String Deduplication During Garbage Collection in Virtual Machines

Konstantin Nasartschuk
University of New Brunswick
Fredericton, Canada
kons.na@unb.ca

Marcel Dombrowski
University of New Brunswick
Fredericton, Canada
marcel.dombrowski@unb.ca

Kenneth B. Kent
University of New Brunswick
Fredericton, Canada
ken@unb.ca

Aleksandar Micic
IBM
Ottawa, Canada
aleksandar_micic@ca.ibm.com

Dane Henshall
IBM
Ottawa, Canada
henshall@ca.ibm.com

Charlie Gracie
IBM
Ottawa, Canada
charlie_gracie@ca.ibm.com

ABSTRACT

Memory management is a significant topic in virtual machine research. As allocation and deallocation of objects is performed automatically, garbage collection (GC) has become an important field of research. It aims to speed up and optimize the execution of applications developed in languages such as Java, C#, Python and others. Even though GC techniques have become more sophisticated, automatic memory management is not optimal. Garbage collection techniques, such as reference counting, mark-sweep, mark-compact, copying collection and generational garbage collection build the base of most automated memory management environments. Most GC policies include a stop-the-world phase that is used to detect live objects.The research presented in this paper aims to improve the automatic memory management and application execution by investigating an optimization of the memory layout. The goal of the approach described is to utilize the stop-the-world phase of the garbage collector in order to detect duplicate strings and to deduplicate them before copying them to a different region. The goal of this algorithm is to reduce memory duplication, 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.

Keywords

virtual machines, memory management, garbage collection, string deduplication

1. INTRODUCTION

In the last few decades, memory management of applications went through a number of iterations in order to simplify and automate the process. Traditional languages such as C or C++ rely on developers to allocate and free objects on the heap as they are created or become unused.

According to Jones et al. [8], 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 in order to allow developers to focus their resources on the core application development.

Reference counting [3], mark-sweep [12], mark-compact [13] and copying garbage collection [2, 6] are commonly used for automatic memory management. The known and common languages that rely on garbage collection include, but are not limited to, Java, C#, Python, and Ruby.

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 managed program is halted completely in order to clean up memory. This can be critical for the application's run time. In addition, the heap size used by the application, on average, is almost never as small as it would be when managed directly by developers. Attempts to close the gap between automatic and manual memory management techniques aim to improve metrics such as garbage collection count, stop-the-world time, as well as to decrease the execution time of the mutator by improving caching, locking, concurrency and others.

The paper is divided into four parts. Section 2 introduces the background of memory management techniques, heap layout in Java VMs and the field of string deduplication. Section 3 discusses the motivation of the project. Section 4 explains the design and implementation before experiments are conducted and discussed in Section 5.

2. BACKGROUND

The structure created by memory on the heap is application specific. However some frequently found features can still be identified. One of those metrics is the amount of memory occupied by string objects. According to Kawachiya et al. [9], on average, 20% of the heap is used by strings. In string-heavy applications such as a J2EE application server, the metric can reach as high as 40%. This high number suggests that improvements made to string objects promise to achieve a high positive impact on a large number of applications.

String objects in Java are immutable; they have the specific property that they can never change their content. If a

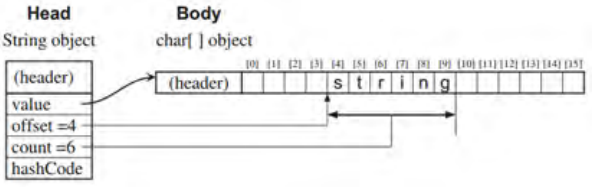

Figure 1: String header structure with the fields necessary [9].

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. This 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. All three declarations are valid, but are being processed in different ways. String `a` is created using the string table. The string table acts as a buffer containing all strings created in this way. 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`), 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. String `c` represents a special case. The function 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 1 presents the string header structure. The `substring()` function 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()` procedure 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 variables. 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 build the majority of the heap, Kawachiya et al. suggest collecting strings first in order to check if this would free enough space to continue execution [9]. 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. [7] 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 the execution flow or require significant computation, but are not executed over and over again, do not benefit using such an approach. 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 [11] 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 duplicated. The limitations this approach presents in regard to offline profiling are the same as with Horie et al. [7]. They can be used as guidelines and profiling tools for developers, but cannot optimize the memory structure dynamically.

