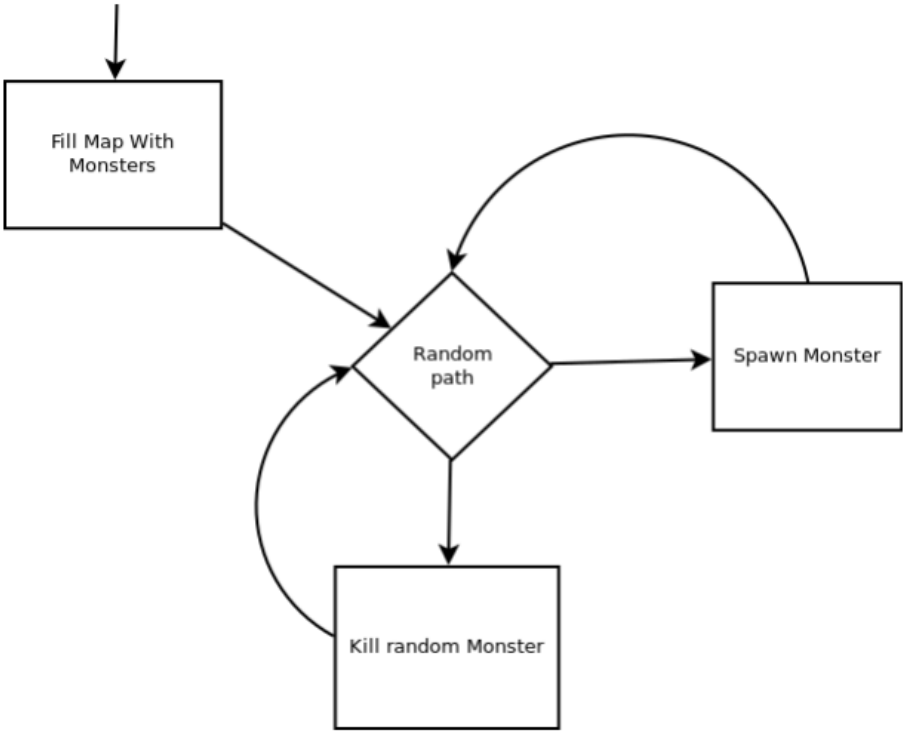

Figure 5.2: Test application structure. Enemies are spawned and killed off at pseudo-random. The Enemy class is pooled and compared to a configuration relying on garbage collection in order to determine the potential benefit of internal VM pools.

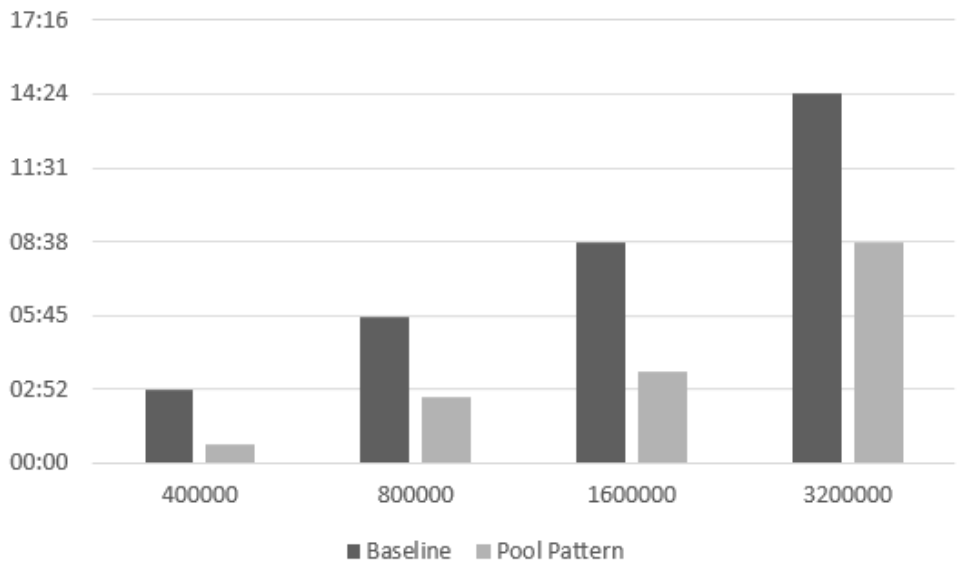

Figure 5.3: Preliminary results comparing the test application runtime when relying on garbage collection compared to using the object pool pattern to store all instances of the enemy class.

pool pattern. Instead of allocating enemies, an instance of the class was requested from the pool. Once an enemy was killed, the instance was returned to the pool. The `acquire()` function was extended to include all constructor values of the instance requested and was reinitialized by the pool before returning it to the client class.

The difference between the two runs is significant. As the application did not have to perform any garbage collection aside from collections in the young space regions of the heap and object promotion to old space, the mutator was almost never halted for a stop-the-world phase. As shown in Figure 5.3 and Table 5.1, runtime results were decreased by an average of 56.57%.

In addition, the number of garbage collections dropped by 90% as most

	Baseline	Object Pool	Speedup
400,000 steps	02:53.71	00:46.30	73.35%
800,000 steps	05:42.35	02:35.53	54.57%
1,600,000 steps	08:37.82	03:35.76	58.33%
3,200,000 steps	14:26.01	08:38.19	40.02%

Table 5.1: Runtime comparison (in min:secs) between the base line application and the Enemy class pool implementation. Each result is the average of 25 runs.

objects required by the mutator application did not have to be cleaned up and reallocated. A small number of partial garbage collections still happened to remove child objects of the pooled objects.

The experiment shows a best case scenario since the pool size was large enough to store all instances of `Enemy` and the application did not perform anything but object allocation and deallocation. In addition, the task was performed by a developer who knew when resources were needed and discarded. Performing this task in such a way by a virtual environment with garbage collection is unlikely, but creating even a portion of the benefit shown would potentially decrease the runtime of a set of applications. The cause for the challenges is the fact that the application flow is not considered by the memory management component of a virtual environment. Objects are usually treated as interconnected chunks of memory. While some metrics are gathered, none of them include the purpose of an object.

5.4 Design and Implementation

Moving the algorithm presented in Section 5.3 from being a mutator based implementation to an algorithm provided by the managed runtime implies adjusting the memory managing part of the environment, namely the allocator and the garbage collector.

The allocator and garbage collector work together to implement a specific garbage collection policy. The allocator's main tasks are to handle memory requests from the mutator, to allocate the memory and to provide a reference to this address space to the mutator. Should there be no contiguous free memory available, the garbage collector is notified and is responsible for a memory clean up.

Garbage collectors traverse the heap or a subset of the heap in order to identify which objects can be referenced by the client application. Those objects are marked alive and therefore kept on the heap. The memory not occupied by those objects can be reclaimed and therefore used for new allocations. The process of detecting live objects and freeing memory is usually performed during a stop-the-world phase where the client application execution is halted.

Modern garbage collection policies divide the heap into multiple regions with the goal to minimize the stop-the-world phase. Fragmentation is prevented by evacuating traversed objects into a different region (evacuation space) and marking the complete old region (collection space) as free. This frees up a

full region of contiguous space and compacts live objects in the process.

The challenge in this process is that unreferenced objects are not detected as such. Only live portions of the memory are identified before the rest is collectively marked as unused without identifying the objects stored within. This means that the algorithm used in Section 5.3 cannot simply be copied over into the managed environment infrastructure. The functionality of the `Pool` class shown in Figure 5.1 therefore has to be split into multiple pieces, which change the allocator and garbage collector based on the GC policy. The following subsections will demonstrate the design, implementation and discussion of the object pool implementation for two specific garbage collection policies provided by the IBM Java J9 VM.

