September 7, 2007

RT0750
Computer Science 12 pages

Research Report

A Quantitative Analysis of Space Waste from Java Strings and its Elimination at Garbage Collection Time

Kiyokuni Kawachiya, Kazunori Ogata, and Tamiya Onodera

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

A Quantitative Analysis of Space Waste from Java Strings and its Elimination at Garbage Collection Time

Kiyokuni Kawachiya Kazunori Ogata Tamiya Onodera
IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato, Kanagawa 242-8502, Japan
<kawatiya@jp.ibm.com>

Abstract

This paper describes a novel approach to reduce the memory consumption of Java programs, by reducing the *string memory waste* in the runtime. In recent Java applications, string data occupies a large amount of the heap area. For example, more than 30% of the live heap area is used for string data when WebSphere Application Server with Trade6 is running. By investigating the string data in real Java applications, we found two types of memory waste in typical string implementations in Java. First, there are many `String` objects which have the same values. Second, there are many unused areas in the `char` arrays used to hold the string values. This string memory waste exists as or in live objects, so it cannot not be eliminated by existing garbage collection techniques, which only remove dead objects. Quantitative analysis of Java heap revealed that such waste occupied up to 17% of the live heap area even in real Java applications.

To remove the string memory waste, we propose a new "string garbage collection" (StringGC) technique for Java. The StringGC works with a usual garbage collector in a JVM, unifying same-value `String` objects and removing the unused areas in `char` arrays. In an IBM production JVM, we implemented a StringGC prototype named "UNITE", where same-value strings are unified when they are tenured by a generational GC. This prototype was able to eliminate more than 90% of the string memory waste, and the live heap size of real Java applications was reduced by up to 15% without noticeable performance degradation.

1. Introduction

Virtually all programming languages provide support for strings together with a rich set of string operations. In Java [8], the standard class library includes three classes, `String` for immutable strings, and `StringBuffer` and `StringBuilder` for mutable strings.

Although the developer of a virtual machine attempts to implement these classes as efficiently as possible, there is little description in the literature on how efficient or inefficient they actually are in terms of time and space.

In this paper, we study *space waste* in a typical implementation of Java strings. We performed a quantitative analysis of the wasted space by running large enterprise applications in a production virtual machine.

The `String` class is typically implemented as in Figure 1. As seen in the figure, a string is represented with two objects, a `String` and a `char` array, which we call the *head* and *body* of the string, respectively. Space waste from Java strings comes in two forms, *duplication* and *fragmentation*. When there exist identical string heads or bodies, we say that there is duplication. When there are unused array elements in a body, we say that there is fragmentation.

Our measurements show that space wasted by Java strings is significant, sometimes as much as 17% of the live heap area in an enterprise application. For instance, we observed 1,067 instances of the string `"name"` and 773 instances of the string `"descriptorType"` in a snapshot of the Java heap (immediately after collecting the garbage) during the execution of the enterprise application.

We propose a new optimization at garbage collection time to eliminate space wasted by Java strings. The optimization, named "StringGC", can be implemented in a standard tracing collector, and eliminates duplication and fragmentation. We also implemented the simplest form of StringGC in an IBM production Java virtual

```
1 public final class String ... { // only has private fields
2   private char[] value; // char array which holds the string value
3   private int offset;   // start offset in the char array
4   private int count;    // length of the string value
5         :
6 }
```

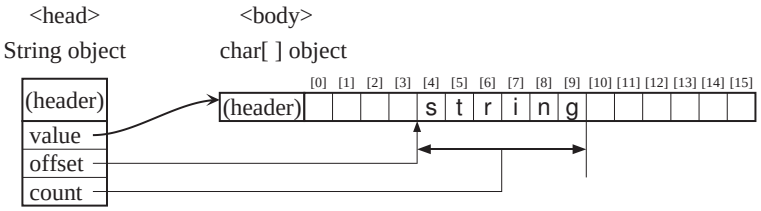


Figure 1. Structure of `String` object in typical Java implementations.

machine (JVM). Experimental results showed that it is able to eliminate more than 90% of the space wasted by strings, and it reduces the size of the live heap area by up to 15%.

In addition, we observed a surprising speedup in a benchmark of SPECjvm98 [17] with StringGC. The execution time of `_209_db` was reduced by more than 40%. We believe that eliminating duplicates allows the equality of two strings to be checked without any structural comparison.

Our contributions in this paper are as follows:

- *A quantitative analysis of space wasted by Java strings.* We measured the space waste by running large-scale enterprise applications in a production virtual machine. We observed that the space wasted by Java strings is 10–17% of the live heap area.
- *A proposal for garbage-collection-time optimization to eliminate the wasted space.* The optimization, called *StringGC*, can be integrated with a standard tracing collector. The optimization allows for trade offs between the computational complexity and the elimination effectiveness.
- *Empirical results of a prototype for GC-time elimination of the space waste.* We implemented the simplest form of StringGC in an IBM production Java virtual machine, and examined large enterprise applications. Results show that it eliminated more than 90% of the wasted space, and reduced the size of the live objects by up to 15%.

The rest of the paper is organized as follows. Section 2 describes how strings are handled in Java, and investigates wasted space due to Java strings in real applications. Section 3 proposes a garbage-collection-time elimination of the wasted space, called StringGC, and discusses its implementation variations. Section 4 shows experimental results using the simplest form of StringGC. Section 5 discusses related work, while Section 6 offers conclusions.

2. String Memory Waste in Java

This section describes the specification and typical implementation of strings in Java, and examines the space waste from it.

