July 17, 2009
RT0867
Computer Science 5 pages

Research Report

FSGC: String Garbage Collection on a Flat Java Heap

Kiyokuni Kawachiya 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

FSGC: String Garbage Collection on a Flat Java Heap

Kiyokuni Kawachiya Tamiya Onodera

IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato, Kanagawa 242-8502, Japan
<kawatiya@jp.ibm.com>

Abstract

This report proposes a technique to reduce the memory footprint of Java programs by eliminating duplicate strings in the heap. Unlike an existing technique based on generational GC, the proposed Flat-heap StringGC (FSGC) works in a flat-heap environment, where the age of objects is not maintained by the GC. We use special counters for String objects to detect and unify long-lived strings. The FSGC can find duplicate strings while retaining the no-write-barriers advantage of the flat-heap GC.

1. Introduction

In Java applications, string-related objects (String and char[ ]) occupy much of the live heap. There are generally many duplicate strings in the heap [1]. In Java, String objects are “immutable” and their values cannot be modified once created. Thus, memory consumption can be reduced by unifying the strings that are the same.[1]

However, if all of the strings are checked for the unification, the overhead becomes large because of the numerous temporary strings. To solve this problem, we created the UNITE (UNIfication at TEnuring) StringGC method [1]. In this method, only *long-lived* strings are unified, and only when a candidate String object is moved to the tenured space by the generational GC (garbage collector). This mechanism works well and can decrease the live heap by 9-14% in real applications such as WebSphere Application Server (WAS) [2].

One problem with the UNITE StringGC is that it needs a generational GC framework to detect the long-lived strings. Therefore, it cannot be used in more general “flat-heap” environments. In this report, we propose a method for StringGC in a flat-heap environment. Like the UNITE StringGC, the proposed Flat-heap StringGC (FSGC) works as an extension of the existing GC in a Java Virtual Machine (Java VM), so Java applications need not be modified at all.

In the FSGC, each String object includes a counter incremented when the String object survives a GC. This counter can be implemented as a hidden field of the String object, without using the pre-

[1] Strictly speaking, the String objects themselves should not be unified. Only the char arrays, which hold the string's value, should be unified. See Section 3.1.1 of [1] for details.

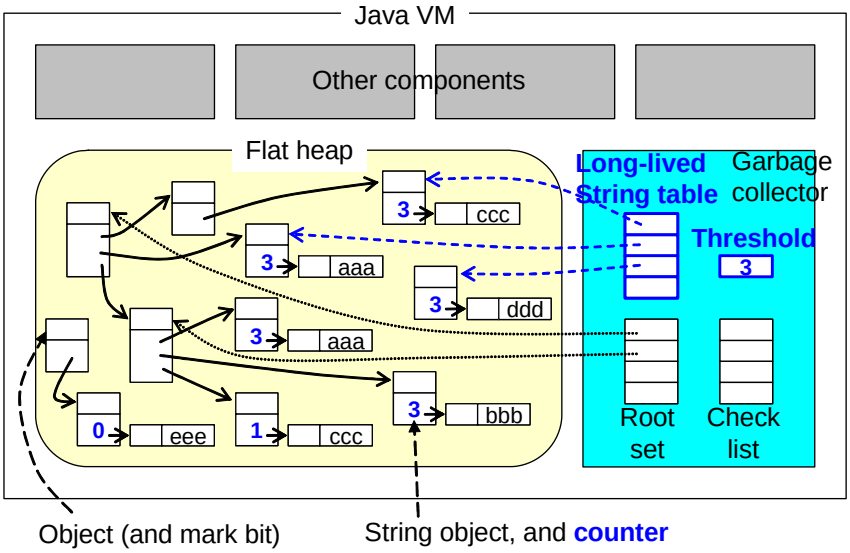

Figure 1: Configuration for the FSGC

cious object-header region. When the counter reaches a threshold value, the String object is checked for possible unification with another String object having the same value. Through this mechanism, duplicate strings can be eliminated without degrading performance. Since the counter is not used for managing object generation, heavy write barriers like those of generational GC are unnecessary. Therefore, the FSGC can retain the main advantage of a flat heap, no overhead in the normal mutator execution [3].