5.4.1 Overall Approach Structure

As the new process does not rely on developers to use it, detectors have to be created within the managed runtime to enable the use of object pools. Three parts of the environment have to be changed: the managed environment start up phase, the allocator and the garbage collector.

5.4.1.1 Environment Startup

The startup phase of the environment has to be adjusted to set up the structure holding and managing the pool objects. This includes a locking mechanism, reference holders for objects and a collection of free and available objects.

The object pool size is provided during startup. There are two possible paths of pool creation: a filled pool structure and a self filling pool. The filled pool structure allocates the maximum number of objects right away and includes them in the free list. The benefit of this approach is faster acquisition of objects during mutator runtime at the cost of an inflated start up time and initial heap size. However, with the objects already on the heap, the probability that they are located close to their future user thread's other objects is low. This can cause worse object locality and therefore increase the probability of cache misses and page faults [19].

The second approach is to initialize an empty set of references, an empty free table and fill the pool during execution time. The tradeoff is a mirror of the creation of a filled pool. The impact on the start up time is reduced and the computational overhead during allocation is applied dynamically based on application behaviour. However, during the initial execution stage, while the pool is being filled with objects, allocating an object of the pooled class will cause a more complex procedure and therefore require more computation. This research focuses on utilizing the second approach as the dynamic nature of time overhead applied is more suitable for short applications as well as applications which have multiple behaviour phases during runtime.

5.4.1.2 Allocation

The `acquire()` function of the object pool pattern shown in Figure 5.1 is included in the allocation process of the environment. The decision of which

objects are pooled and how many of them are included in the pool is provided when the environment is started. A startup argument with the class name(s) is provided in order to decide which allocation method is chosen for the specific object.

In addition to the garbage collection specific policy on where new objects are allocated, there are two main allocation flows in the IBM Java VM: slow path allocation and fast path allocation. Slow path allocation treats the new object space as a shared resource, looks for a large enough memory space for the current request and allocates it before returning the start address to the mutator. Since the free space is shared by all mutator threads, this procedure relies on locking and one request is handled by the allocator at a time. This can be further improved by having multiple locks for different regions if multiple new-object-regions are available.

The fast path allocation aims at reducing locking by allocating a relatively large piece of memory for each running thread. The allocation of this thread local heap is performed using the slow path allocation. Once received by a thread, it is used to place newly created objects of this thread inside this piece of memory without the need of locking. In addition, objects created by the thread are placed closer together and have therefore better object space locality.

In these experiments, an additional stage is introduced before one of those routes is taken, which is initiated when an allocation of a pooled object is requested by the client application. In this stage the pool is checked for an

available object. If there is one, the memory of the object is set to 0 to comply with the Java language specification and returned to the mutator, where the constructor is executed in order to prepare the object for its task. Should no object be available, the number of objects registered by the pool is determined. If this number is smaller than the maximum capacity of the pool, a new object is allocated and added to the pool before it is assigned to the mutator. The pool therefore grows. In the last case, where no pooled objects are available and the maximum number of objects in the pool is reached, the original VM allocation flow continues as it would without the use of a pool structure.

Using this flow increases the amount of work performed for pooled object allocations. The idea behind it is to prevent the application from utilizing slow path and fast path allocation in as many cases as possible and therefore let the thread local heap last longer. This reduces the number of locking procedures during allocation and the amount of new memory required during execution.

The pool size has to be chosen carefully. A small pool increases the probability of using the new preliminary stage of the allocation without getting a frequent benefit. A too large pool is less costly during allocation as new objects are created and the pool is filled. However, a large pool means more objects on the heap, which are kept alive by the fact that they are part of this pool. Even if the mutator cannot reference them any longer, they are kept in the free table of the pool waiting for the next allocation request of

the user. This can potentially trigger additional garbage collections.

5.4.1.3 Garbage Collection

The garbage collection component holds the most challenges to the approach. Its tasks are detecting pooled objects that are about to be collected, marking them as alive and adding those objects to the free table of the pool structure so they can be reused.

This creates a slightly unnatural behaviour of the garbage collector as traversing floating garbage objects is usually considered wasteful computation. Usually, live objects are evacuated before a full region is marked free. Using this approach, the number of objects evacuated is increased by the number of pooled objects in the evacuation region.

The main loss in performance when comparing the case study and the automatic approach is the time between the object becoming not referenceable by the mutator and the time the object is saved from a garbage collection. The shorter this time, the closer the performance gain shown in Section 5.3 can be reached.

In many collection techniques, memory regions are split into object age groups. An object ages when it survives garbage collections. Objects that survive a large number of garbage collections are more likely to survive longer. The same behaviour can be used by the pool to prevent a scenario where objects that are likely to never die or not die for a very long time crowd the pool. In doing so, the pool is not serving its purpose of providing preallocated

memory for frequently allocated objects. There are two main metrics used to estimate how long an object will survive at a certain point of execution: the size of the object and the age of the objects. The age is indicated by the number of garbage collections survived by the object.

Since the size of pooled objects remains stable, the remaining metric is the age. Age is used at multiple points in the virtual environment. Most of the time, the decision about where to relocate an object during a garbage collection is made using this metric. In the IBM J9 Java VM, the object age is used in GenCon to determine when it can be considered old enough to relocate to the less frequently collected old space. Starting with this relocation, the frequency of detecting this object's alive status is drastically reduced to prevent unnecessary GC activity.

Once a pooled object survives this long, the probability of it ever returning to the pool as a free object decreases. In such cases, this object can be removed from the pool and the slot used by a more frequently returning instance of this particular class.

5.4.2 GenCon Object Pools

GenCon is a garbage collection policy in the IBM Java J9 VM that couples a generational garbage collection policy with a concurrent mark stage. It is the default GC option in the IBM Java J9 VM. A young space and an old space are maintained to reduce the size of regions that are collected frequently.

This subsection describes the implementation of object pools using GenCon

and discusses the results gathered from experiments. This GC policy was chosen as the preliminary tests showed potential using this policy and because it is the default option for applications running on the IBM Java J9 VM. In addition, it was found to be the best choice for most general application scenarios by Neu et al. [51].

5.4.2.1 GenCon Pool Design and Implementation

The basic structure of the heap, allocation policy and structures as well as the garbage collection are prepared for execution during the VM startup phase based on parameters configured by the user. A mutator does not have an opportunity to change the heap structure and object allocations/deallocations are treated as a given by the mutator developer.

Allocation is performed based on programming language keywords while garbage collections are started based on different metrics such as free heap size, free region size, allocations since the last garbage collection and others. Those three procedures have to be changed in order to create an automatic object pool.

The GenCon heap is structured into the new space and the old space. The new space is further equally divided into the nursery and the evacuation space.

Objects are allocated in the nursery. Each allocation changes the remember set, which stores the incoming references of the nursery region. Once the nursery is filled and allocations begin to fail, a partial garbage collection

using the copy-forward technique is initiated. Starting with the remember set, all live objects are traversed and evacuated onnce encountered. Once the traversal concludes, the designation of nursery and evacuation are swapped and the former nursery region is marked as free memory space.

Every time an object survives a collection, the object age counter attached to each object is increased. Once the counter reaches a high enough count, the object is not moved to the evacuation space, but to the old space.

The old space can be seen as a large memory region created to store long-living objects. The GC algorithm does not subdivide the region further in order to not reduce the usable space to 50% as is done with the young space. The old space is only collected during global collections, when partial GC gains are not sufficient enough to maintain fast execution.

A global collection does not utilize remember sets, but a stack frame, static fields and references stored in the registers as the root set for object traversal. All regions are collected and the remember set for the young space is adjusted to reflect references from the old space to the young space accordingly. The different heap structure of the old space requires the use of a separate collection policy, in this case mark-compact is utilized.