2.1 String Handling in Java

In Java, string data is mainly handled through `String`, which is one of the system-provided core classes. One important characteristic of a `String` object is that it is immutable [8]. User programs cannot modify its value by any method, except creating another `String` object. To manipulate string data, Java provides another class, `StringBuffer`. Objects of these string-related classes may also be created and converted implicitly by Java compilers such as `javac`.

Although the Java Language Specification [8] does not specify the details, the `String` class is typically implemented as in Figure 1. Actually, this is a simplified version of the implementation in the Apache Harmony open source Java SE project [1], but we know that many JVMs implement the class in a very similar way. As seen in the figure, a string is represented with two objects, a `String` and a `char` array (`char[]`), which we call the *head* and *body* of the string, respectively. The head points to the body through the `value` field. The

```
 1 class StringSample {
 2   public static void main(String[] args) {
 3     String sysjar = "system.jar", usrjar = "user.jar";
 4     String tmpstr = sysjar + ":" + usrjar + ":" + ".";
 5     int colon1 = tmpstr.indexOf(":"),
 6     int colon2 = tmpstr.indexOf(":", colon1+1);
 7     String jar2 = tmpstr.substring(colon1+1, colon2);
 8     tmpstr = null;
 9
10     System.out.println(jar2); // "user.jar" will be printed
11   }
12 }
```

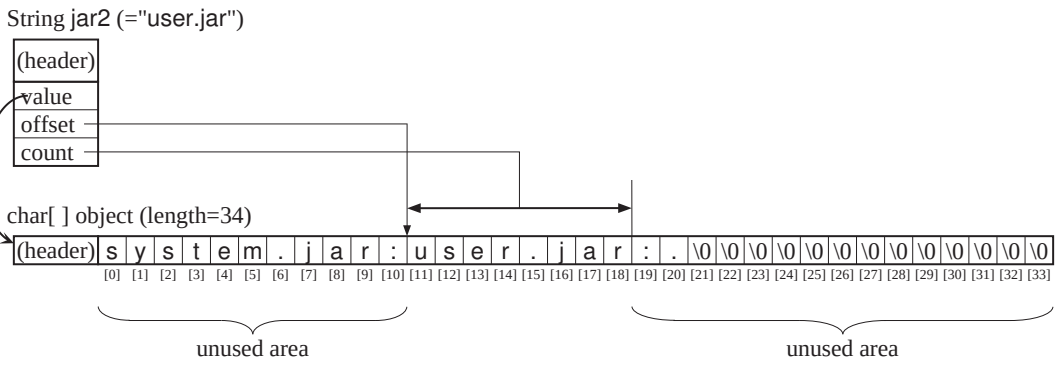


Figure 2. Sample program for string manipulation, and a resulting `String` structure.

actual value of the string is stored as a subarray of the body, where the `offset` field of the head gives the index of the first character, and the `count` field the length of the string. All of these fields are declared as `private`, and cannot be directly accessed by a user program. As an example, the `String` object[1] in Figure 1 represents a string `"string"`.

This structure is effective to reduce the copy of string values because a `char` array can be shared by multiple `String`s and `StringBuffer`s[2]. The sharing occurs, for example, when a new `String` object is created by `String.substring()` or `StringBuffer.toString()`. However, it introduces memory waste since *unused areas* may remain in the `char` array. The program in Figure 2 is an example. Although it depends on the JVM implementation, the string `jar2` will be as shown in the lower part of the figure, where 26 of the total 34 character slots of the `char` array are not used. This happens because a `StringBuffer` is internally created at line 4 and a generous `char` array is allocated for the string manipulation. This is an arbitrary example for the explanation, but similar string manipulations are performed frequently during Java program execution.

Another kind of memory waste in Java strings is that there are multiple `String` objects (and `char` arrays) that have the same string value. In the example in Figure 2, the strings `jar2` and `usrjar` have the same value `"user.jar"`. However, the `String` objects and the `char` arrays for these two strings exist independently in the heap. Such duplication occurs very frequently while executing real Java applications. As explained at the beginning of this section, `String` objects are immutable in Java, so it is possible to unify these duplicated strings without affecting program execution.

In Java, objects that are not referred to from anywhere are removed by garbage collection (GC) [13] as *dead* objects. However, the two kinds of *string memory waste* discussed here exist in *live* `char` arrays or as

[1] The "header" in each object in Figure 1 is an area used by JVM to store object-management information such as class pointer and lockword [14].

[2] The value of `StringBuffer` can be modified through its methods such as `insert()` and `delete()`. However, if the target `char` array is shared with `String` objects, a copy of the array is created before the modification.

Metrics	Trade6 count	Trade6 size	Tuscany count	Tuscany size
Total live objects	621,463	33,938 KB	281,973	15,332 KB
`String` objects	106,957	2,995 KB	57,922	1,622 KB
- duplicated	42,163	1,181 KB	33,939	950 KB
`char[]` objects	97,306	9,641 KB	49,150	4,470 KB
- used only for `String`	95,639	8,810 KB	48,118	3,628 KB
- duplicated	32,839	2,107 KB	25,578	1,596 KB
- unused in remaining	5,923	139 KB	2,019	46 KB
Total eliminable	75,002	3,427 KB	59,517	2,593 KB
- ratio to the heap	12.1%	10.1%	21.1%	16.9%

Table 1. String memory waste in real Java applications.

live `String` objects. Therefore, existing GC techniques cannot remove them. New mechanism to remove the waste is required for more efficient memory management in Java.

2.2 Investigation of the String Memory Waste

In recent Java applications, there are an increasing needs to handle string data, such as for processing XML and accessing databases, so the string memory waste explained above may become significant. Therefore, we did quantitative analysis of the waste in real Java applications.

