Analysis and Reduction of Memory Inefficiencies in Java Strings

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 focusing on their *string memory inefficiencies*. In recent Java applications, string data occupies a large amount of the heap area. For example, about 40% of the live heap area is used for string data when a production J2EE application server is running.

By investigating the string data in the live heap, we identified two types of memory inefficiencies — *duplication* and *unused literals*. In the heap, there are many string objects that have the same values. There also exist many string literals whose values are not actually used by the application. Since these inefficiencies exist as live objects, they cannot be eliminated by existing garbage collection techniques, which only remove dead objects. Quantitative analysis of Java heaps in real applications revealed that more than 50% of the string data in the live heap is wasted by these inefficiencies.

To reduce the string memory inefficiencies, this paper proposes two techniques at the Java virtual machine level, *StringGC* for eliminating duplicated strings at the time of garbage collection, and *Lazy Body Creation* for delaying part of the literal instantiation until the literal's value is actually used. We also present an interesting technique at the Java program level, which we call *BundleConverter*, for preventing unused message literals from being instantiated.

Prototype implementations on a production Java virtual machine have achieved about 18% reduction of the live heap in the production application server. The proposed techniques could also reduce the live heap of standard Java benchmarks by 11.6% on average, without noticeable performance degradation.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—frameworks

General Terms Languages, Design, Performance, Experimentation

Keywords Java, string, memory management, garbage collection, footprint analysis and reduction

1. Introduction

Virtually all programming languages provide support for strings together with a rich set of string operations. In Java [10], the standard class library includes three classes, `String` for immutable strings, and `StringBuffer` and `StringBuilder` for mutable strings. Although the developer of a Java virtual machine (JVM) 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 inefficiencies* in a typical implementation of Java strings.

Actually, the most frequently created objects during Java program execution are `String` objects, which represent strings, and `char` array objects to hold the string values[1]. In this paper, we call these objects *string-related objects*. For example, Figure 1 shows a live heap area when IBM WebSphere Application Server (WAS) [14] Version 6.1 with Trade6 benchmark application [13] is executed on IBM J9 Java VM [5, 12] 5.0 for Linux. String-related objects occupy about 40% of the live heap.

From exhaustive analysis of string data in the live heap, we found two types of memory inefficiencies. First, there are many string objects that have the same values. For instance, we observed 1,067 live instances of the string `"name"` in a snapshot of the Java heap during the execution of an enterprise application. Second, there are many string literals (constants) whose values are not actually used by the application. In particular, many strings for error messages are instantiated in the heap when message classes are initialized, but most of them are never used in normal execution.

[1] In this paper, `char` arrays directly used by applications are not treated as the string-related objects.

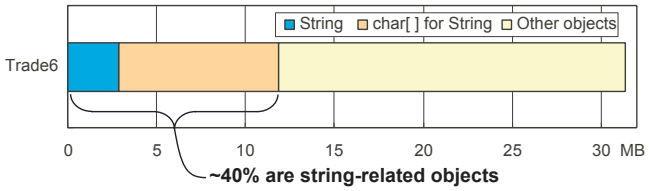

Figure 1. Live heap analysis of WAS 6.1 running Trade6. String-related objects occupy a large part of the live heap.

These *string memory inefficiencies* waste a considerable amount of the live heap in Java. Further quantitative analysis by running large enterprise applications in a production JVM showed that such wasted data occupies more than 50% of the live string data. Since these inefficiencies exist as live objects, they cannot be removed by existing garbage collection (GC) techniques. To remove the string memory inefficiencies, this paper proposes three new optimization techniques for each of the three major components, JVM, GC, and application classes.

The first technique, *StringGC*, is a new memory optimization at GC time to eliminate space wasted by duplicated Java strings. It is implemented with a standard generational collector, and unifies duplicated `String` objects when they are long-lived and moved to the tenured space. Experimental results showed that it is able to eliminate more than 90% of the space wasted by duplicated strings.

The second technique, *BundleConverter*, is a program converter to suppress the instantiation of message literals. It converts the internal structure of Java classes that extend `java.util.ListResourceBundle`. After the conversion, `String` objects are created only when messages are actually requested, rather than when the class is initialized. Converted classes can be used without modifying other part of applications. Since most error messages are not used during normal execution, this conversion can suppress 99% of message instantiations.

The third technique, *Lazy Body Creation*, is also targeting the reduction of unused literals, and is implemented in the JVM and its `String` class. When the JVM instantiates a string literal, it creates only *part* of the data structure. The remaining part, which contains the value of the string, will be created when the literal is actually used. This technique can be used for all literals, even for classes that cannot be converted to suppress the instantiation of string literals. With this technique, the space wasted by unused literals was reduced by more than 60%.

Combination of these three techniques on a production JVM has removed about 90% of the string memory inefficiencies and achieved about an 18% reduction of the live heap in real enterprise applications. The proposed techniques are also effective for client-type applications. Using SPECjvm98 [24] and DaCapo [6] benchmark programs, we confirmed that live heap was reduced by 11.6% on average without noticeable performance degradation.

Our contributions in this paper are as follows:

- *A quantitative analysis of string memory inefficiencies in Java.* We identified several types of memory inefficiencies in Java strings and measured the wasted space by running large-scale enterprise applications in a production JVM. We observed that the space wasted by Java strings can be about 20% of the live heap area.
- *Three proposed techniques to reduce the inefficiencies.* To remove the string memory inefficiencies, we introduced three new techniques, one for each of the three major components, JVM, GC, and application classes. StringGC eliminates duplicated strings at the time of garbage collection. BundleConverter converts application classes to avoid instantiating unused message literals. Lazy Body Creation delays part of the literal instantiation in the JVM.
- *Empirical results of reducing the inefficiencies using a prototype on a production JVM.* We implemented these three techniques for a production Java virtual machine, and examined large enterprise applications. Results show that it eliminated about 90% of the string memory inefficiencies, and reduced the size of the live heap by about 18%. Using standard Java benchmarks, we also verified that our techniques are applicable to most Java applications.

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 three techniques to remove the string memory inefficiencies. Section 4 shows experimental results of these three techniques using a prototype on a production JVM. Section 5 discusses related work, while Section 6 offers conclusions.

2. String Memory Inefficiencies in Java

This section describes the specification and typical implementation of strings in Java, and examines the resulting memory inefficiencies.

2.1 String Handling in Java

In Java, string data is mainly handled through a class `String`, which is one of the core classes provided with the JVM. One important characteristic of `String` objects is that they are immutable. User programs cannot modify their values by any method. To manipulate string data, Java provides another class, `StringBuffer`[2]. These classes can explicitly be used by user programs, but objects of these classes may also be created and converted to each other implicitly by Java compilers such as `javac`.

Although the Java Language Specification [10] does not specify the details, the `String` class is typically implemented as in Figure 2. Actually, this is a simplified version

[2] There is one more string-related class, `StringBuilder`, which is a thread-unsafe (but faster) version of `StringBuffer`.

```
1 public final class String ... { // only has private fields
2   private char[] value; // char array to hold the value
3   private int offset;   // start offset in the char array
4   private int count;    // length of the string value
5   private int hashCode; // hash code of this object, or 0
:       :
6 }
```

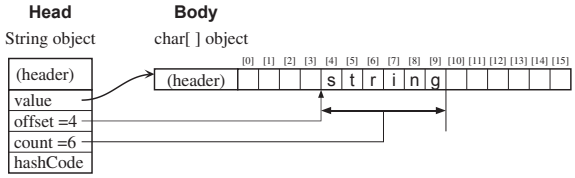

Figure 2. Structure of a Java string in typical implementations. Two objects, a `String` (head) and a `char[]` (body), are used to represent each Java string.

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 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. The `hashCode` field is used to store the hash code of the `String` object. All of these fields are declared as `private`, and cannot be directly accessed by a user program. As an example, the `String` object[3] in Figure 2 represents a string `"string"`.

The `String` class defines its own `hashCode()` method to get the object's hash code, which is calculated from the value of the string using a formula defined in the class specification [25]. Since the hash code is uniquely determined for each `String` object, it is stored into the `hashCode` field for future use, once it is calculated. 0 in this field means that a hash code has not been calculated yet.

String literals are constant values declared in Java programs using double-quotes, such as `"literal"`. Their values are stored as UTF-8 [26] data in the constant pool of each class file [19]. The JVM *instantiates* each literal into the Java heap as a `String` object and a `char` array when it is first used by the program with the `ldc` (load constant) bytecode. Since each `char` data represents a 16-bit Unicode (UTF-16) [26] character, the UTF-8 data in the constant pool is converted to the corresponding Unicode data and put into the created `char` array in the heap.