In order to create an object pool for the GenCon policy, an off heap structure storing pool information was created. The structure consists of a two-dimensional linked list. The dimensions are dictated by the classes to be stored and the number of objects each class can add to the pool. One such table stores all pooled object references, while a second one only stores object

pools that can be assigned to the mutator. The structure is initialized during the start up time of the environment.

The object pool is implemented as a growing structure. At start up time, all tables are empty and only store the class names of interest. The beginning of the mutator application marks a learning phase for the approach as the class names of each object allocated are examined. This includes using the class pointer of the allocation request to retrieve the class name string and to compare it to the list of pooled classes. Once a match is found, the class pointer is used for future comparisons instead of creating the overhead of string retrieval and comparison.

The previously described allocation flow of the virtual environment was adjusted to accommodate the object pools. Since pooled objects are stored in a common space not attached to specific threads, object assignment has to prevent race conditions and therefore utilize locking. This excludes fast path allocation from all object pool handling. The only way to change that would be to create a lock free pool that creates a pool for each thread separately or to improve the pool structure flow to a lock free operation that would prevent or deal with race conditions.

Each allocation performs a check if fast path allocation is possible. This check is adjusted to collect the following additional information:

- Is a pooled class being allocated?
- Does the pool have free objects?

- Is the maximum pool size reached?

The first check is performed on each object. This is also where class pointers of the pooled classes are collected as mentioned above. If a pooled class allocation is encountered, a free object in the pool is returned, if available. This is the most beneficial case as no new object has to be allocated. The object is removed from the free table, reinitialized using the class constructor and the mutator execution may continue. The reinitialization stage is added in order to maintain the ability to retroactively add the optimization to a mutator without source code modifications. A traditional pool object is not initialized multiple times by the pool.

The third check is necessary since the pool is a growing structure. Should no objects be available, the newly allocated object may still be able to grow the pool. In this case, a fast path allocation cannot be started and a redirect to a slow path allocation begins. Once allocated, the new object is added to the object pool's list as an unavailable object. Should the maximum pool size be reached and no free object be available, the allocation may pass the fast path check and can use the thread local heap, therefore reducing unnecessary overhead.

The slow path allocation flow was changed to include table handling. Once the object allocation is completed, it is added to the object pool, but not the free table, therefore remembering it for potential reuse opportunities.

The garbage collector of GenCon had to be partially adjusted to accommodate object pools. Since the largest benefit of the pool occurs when there

are free objects available in the pool, it was not deemed beneficial to include old region objects into the pool. Whenever an object ages enough to be promoted to the old region, the probability of it dying is low enough that it is excluded from the pool. The old region garbage collection did not have to be adjusted further.

Partial garbage collection handles all short lived objects. It is intended to be frequent, but very short. Pooled objects are created in the nursery and remembered using the object pool table. Whenever a partial collection occurs, three flow adjustments are triggered:

- Object exclusion
- Reference update
- Object rescue

Object exclusion happens when a pooled object is promoted to the old space. A promoted object is considered unlikely to become garbage in the near future and is therefore unlikely to create a free pool object. Before it is copied over to the old space, the reference to this object is removed from the pool, the linked list pointers are fixed and the pool size corrected. This allows the next allocation of this class to fill up the pool once again with a potentially short lived object.

Similar to object exclusion, reference updates have to be performed during object traversal. The class pointer of each object is examined. If the current

class is pooled, the reference of the object is located in the pool table. Should this object be part of the pool, the new location of the object is stored in the pool table once the evacuation is indicated as successful.

Object rescue is a new stage of the partial garbage collection introduced before the region assignments are swapped. Once all objects are evacuated, the pool table is traversed to examine all objects stored. Should an object be in the collection area, the object is forced to be evacuated even though its reference is not accessible to the mutator. The new location of the object is updated using the reference update flow. In addition, the memory of the new object is set to store only zeroes and the object itself is added to the pool free table. This stage is an additional garbage collection for pool objects only.

5.4.2.2 GenCon Pool Discussion

The object pool was examined using the best case scenario application described in Section 5.3. Each experiment was performed on the same machine as the preliminary results. To reduce the impact of noise and get a more reliable value, each variable set went through testing at least 25 times.

As shown in Figure 5.4, the proposed approach was outperformed by the baseline by an average of 11.75%. When looking at the relative change compared to the baseline (Figure 5.5), the approach shows that the performance loss increases when a longer execution is performed.

In order to investigate the reasons for this behaviour, multiple experiments

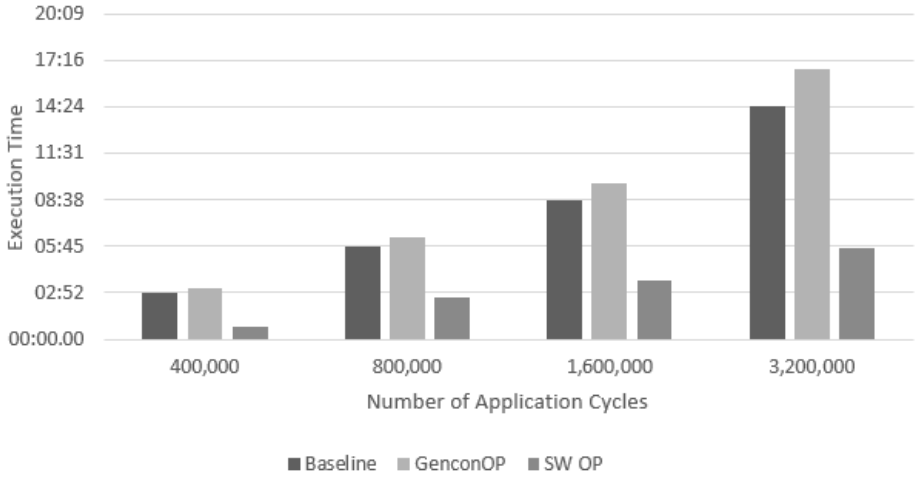

Figure 5.4: Performance of automatic object pools in GenCon (GenconOP) compared to object pools implemented inside the mutator (SW OP) and the baseline configuration without object pools. Object pools created by developers in the mutator outperform the baseline implementation as well as the VM internal object pools. The difference between the baseline setup and the VM internal object pools becomes more significant as time continues.

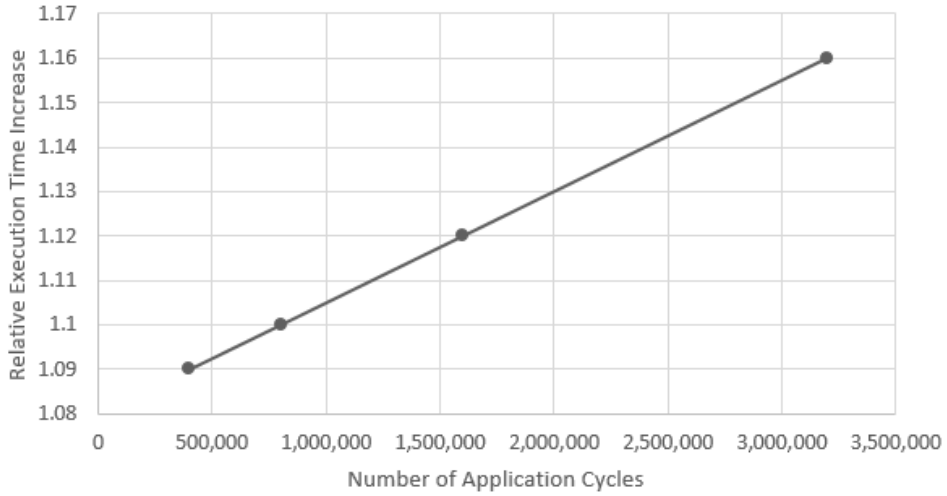

Figure 5.5: Relative execution time change when using GenCon object pools. The relative difference between the approaches is increasing as execution time progresses.