The investigation was done with IBM's latest production JVM, the J9 Java VM [5, 10] 1.5.0 for Linux. For each investigation defined below, the Java heap area was dumped after a system GC and exhaustively analyzed using off-line tool. The following large-scale enterprise Java applications were chosen for the analysis.

Trade6: Running the Trade6 benchmark application [11] on IBM WebSphere Application Server (WAS) [12] Version 6.1. This investigates the heap after processing continuous requests from a test client for three minutes.

Tuscany: Running the Tuscany [3] Incubating-M1, open source middleware for the Service Component Architecture (SCA) [16], and a BigBank sample SCA application in the Apache Tomcat [2] servlet container. This investigates the heap after continuously sending a sequence of requests for three minutes.

The results are summarized in Table 1. In the table, the "Total live objects" row shows the numbers and sizes of the live objects at the time of the investigation. The J9 Java VM used for the investigation uses [5] type-accurate GC [13], so no dead objects are mixed into the measurement results.

The "`String` objects" row shows the numbers and sizes of live `String` objects, without including the `char` array objects used for the string bodies. The next "duplicated" row shows the `String` objects that can be removed because their value is the same as another `String` object[3]. For example, in Trade6, 42,163 of 106,957 (39.4%) `String` objects could be removed, or unified, because of the duplications.

Table 2 shows the top 15 duplicated strings for each application[4]. For example, there were 1,067 independent `"name"` strings in Tuscany.

The "`char[]` objects" row in Table 1 shows the number and size of live `char` array objects, and the next "used only for `String`" row shows the number excluding those referred to by `StringBuffer` or places other than `String` objects. The result shows that most (98%) `char` arrays are used just for holding the value of strings. Among these `char` arrays used for string bodies, the "duplicated" row shows the `char` arrays that can be removed when duplicated values are unified. The "unused in remaining" row shows the numbers of remaining `char` arrays which have unused areas as shown in Figure 2, and the total size of the unused area.

These kinds of string memory waste are accumulated in the "total eliminable" row, which indicates the totals for objects and for heap size that could be elimi-

[3] If there are 100 `String` objects which have the same value, "99" is shown in the table as the number of "duplicated" objects.

[4] The `"dolly"` in Trade6 result is the name of the machine that was running WAS.

Trade6		Tuscany	
1,055	`""`	1,067	`"name"`
930	`"java.lang.String"`	891	`"\n    "`
586	`"TRADE61 "`	773	`"descriptorType"`
586	`"TRAD61DB"`	620	`"\n        "`
370	`"Q1"`	527	`"displayName"`
361	`"1.0"`	504	`"attribute"`
345	`"server1"`	368	`"operation"`
327	`"name"`	363	`"java.lang.String"`
279	`"dollyNode04Cell"`	342	`"\n  "`
253	`"dollyNode04"`	304	`"\n\t\t\t"`
242	`"true"`	304	`"\n      "`
216	`"T1"`	267	`"none"`
215	`"en-US"`	252	`"role"`
214	`"type"`	248	`"displayname"`
207	`"void"`	246	`"false"`

Table 2. Top 15 duplicated string values and their counts in real Java applications.

nated[5]. The last row of the table shows the ratios of the eliminable amount to the heap.

From these results, the following can be observed:

- In the investigated Java applications, 5–12MB are used for holding string values, which occupies about 35% of the live heap area.
- 30–50% of the string memory is wasted by duplicates or unused areas. If we can eliminate this string memory waste, the heap area could be reduced by 10–17%.

This investigation discovered the importance of reducing the string memory waste for real Java applications. In next section, we will discuss concrete methods for doing this.

3. A Proposal for String Garbage Collection

The investigation in the previous section revealed that the Java heap can be reduced by up to 17% by removing the following two kinds of waste.

Waste-A: There are many `String` objects that have the same values.

Waste-B: There are many unused areas in the `char` arrays used for holding the string values.

In this section, we propose a "string garbage collection" (StringGC) technique to remove the string memory waste in Java.

[5] The `char[]` objects which have unused areas are not counted in the number of eliminable objects, but the unused areas are added to the eliminable size.

3.1 Core Algorithm

The StringGC we propose can be implemented as additional steps with the standard garbage collection methods of Java. First, we show the core algorithm of StringGC. A stop-the-world type of GC is assumed to simplify the explanation, but we believe various concurrent and parallel GC techniques [13] could also be used to improve the StringGC, some of which will be discussed in Section 3.2.

Step 1: Create `String` and `char` array tables.

Scan the heap, and create two tables that contain all live `String` objects and the `char` arrays used for string values. The tables should also contain *referer* information to retrieve all of the places each `String` or `char` array object is referenced. If a `char` array is referred to xfrom places other than `String` objects, remove it from the `char` array table, since such a `char` array cannot be restructured by StringGC. This step can be done during the mark phase in standard collectors.

Step 2: Unify duplicated `String` objects.

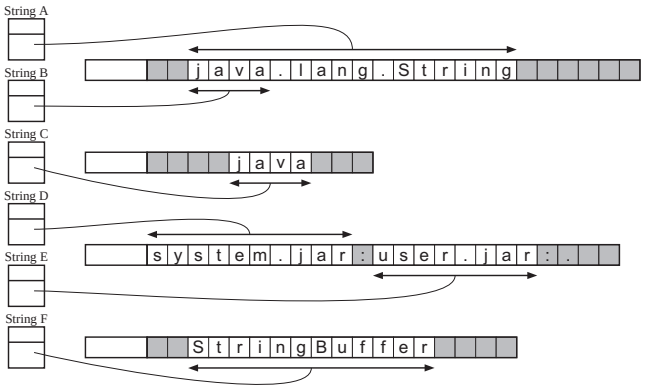
Scan the `String` object table, and unify the `String` objects that have the same string content. Retrieve the referer information of each duplicated `String` object and update the references to point to the unified `String` object. This is a mechanism similar to object relocation for heap compaction in standard collectors. With this step, the waste-A type can be eliminated. We will discuss some subtle issues on this string-head unification in Section 3.3.