2. Flat-heap StringGC (FSGC)

This section describes the components of and the algorithm for the FSGC.

2.1 Components

The FSGC can be implemented by extending a conventional flat-heap garbage collector in a Java VM. Figure 1 shows the most typical configuration for an FSGC, and the blue labels are for the new items. In a suitable Java VM, the objects are held in a “heap”, which is flat (single) and not divided by the objects' generations. For each object in the heap, a “mark bit” is prepared for the GC, although these bits are not explicitly shown in the figure because they are usually allocated outside of the heap. In addition, a counter is added to each String object to track how many times the object has survived the GC.

The GC periodically collects and disposes of the dead objects that are not reachable from the “root set”. While checking the reachability, the GC uses a “check list” as a work area. If a String object is reachable, the GC increments its counter. When the counter reaches a threshold value, the String object is registered in the “long-lived string table”, which was also added to the FSGC. When a

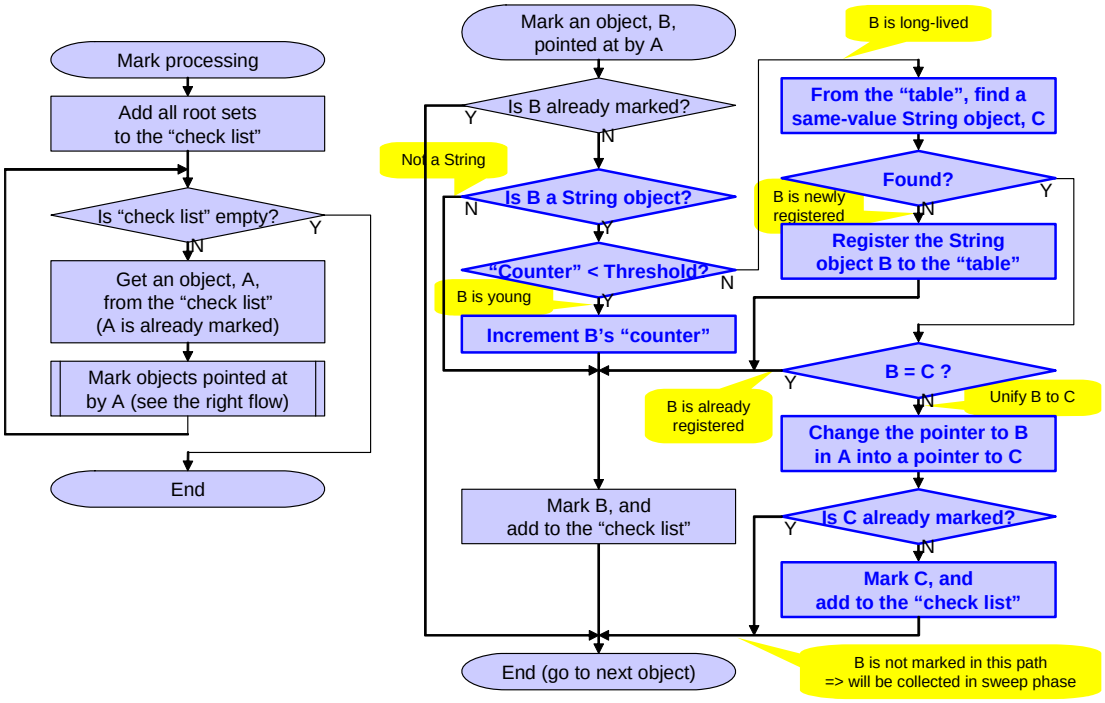

Figure 2: Mark algorithm of the FSGC

string becomes long-lived, the string's value is checked against the strings already registered in the table. If a String with the same value is found, then the String object being registered is unified with the String object already in the table.