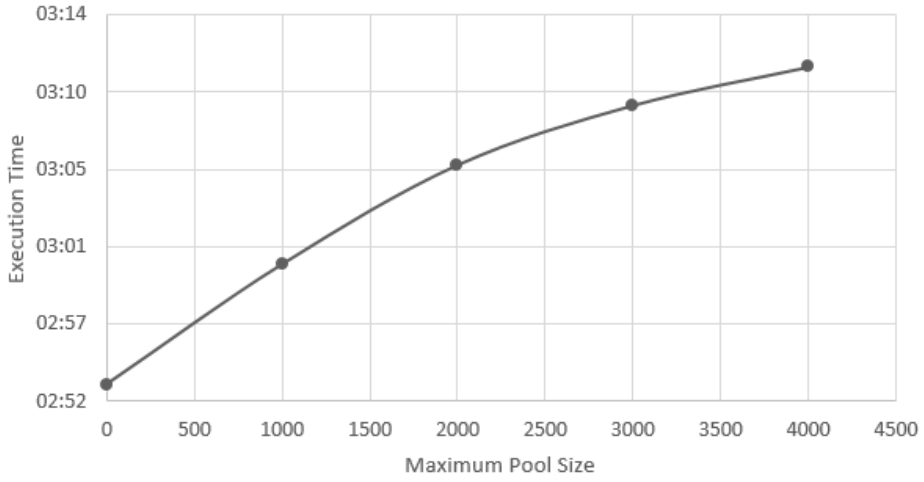

Figure 5.6: Execution time when varying the maximum pool size parameter.

were conducted. At first, the impact on fast path and slow path allocation was tested. When running the application in the baseline environment, about 83% of all allocations qualify for the fast path allocation track.

Introducing more objects which do not qualify for this flow and utilize locking based allocation is likely to slow down the mutator. An experiment with varying object pool sizes was conducted to quantify the change in this metric. Figure 5.6 shows the results of an experiment where the number of application cycles was left fixed at 1,600,000 cycles while varying the maximum number of objects in an object pool. It shows that the execution time increases as more objects have to pass through the lock protected allocation.

Since the application presents a possible best case scenario for a successful pool, almost all objects created are objects that are part of the object pool. Hence, until the pool fills up, all objects are going through the slow path

allocation. Once the pool is full, the object pool flow is skipped and therefore the metric is once again comparable to the baseline.

After each garbage collection, the first allocations of the pooled class do not qualify for the fast path once again, since they are assigned reused objects. This means no allocation is necessary.

Further, the garbage collection times of the application were compared. The distance between partial collections is reduced compared to the baseline and more collections are necessary. The increase in the number of overall GCs is the result of both the reduced distance between the collections and the increase in the overall execution time.

Experiments showed that the distance between the collections can be manipulated by changing the pool size variable. The larger the pool, the shorter the time between garbage collections. This seems to be the result of the rescue phase. As dead objects are copied over, they decrease the initial free space of the new nursery. This causes more copy processes, additional traversal of the old region and the earlier trigger for a new garbage collection.

In addition, during the time between the object not being referenced by the mutator any longer and the next GC, the object is considered floating garbage. It is not useful to the object pool as it is not clear that it can be reassigned. This creates a trade off when looking at the times between garbage collections. On one side, it is desirable to increase the time between collections in order to allow the mutator to have as much execution time as possible. On the other side, rapid garbage collections reclaim pooled objects

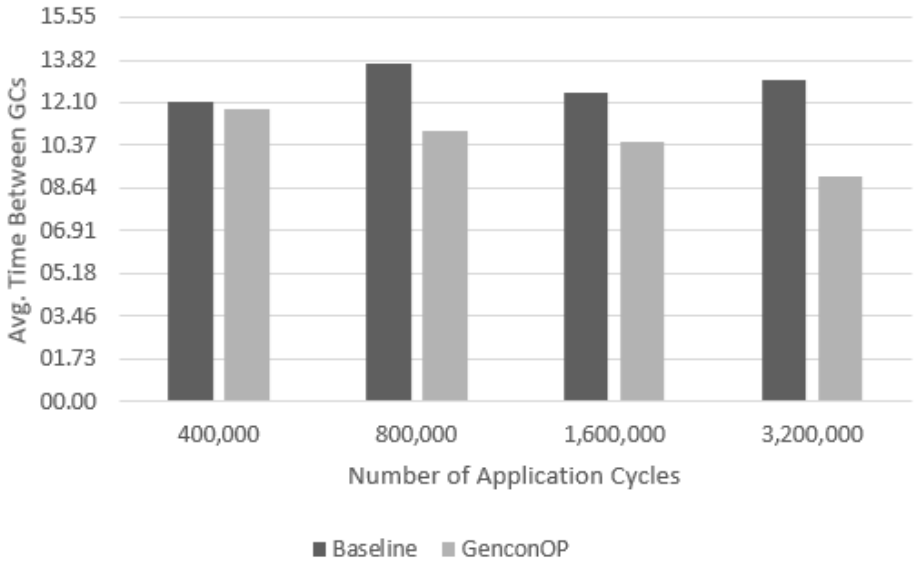

Figure 5.7: Average time between garbage collections when comparing the baseline to object pools.

to the pool and can therefore speed up the mutator's execution. A comparison of the average time between garbage collections for the configurations is presented in Figure 5.7.

5.4.3 Region Based GC Object Pools

The balanced garbage collection policy of the Java VM divides the heap into equal sized regions. The regions are managed separately. During the startup phase of an execution, a number of regions are selected to represent the nursery space, where new objects may be allocated.

Using a number of metrics such as fragmentation, potential reusable space and free space in a region, a collection set is created. Only those regions are

cleaned up by the garbage collector. A concurrent marker is responsible for traversing the regions during execution time and helps gather the metrics required.

A backup global collection policy is in place to clean up the full heap space if partial collections cannot satisfy allocation requirements. The balanced GC policy aims at reducing and potentially eliminating the number of global collections.

5.4.3.1 Region Pool Design

Two main factors causing the slow down of the object pool approach in Section 5.4.2 are: the inflated nursery space and the time between an object becoming free and the next garbage collection. Additional region control and structure of the balanced GC policy was the reason for porting the approach. Object pools are created and maintained in separate regions. The number of regions depends on the size of the pool. Initially, only one region is assigned for pooled objects. Similar to the pool structure in Section 5.4.2, the region is filled with objects as the mutator is allocating them. Once a region is filled, another region is added to the pool if possible. Given that regions use a copy-forward collection technique, free regions are generated at each partial collection stage.

To prevent garbage collections caused by too many instances of floating garbage created by the pool, pool regions were excluded from the process of deciding when to trigger a collection. A number of metrics are used to

decide when to trigger partial collections and which regions to collect. While object pool regions are never the leading cause for a collection, they are a part of each partial collection set.

The allocation process described in the previous section remains mostly the same with a few modifications. The allocation space is the current object pool nursery. Object allocations are still performed when requested. Once a region is filled, a new region is added to the pool and marked as the current object pool allocation space.

The only case in which the object pool is responsible for a garbage collection phase occurs when no free region is found while the pool has to be grown. In this case a partial GC is initiated, which usually produces free regions.

Changes to the garbage collection process had to be restructured. Instead of interrupting the partial collections, the balanced GC implementation introduces a separate stage used only for object pools. All object pool regions are processed by the garbage collector. The collection does not perform any copying operation, which removes the requirement of a target region.

The traversal technique used for regular regions is also applied to object pool regions. An object traversed is marked as alive instead of copying it to a target region. Once the region is processed, all objects in the current region are checked. Each unmarked object's memory is reset and the object is added to the free list of the object pool for future assignment.