A `String` can be *interned* by explicitly invoking its `intern()` method, which returns a unified `String` object with the same value. The Java Language Specification [10] specifies that all string literals are automatically interned when it is instantiated. Therefore, there is no duplication of string literals in the Java heap.

[3] The "header" in each object in Figure 2 is an area used by JVM to store object-management information such as class pointer and lock status.

```
1 class WasteSample {
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" is printed
11 } }
```

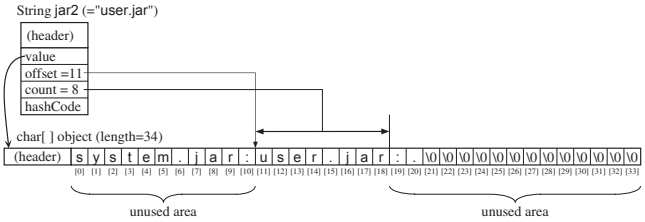

Figure 3. Sample program for string manipulation, and a resulting `String` structure, whose `char` array contains large unused areas. There also exist multiple `String` objects which have the same value `"user.jar"`.

2.2 String Memory Inefficiencies

The string structure shown in Figure 2 is effective for reducing the copying of string values because the `char` array can be shared by multiple `String` and `StringBuffer` objects[4]. The sharing occurs, for example, when a new `String` object is created by `String.substring()` or `StringBuffer.toString()`. However, it may result in memory being wasted differently, since unused areas may remain in a `char` array. In this case, we say that there is *fragmentation* in the array.

The program in Figure 3 is an example of such waste. Although it depends on the JVM implementation, the resulting string `jar2` will be as shown in the lower part of the figure, where 26 of the total 34 elements of the `char` array are not used. This happens because a `StringBuffer` was implicitly created in line 4 and a generous `char` array was 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 may exist multiple `String` objects (and `char` arrays) that have the same string value. In the example in Figure 3, 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 in real Java applications.

The fragmentations and duplications explained above typically occur for strings dynamically generated during the program execution. In contrast, string literals basically do not have such problems, since they are directly created from

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

```
 1 class MyMessages {
 2   private static String[] messageList = {
 3     "message0",  // 0
 4     "message1",  // 1
 5     "message2",  // 2
 6     "message3"   // 3
 7   };
 8   String getMessage(int key) { // returns a message
 9     return messageList[key];   // specified by the key
10 } }
11
12 class MessageTest {
13   public static void main(String[] args) {
14     MyMessages myMsg = new MyMessages();
15     System.out.println(myMsg.getMessage(2));
16 } }
```

Figure 4. An example where unused literals are created. All of the messages are instantiated, even though only one of them is used.

constant values in classes and automatically unified by interning. However, there is another type of inefficiency for string literals in Java. It is very common that string literals are instantiated too eagerly, even though their values are not actually used by an application.

Figure 4 is an example of such *unused literals*. When this program is executed, all of the literals `"message0"` to `"message3"` are instantiated at line 14, where the class `MyMessages` is loaded and initialized, although only `"message2"` is actually used by the program. Once a string literal is instantiated, it continues to exist as a live object unless the class is unloaded. Therefore, the instantiation of unused literals may consume much of the Java heap.

In this section, we defined three kinds of memory inefficiencies in Java strings — fragmentation, duplication, and unused literals. In Java, objects that are not referred to from anywhere are removed by garbage collection [15] as *dead* objects. However, all of the memory inefficiencies discussed here exist as *live* `String` objects or in *live* `char` arrays. Therefore, existing GC techniques cannot remove them. New mechanisms to remove the waste are required for more efficient memory management in Java.

2.3 Quantitative Analysis of the Inefficiencies

In recent Java applications, there are increasing needs to handle string data, such as for processing XML and accessing databases, so the string memory waste discussed 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, 12] 5.0 for Linux. For each of the investigations defined below, the Java heap area was dumped after a system GC and exhaustively analyzed using an off-line tool. The following large-scale enterprise Java applications were chosen for the analysis:

Trade6: Running the Trade6 benchmark application [13] on IBM WebSphere Application Server (WAS) [14] Version 6.1. The benchmark models a financial services trading platform, whose data is stored in an external database. This test investigates the heap after processing requests based on a trading scenario for three minutes.

Tuscany: Running the Tuscany [3] Incubating-M1, open source middleware for the Service Component Architecture (SCA) [23], on the Apache Tomcat [2] servlet container. This test investigates the heap after processing requests to a BigBank sample SCA application for three minutes.

Metrics	Trade6 count	Trade6 size	Tuscany count	Tuscany size
Total live objects	582,012	31,346 KB	281,210	15,193 KB
`String` objects	103,370	2,894 KB	57,773	1,618 KB
- duplicated	36,468	1,021 KB	33,914	950 KB
- literals	35,498	994 KB	9,342	262 KB
- unused literals	24,636	690 KB	5,619	157 KB
`char[]` objects	94,553	9,445 KB	48,985	4,428 KB
- used for `String`	92,966	8,967 KB	47,991	3,649 KB
- duplicated	28,166	2,005 KB	25,564	1,606 KB
- unused in remaining	6,171	154 KB	2,026	47 KB
- literals	35,498	3,100 KB	9,342	533 KB
- unused literals	24,636	2,152 KB	5,619	283 KB
String-related objects	196,336	11,861 KB	105,764	5,267 KB
- duplication	64,634	3,026 KB	59,478	2,556 KB
- fragmentation	6,171	154 KB	2,026	47 KB
- unused literals	49,272	2,842 KB	11,238	441 KB
Total waste	113,906	6,022 KB	70,716	3,043 KB
- ratio to str-rel obj	58.0%	**50.8%**	66.9%	**57.8%**
- ratio to the live heap	19.6%	**19.2%**	25.1%	**20.0%**

Table 1. String memory inefficiencies in real Java applications. Total waste reaches about 20% of the live heap.

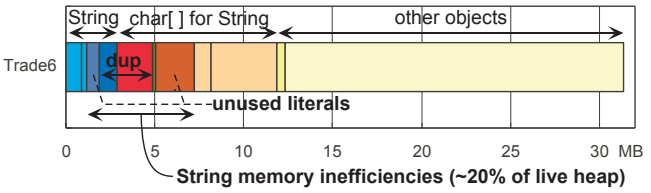

Figure 5. String memory breakdown in Trade6. Much memory is wasted by duplications and unused literals.

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 analysis. The J9 Java VM used for the investigation uses type-accurate GC [5, 15], 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[5]. For example, in Trade6, 36,468 of 103,370 (35.3%) `String` objects could

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

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\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. There are more than 1,000 independent `"name"` strings in Tuscany.

be removed, or unified, because of the duplications. Table 2 shows the top 15 duplicated strings[6] for each application. For example, there were 1,067 independent `"name"` strings in Tuscany.

Returning to Table 1, the "literals" row in the `String` section shows the number and size of `String` objects created from string constants. Among these literals, the "unused literals" row shows the ones whose values are not used by the applications. We measured these numbers by modifying the JVM and its `String` class library to mark each `String` object when it is created by the `ldc` bytecode (literal creation) and when its `value` field is accessed (literal use). The result in the table shows that 16–34% of the live `String` objects are literals. Surprisingly, more than 60% of the string literals are not used even though they exist in the live heap. Table 3 shows some examples of such unused literals in Trade6. It is observed that many error messages are unnecessarily instantiated.

The "`char[]` objects" row in Table 1 shows the number and size of live `char` array objects, and the next "used for `String`" row shows the number used for holding `String` values. The result shows that almost all (98%) of the `char` arrays are used for holding the string values. Among these `char` arrays used for string bodies, the "duplicated" row shows the ones 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 (fragmentations) as shown in Figure 3, and the total size of the unused elements. The rows "literals" and "unused literals" in the `char[]` section have the same meanings as those in the `String` section, except that the size columns show the size of `char` arrays used for holding the literal values.

The remaining part of Table 1 summarizes the three kinds of string memory inefficiencies measured above. Here,

[6] The `"dolly"` in Trade6 results is the name of the machine that was running the application.