Step 3: Remove unused areas in the `char` arrays.

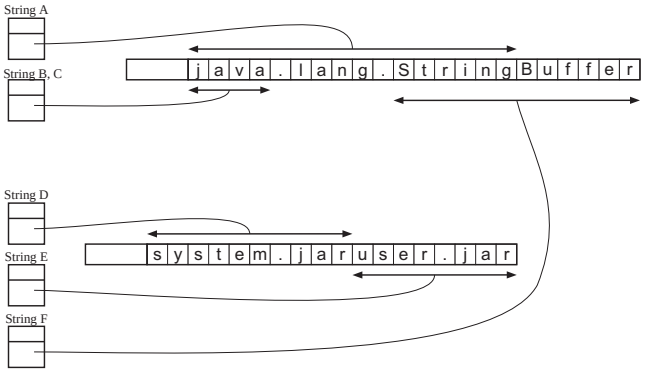
For each `char` array in the `char` array table, check its referer `String` objects. If the `char` array contains character slots not used by any `String` objects, restructure the `char` array by truncating the unused slots. Update the `value` and `offset` fields in the referer `String` objects to point to the appropriate substring of the new `char` array. With this step, the waste-B type can be eliminated.

Step 4: Unify `char` arrays used for `String`s.

Scan the `char` array table, and unify `char` arrays which have the same substring. An example is that if a `char` array's value is part of another `char` array's, the short array can be unified to use the long one. Similar to the previous step, update the referer



(a) Before the StringGC.



(b) After the StringGC.

Figure 3. String memory waste, and its elimination by StringGC.

`String` objects to point to the appropriate substring of the new `char` array. Actually, this step can be performed at the same time as Step 3.

Figure 3 shows an image of how string memory waste is removed by the StringGC. The upper figure (a) shows the heap before the StringGC, where the gray portion indicates the unused areas in `char` arrays (waste-B). There also exists identical (or partially identical) string values such as `"java"` and `"String"` (waste-A). When all the steps of the StringGC are applied, the heap becomes like the lower figure (b). The `String` objects B and C that have the same value `"java"` are unified (in Step 2), and the gray unused areas in the `char` arrays are removed (in Step 3). The `String` objects A and F, whose values are partially the same now point to a unified `char` array (in Step 4). Note that the references to the modified objects are appropriately updated in each step.

3.2 Variations

Several variations can be considered for each step of the StringGC explained above, according to the required space and time efficiency or combined GC algorithms.

First, the tables in Step 1 can be created independently for each `String` object or `char` array. Therefore, it is possible to parallelize this step for multiple processors (threads). It is also possible to create tables that contain only some of these objects if there is not enough memory, although that reduces the chances of unification.

For some combined GCs, the tables in Step 1 need not be explicitly created. If the collector maintains a structure to retrieve an object's referers, as might be used for implementing heap compaction, it could be utilized for the StringGC. The remaining steps could be done by scanning all of the live objects and performing the necessary steps if the object is a `String` or `char` array.

Step 3 could be simplified not to check all of the character slots one-by-one but to check only the start and end indexes of the used slots. This makes it impossible to remove the internal unused slots as in the third `char` array in Figure 3(a), but we believe this is a very rare case and negligible in real programs. Actually, further simplification to check only the end index of the used slot may be sufficient, since an unused area at the start can be created only when a substring is extracted with `String.substring()`, etc.

Many variations can be considered for the `char` array unification of Step 4. One simple implementation is to sort the `char` arrays in the table by their content and find an array whose content is part of the next array. The string data `"java"` in Figure 3 can be unified in this way. A more advanced implementation would be to find a `char` array whose first portion is the same as a latter portion of another `char` array and unify them, like for the `"String"` in Figure 3. However, it may need exhaustive search and increase the StringGC time. In real implementation, it is better to combine Step 4 with Step 3, to reduce redundant data copies and `char` array restructuring.

What variation should be used depends on the required space and processing-time efficiency. From the investigation of Table 1, simply unifying duplicated string values can eliminate 30–50% of the `char` arrays used for string values. If the combined GC has several processing levels, it is possible to apply aggressive

```
 1 class BadManner {
 2  public static void main(String[] args) {
 3   String s1 = new String("java");
 4   String s2 = new String("java");
 5   if (s1 == s2) // should use s1.equals(s2)
 6    System.out.println("Same Strings");
 7   else
 8    System.out.println("Different Strings");
 9  }
10 }
```

Figure 4. A bad-manner program incompatible with the string-head unification.

StringGC methods only at the time of the full collection.

3.3 Discussion

Step 2 of StringGC explained in Section 3.1 unifies `String` objects if they have the same value. For example, in Figure 3, `String` objects B and C whose values are `"java"` are unified. Through this unification of string heads, Java programs that do some *identity-based* operations on `String` objects may behave differently. The identity-based operations include reference comparisons with "`==`", monitor entrances and exits, and calls to `System.identityHashCode()`. Figure 4 shows an example that behaves differently after the `String` unification. If the `String` objects `s1` and `s2` are unified by StringGC, the program will print `"Same Strings"`, while it originally printed `"Different Strings"`.