Similar to the previous implementation, an allocation in the object pool never fails. Should the pool be full, a regular allocation is initiated. If the

pool is not successful in growing because no free region was produced by a garbage collection, a regular allocation is initiated as well. During the testing stage, this case never occurred using real life applications. A corner case test was created to force this behaviour for a proof of concept study where pool objects were allocated and never freed while the maximum number of pooled objects was set to be higher than the heap was capable of accommodating.

5.4.3.2 Region Pool Discussion

Experiments using this approach were performed following the same structure as those described in Section 5.4.2. At first the overall execution time was measured and compared to the baseline and the GenCon implementation. The approach proved to have a faster execution time than the GenCon implementation, while still being outperformed by the baseline.

The balanced GC implementation performed an average of 4% better than the GenCon object pool implementation. The difference between the two approaches increased with longer execution times. For the longest test in the experiment, the newly introduced approach performed 6% faster (Figure 5.8).

An experiment evaluating the time between garbage collections was performed. In order to measure the impact of object pools on the collection policy, the time between GCs was measured for an unchanged balanced collector and one using object pools. The difference between the numbers was insignificant. The variation in garbage collection numbers for both approaches was

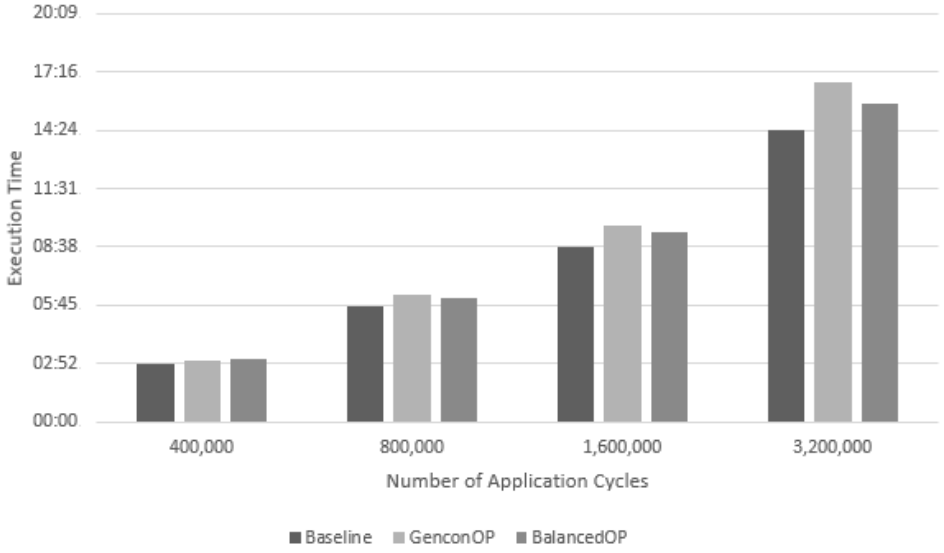

Figure 5.8: Execution time of balanced object pools in comparison.

within the same range, which leads one to believe that other collector metrics were more significant for the decision to perform a collection. This can be explained by the design of the pool structure.

Pool structures usually do not initiate garbage collections. Those happen when the pool is full or when too many other objects are filling up the heap. These can be objects that do not fit into the pool structure as well as pool object children, which are no longer used.

Overall, the iteration proved to be performing better and the initially anticipated benefits were achieved by moving the object to separate regions. On the flipside, as the time between collections was not influenced, reclaiming objects in a timely fashion remained problematic.

5.4.3.3 Pseudo Reference Counting Design

Addressing the issue of reducing the time between an object dying and being added to the free list of the pool structure required an additional mechanism to the region based object pool approach. Adding additional periodical garbage collections only for the pooled regions was found to be too costly as not enough objects were reclaimed in the process. Most remember set changes are done when non-pooled regions were traversed. This approach was therefore too costly. Thus, another way to detect such objects between garbage collections had to be found.

The only widespread garbage collection technique that is capable of detecting unreferenced objects as they occur is reference counting [32]. The number of incoming references to an object is stored and maintained and as soon as this count reaches 0, the object is considered garbage and can be reclaimed. There are a number of challenges using this approach, especially in an environment relying on garbage collection phases. Objects usually do not have the infrastructure to hold such a count, reference operations are not built for this kind of operation and the fact that reference counting fails in cyclic structures. Two objects referencing each other are enough to maintain a live state. In order to still evaluate potential benefits of such a structure using the implementation available, pseudo reference counting for pooled objects was created in the IBM Java J9 VM.

The allocation and garbage collection handling remained unchanged. The garbage collection in pooled regions was kept as a back-up system in case

of cyclic structures, however the flow was only initiated during global collections. The pool object structure was extended to hold a lock-protected reference count for each pooled object. The initial state of the count was defined to be -1 in order to recognize a newly created object.

In order to modify and maintain the count, the write barrier of objects had to be adjusted. This usually comes at a very high cost, as each object modification has to pass this barrier. Adding slight complexity to this flow affects such a high number of operations that it is usually not beneficial.

The effects were minimized by using a number of special cases in the testing environment. Since the classes of pooled objects are defined prior to execution, the class pointer can be compared to determine whether or not a pool object is affected.

In addition, the address space of pooled regions is remembered to determine when a new or old reference is leading into this space. If that is the case, the operation is processed for reference counting purposes. If an object creates a reference to a pooled object, the object's counter increases (-1 increases to 1). When an old reference was pointing to a pooled object, the reference counter is decreased. Once the counter reaches 0, the object is returned to the free list and can be reassigned to the mutator. The initial -1 symbolizes a pooled object, which is not part of the free table. As the counter is not set upon creation, but rather when it passes the write barrier, operations such as removing the object from the free table are not necessary.

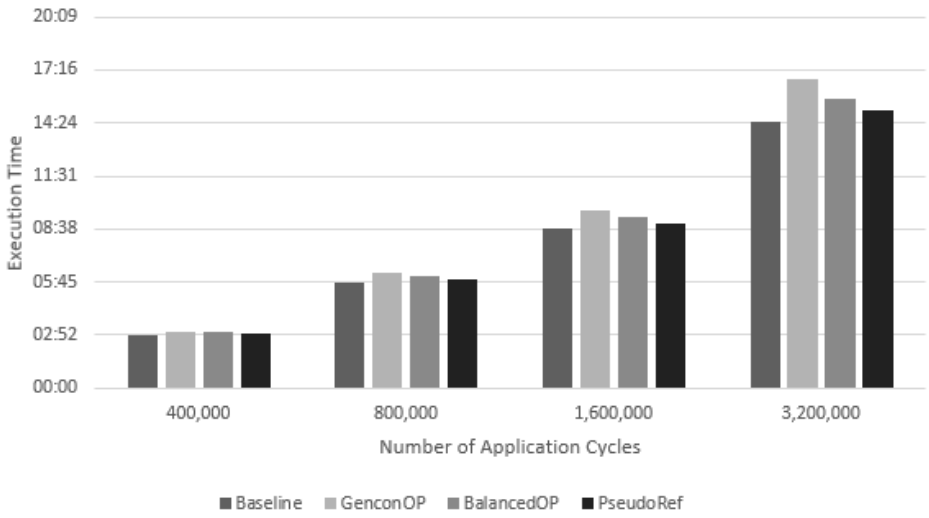

Figure 5.9: Execution time of balanced object pools in comparison.

5.4.3.4 Pseudo Reference Counting Discussion

The execution time comparison of all three iterations of the approach are shown in Figure 5.9. The third iteration of the approach performed best compared to the other object pool implementations. While modifications to the write barrier were made, the overall benefit of the approach compared to GenCon object pools averaged at 6%. Comparing this implementation to the baseline however still shows a performance degradation of about 4%.

5.5 Conclusions and Future Work

This chapter presented a feasibility study of automatic object pools in virtual environments. Three implementations were presented, tested and discussed in order to identify their benefits and challenges while attempting to address

them. Two main problems of the initial approach led to the design and evaluation of the subsequent iterations.