Trade6
`"SECJ0040E: Error occurred while generating new LTPA keys. The exception is {0}."`
`"SECJ0041E: Can't set Authentication Mechanism to LTPA when LTPAConfig is null"`
`"SECJ0270E: Failed to get actual credentials. The exception is {0}."`
`"WSWS3221E: Error: Bean attribute {0} is of type {1}, which is not a simple type."`
`"WSWS3222E: Error: Attribute is of type {0}, which is not a simple type."`
`"WSWS3220E: Error: Error: Empty or missing service name."`

Table 3. Example of unused literals in Trade6. Many error messages are unnecessarily instantiated.

string-related objects mean `String` objects and `char` arrays used for holding their values. Finally, the "Total waste" row indicates the totals for objects and for heap memory that were wasted[7] by the string memory inefficiencies. The last row of the table shows the ratios of the wasted amount to the heap. Figure 5 is a visualization of the result for Trade6.

From these results, the following can be observed:

- In the investigated Java applications, 35–40% of the live heap area is used for holding string values.
- 25–50% of the string memory is wasted by duplications.
- Compared to the duplication, the impact of fragmentation in `char` arrays is small. It is less than 2% of the string memory.
- 16–34% of live `String` objects are literals, but more than 60% of them are not used by the program. The unused literals occupies 8–24% of the string memory.
- As a result of these inefficiencies, more than 50% of the string memory is wasted, which is about 20% of the live heap.

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

3. Reduction of the String Memory Inefficiencies

The investigation in the previous section revealed that the Java heap can be reduced by up to 20% by removing the following string memory inefficiencies:

Duplication: There are many `String` objects that have the same values.

Fragmentation: There are unused areas in the `char` arrays used for holding the string values.

Unused literals: Many literals are instantiated even though they are not used.

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

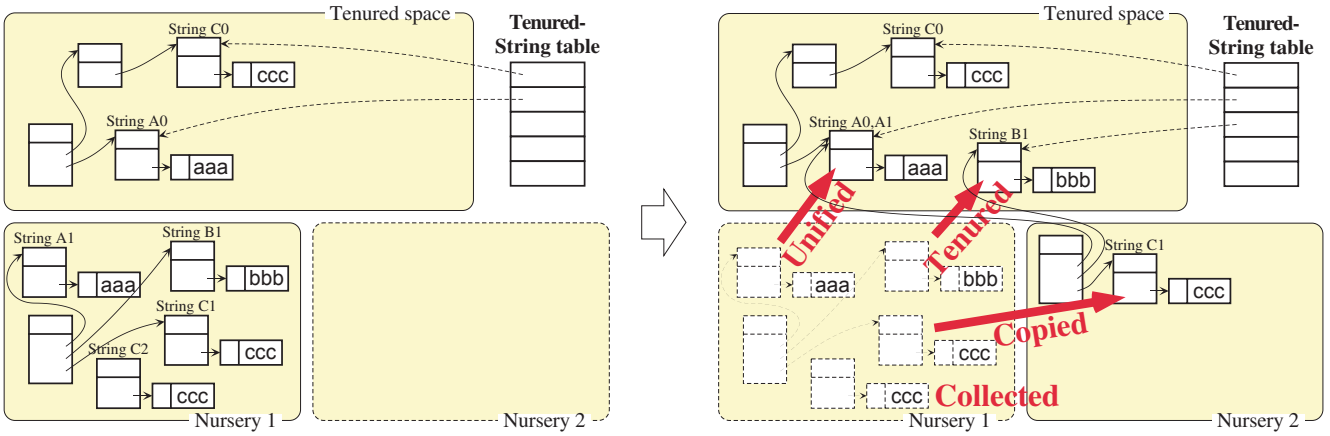

Figure 6. Behavior of the "UNITE" StringGC. String A1 is UNIfied with the same-value string A0 when it is TEnured.

In this section, we propose three independent techniques to reduce the inefficiencies. To reduce the duplication and fragmentation, we propose an enhancement of garbage collectors, named *StringGC*. To reduce the unused literals, we propose two approaches, program conversion by *Bundle-Converter* and *Lazy Body Creation* in the JVM.

3.1 String Garbage Collection

First of all, we propose a "string garbage collection" (StringGC) technique to address the duplication and fragmentation in Java strings. The key idea is to restructure string-related objects at the time of garbage collection. To remove the duplicates, `String` objects with the same value are unified. To eliminate the fragmentation, `char` arrays are shrunk to contain only the characters used as string values.

The StringGC generally aims at the elimination of duplicates and fragments, but the investigation results of Table 1 showed that the impact of fragmentation is not as large as that of duplication. Therefore, as a practical approach to StringGC, we focused on unifying the `String` objects with the same values[8].

One naive way to reduce the duplication would be 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-lived `String` objects. Our solution for this problem is to combine StringGC with a generational garbage collector [15], and perform the `String` search and unification at the time of tenuring a `String` object, rather than when creating the object. We call this pragmatic StringGC approach "UNITE (UNIfication at TEnuring)".

[8] Refer to our research report [18] for the full algorithm of the generic StringGC, which also tries to eliminate the fragmentation.

We implemented the UNITE StringGC in IBM's latest production JVM, the J9 Java VM 5.0 for Linux. The JVM can choose several GC policies [5], and 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 the nursery space to the tenured space, the table is searched to check if there is a `String` object with the same value. If found, the nursery `String` object is unified with 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, the registration is removed when the tenured-space GC collects the dead object.

Figure 6 shows an example of the behavior of this UNITE StringGC, in which Nursery Space 1 became full and GC was performed. A long-lived `String` object A1 was tenured and unified with A0 that has the same value. In contrast, B1 was moved to the tenured space and registered in the table, since there was no `String` object with the same value. `String` object C1 was not old enough so just moved to Nursery Space 2, although there was a same-value `String` object C0 in the tenured space. `String` object C2 was not moved, and therefore was collected, since it was not reachable from GC root sets. The references to the unified or moved objects were appropriately updated by the GC.

3.1.1 Discussion

The StringGC explained above unifies `String` objects if they have the same value. For example, in Figure 6, `String` objects A0 and A1 whose values are `"aaa"` are unified. Due to 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 7

```
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 } }
```

Figure 7. A bad-mannered program incompatible with the string-head unification. In this program, `s1` and `s2` are "Different Strings" although they have the same value.

```
1 public final class String ... {
:       :
2   public boolean equals(Object o) {
3     if (o == this) return true;
4     if (!(o instanceof String)) return false;
5     String s = (String)o;
6     if (s.value  == value  &&           // newly
7         s.offset == offset &&           // added
8         s.count  == count) return true; // check
9     // original comparison code follows
:       :
10 } }
```

Figure 8. Modified `String.equals()` to exploit the string-body unification. The comparison is accelerated if the string bodies have been merged.

shows an example that behaves differently after the string-head 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 strongly believe that well-written Java programs should not use such identity-based operations for `String` objects. There are Java programs that intentionally use "==" for the string comparison, by interning the involved strings in advance. Such programs still work correctly even if the string heads are unified. The issue only occurs if a program relies on the "non-identity" of same-value `String` objects, which should be very rare.

If the string-head unification is still considered to be problematic, the issue can be avoided by unifying only the bodies (`char` arrays) of the same-value strings. 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, but the analysis of Table 1 shows that we still have a chance to remove more than 60% of the waste by duplication, since `char` arrays occupy the main part of the string memory.

Unifying the `String` objects has a good side effect in speeding up the execution of `String.equals()`, because typical implementations of the method first check the two objects with "==". Slow character-by-character comparisons

```
1 class MyMessages {
2   String getMessage(int key) {
3     switch (key) {
4     case 0: return "message0";
5     case 1: return "message1";
6     case 2: return "message2";
7     case 3: return "message3";
8     default: // error handling
9 } } }
```

Figure 9. An example of program conversion to reduce unused literals. Only the requested messages will be instantiated.

are unnecessary if the same-value `String` objects are unified by StringGC. Even if the string heads are not unified, StringGC can speed up `String.equals()` by modifying the method to compare the `value`, `offset`, and `count` fields as shown in Figure 8. This is because these fields become the same if string bodies with the same value are unified by StringGC.

When a `String` is unified, the values of its `value` and `offset` fields may change. 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 the Java runtime and its fields are accessed only by the `String` class or runtime. One possible solution is to minimize the sections which use the `String` fields and to 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. For the evaluation in the next section, we adopted the former approach.

3.2 Program Conversion to Reduce Unused Literals

The StringGC aims at the reduction of the duplication by unifying strings at the time of garbage collection. However, it cannot reduce the waste from unused literals, since it is difficult for the GC to know which string literals have not been (and will not be) used by the application. It is better to remove the waste from unused literals *before* the literals are fully instantiated in the Java heap.