2.1 Algorithm

The garbage collection for a flat heap is usually done with a mark-and-sweep method, which consists of a mark phase followed by a sweep phase [3]. In the mark phase, objects reachable from the root set are marked, and the unmarked objects are collected in the sweep phase.

Figure 2 shows a plausible algorithm for the mark phase of the FSGC, where thick blue frames mark the processes added to the conventional GC. Here, only when a String object is marked does the FSGC check and increment its counter. When the counter reaches the threshold value, the String object is regarded as a long-lived string and checked for unification. If a string with the same value is already registered in the long-lived string table, then the strings are unified. If no match is found, the String object is registered. When a String object is unified, its mark bit is not set, so it will be collected in the following sweep phase.

Figure 3 is a plausible algorithm for the sweep phase, which is basically the same as the conventional flat-heap GC. However, when a String object is being collected, the algorithm checks to see if it is registered in the long-lived string table. If so, the string is removed from the table.

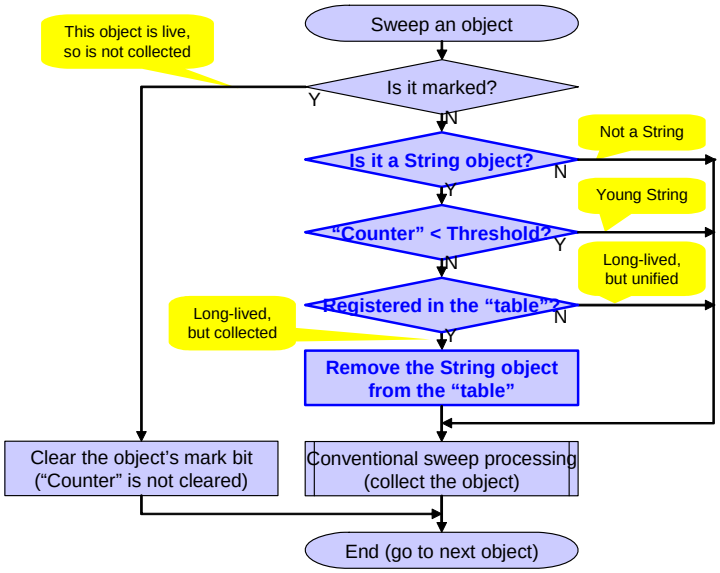

Figure 3: Sweep algorithm of the FSGC

3. Summary and Discussion

This report described a technique to remove the duplicate strings from a flat Java heap. Unlike our original UNITE StringGC [1], which used a generational GC, the proposed Flat-heap StringGC (FSGC) can unify long-lived strings in flat-heap environments. Since no unification is done for short-lived strings, the performance degradation is minimized.

The proposed FSGC uses a counter in each String object to check whether it is long-lived. It resembles the age counter in a generational GC, but is used only for String objects and without any generation management in the GC. Therefore, the FSGC has no heavy "write barrier" operations when old generation objects refer to newer objects. The flat-heap GC advantage of "no overhead in the normal mutator execution" is preserved.

Several variations are possible when implementing the FSGC. If the base flat-heap GC has a compaction phase, then the mark phase only has to increment the counters, and string unification can be handled during compaction.

It would also be possible to set the threshold to 0, which eliminates the need for actual counters in the String objects. In this variation, any string that survives a GC is treated as long-lived and unified. Even in this variation, most temporary strings will never be checked for the unification, because they rarely survive even one GC.

Another variation would be to dynamically change the threshold value. If memory becomes scarce, then a smaller value should be used to do the unifications more eagerly. If performance is more important, then a larger value should be used to reduce the unification overhead.

References

[1] K. Kawachiya, K. Ogata, and T. Onodera. Analysis and Reduction of Memory Inefficiencies in Java Strings. In *Proceedings of the 23rd Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '08)*, pp. 385-401, 2008.

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

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