However, in Java programming, using "`==`" to compare strings should be avoided and `String.equals()` is recommended. We believe identity-based operations should not be used for `String` objects in well-written Java programs.

If the string-head unification is still considered to be problematic, it is possible to unify only the bodies (`char` arrays) of the same-content strings in Step 2. Since the string body is not directly accessible by Java programs, this version is compatible with the identity-based operations. The heap area reduced by StringGC is decreased by this modification. However, from the investigation of Table 1, it can still remove 65% of the total string memory waste.

At the same time, unifying the `String` objects has a good side effect of speeding up the execution of `String.equals()`, because typical implementations of the method first check the two objects with "`==`". Slow character-by-character comparisons are unnecessary if the same-value `String` objects are unified by StringGC.

During the StringGC, the fields of a `String` object, `value` and `offset`, may be modified. Therefore, special care must be taken not to run StringGC while a Java program is accessing these fields. Fortunately, the `String` class is a final class provided by Java runtime and its fields are accessed only by the class or runtime. One possible solution is to minimize the sections which use the `String` fields and make GC-safe points out of these sections to suppress StringGC. Another practical solution is to unify `String` or `char` array objects only if the values of the `offset` fields are the same.

3.4 A Practical Prototype

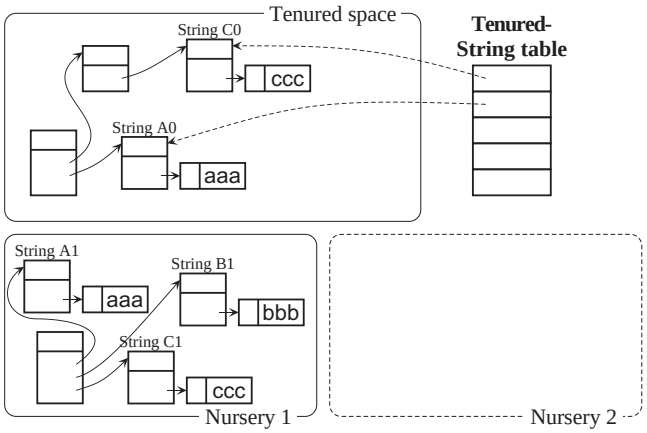
If the StringGC steps explained in Section 3.1 are completely applied, all of the string memory waste discussed in Section 2 can be removed. However, as shown in Table 1, many `String` objects exist in real Java applications, and handling them in a table may take long time. Therefore, we discuss a more practical subset of StringGC here which can remove most of the string memory waste without degrading the performance so greatly.

The investigation of Table 1 shows that more than 90% of the string memory waste comes from duplicated string values. Therefore, for the practical StringGC subset, we focused on unifying the same-value `String` objects.

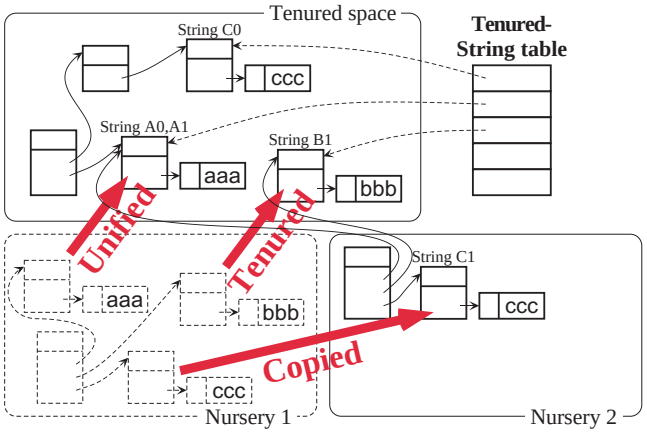
One naive implementation for this is to unify each `String` object at the time of its creation, by modifying the `String` constructors to reuse an existing `String` object that has the same value. However, searching for a `String` object with the same value takes time, and should not be done for temporary strings. To get better results for StringGC without degrading the performance, the search and unification should be applied only to long-living `String` objects.

Our solution for this problem is to combine StringGC with a generational garbage collector [13], and perform the `String` search and unification at the time of tenuring a `String` object, rather than when creating the object. We call this practical StringGC approach "UNITE (UNIfication at TEnuring)".

We have implemented the UNITE StringGC in IBM's latest production JVM, the J9 Java VM 1.5.0



(a) Before GC



(b) After GC

Figure 5. Behavior of the "UNITE" StringGC.

for Linux. The JVM can choose several GC policies [5], and of course generational GC was chosen for the prototype. In the prototype, all `String` objects in the tenured space are registered in a "tenured-`String` table". When the generational GC decides to move a `String` object from a nursery space to the tenured space, the table is searched to check if there is a same-value `String` object. If found, the nursery `String` object is unified to the found `String` object, which is already tenured. If not found, the nursery `String` object is moved to the tenured space and registered in the table. If a `String` object in the tenured-`String` table is no longer referred to from anywhere, the registration is removed when the tenured-space GC collects the dead object.

Figure 5 shows the example behavior of this UNITE StringGC, where the nursery space 1 became full and GC was performed. A long-living `String` object A1

```
 1 class MicroBench {
 2  static String[] doCreate(int dupRatio) {
 3   String[] strs = new String[1000000];
 4   int n = 0, dupCount = 1000000 * dupRatio/100;
 5   for (int i = 0; i < dupCount; i++)
 6    strs[i] = "STR_"+n;     //"STR_0"
 7   for (int i = dupCount; i < 1000000; i++)
 8    strs[i] = "STR_"+(++n);//"STR_1","STR_2",...
 9   return strs;
10  }
11  static int doCompare(String[] strs) {
12   String str0 = "STR_0";
13   int dupCount = 0;
14   for (int i = 0; i < 1000000; i++)
15    if (strs[i].equals(str0)) dupCount++;
16   return dupCount*100 / 1000000;//==dupRatio
17  }
18       :
19 }
```

Figure 6. Micro-benchmark to test the StringGC.

was tenured and unified to A0 which has the same value. On the other hand, B1 was moved to the tenured space and registered in the table, since there was no `String` object that has the same value. `String` object C1 was not old enough so just moved to the nursery space 2, although there was a same-value `String` object in the tenured space. The references to the unified or moved objects were appropriately updated.