To reduce the unused string literals, we first propose a program conversion. In Java programming, a typical coding pattern can instantiate almost all string literals at the time of class initialization regardless of whether or not they are actually used. This pattern is often used for handling message strings, as shown in Figure 4. Therefore, we propose converting (or rewriting) such message classes to a structure that instantiates the messages only when they are actually used by the application.

Figure 9 is an example of such a conversion for the `MyMessages` class in Figure 4. In the original class, all messages are instantiated at the time of class initialization. In contrast, in the converted class, messages are instantiated when they are first requested through the `getMessage()` method, so memory waste by unused literals is reduced.

Trade6		
#chars	#strs	Class name
86,702	1,628	security_en
86,702	1,628	security
72,594	1,340	engineMessages_en
72,594	1,340	engineMessages
31,564	492	CWSICMessages_en
31,564	492	CWSICMessages
26,997	382	CWSIPMessages_en
26,997	382	CWSIPMessages
25,782	466	HAManagerMessages_en
25,782	466	HAManagerMessages
:	:	:
2,181,795	64,887	TOTAL

Table 4. Top 10 classes that have many instantiated string literals in Trade6. All are subclasses of `ListResourceBundle`.

Actually, only a string `"message2"` is instantiated when the converted class is used with the `MessageTest` program in Figure 4.

Such program conversion can be done manually by programmers. However, if the structure of message classes are well defined and known, there is a chance to automate the conversion. To find such opportunities, we performed additional investigations for unused literals. Table 4 shows the top 10 classes[9] that have many instantiated string literals in the Trade6 test case. For each class shown in the third column, the second column shows the number of string literals instantiated from the class, and the first column shows their total number of characters. The bottom line is the total for all classes (including the top 10), which shows that about 65,000 literals[10] were instantiated for the application.

Here we noticed that all of the top 10 classes are subclasses of `java.util.ListResourceBundle`, which is a framework to provide internationalized message sets in Java [7]. Actually, all unused messages shown in Table 3 came from these classes. Figure 10 shows a typical program using `ListResourceBundle`. By extending the class, a message class for English locale, `MyResources_en`, is defined. `BundleTest` is a sample program to use the message class, where a message for a specific key is retrieved by `getString()`. In this example, all of the messages and keys are instantiated when the message class is loaded and initialized at line 15.

The `getString()` method, defined in `ResourceBundle`, internally invokes `handleGetObject()` of the specified bundle to retrieve an object from a key string. By replacing this method using a similar conversion technique as the one used in Figure 9, we can suppress the instantiation of unused literals. Figure 11 shows the converted `MyResources_en` class. Here the converted class is a subclass of `ResourceBundle`, since `handleGetObject()` is defined as a `final` method of `ListResourceBundle` and cannot be overridden. This change should not affect applications that conform to the `ResourceBundle` interface.

Note that the conversion shown in Figure 11 includes an additional technique to reduce unused literals. Since the key passed to `handleGetObject()` is a string, such as `"k2"` in Figure 10, direct comparison of the passed key with the message keys will result in instantiating all those keys that appear in the path executed. To avoid this, keys are first checked for their hash codes. In Figure 11, the numbers 3365–3368 correspond to the hash codes of `"k0"`–`"k3"`, respectively. As explained in Section 2.1, the hash code of a `String` is determined from its value using a formula defined in the specification [25]. Therefore, it is possible to precalculate them at the time of conversion. String comparison is done only when the hash code matches the specified key.

```
 1 import java.util.*;
 2 public class MyResources_en extends ListResourceBundle {
 3   public Object[][] getContents() { return contents; }
 4   private static final Object[][] contents = {
 5     {"k0", "message0"},
 6     {"k1", "message1"},
 7     {"k2", "message2"},
 8     {"k3", "message3"}
 9   };
10 }
11
12 class BundleTest {
13   public static void main(String[] args)
                                        throws Exception {
14     Locale l = Locale.ENGLISH;
15     ResourceBundle myRes =
             ResourceBundle.getBundle("MyResources", l);
16     System.out.println(myRes.getString("k2"));
17 } }
```

Figure 10. An example that uses `ListResourceBundle`. All messages (and keys) are instantiated in line 15.

```
 1 import java.util.*;
 2 public class MyResources_en extends ResourceBundle {
 3   protected Object handleGetObject(String key) {
 4     int hc = key.hashCode();
 5     if (hc == 3365 &&
                 key.equals("k0")) return("message0");
 6     else if (hc == 3366 &&
                 key.equals("k1")) return("message1");
 7     else if (hc == 3367 &&
                 key.equals("k2")) return("message2");
 8     else if (hc == 3368 &&
                 key.equals("k3")) return("message3");
 9     else return null;
10   }
11   public Enumeration<String> getKeys() {
 :       :
12 } }
```

Figure 11. A message bundle converted to reduce unused literals. Only the requested messages (and keys) will be instantiated.

[9] Package names of classes are omitted in this table because of the space limitation.

[10] In this test case, `FooMessages_en` and `FooMessages` contain exactly same message sets, so the number of literals in the live heap is smaller than this.

DateFormatZoneData	
"localPatternChars"	:
"Acre Time"	"US/Mountain"
"ACT"	"US/Pacific"
"Acre Summer Time"	"US/Pacific-New"
"ACST"	"US/Samoa"
"Central Standard Time (South Australia)"	"VST"
	"W-SU"
"CST"	"Zulu"
:	"GyMdkHmsSEDFwWahKzZ"

Table 5. Some of the string literals instantiated when `DateFormatZoneData` is initialized. Most remain unused.

By this conversion, we can almost completely suppress the instantiation of unused literals. For example, when the converted class in Figure 11 is used with the `BundleTest` program in Figure 10, only two strings, `"k2"` and `"message2"`, are instantiated from the converted class.

Since the structure and usage of the message classes extending the `ListResourceBundle` are well known, it is possible to automatically convert such classes to the structure shown in Figure 11. For this purpose, we created a program named *BundleConverter*. This program loads a specified subclass of `ListResourceBundle`, extracts all keys and messages defined in the class, and generates a class converted as shown in Figure 11. By replacing the original `ListResourceBundle` subclasses with the converted ones, applications can reduce the instantiation of unused literals.

3.3 Lazy Body Creation

The program conversion shown above is very effective in suppressing the instantiation of unused literals. However, such conversion is difficult for classes whose internal structures are not well known by the converter or by the programmer. One example of such classes is `DateFormatZoneData`, which instantiates more than 1,000 literals at its initialization, some of which are shown in Table 5. By checking these values, it is considered that only few literals are used by the application, but the program conversion cannot be applied to the class since its structure is not known.

For such cases, this section proposes another technique to reduce the memory waste of unused literals. The technique, named *Lazy Body Creation*, is implemented inside the JVM. As explained in Section 2.1, a Java string is represented with two objects, a `String` (head) and a `char` array (body). The key idea of Lazy Body Creation is to *delay* the creation of the string body until the literal's value is actually used. For unused literals, their bodies are not created in the Java heap, so the memory waste by unused literals can be partially avoided. Since this technique is implemented in the JVM, it works for all literals unlike the program conversion.

The Java virtual machine with Lazy Body Creation separates the instantiation of string literals into the following two phases:

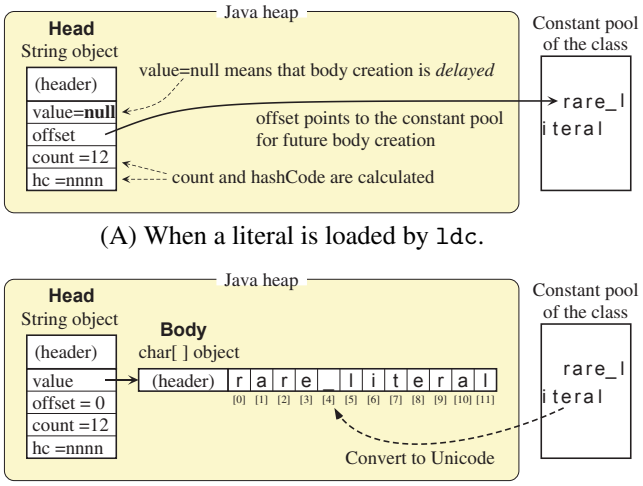

Figure 12. Behavior of Lazy Body Creation. The string body is not created if the value is not used.

A. When a string literal is being loaded by the `ldc` bytecode,

1. Calculate the hash code and length of the literal, and search for an already interned `String` object of the same value. If found, return it. Otherwise, perform the following steps.
2. Create a `String` object (head) with `null` in its `value` field, which means that the creation of the body is delayed.
3. Store the address of the UTF-8 data in the constant pool into the `offset` field.
4. Store the values calculated in Step 1 into the `count` and `hashCode` fields.
5. Perform the interning of the created `String` object, and return the result.