While not outperforming the baseline, the results of the last iteration still show promise. The main issues of the initial implementation were mitigated by adjusting a better suited collection policy and introducing a reference counting mechanism. The final iteration shows that there is potential in the approach, especially for virtual environments that are built around a reference counting collection policy.

In addition, it was shown that an automatic pool implementation is possible and can be used for specific cases. Examples would be cases where developers aim at isolating objects of a specific class from others. This can be the case for a class that frequently produces cold objects, classes that should be stored in a specific location for performance purposes, etc.

Future work in the field can examine the impact of creating object pools on object locality since objects might not be allocated closely to their hierarchical neighbours [16]. An implementation of the structures in a reference counting environment would be a good comparison to the work presented here. Hot and cold object structures and how they qualify for pooling is another field worth investigating [19].

Moving the approach described to an environment using reference counting would mean that the initial structure of the pools described in this chapter can be reused. The allocation and garbage collection modifications would have to be adjusted and recreated. Allocations of predefined classes and

their reference count reaching zero would have to be targeted in order to perform the maintenance required by the structure.

Chapter 6

Conclusions

The dissertation presented research on the heap structure of virtual environments. Specifically, the concepts of String deduplication and Object pooling were explored as techniques to reduce memory footprint and allocations/deallocations respectively. While these techniques are generally applicable to all application virtual machines, the proposed improvements were explored and validated through the use of the IBM Java J9 virtual machine.

String deduplication aimed at reducing the number of memory objects occupied by finding and eliminating duplicate structures. While the initial implementation proved to be creating a benefit for specific applications, others were slowed down by a high computational overhead caused by hash table maintenance and object indexing. The impact of the approach was examined more closely and optimizations were added in order to perform the process faster and to decide whether or not this optimization is beneficial for the

currently running application.

While the string deduplication approach may provide a benefit for a subset of applications, it does apply a significant overhead to others. The overhead is mitigated by the introduction of a threshold used to determine during global collections phases whether or not deduplicating strings is beneficial. Not only was the negative execution time overhead reduced for nearly all applications, the approach also gained the ability to react to phases of an application where duplicate strings are present. Once this phase of the application has passed, the optimization is disabled once again. Overall, the approach showed promising results for string heavy applications and can be used for applications with memory size constraints.

In addition to deduplication aiming specifically on string objects, VM internal object pools were presented as a feasibility study on object classes in general. While an implementation was successfully added to the IBM Java VM, the approach was found not to be providing the anticipated performance boost indicated by a preliminary case study. After examining the limitations and impact of an implementation in the GenCon garbage collector of the IBM Java J9 VM, the drawbacks of the approach were mitigated by examining different techniques.

The main drawbacks were found to be the reduced time between garbage collections, which causes more stop-the-world phases to be scheduled by the environment, and the time an object spends as undetected garbage. The additional garbage collections are caused by the addition of object pool ob-

jects to a constant pool of objects that have to be evacuated by the garbage collection.

Addressing the drawbacks of the object pool algorithm included the migration of the implementation to a more suited garbage collection policy and by assisting the garbage collector of the need to detect floating garbage. In addition, a pseudo reference counting algorithm was added to the approach. The reference counting algorithm aimed at reducing the time an object spends in the floating garbage state to provide the instance of the object back to the mutator.

While the migration of the approach to a region-based garbage collection policy did not outperform the baseline implementation in all cases, it did reduce the overhead of the implementation. Additionally, the pseudo reference counting indicated possible future research fields, which can be explored further.

Chapter 7

Future Work

While the work completed investigated many aspects of string deduplication and object pooling to enhance memory management, there are many areas in which the work can be further explored. One key area is in the applicability of the findings to other application virtual machines. The research presented was conducted and validated in the context of the very popular Java virtual machine. Specifically, the IBM implementation J9 was modified to provide the optimizations. Given the rigid specification of the JVM, the author is confident that the findings presented are applicable to other Java VM implementations. However, there are other application virtual machines that should be explored for applicability and validation of the techniques presented.

The string deduplication approach only considered full string contents and did not look into deduplication for parts of those strings, which could be

investigated in the future. In addition, the impact of string deduplication can be evaluated in a *non-uniform memory access* (NUMA) environment, where deduplicating structures might be more costly. NUMA environments might benefit from purposefully duplicating frequently used strings, which would be placed on each node for quicker access. Further, the caching and object locality impact based on the caching strategy and architecture can be examined in order to find potential optimizations to the process. This can be combined with findings of Patrou et al. [53] to optimize the allocation and garbage collection structure in virtual environments.

Related to the NUMA approach, an investigation about the locality impact of deduplicating strings can be considered. As deduplicating a string points two object headers to the same character array, the distance between one of the headers and the array may be larger than in an unchanged VM. This can have a disadvantageous effect, as the probability of the header and the string contents to be in the same cache line are reduced. This can have an effect on the performance of an application as shown by Eimouri et al. [18]. Further, data about the dynamic deduplication threshold chosen for certain combinations of classes and applications can be collected and saved between runtimes. One possibility would be to use the shared class cache, which saves data between runtimes for performance purposes [3]. Once this data is collected, the application could explore different values for applications to determine the best trade off between deduplication benefit and maintenance cost. While this procedure might have some significantly slower executions,

it can also speed up a frequently started application.

Future work in the field of VM internal object pools might include the implementation of the presented algorithm in a reference counted environment to see if the hypothesis holds. Further, changing the object pool maintenance to an algorithm less reliant on locking could create a potential speed up for multithreaded applications. Approaches used could utilize a data race detector as proposed by Hundon et al. [59], introduce lock free data structure manipulation as invented by Moir et al. [46], or use hazard pointers associated with each thread as invented by Michael [45].

In addition, object pools can be established as a shared resource between multiple virtual environments as suggested by Richard et al. [57]. A pool could be established in a separate space and referenced by the environments as needed. Challenges for such an approach include a decision about which entity is responsible for the maintenance of the structure and how to ensure environment isolation.

Additionally, shared structures might cause a security vulnerability, which could also create a research opportunity. The biggest challenge is once again to minimize the time between the death of an object to its detection.

Bibliography

[1] Andrew W Appel, *Simple generational garbage collection and fast allocation*, Software: Practice and Experience **19** (1989), no. 2, 171–183.

[2] U Berkeley, *Berkeley logic interchange format*, tech. rep., Technical report, Technical report, University of California at Berkeley, Tech. Rep. (1998).

[3] Dev Bhattacharya, Kenneth B Kent, Eric Aubanel, Daniel Heidinga, Peter Shipton, and Aleksandar Micic, *Improving the performance of JVM startup using the shared class cache*, Communications, Computers and Signal Processing (PACRIM), 2017 IEEE Pacific Rim Conference on, IEEE, 2017, pp. 1–6.

[4] Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley, *Myths and realities: The performance impact of garbage collection*, ACM SIGMETRICS Performance Evaluation Review **32** (2004), no. 1, 25–36.

[5] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg,

Daniel Frampton, Samuel Z Guyer, et al., *The DaCapo benchmarks: Java benchmarking development and analysis*, ACM Sigplan Notices, vol. 41, ACM, 2006, pp. 169–190.

[6] Hans-J Boehm, *The space cost of lazy reference counting*, ACM SIGPLAN Notices **39** (2004), no. 1, 210–219.

[7] Chris J Cheney, *A nonrecursive list compacting algorithm*, Communications of the ACM **13** (1970), no. 11, 677–678.

[8] Trishul M Chilimbi, Mark D Hill, and James R Larus, *Cache-conscious structure layout*, ACM SIGPLAN Notices, vol. 34, ACM, 1999, pp. 1–12.