4. Evaluation

Using the UNITE StringGC prototype, we measured its effectiveness with various Java programs. All of the measurements were done on a 3.06 GHz dual Xeon PC with 4 GB of memory, running the Red Hat Enterprise Linux 3 AS operating system.

4.1 Micro-Benchmarks

First, micro-benchmarks were performed to analyze the basic characteristics of the StringGC, using the program shown in Figure 6. The `doCreate()` method in the program creates 1,000,000 `String` objects through `StringBuffers`. The `dupRatio` percent of those created `String` objects had the same value (`"STR_0"`). In the measurements, this program was executed by specifying various `dupRatio` values on the JVMs with and without the StringGC. We confirmed that all of the related methods were JIT-compiled similarly in both JVMs.

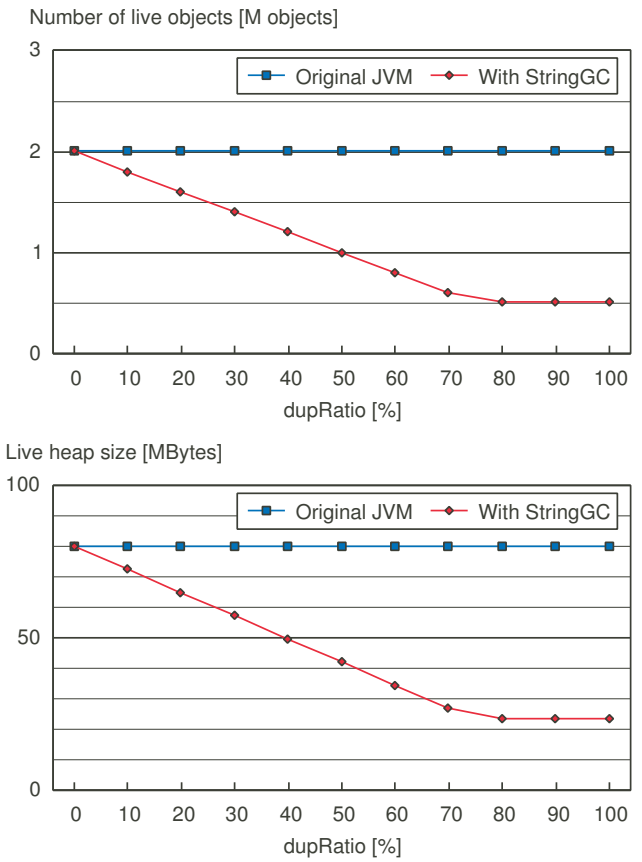


Figure 7. Heap analysis of the micro-benchmark.

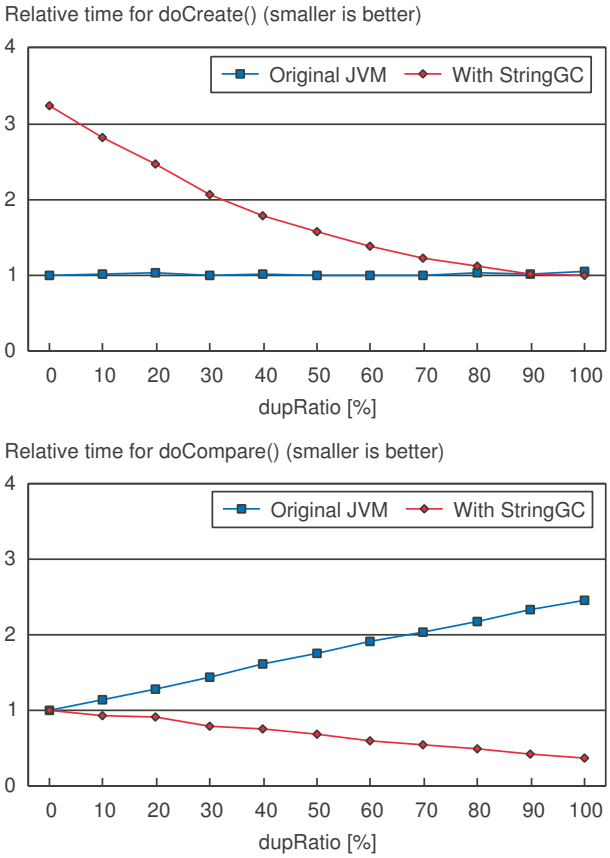


Figure 8. Relative times for `doCreate()` and `doCompare()` in the micro-benchmark.

The graphs in Figure 7 show the status of the Java heap just after the `doCreate()` method is executed. The upper graph shows the numbers of live objects and the lower graph shows their total size for each `dupRatio` value. Without the StringGC, the heaps were almost same for all of the `dupRatio` values. There were about 2 million live objects in the heap because one string is represented by two objects in Java, a `String` and a `char` array, as shown in Figure 1.

When the StringGC was enabled, both the number of live objects and their total size decreased as the `dupRatio` increased. The ratio of decrease matches the `dupRatio`, which indicates that `String` unification of the StringGC worked effectively. One interesting observation is that the heap decrease stopped around the 70% `dupRatio`. This is because the newest `Strings` were not unified since they remained in the nursery space, which was 32 MB in these measurements.