B. When a `String`'s `value` is being used,

1. If the `value` field is `null`, perform the following steps before using the `value` field.
2. Create a `char` array (body) with the length of `count` to hold the literal's value.
3. Fill in the `char` array with Unicode characters converted from the UTF-8 data pointed at by the `offset` field.
4. Store 0 into the `offset` field, then store the reference to the created `char` array into the `value` field.
5. The other fields, `count` and `hashCode`, need not be modified, since appropriate values have already been set.

Since the value of each literal exists as UTF-8 data in the constant pool of the class structure, it becomes possible to delay the creation of the string body by remembering the address of the UTF-8 data. The address is stored in the `offset` field rather than in the `value` field, to avoid confusing the garbage collector.

String is a core class provided with each JVM. As shown in Figure 2, it cannot be extended and all of its fields are private. Therefore, the check in Phase B only needs to be added to the methods in the String class and the JVM itself, where the value field (or offset field) is directly accessed. Note that some methods in the String class can be executed without creating the string body, since they do not access the value field. Examples of such methods are String.length(), hashCode(), and intern()[11].

Figure 12 illustrates the behavior of Lazy Body Creation. The upper figure (A) shows when a literal "rare_literal" is loaded by the ldc bytecode, where only a String object (head) is created in the Java heap. The lower figure (B) shows when the literal is actually used, where the char array (body) is created. Once the body is created, the literal can be used in the same way as regular String objects.

The Lazy Body Creation proposed here is applicable to all Java programs without modification. Although it does not prevent string heads from being created, we believe this technique is effective to reduce the memory waste by unused literals because 64–75% of such waste comes from the string bodies (char arrays) according to our investigations summarized in Table 1.

3.3.1 Discussion

In the implementation of Lazy Body Creation, Phase A (string head creation) is executed by only one thread for each literal, using the original mutual exclusion mechanism of literal initialization. In contrast, Phase B (string body creation) may be executed by multiple threads in parallel. Our algorithms prevent race conditions without requiring additional locks.

In Step B4, the value field is set *after*[12] the offset field is set to 0, which prevents any other thread from accessing the char array with an invalid offset index. With this ordering, it is possible that the offset has already become 0 in Step B3. That would mean that some other thread is executing Step B4, so the thread in Step B3 just can wait until the value field is set. Similar waiting loop is inserted for all the places that directly access the UTF-8 data through the offset field.

There may be a different kind of racing condition if two threads execute Step B4 in parallel. However, this does not cause any problem even if the offset or value fields are overwritten, because the same values (0 or the same-value char array) are stored by both threads. The previous char array will be collected as garbage when it is no longer referred to. It is also possible to avoid the overwriting using atomic compare-and-swap.

[11] The length and hash code are already calculated and stored in the count and hashCode fields, so can be returned by length() and hashCode(), respectively. Literals are interned at the time of the instantiation, so intern() can simply return the object if the value field is null.

[12] An appropriate memory barrier must be inserted to enforce the write order.

```
 1 class DuplicationBench {
 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 }
```

Figure 13. DuplicationBench, to test the string duplication. 1,000,000 String objects are created and compared, where dupRatio percent of them have the same value.

Additional care must be taken when StringGC is used with Lazy Body Creation. Since a new object usually cannot be created during GC, the StringGC code must be written so as to not trigger the string body instantiation. In our implementation, it directly accesses the UTF-8 data in the constant pool if a String object does not have a body.

4. Evaluation

This section evaluates the effectiveness of the three techniques in reducing the string memory waste described earlier. As the implementation target, we used IBM's latest production JVM, the J9 Java VM 5.0 for Linux. StringGC was implemented in its generational collector. Lazy Body Creation was implemented in its core virtual machine component, while also modifying codes in the String class and the JVM that directly use the String.value. BundleConverter was prepared as an independent Java program.

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. The Java heap size was set to 256 MB, where two 32 MB areas are used as the nurseries of the generational GC.

4.1 Micro-Benchmarks

First, micro-benchmarks were used to analyze the basic characteristics of the proposed techniques.

4.1.1 Micro-Benchmark for String Duplication

The program shown in Figure 13, called DuplicationBench, was used to analyze StringGC. 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 vari-

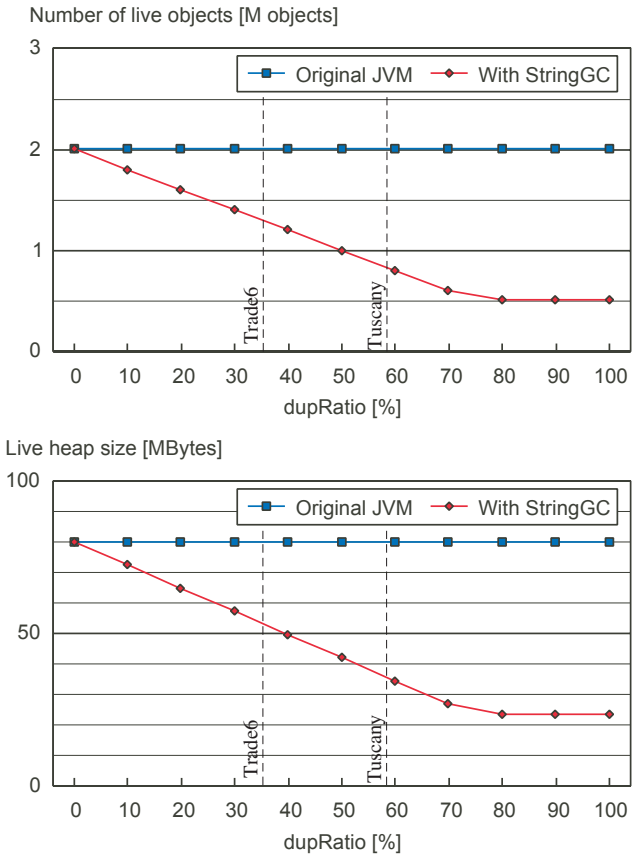

Figure 14. Heap analysis of DuplicationBench. StringGC worked effectively to reduce the live heap. The two dotted lines show the actual dup ratios in Trade6 and Tuscany.

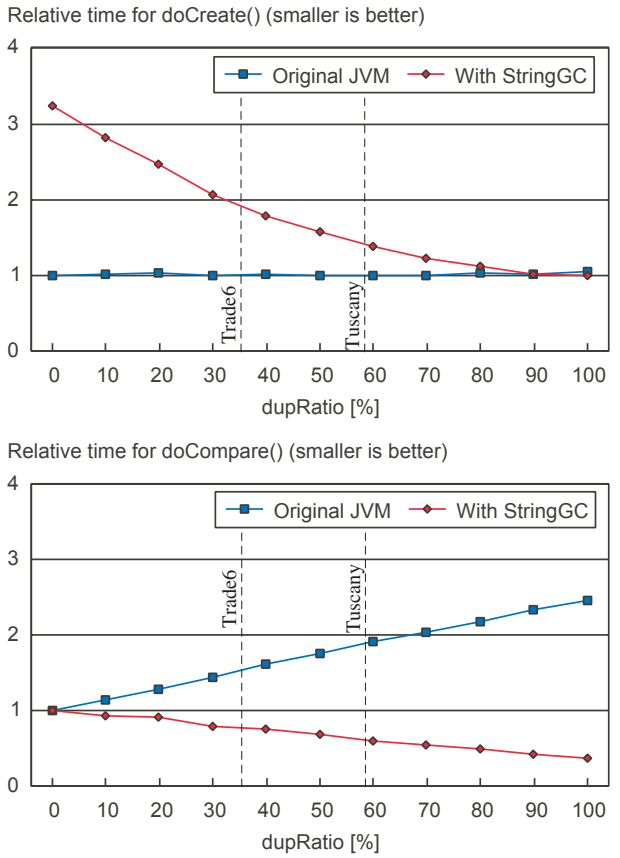

Figure 15. Relative times for `doCreate()` and `doCompare()` in DuplicationBench. With StringGC, the creation became slower but the comparison became faster.

ous `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.

The graphs in Figure 14 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. The two vertical dotted lines labeled "Trade6" and "Tuscany" show the actual duplication ratios measured in Section 2.3. Without the StringGC, the heaps were almost the 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, a `String` and a `char` array, as shown in Figure 2.