3. 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.

An initial analysis of live objects on the heap while running SPECjbb2005 [4], SPECjbb2013 [5], and DaCapo [1] benchmarks revealed that some applications have many duplicate strings even when considering the deduplication performed by the string table and `substring()`. Figures 2, 3, and 4 show an extract of a complete analysis of all live strings after each garbage collection for the DaCapo benchmarks `jython`, `lusearch`, and `h2`. The graphs show the percentage of duplicate strings encountered as well as the possible heap space, which could be saved if all possible strings were deduplicated. The capturing process is performed before and after each garbage collection.

Note that the results in graphs for the benchmarks are quite different. Even though all three of the benchmarks show a large percentage of duplicate strings, the heap impact of the strings varies by a large margin. While most duplicate strings are collected in `jython`, the heap impact for `lusearch` and `h2` after garbage collections 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 2, 3 and 4 merely show what could be achieved in the optimal case for those specific applications. A preliminary test was

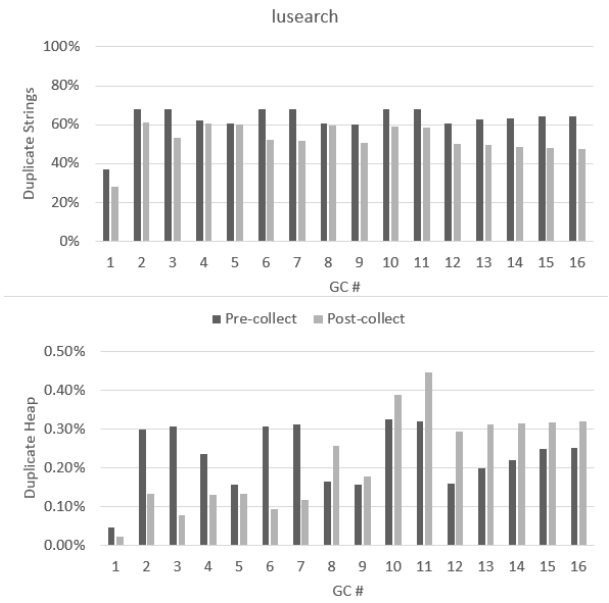

Figure 2: Lusearch: Number of duplicate string objects which do not share a character array (top) and the possible heap savings based on the string lengths and heap size (bottom).

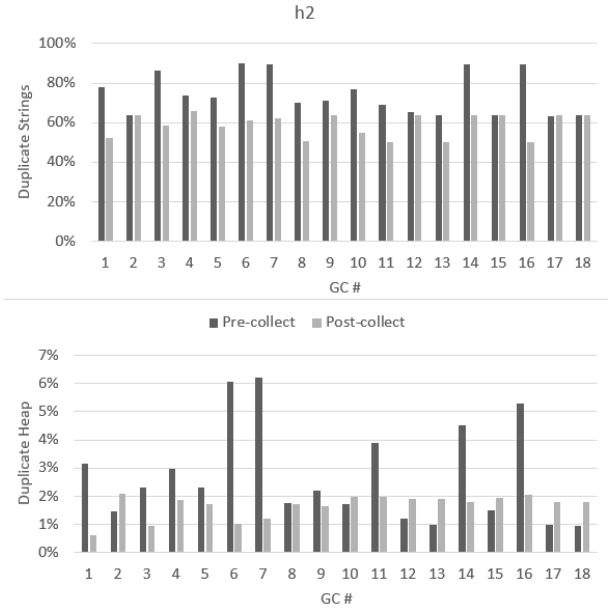

Figure 3: H2: Number of duplicate string objects which do not share a character array (top) and the possible heap savings based on the string lengths and heap size (bottom).