[9] George E Collins, *A method for overlapping and erasure of lists*, Communications of the ACM **3** (1960), no. 12, 655–657.

[10] Standard Performance Evaluation Corporation, *SPEC JBB2005*, `https://www.spec.org/jbb2005/` [Online. Last accessed: February 28, 2019], 2018.

[11] ———, *SPECjbb2013*, `https://www.spec.org/jbb2013/` [Online. Last accessed: February 28, 2019], 2018.

[12] Sylvia Dieckmann and Urs Hölzle, *A study of the allocation behavior of the SPECjvm98 Java benchmarks*, European Conference on Object-Oriented Programming, Springer, 1999, pp. 92–115.

[13] ———, *The allocation behavior of the SPECjvm98 Java benchmarks*, Performance evaluation and benchmarking with realistic applications, MIT Press, 2001, pp. 77–108.

[14] Damien Doligez and Georges Gonthier, *Portable, unobtrusive garbage collection for multiprocessor systems*, Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1994, pp. 70–83.

[15] Damien Doligez and Xavier Leroy, *A concurrent, generational garbage collector for a multithreaded implementation of ml*, Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1993, pp. 113–123.

[16] Marcel Dombrowski, Konstantin Nasartschuk, Kenneth B Kent, and Gerhard W Dueck, *A survey on object cache locality in automated memory management systems*, 2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE), IEEE, 2015, pp. 349–354.

[17] Daniel Edelson and I Pohl, *Smart pointers: They're smart, but they're not pointers*, Citeseer, 1992.

[18] Taees Eimouri, Kenneth B Kent, and Aleksandar Micic, *Effects of false sharing and locality on object layout optimization for multi-threaded ap-*

plications, Electrical and Computer Engineering (CCECE), 2016 IEEE Canadian Conference on, IEEE, 2016, pp. 1–5.

[19] Taees Eimouri, Kenneth B Kent, Aleksandar Micic, and Karl Taylor, *Using field access frequency to optimize layout of objects in the JVM*, Proceedings of the 31st Annual ACM Symposium on Applied Computing, ACM, 2016, pp. 1815–1818.

[20] Robert R Fenichel and Jerome C Yochelson, *A LISP garbage-collector for virtual-memory computer systems*, Communications of the ACM **12** (1969), no. 11, 611–612.

[21] Eclipse Foundation, *Openj9*, `https://github.com/GarCoSim/` [Online. Last accessed: February 28, 2019], 2018.

[22] The Apache Software Foundation, *DBCP - Overview*, `http://commons.apache.org/proper/commons-dbcp/` [Online. Last accessed: February 28, 2019], 2016.

[23] James Gosling, *The Java language specification*, Addison-Wesley Professional, 2000.

[24] Mark Grand, *Patterns in Java: a catalog of reusable design patterns illustrated with UML*, John Wiley & Sons, 2003.

[25] Graham Hamilton, Rick Cattell, Maydene Fisher, et al., *JDBC database access with Java*, vol. 7, Addison Wesley, 1997.

[26] Michihiro Horie, Kazunori Ogata, Kiyokuni Kawachiya, and Tamiya Onodera, *String deduplication for Java-based middleware in virtualized environments*, Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, ACM, 2014, pp. 177–188.

[27] R John M Hughes, *A semi-incremental garbage collection algorithm*, Software: Practice and Experience **12** (1982), no. 11, 1081–1082.

[28] IBM, *Components of the IBM Virtual Machine for Java*, `https://www.ibm.com/support/knowledgecenter/SSYKE2_7.0.0/com.ibm.java.win.70.doc/diag/understanding/jvm_components.html` [Online. Last accessed: February 28, 2019], 2018.

[29] ———, *Components of the IBM Virtual Machine for Java*, `http://www.eclipse.org/openj9` [Online. Last accessed: February 28, 2019], 2018.

[30] ———, *JVM -X options: -Xgcpolicy*, `https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/openj9/xgcpolicy/index.html` [Online. Last accessed: February 28, 2019], 2018.

[31] Peter Jamieson, Kenneth B Kent, Farnaz Gharibian, and Lesley Shannon, *Odin II-an open-source Verilog HDL synthesis tool for CAD research*, 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, IEEE, 2010, pp. 149–156.

[32] Richard Jones, Antony Hosking, and Eliot Moss, *The garbage collection handbook: the art of automatic memory management*, CRC Press, 2016.

[33] Richard Jones and Rafael D Lins, *Garbage collection: Algorithms for automatic dynamic memory management*, (1996).

[34] Kiyokuni Kawachiya, Kazunori Ogata, and Tamiya Onodera, *Analysis and reduction of memory inefficiencies in Java strings*, ACM Sigplan Notices **43** (2008), no. 10, 385–402.

[35] M Kircher and P Jain, *Pooling pattern*, Proceedings of EuroPlop 2002 (2002).

[36] Reinhard Klemm, *Practical guidelines for boosting Java server performance*, Proceedings of the ACM 1999 conference on Java Grande, ACM, 1999, pp. 25–34.

[37] Leslie Lamport, *Garbage collection with multiple processes: an exercise in parallelism*, Proc. of the 1976 International Conference on Parallel Processing, 1976, pp. 50–54.

[38] Per-Åke Larson, *Dynamic hashing*, BIT Numerical Mathematics **18** (1978), no. 2, 184–201.

[39] Henry Lieberman and Carl Hewitt, *A real-time garbage collector based on the lifetimes of objects*, Communications of the ACM **26** (1983), no. 6, 419–429.

[40] Yibei Ling, Tracy Mullen, and Xiaola Lin, *Analysis of optimal thread pool size*, ACM SIGOPS Operating Systems Review **34** (2000), no. 2, 42–55.

[41] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, et al., *VTR 7.0: Next generation architecture and CAD system for FPGAs*, ACM Transactions on Reconfigurable Technology and Systems (TRETS) **7** (2014), no. 2, 6.

[42] Darko Marinov and Robert O'Callahan, *Object equality profiling*, ACM SIGPLAN Notices, vol. 38, ACM, 2003, pp. 313–325.

[43] Simon Marlow, Tim Harris, Roshan P James, and Simon Peyton Jones, *Parallel generational-copying garbage collection with a block-structured heap*, Proceedings of the 7th international symposium on Memory management, ACM, 2008, pp. 11–20.

[44] John McCarthy, *Recursive functions of symbolic expressions and their computation by machine, Part I*, Communications of the ACM **3** (1960), no. 4, 184–195.

[45] Maged Michael, *Method for efficient implementation of dynamic lock-free data structures with safe memory reclamation*, June 3 2004, US Patent App. 10/308,449.

[46] Mark S Moir, Victor Luchangco, and Maurice Herlihy, *Lock-free implementation of dynamic-sized shared data structure*, August 7 2007, US Patent 7,254,597.

[47] Konstantin Nasartschuk, Marcel Dombrowski, Tristan Basa, Mazder Rahman, Kenneth Kent, and Gerhard Dueck, *GarCoSim: A framework for automated memory management research and evaluation*, Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016, pp. 263–268.

[48] Konstantin Nasartschuk, Marcel Dombrowski, Kenneth B Kent, Aleksandar Micic, Dane Henshall, and Charlie Gracie, *String deduplication during garbage collection in virtual machines*, Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering, IBM Corp., 2016, pp. 250–256.

[49] Konstantin Nasartschuk, Kenneth B Kent, Stephen A MacKay, and Aleksandar Micic, *Feasibility of internal object pools to reduce memory management activity*, Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering, IBM Corp., 2018, pp. 136–144.

[50] Konstantin Nasartschuk, Kenneth B Kent, Stephen A MacKay, Aleksandar Micic, and Charlie Gracie, *Improving garbage collection-time*

string deduplication, Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering, IBM Corp., 2017, pp. 113–119.