Next, the upper graph of Figure 8 shows the times for executing the `doCreate()` method, normalized to the time with no duplications on the original JVM. Without the StringGC, the times were constant for all `dupRatio` values. When the StringGC was enabled, the method became slower especially for low `dupRatio` cases. The reason is evidently the cost for maintaining the tenured-`String` table, where millions of strings were eventually registered. The time became better as the `dupRatio` increased, since the table size became smaller. The performance overhead with the StringGC peaked at the 3.2 times slowdown even for this artificial string-intensive program where millions of `String` objects exist in the tenured space. Performance with more realistic programs will be measured in the next section.

As discussed in Section 3.3, unifying the `String` objects has a side effect of speeding up some string comparisons. We also measured this effect in the micro-benchmark. The `doCompare()` method in Figure 6 was used for the measurements. This method compares the 1,000,000 `String` objects created by `doCreate()` with a string `"STR_0"` by using `String.equals()`. Therefore, `dupRatio` percent of the comparisons will succeed.

Metrics	Trade6 count	Trade6 size	Tuscany count	Tuscany size
Original JVM				
Total live objects	621,463	33,938 KB	281,973	15,332 KB
- `String` objects	106,957	2,995 KB	57,922	1,622 KB
- `char[]` objects	97,306	9,641 KB	49,150	4,470 KB
With StringGC				
Total live objects	547,999	30,824 KB	228,379	13,026 KB
- `String` objects	65,584	1,836 KB	27,306	765 KB
- `char[]` objects	66,149	7,662 KB	26,370	3,032 KB
Total eliminated	72,530	3,137 KB	53,396	2,295 KB
- ratio to the original heap	11.7%	9.2%	18.9%	15.0%
- ratio to the eliminable	89.6%	91.5%	86.8%	88.5%

Table 3. Heap memory reduction by the StringGC in real applications.

The lower graph of Figure 8 shows the normalized time for executing the `doCompare()` method for various `dupRatio` values. In the original JVM, the time became worse for higher `dupRatios`, because comparisons of same-value `String` objects eventually needed to execute character-by-character comparison, while hashcode comparison was sufficient for different strings. However, when the StringGC was enabled, the comparison became faster for higher `dupRatios`, because the same-value `String` objects had been unified. Consequently, for this micro-benchmark, comparing same-value `String`s was 6.4 times faster than on the original JVM.

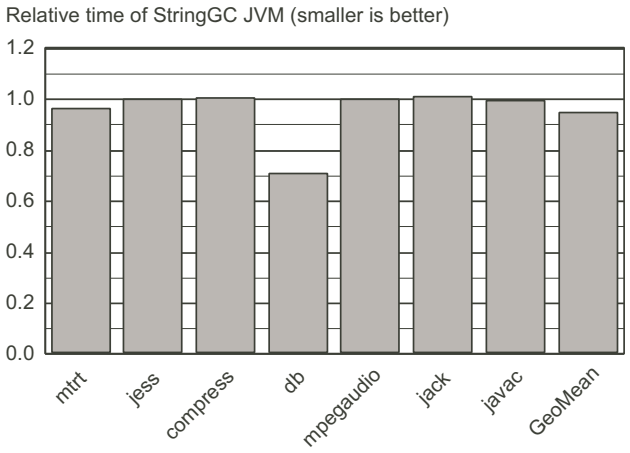


Figure 9. Relative times of the StringGC JVM in the SPECjvm98 benchmarks.

4.2 Macro-Benchmarks

Next, we measured the performance impact of the StringGC for real Java applications, by using the SPECjvm98 [17] benchmark programs. In the evaluation, each of the seven programs was run separately in the application mode, specifying the problem size as 100%. For each configuration, we took the average of the middle three scores of five independent runs.

Figure 9 shows the results, where the relative execution time on the StringGC JVM to the original JVM is shown. Surprisingly, we observed about 30% time reduction in `_209_db`. The reason is not clear yet, but we suspect that many string-compare operations in this benchmark were accelerated by the unification of `String` objects, as shown in the lower graph of Figure 8. For `_227_mtrt`, we also measured 3.6% performance improvement with the StringGC. Other benchmarks had almost the same scores for the two JVMs, and no significant degradation was observed.

Finally, we measured the effects for heap reduction with the StringGC in large-scale enterprise Java applications, which was the original concern of this research. The same programs, Trade6 and Tuscany described in Section 2.2 were used for the evaluation.

Table 3 shows the result of live heap analysis similar to Table 1, with and without the StringGC. In Trade6, 38.7% of the `String` objects and 32.0% of the `char` arrays were removed by the StringGC. Due to this, 9.2% of the heap area in the original JVM was eliminated. The StringGC was more effective for Tuscany, where 15.0% of heap area was eliminated. Comparing the Table tbl:realapp with Table 1, the StringGC prototype removed about 90% of the string memory waste.

5. Related Work

This section will introduce related work and compare that work with our StringGC.

In Java, a string can be *interned* by the `String.intern()` method, which returns a `String` object whose value is the same as the target `String`. A unique object is returned for the same string value. Therefore, interning can be used to unify duplicated `String` objects. However, it must be done explicitly in each Java application, and as far as we know, no application utilizes interning to reduce the string memory overhead. Actually, it is usually used to make `String` objects comparable by using "`==`". The simple idea of interning all `String`s when they are created does not work well, since many temporary `String`s will also be interned. In fact, our UNITE StringGC in Section 3.4 is considered to be a solution to this problem, by effectively interning only the long-living `String` objects.