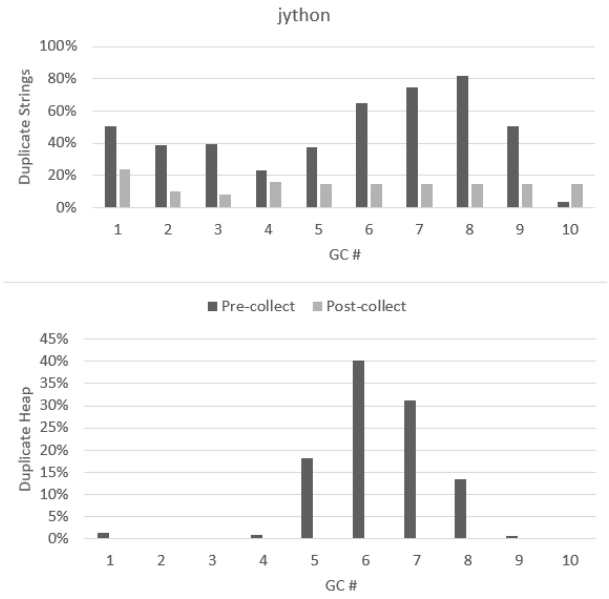

Figure 4: Jython: Number of duplicate string objects which do not share a character array (top) and the possible heap savings based on the string lengths and heap size (bottom).

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. The DaCapo benchmark suite [1] offers a suite of deterministic applications from different fields. Even though it has not been maintained for a number of years, it still contains representative real life application examples that can be used in order to evaluate the approach presented.

4. 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-sweep 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 the 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 5.

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.

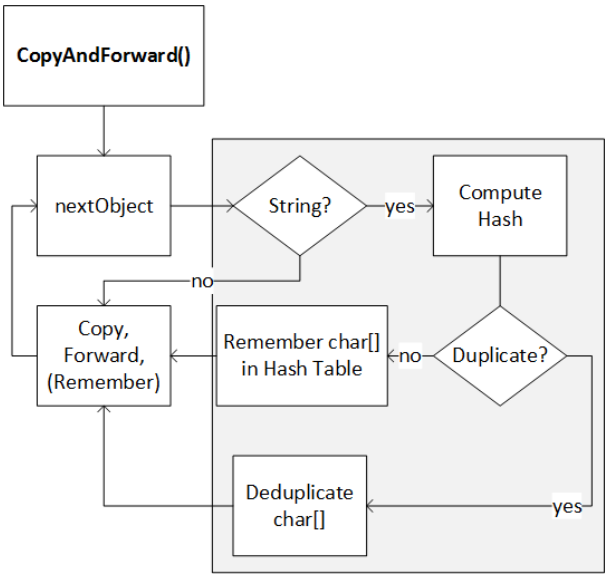

Figure 5: The flow of the string deduplication process in the garbage collector. Proposed changes to the flow are highlighted by a grey background.

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 will remain static and is subject to experiments. Collision is dealt with by adding colliding strings into the same slot. 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 [10].

5. EXPERIMENTAL RESULTS

The described approach was evaluated using a set of DaCapo benchmarks and a 2 GB 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. Each of the benchmarks was executed in three configurations:

(A) Unchanged VM

Benchmark	A	B	C
avrora	137.723	136.253	137.600
batik	97.773	98.186	98.881
fop	48.035	48.945	47.500
h2	2239.564	2290.825	2235.637
jython	765.020	784.355	780.386
luindex	124.723	124.672	124.930
lusearch	50.091	54.390	48.410
pmd	61.215	62.483	61.035
sunflow	151.774	171.973	147.738
tomcat	93.845	95.695	96.385
xalan	26.776	28.803	27.909

Table 1: Benchmark runtime for the three configurations. 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.

(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 amortized 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 duplicate number to the ones of configuration two. 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 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 1. 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 2 shows the relative change of the configurations compared to the unchanged VM results.

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 deduplica-

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	+13.31%	-2.66%
tomcat	+1.97%	+2.71%
xalan	+7.57%	+4.23%

Table 2: The relative impact of the optimization on the runtime compared to running the benchmarks in an unchanged VM.

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: Number of garbage collections for each benchmark at 2GB of heap space.

tion is a very specific optimization. The largest benefit is achieved for applications that handle a lot of strings. In addition, the approach changed the garbage collection process and does not interfere with any other part of the program.

In order to explain the results in Tables 1 and 2, two main factors have to be considered: the application structure and the number of 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 is started. The same behaviour is observed for `batik`, `fop`, and `luindex`. The full list of GCs is shown in Table 3. All but one of the collections were partial for each of the runs.

Negative relative impact in column B in Table 2 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 with 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 erased by the benefit of string deduplication. Even though an overall benefit was not achieved, the cost of the data structures was nullified by the benefit. The reasons for this behaviour is a large number of strings that are used for the database communication in the application. The benefit was limited by the fact, that most of the strings used were part of the internal string table and did hot 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 5 garbage collections, were `lusearch` and `sunflow`. A relative benefit of about 3.36% was achieved for `lusearch`. When compared with the second experiment configuration which shows that not deduplicating while comparing strings adds 8.5% to the run time. The characteristics 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 and 4. 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.