When the StringGC was enabled, both the number of live objects and their total size decreased as the `dupRatio` increased. The ratio of decrease matched 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 15 shows the times for executing the `doCreate()` method, normalized against 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. This is evidently due to the cost for maintaining the tenured-string table, where one-million 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 a slowdown of 3.2 times lower, even for this artificial string-intensive program where millions of string-related objects exist in the tenured space. Performance with more realistic programs will be measured in the next section.

As discussed in Section 3.1.1, 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 of DuplicationBench 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.

```
 1 import java.util.*;
 2 class LiteralsBench {
 3   public static void main(String[] args)
                                      throws Exception {
 4     Locale l = Locale.ENGLISH;
 5     ResourceBundle myRes =
          ResourceBundle.getBundle("MyResources1000", l);
 6     useMsg(myRes, 1000); // use "message1000"
 7     /*---- Measurement point A ----*/
 8     for (int i = 1001; i <= 1999; i++)
 9       useMsg(myRes, i);  // use "message1001",...
10     /*---- Measurement point B ----*/
11   }
12   volatile static char c;
13   static void useMsg(ResourceBundle res, int idx) {
14     String msg = res.getString("k" + idx);
15     c = msg.charAt(0);   // msg's value is used here
16 } }
17
18 public class MyResources1000_en
                             extends ListResourceBundle {
19   public Object[][] getContents() { return contents; }
20   private static final Object[][] contents = {
21     {"k1000", "message1000"}, // 2,000 literals here
 :                 :             //  (1,000 keys and
22     {"k1999", "message1999"}  //   1,000 messages)
23   };
24 }
```

Figure 16. LiteralsBench, to test unused message literals. It first uses just one message in the message class, then uses the remaining messages.

The lower graph of Figure 15 shows the normalized time for executing the `doCompare()` method for various `dupRatio` values. In the original JVM, the time becomes worse for higher `dupRatio`s, because the comparisons of `String` objects with the same value eventually needed to execute character-by-character comparison, while hash-code comparison was sufficient for different strings. However, when the StringGC was enabled, the comparison became faster for higher `dupRatio`s, because the same-value `String` objects had been unified. Consequently, for this micro-benchmark, comparing same-value `String` objects was 6.4 times faster than on the original JVM.

4.1.2 Micro-Benchmark for String Literals

Next, we analyzed BundleConverter and Lazy Body Creation using another micro-benchmark called LiteralsBench, shown in Figure 16. This program first loads a message class `MyResources1000_en`, which is a subclass of `ListResourceBundle` containing 1,000 messages, then uses the messages one-by-one.

Measurements were done at two points shown in the program: (A) after one message is used, and (B) after all of the messages are used. Using this micro-benchmark, the following three cases were tested:

1. Run LiteralsBench on the original JVM.
2. Run LiteralsBench on a JVM with Lazy Body Creation.
3. Run LiteralsBench with a new, BundleConverted `MyResources1000_en` class, on the original JVM.

Measured point	1 Original JVM	2 Lazy Body Creation	3 Bundle Converter
A `String`	2,000	2,000	2
`char[]`	2,000	2	2
B `String`	2,000	2,000	2,000
`char[]`	2,000	2,000	2,000

Table 6. Number of objects created from the literals of `MyResources1000_en` in LiteralsBench. Both of the proposed techniques could reduce the instantiation of unused literals.

Measured section	1 Original JVM	2 Lazy Body Creation	3 Bundle Converter
Entry to A	100	89	158
(noverify)	(93)	(82)	(61)
A to B	50	66	120

Table 7. Relative execution times of LiteralsBench. The BundleConverted class is slower than the original class.

BundleConverter and Lazy Body Creation were tested separately here, to understand the behavior of each optimization.

Table 6 shows the number of `String` and `char[]` objects in the Java heap created from the 2,000 literals of the `MyResources1000_en` class. These results are just as expected. At the measurement point A, all literals were fully instantiated in the original JVM, while the creation of `char` arrays were suppressed in Lazy Body Creation. If the message class is converted by BundleConverter, only two literals (`"k1000"` and `"message1000"`) were instantiated at this point. At Point B, there is no difference among the three test cases, since all messages had been used by this point.

Next, Table 7 shows the relative execution time of LiteralsBench in each test case. The times from the entrance of `main()` to Points A and B were measured and normalized. The JIT compiler was turned off in this measurement.

Lazy Body Creation arrived at Point A faster than the original JVM, because it did not create string bodies for most literals. Instead, the time from A to B became longer, because the delayed body creations were done here. The total time to Point B was almost the same as the original JVM.

For the program converted by BundleConverter, we expected that it would reach Point A much faster because fewer objects need to be created. However, the result was 1.6 times slower than original JVM. This is because bytecode verification took longer time for the converted class, since it contains a long `handleGetObject()` method as shown in Figure 11. The "(noverify)" row in the table shows the times if the verification is turned off, in which the time to Point A became much faster than the original JVM. As for the total time to arrive at Point B, it was slower than for the original class. This is because many `if`-statements needed to be executed in the converted class.

4.2 Macro-Benchmarks

Next, macro-benchmarks were run using more realistic Java programs.

4.2.1 Execution Performance

First, the execution performance was measured by using the SPECjvm98 [24] 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 best time from repeated runs.

Figure 17 shows the results, where the execution time relative to the original JVM is shown for the StringGC JVM and the Lazy Body Creation JVM. Since `ListResourceBundle` class is not used in SPECjvm98, BundleConverter was not tested here.

Surprisingly, for StringGC, we observed a time reduction of about 30% in `db`. The reason is that many string-compare operations in this benchmark was accelerated by the unification of `String` objects, as shown in the lower graph of Figure 15. In the measured environment, about 80,000 `String` objects were tenured during one execution of the benchmark, and about 50,000 of them were unified to other `String` objects. `String.equals()` was executed about 1,460,000 times. For Lazy Body Creation, we could not observe any significant differences.

We also measured the DaCapo [6] benchmarks for the three JVMs, but did not find any significant differences. Actually, the deviations among the runs were much larger than the differences among the JVMs.

To summarize, no performance degradations due to StringGC or Lazy Body Creation were observed in these macro-benchmarks.

4.2.2 Heap Reductions for Client-Type Applications

Next, we measured the heap reductions for these macro-benchmarks. Our original motivation was to reduce the memory consumption of large enterprise applications. However, our string reduction techniques also work effectively for most of these client-type applications.

Figure 18 shows the analyses of the live heaps of the SPECjvm98 and DaCapo benchmarks. Since these programs have no "stable state" for consistent analyses among multiple runs, we analyzed each program when it reached 70% of its total object allocations. For example, a total of 222 MB of objects are allocated to execute `antlr` in DaCapo, so we analyzed its live heap when 155 MB (222×0.7) of the objects had been allocated. Note that the analyzed live heap was much smaller than the allocated size, because many objects had already died at the measurement point.

For each benchmark in the figure, the upper bar shows the live heap in the original JVM, and the lower bar shows the live heap when StringGC and Lazy Body Creation are enabled. Since the actual sizes of the live heaps vary among the benchmarks, they are normalized so the size in the original JVM is 100%. The memory for `String` objects and the `char` arrays for holding their actual values are separately shown in the bars. This shows that these string-related objects occupy large portions of the live heap in most of the benchmarks.

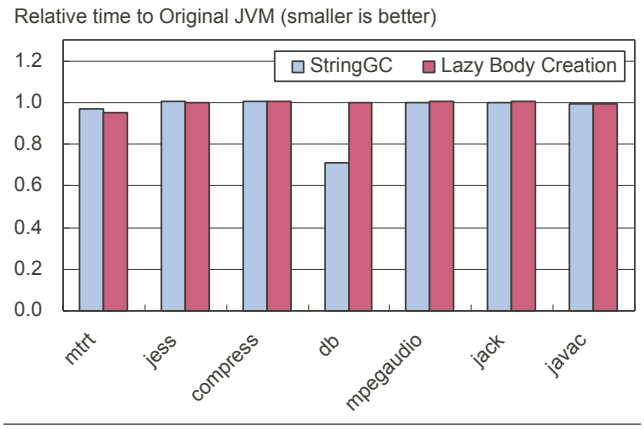

Figure 17. Relative times to the original JVM in the SPECjvm98 benchmarks. There are no large differences, except `db` was accelerated by StringGC.

The number under each benchmark name on the left side is the reduction ratio of the live heap for that benchmark. The live heaps were reduced in *all* of these benchmarks. In particular, more than 10% reductions were observed for seven benchmarks: `db`, `mpegaudio`, `jack`, `bloat`, `chart`, `luindex`, and `lusearch`. The geometric mean indicates that our string reduction techniques can reduce the live heaps by 11.6% for these client-type applications.