It is common to eliminate duplicates of *string literals*. The Java Language Specification [8] dictates that all string literals and string-value constant expressions be interned in Section 3.10.5. The Unix operating system provides a tool named "`xstr(1)`" to extract strings from C programs and unify them.

Dieckmann and Hölzle [6] studied the allocation behavior of the SPECjvm98 benchmarks, and measured low-level data including age and size distribution, type distribution, and the overhead of object alignment. Their study did not include duplication or fragmentation due to unused areas, or specifically focus on strings. They mentioned in the conclusions that they are working on additional experiments for an extended version of the paper, suggesting that they are gathering new data which will illuminate the influence of strings on the heap composition. However, to the best of our knowledge, the new data remains unpublished.

Marinov and O'Callahan [15] presented Object Equality Profiling (OEP) for discovering opportunities for replacing a set of equivalent object instances with a single representative object. While we focus on duplication and fragmentation in strings, they studied duplication in general, or *mergeability* in their terminology, for arbitrary Java objects. They precisely define the mergeability problem, and describe how to efficiently compute mergeability. They developed a tool, a combination of an online profiler and postmortem analyzer, and revealed significant amount of object equivalence in real Java programs.

Automatic optimizations exploiting object equivalence is beyond the scope of their paper. However, they performed a case study for two SPECjvm98 benchmarks, `_209_db` and `_227_mtrt`, and manually optimized them to reduce the space used by live objects. Their tool showed that in `_209_db` only two classes, `String` and `char[]`, matter for mergeability, and that the relevant allocation site for the mergeable `String` was line 191 in the file `Database.java`. They then modified the line to call `String.intern()` immediately after the allocation. They observed a 47% reduction in the average space for live objects, and a slight improvement in the execution from 19.1 sec to 18.8 sec. This is an interesting contrast with the results from our optimization.

Hash-consing [7, 9] in functional languages guarantees that two identical objects share the same records in the heap. Hash-consing eliminates duplicated records, and allows the equality of two records to be determined without structural comparison. However, it must maintain a hash table to check whether there is already an identical record, and check the hash table at every allocation. Appel and Goncalves [4] proposed to integrate hash-consing with generational garbage collection. They only hash-cons records that survive a garbage collection, thereby avoiding the cost of a table lookup for short-lived objects. They implemented the scheme for Standard ML of the New Jersey compiler. Unfortunately, the space savings in the programs they measured were not impressive, less than 1% in most cases. They argue that hash-consing would show real promise when coupled with function memorization.

6. Conclusion

This paper described the proposal, implementation, and evaluation of "string garbage collection" (StringGC) to reduce the *string memory waste* in the Java heap caused by duplication and fragmentation of strings. The waste exists as or in live objects, so it cannot be reduced by existing GC techniques.

Recent Java applications handle more and more string data, such as for processing XML and accessing databases, and the string memory waste is becoming problematic. Quantitative analysis of real Java applications revealed that 10–17% of the live heap area is wasted.

In IBM's production JVM, we implemented a StringGC prototype named UNITE, where same-value `String`

objects are unified at the time of tenuring in the generational GC. Measurements using this prototype showed that the live heap area of real Java applications could be reduced by up to 15%, without noticeable performance degradation. Actually, for some programs the performance was even improved because of the reduced heap size and accelerated string comparisons.

As future work, we are considering further reduction of the execution overhead, implementation of more complete StringGC, and continuous measurements of heap status while executing Java applications.

Acknowledgments

We thank the members of the Systems Group in IBM Tokyo Research Laboratory, who have developed many Java-related optimization techniques and who always gave us valuable comments.

References

[1] The Apache Software Foundation. Apache Harmony. `http://harmony.apache.org/`

[2] The Apache Software Foundation. Apache Tomcat. `http://tomcat.apache.org/`

[3] The Apache Software Foundation. Apache Tuscany. `http://incubator.apache.org/tuscany/`

[4] A. W. Appel and M. J. R. Goncalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Department of Computer Science, Princeton University, 1993.

[5] C. Bailey. Java Technology, IBM Style: Introduction to the IBM Developer Kit: An overview of the new functions and features in the IBM implementation of Java 5.0, 2006. `http://www.ibm.com/developerworks/java/library/j-ibmjava1.html`

[6] S. Dieckmann and U. Hölzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmark. *Proc. ECOOP '99*, 92–115, 1999.

[7] A. P. Ershov. On Programming of Arithmetic Operations. *Communications of the ACM*, 1(8), 3–9, 1958.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*, Addison Wesley, 2005.

[9] E. Goto. Monocopy and Associative Algorithms in an Extended Lisp. Technical Report 74-03, Information Science Laboratory, University of Tokyo, 1974.

[10] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java Just-In-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications. *Proc. USENIX VM '04*, 151–162, 2004.

[11] IBM Corporation. IBM Trade Performance Benchmark. `https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=trade6`

[12] IBM Corporation. WebSphere Application Server: Product Overview. `http://www.ibm.com/software/webservers/appserv/was/`

[13] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley, 1996.

[14] K. Kawachiya. *Java Locks: Analysis and Acceleration*, Ph.D. Thesis, Graduate School of Media and Governance, Keio University, 2005. `http://www.trl.ibm.com/people/kawatiya/Kawachiya05phd.pdf`

[15] D. Marinov and R. O'Callahan. Object equality profiling. *Proc. OOPSLA '03*, 313–325, 2003.

[16] Open SOA. Service Component Architecture Home. `http://osoa.org/display/Main/Service+Component+Architecture+Home`

[17] Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. `http://www.spec.org/osg/jvm98/`