In addition to the runtime analysis, the times when garbage collections are compared in order to visualize the impact of the string deduplication. In order to analyze this metric, the fact that GC phases are longer due to a changed flow has to be considered. Figure 6 and 7 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 1.

$$T_n = OverallTime - \sum_{k=0}^{n-1} GCDuration_k \qquad (1)$$

Figures 6 and 7 show that an application, that benefits from string deduplication 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.

6. CONCLUSIONS AND FUTURE WORK

This paper presented an approach of detecting and handling duplicate strings on the heap in virtual machines. It introduced the problem 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.

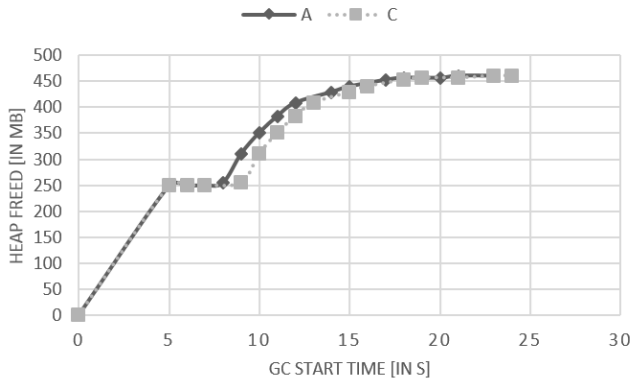

Figure 6: Start time and memory freed for each GC when running the lusearch benchmark.

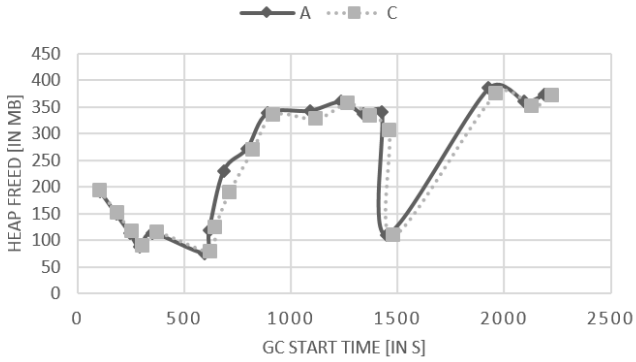

Figure 7: Start time and memory freed for each GC when running the h2 benchmark.

The experiments showed that benefits of the approach are very application specific. Especially string heavy applications showed promising results and revealed 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.

Deduplicating strings changes the objects 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 initially unrelated objects are placed in the same cache line. Those relations will be examined and investigated further as future work as well.

7. ACKNOWLEDGMENTS

The authors would like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program. Furthermore, we would also like to thank the New Brunswick Innovation Foundation for contributing to this project. Finally, we would like to thank the Centre for Advanced Studies—Atlantic for access to the resources for conducting our research.

8. REFERENCES

[1] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

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

[3] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.

[4] S. P. E. Corporation. SPEC JBB2005. https://www.spec.org/jbb2005/ [Online. Last accessed: 2016-Aug-24], 2016.

[5] S. P. E. Corporation. SPECjbb2013. https://www.spec.org/jbb2013/ [Online. Last accessed: 2016-Aug-24], 2016.

[6] R. R. Fenichel and J. C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, 1969.

[7] M. Horie, K. Ogata, K. Kawachiya, and T. Onodera. String deduplication for Java-based middleware in virtualized environments. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 177–188. ACM, 2014.

[8] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 2011.

[9] K. Kawachiya, K. Ogata, and T. Onodera. Analysis and reduction of memory inefficiencies in Java strings. *ACM Sigplan Notices*, 43(10):385–402, 2008.

[10] P.-Å. Larson. Dynamic hashing. *BIT Numerical Mathematics*, 18(2):184–201, 1978.

[11] D. Marinov and R. O'Callahan. Object equality profiling. In *ACM SIGPLAN Notices*, volume 38, pages 313–325. ACM, 2003.

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

[13] R. A. Saunders. The LISP system for the Q-32 computer. *The Programming Language LISP: Its Operation and Applications*, pages 220–231, 1974.