In Figure 18, we only showed the live heap at the "70% allocation point" of each benchmark. However, we confirmed that each benchmark basically exposed similar demography in the live heap for all of the 10%, 20%, ..., 90% allocation points. For example, Figure 19 shows the comprehensive analysis for `eclipse` in DaCapo. The live heaps in the original JVM (left bar) and our modified JVM (right bar) are compared for each of the 10% to 90% allocation points. Although the size of the live heap changes during the execution, its 40–50% region is always occupied by string-related objects in this benchmark. As shown in the dotted lines, our string reduction techniques were able to reduce the live heap size by 8–13%.

4.2.3 Heap Reductions for Real Applications

Finally, we measured the effects for heap reduction in large-scale enterprise Java applications, which was the original concern of this research. The same programs described in Section 2.3, Trade6 and Tuscany, were used for the evaluation.

Table 8 shows the results of live heap analysis as reported in Table 1, when incrementally enabling StringGC, BundleConverter, and Lazy Body Creation. BundleConverter was

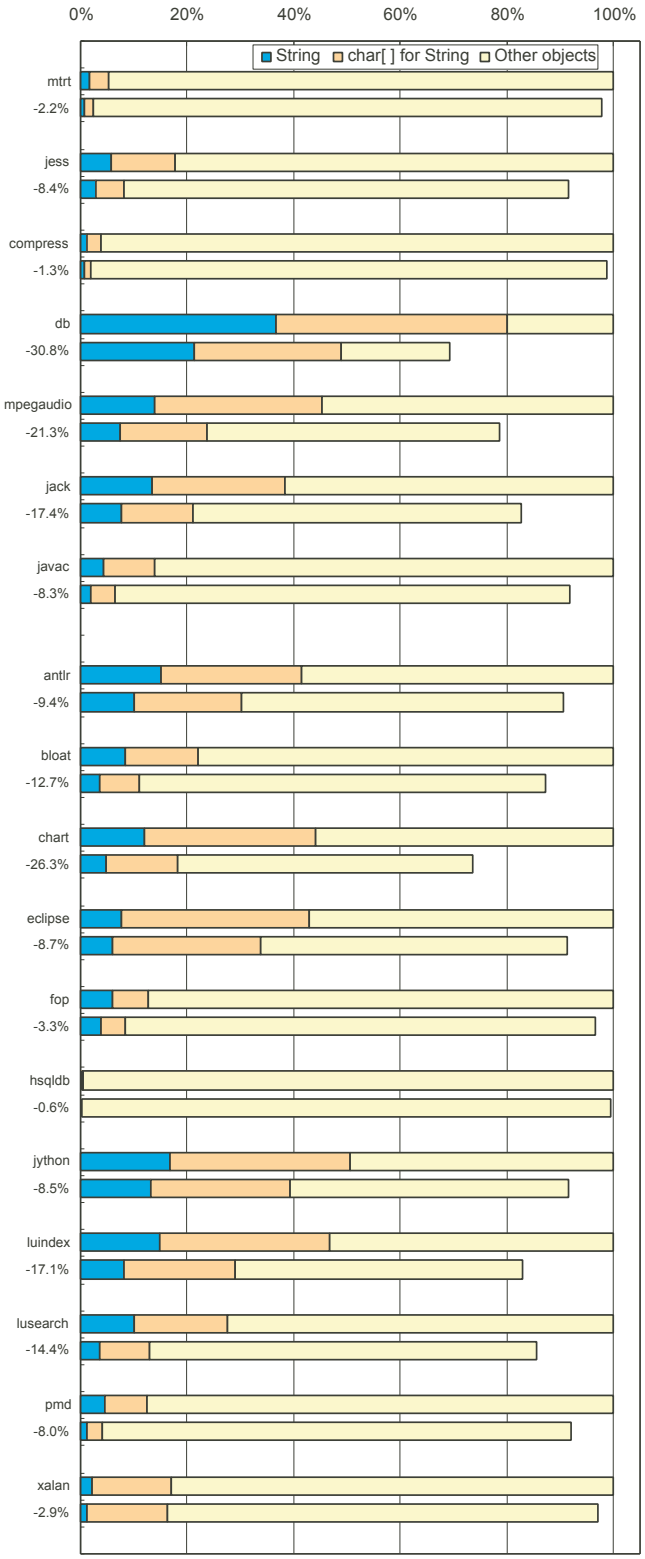

Figure 18. The live heaps of the SPECjvm98 and DaCapo benchmarks at the 70% allocation point. For each pair of bars, the upper bar shows the original JVM and the lower bar shows the modified JVM with StringGC and Lazy Body Creation, which reduced the live heaps by 11.6% on average.

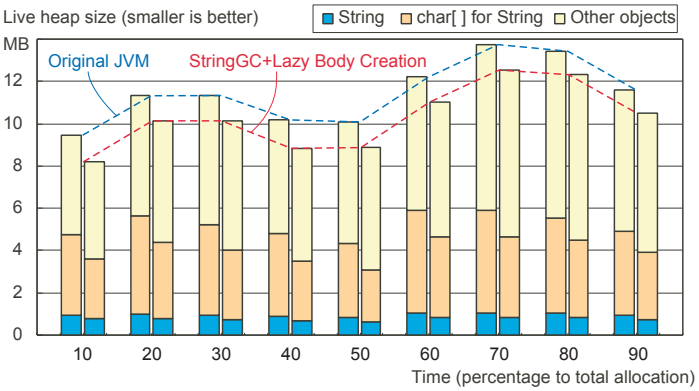

Figure 19. Live heap of `eclipse` in DaCapo at the 10–90% allocation points. Proposed techniques consistently reduced the live heap size by 8–13%.

Metrics	Trade6 count	Trade6 size	Tuscany count	Tuscany size
Original JVM				
Total live objects	582,012	31,346 KB	281,210	15,193 KB
- `String` objects	103,370	2,894 KB	57,773	1,617 KB
- `char[]` for `String`	92,996	8,967 KB	47,991	3,649 KB
+ StringGC				
Total live objects	520,387	28,506 KB	227,684	13,007 KB
- `String` objects	69,162	1,937 KB	26,527	743 KB
- `char[]` for `String`	67,236	7,157 KB	25,129	2,186 KB
+ BundleConverter				
Total live objects	487,363	26,643 KB	—	—
- `String` objects	59,574	1,668 KB	—	—
- `char[]` for `String`	57,684	5,962 KB	—	—
+ Lazy Body Creation				
Total live objects	472,380	25,695 KB	220,272	12,544 KB
- `String` objects	59,693	1,671 KB	26,093	731 KB
- `char[]` for `String`	43,125	5,055 KB	18,743	1,887 KB
Total eliminated	109,632	5,651 KB	60,938	2,650 KB
- ratio to the waste	96.2%	**93.8%**	86.2%	**87.0%**
- ratio to the orig. heap	18.8%	**18.0%**	21.7%	**17.4%**

Table 8. Cumulative results for heap memory reductions in real applications. From 87% to 94% of the string memory inefficiencies have been removed.

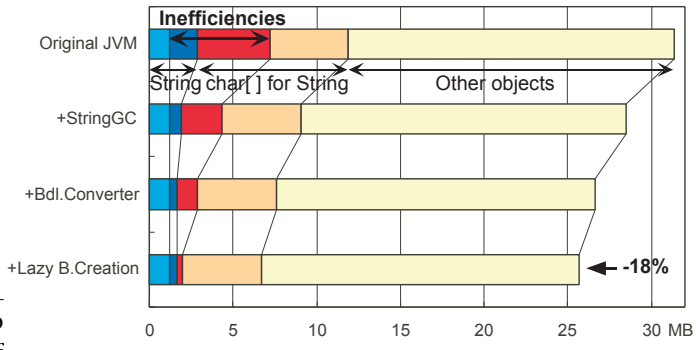

Figure 20. Live heap reduction for Trade6 by the proposed methods. The inefficiencies were eliminated in each step, and the live heap size was reduced by 18%.