[51] Nicolas Neu, Kenneth B Kent, Charlie Gracie, and Andre Hinkenjann, *Automatic application performance improvements through vm parameter modification after runtime behavior analysis*, Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools, ICST (Institute for Computer Sciences, Social-Informatics and , 2014, pp. 147–152.

[52] Clifton Malcolm Nock, *Patent US 6832228 B1*, `http://www.google.com/patents/US6832228` [Online. Last accessed: February 28, 2019], 2018.

[53] Maria Patrou, Kennth B. Kent, Gerhard W. Dueck, Charlie Gracie, and Aleksandar Micic, *NUMA awareness: Improving thread and memory management*, 2018 44th Euromicro Conference on Software Engineering and Advanced Applications, Proceedings of (2018), 119–123.

[54] DaCapo Project, *DaCapo Benchmarks*, `http://www.dacapobench.org/` [Online. Last accessed: February 28, 2019], 2018.

[55] Irfan Pyarali, Marina Spivak, Ron Cytron, and Douglas C Schmidt, *Evaluating and optimizing thread pool strategies for real-time CORBA*, ACM SIGPLAN Notices **36** (2001), no. 8, 214–222.

[56] Zhu Quan-Yin, Jin Yin, Xu Chengjie, and Gen Rui, *A UML model for mobile game on the Android OS*, Procedia Engineering **24** (2011), 313–318.

[57] Adam Richard, Lai Nguyen, Peter Shipton, Kenneth B Kent, Azden Bierbrauer, Konstantin Nasartschuk, and Marcel Dombrowski, *Inter-JVM sharing*, Software: Practice and Experience 2016; 46:(09) (2016), 1285–1296.

[58] Robert A Saunders, *The LISP system for the Q-32 computer*, The Programming Language LISP: Its Operation and Applications (1974), 220–231.

[59] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson, *Eraser: A dynamic data race detector for multi-threaded programs*, ACM Transactions on Computer Systems (TOCS) **15** (1997), no. 4, 391–411.

[60] Ronald Servant, *How did the J9 in OpenJ9 get its name?*, `https://medium.com/@rservant/how-did-the-j9-in-openj9-get-its-name-95a6416b4cb9` [Online. Last accessed: February 28, 2019], 2017.

[61] Joseph Weizenbaum, *Recovery of reentrant list structures in SLIP*, Communications of the ACM **12** (1969), no. 7, 370–372.

[62] Paul R Wilson and Thomas G Moher, *Design of the opportunistic garbage collector*, ACM SIGPLAN Notices, vol. 24, ACM, 1989, pp. 23–35.

Vita

Candidate's full name:	Konstantin Nasartschuk
University attended:	Master of Computer Science 2013 University of New Brunswick Fredericton, New Brunswick, Canada
	Master of Science in Autonomous Systems 2013 Bonn-Rhein-Sieg University of Applied Sciences Sankt Augustin, Nordrhein-Westfalen, Germany
	B.Sc Computer Science 2010 Bonn-Rhein-Sieg University of Applied Sciences Sankt Augustin, Nordrhein-Westfalen, Germany

Publications:

K. Nasartschuk, K. B. Kent, S. A. MacKay, A. Micic, *Feasibility of Internal Object Pools to Reduce Memory Management Activity*, Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering, IBM Corp., 2018,

K. Nasartschuk, K. B. Kent, S. A. MacKay, A. Micic, C. Gracie, *Improving Garbage Collection-Time String Deduplication*, Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering, IBM Corp., 2017, pp. 113 – 119

K. Nasartschuk, K. B. Kent, S. A. MacKay, A. Micic, C. Gracie, *String deduplication During Garbage Collection in Virtual Machines*, Proceedings of the

26th Annual International Conference on Computer Science and Software Engineering, IBM Corp., 2016, pp. 250 – 256

M. Dombrowski, K. Nasartschuk, C. Gracie, G. Dueck and K. B. Kent, *Thread Group Based Local Heap Garbage Collection in a Simulated Runtime Environment*, IEEE Canadian Conference on Electrical and Computer Engineering, Vancouver, Canada, (6 pages), May 15-18, 2016.

M. Rahman, K. Nasartschuk, G. Dueck and K. B. Kent, *Trace Files for Automatic Memory Management Systems*, International Workshop on Validating Software Tests, Osaka, Japan, pp. 9-12, March 15, 2016.

Richard, A., Nguyen, L., Shipton, P., Kent, K.B., Bierbrauer, A., Nasartschuk, K. and Dombrowski, M., *InterJVM Sharing*. Software: Practice and Experience, 46(9), pp.1285-1296, 2016

K. Nasartschuk, M. Dombrowski, T. M. Basa, M. M. Rahman, G. W. Dueck and K. B. Kent, *GarCoSim: A Framework for Automated Memory Management Research and Evaluation*, 9th International Conference on Performance Evaluation Methodologies and Tools, Berlin, Germany, pp. 263-268, December 14-16, 2015.

B. Narayanan, L. Cambuim, K. Nasartschuk, P. Ploeger and K. B. Kent, *Improved Language Support for Verilog Elaboration in Odin II and FPGA Architecture Benchmarking in the VTR Project*, IEEE Pacific Rim Conference on Communications Computers and Signal Processing, Victoria, Canada, pp. 309-314, August 24-26, 2015.

M. Dombrowski, K. Nasartschuk, G. Dueck and K. B. Kent, *A Survey on Object Cache Locality in Automated Memory Management Systems*, IEEE Canadian Conference on Electrical and Computer Engineering, Halifax, Canada, pp. 349 - 354, May 3-6, 2015.

J. Luu, J. Goeders, T. Yu, T. Liu, M. Wainberg, A. Somerville, K. Nasartschuk, S. Wang, M. Nasr, N. Ahmed, K. B. Kent, J. Anderson, J. Rose and V. Betz, *VTR 7.0: Next Generation Architecture and CAD System for FPGAs*, ACM Transactions on Reconfigurable Technology and Systems, vol 7, issue 2, article 6, pp. 1-30, June 2014.

J. Li, K. Nasartschuk, and K. B. Kent, *System-on-Chip Processor using Different FPGA Architectures in the VTR CAD Flow*, 2014 IEEE International Symposium on Rapid Systems Prototyping, New Delhi, India, pp. 72 - 77, October 16-17, 2014.

K. Nasartschuk, R. Herpers, and K. B. Kent, *Visual Exploration of Changing FPGA Architectures in the VTR Project*, 2013 IEEE International Symposium on Rapid Systems Prototyping, Montreal, Canada, pp. 16 - 22, October 3-4, 2013.

Konstantin Nasartschuk, Kenneth B. Kent, and Rainer Herpers, *Visual Exploration of Simulated FPGA Architecures in Odin II*, Tech. report, University of Applied Science Bonn-Rhine-Sieg, University of New Brunswick, TR 12-220, 2012.

Konstantin Nasartschuk, Kenneth B. Kent, and Rainer Herpers, *Visualization Support for FPGA Architecture Exploration*, Rapid System Prototyping (RSP), 2012 23rd International Conference on, 2012, pp. 128-136.

K. Nasartschuk, R. Herpers and K. B. Kent, *Visual Exploration of Changing FPGA Architectures in the VTR Project*, Euromicro Conference on Digital System Design (DSD) Work-In-Progress Session 2012 (*Extended Abstract*), Izmir, Turkey, 2 pages, September 5 - 8, 2012.

Konstantin Nasartschuk, Kenneth B. Kent, and Rainer Herpers, *Visualization Support for FPGA Architecture Exploration*, Tech. report, University of Applied Science Bonn-Rhine-Sieg, University of New Brunswick TR 11-213, 2011.

Marcel Dombrowski, Konstantin Nasartschuk, *Evaluation of an Object Recognition Algorithm On the Volksbot Platform*, Fraunhofer ePub, 2010.