Trade6				
#chars		#strs		Class name
706	(-99.2%)	16	(-99.0%)	`security_en`
0	(-100%)	0	(-100%)	`security`
205	(-99.7%)	6	(-99.6%)	`engineMessages_en`
0	(-100%)	0	(-100%)	`engineMessages`
76	(-99.8%)	2	(-99.6%)	`CWSICMessages_en`
0	(-100%)	0	(-100%)	`CWSICMessages`
294	(-98.9%)	4	(-99.0%)	`CWSIPMessages_en`
0	(-100%)	0	(-100%)	`CWSIPMessages`
177	(-99.3%)	4	(-99.1%)	`HAManagerMessages_en`
0	(-100%)	0	(-100%)	`HAManagerMessages`
:		:		:
1,109,686		44,232		TOTAL
(-49.1%)		(-31.8%)		- reduction due to the conversion

Table 9. Literals instantiated from the converted classes. Compared to the result in Table 4, BundleConverter successfully suppressed the unnecessary literal instantiations.

not used for Tuscany, because this application was not internationalized and was not using `ListResourceBundle`.

When StringGC was enabled, 33.1% of the `String` objects and 27.7% of the `char` arrays for strings were removed in Trade6. Due to this, 9.1% of the live heap area in the original JVM was eliminated. StringGC was more effective for Tuscany, where 14.4% of the heap usage was eliminated. Comparing this result with Table 1, the StringGC prototype removed about 90% of the string memory waste from duplication.

When BundleConverter was used, the number of string literals in Trade6 was reduced by 30.0%, from 35,498 to 24,882. Interestingly, non-string-related objects were also reduced. This is because the conversion shown in Figure 11 eliminates the use of `HashMap` in `ListResourceBundle`. Due to these effects, BundleConverter could further reduce the live heap size of Trade6 by 1.9 MB, which was 5.9% of the original live heap.

Table 9 shows the number of characters and strings instantiated from the converted message classes, which originally had the top 10 literals as shown in Table 4. The numbers in parentheses are reduction ratios from original classes. For example, the instantiated literals of a class `security_en` were reduced by 99.0%, from 1,628 to 16. Since there is a key string for each message, this means that only 8 messages of this class were actually used by the Trade6 test case.

Returning to Table 8, when Lazy Body Creation was added, `char` arrays were not created for 56.3% of the remaining 24,882 literals in Trade6. This shows that there still exist classes that instantiate unused literals besides `ListResourceBundle`. Lazy Body Creation further reduced the `char` arrays by 0.9 MB, which was 3.0% of the original live heap. For Tuscany, Lazy Body Creation also saved 3.0% of the live heap.

Figure 20 illustrates the reduction of the live heap for Trade6. This clearly shows that the inefficiencies were reduced in each step. The remaining inefficiencies we could not remove were: (1) duplications of `String` objects in the nursery space, (2) fragmentations in `char` arrays, and (3) string heads of unused literals which cannot be BundleConverted, the total of which was less than 3% of the reduced new live heap.

Summarizing these results, the combination of the three proposed techniques eliminated 87–94% of the string memory inefficiencies measured in Table 1, achieving about 18% reduction of the live heap usage.

5. Related Work

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

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, this 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 is not practical, since many temporary `String`s will also be interned. Our UNITE StringGC in Section 3.1 is one solution to this problem, by interning only the long-lived `String` objects.

It is common to eliminate the duplicates of string literals. The Java Language Specification [10] specifies that all string literals and string-value constant expressions must be interned. In the Unix operating system, a tool named "`xstr(1)`" is provided to extract strings from C programs and unify them.

Dieckmann and Hölzle [8] 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 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 [20] presented Object Equality Profiling (OEP) to discover opportunities for replacing a set of equivalent object instances with a single representative object. While we focus on duplication 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 amounts of object equivalence in real Java programs.

Automatic optimizations exploiting object equivalence are beyond the scope of their paper. However, they performed a case study for two SPECjvm98 benchmarks, `db` and `mtrt`, and manually optimized them to reduce the space used by live objects. Their tool showed that in `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 optimizations.

Hash-consing [9, 11] 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 if 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 the 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.

Vaziri, Tip, Fink, and Dolby [27] proposed a way to formally declare *object identity* using relation types. This model can be used for hash-consing of general objects. However, the classes must be declared using their constructs, which are not compatible with current Java syntax. In the future, it may become possible to combine their model with our StringGC to further reduce the memory footprint.

Mitchell and Sevitsky [22] investigated the causes of memory bloat in various Java programs, and showed that memory *health* is limited by the structural overhead of the design of each data type. Some of the string memory inefficiencies we showed in this paper can be regarded as such overhead that limits the memory health.

Lazy evaluation is a well-known technique, especially in functional programming languages such as Haskell [16], where computations are delayed until their results are actually needed. Our two techniques to reduce unused literals can be considered as applying the concept of lazy evaluation to a specific mechanism of the language implementation.

One reason of many unused literals is that there is no way for a Java program to directly declare a static hash map. Therefore, all literals are instantiated at the initialization time to construct a `HashMap` dynamically. Adding a new language construct may be another solution to reduce the unused literals.

Memory leaks are one of the largest causes of footprint expansion even in garbage-collected languages such as Java. However, they are basically a problem within each application, so research has been focused on tools to detect memory leaks effectively, such as LeakBot [21] and Cork [17]. In contrast, the string memory inefficiencies we revealed come from the design and implementation of the Java language itself. Therefore, we could create several techniques to reduce these inefficiencies which are widely applicable to almost all Java applications, as shown in Figures 18–20.

6. Conclusion

This paper described proposals, implementations, and evaluations of three techniques to reduce the string memory consumption of Java programs.

Recent Java applications handle more and more string data, such as for processing XML and accessing databases. For example, in a production J2EE application server, about 40% of the live heap area is used for string data. Through exhaustive investigation of this string data, we identified two types of *string memory inefficiencies* in Java. First, there are many duplicated strings in the heap. Second, there are many string literals whose values are not actually used by the program. These inefficiencies exist as live objects, so they cannot be addressed by existing GC techniques. Quantitative analysis of real Java applications revealed that more than 50% of the string data is wasted, occupying about 20% of the live heap area.

To remove the inefficiencies, we developed three new techniques. *StringGC* is a new optimization at GC time to eliminate duplicated strings. *BundleConverter* is a program converter that converts message classes in an application to a form that does not instantiate unused message literals. *Lazy Body Creation* is a new JVM mechanism to delay part of the creation of literals until they are actually used.

We implemented three techniques for a production Java virtual machine, and examined large enterprise applications. Results show that our techniques eliminated about 90% of the string memory inefficiencies, and reduced the size of the live heap by 18%. We also showed that the proposed techniques can reduce the live heap size of standard Java benchmarks by 11.6% on average, without any noticeable performance degradation.

Acknowledgments

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

We also thank Andrew Low and Thomas Gissel of IBM Software Group for their various suggestions to our memory investigation work.

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://tuscany.apache.org/`

[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. M. Blackburn, et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '06)*, pp. 169–190, 2006.

[7] P. Chan, R. Lee, and D. Kramer. *The Java Class Libraries, Second Edition*, Addison Wesley, 1998.

[8] S. Dieckmann and U. Hölzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmark. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*, pp. 92–115, 1999.

[9] A. P. Ershov. On Programming of Arithmetic Operations. In *Communications of the ACM*, Vol. 1, No. 8, pp. 3–9, 1958.

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

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

[12] 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. In *Proceedings of the 3rd USENIX Virtual Machine Research and Technology Symposium (VM '04)*, pp. 151–162, 2004.

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

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

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

[16] S. P. Jones. *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press, 2003.

[17] M. Jump and K. S. McKinley. Cork: Dynamic Memory Leak Detection for Garbage-Collected Languages. In *Proceedings of the 34th ACM Symposium on Principles of Programming Languages (POPL '07)*, pp. 31–38, 2007.

[18] K. Kawachiya, K. Ogata, and T. Onodera. A Quantitative Analysis of Space Waste from Java Strings and its Elimination at Garbage Collection Time. Research Report RT0750, IBM Tokyo Research Laboratory, 2007.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*, Addison Wesley, 1999.

[20] D. Marinov and R. O'Callahan. Object equality profiling. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, pp. 313–325, 2003.

[21] N. Mitchell and G. Sevitsky. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP '03)*, pp. 351–377, 2003.

[22] N. Mitchell and G. Sevitsky. The Causes of Bloat, The Limits of Health. In *Proceedings of the 22nd ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '07)*, pp. 245–260, 2007.

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

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

[25] Sun Microsystems. Java2 Platform Standard Edition 5.0 API Specification: java.lang.String. `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html`

[26] The Unicode Consortium. *The Unicode Standard, Version 5.0*, Addison Wesley, 2006.

[27] M. Vaziri, F. Tip, S. Fink, and J. Dolby. Declarative Object Identity Using Relation Types. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP '07)*, pp. 54–78, 2007.