Sizing Multi-Space in Heap for Application Isolation

Kewei Sun
IBM China Research Laboratory
Beijing, P.R.China
86-10-58748485
sunkewei@cn.ibm.com

Ying Li
IBM China Research Laboratory
Beijing, P.R.China
86-10-58748054
lying@cn.ibm.com

Matt Hogstrom
IBM Software Group
Durham NC
1-919-656-0564
hogstrom@us.ibm.com

Ying Chen
IBM China Research Laboratory
Beijing, P.R.China
86-10-58748010
yingch@cn.ibm.com

Abstract

In this paper, we present a sizing algorithm using resonant model based on a proposed novel heap structure – multiple spaces in heap. Experiments using the algorithm and selected GC method show that, in average, the performance overhead from managing multi-spaces in heap can be reduced from 8.38% to 4.25% when CPU utilization of server is 40% and from 42.42% to 3.52% when CPU utilization of the server is 70%.

Categories and Subject Descriptors D.4.8 [**Operating Systems**]: Performance – *modeling and prediction.*

General Terms Reliability, Performance, Experimentation, Algorithms

Keywords Java heap, multiple spaces, isolation, resonant model

1. Motivation: Building Multiple Spaces in Heap for Isolating Application

Typically, the Java middleware, like web server, or application server, runs in a single Java virtual machine (JVM) instance continuously. Given this single JVM model and the existing JVM technology, the heap is shared and objects are allocated and reclaimed randomly among middleware and applications. The sharing heap strategy brings big negative impact to the system that the bad logic (i.e., memory leak) or malicious intension in one application may cause the degraded performance or even crash of the whole system due to the heap memory competition or deprivation among middleware and applications

We built a revised Java heap for Java middleware-based applications by leveraging memory spaces in IBM JVM. To isolate one application's memory leaking behavior to prevent middleware and other applications from being affected, traditional Java heap is partitioned into multiple memory spaces, each of which is associated with one application and bound by a limit size Capped by the guaranteed heap size while allocating objects in run time, application's memory leaking or malicious intension can not interfere with the others in terms of heap consumption, and most importantly, the whole system including the middleware can continue to operation with maintained performance, in face of memory leaks, during runtime. The experiment result illustrates the isolation effectiveness of the prototype based on the revised Java heap.

One of the key challenges of multi-spaces Java heap is determining the appropriate size of memory space which has a big impact on the performance of the application binding to the memory space. A memory space that barely meets the application's minimum requirements results in excessive garbage collection overhead. While a memory space that always exceeds the application's maximum requirements leads to low utilization of the heap. This paper presents a memory space sizing algorithm that can select sizes of the memory spaces by both accommodating applications' requirement and minimizing the overhead of garbage collection in whole heap. To achieve the goal, the algorithm employs an analytic model – resonate model that characterizes the relationship between memory space size and GC frequency. The experiments results show that, on average, based on the selected GC method and the memory space sizing algorithm, the performance overhead caused by managing multi-spaces in heap can be reduced from 8.38% to 4.25% when CPU utilization of server is 40%, and from 42.42% to 3.52% when CPU utilization of the server is about 70%.

Ying Li is the corresponding author

OOPSLA'06 October 22-26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

2. Selecting Garbage Collection Method for Multiple Spaces in Heap

Firstly, considering the major contribution of GC to performance overhead on Java system, we investigated two typical garbage collection (GC) methods supported by IBM JVM, global GC, and scavenger GC, to evaluate their applicability to multiple memory spaces in heap. Under the environment of multiple memory spaces in a single Java heap, both global and scavenger GC lead to scan the entire heap to identify living objects. The different is that, after the heap is scanned, global GC reclaims the memory from the entire heap, while scavenger GC usually limit the memory reclamation action within the boundary of the memory space that triggered the GC activity. Under such circumstance, scavenger GC reclaims less memory than global GC, and is prone to cause higher GC frequency. Correspondingly, compared with scavenger GC, it seems that global GC is more suitable for multiple spaces in heap. Our experiment confirms the above analysis. We took Jetty 5.1.4 as the experimental Java middleware, and the servlets of the Web application benchmark, TPC-W, as the testing applications. TPC-W and Jetty5.1.4 are run on IBM JVM. Jetty itself, as a middleware, has been assigned with a dedicated memory space, and then each servlet of TPC-W is bound to its dedicated memory space. Since this experiment is just to evaluate the applicability of GC methods, we selected the same size for each memory space of TPC-W servlets. The GC overhead ratio, the portion of time that GC activity occupied in the whole application execution time is defined as the metric to compare the contribution to performance overhead of these two GC methods. The experiment result shows that, on average, GC overhead ratio by global GC is 1%, while by scavenger GC, it is up to 23%. Based on the above empirical analysis and

experimental results, we select global GC supported by IBM JVM as the GC method for multiple spaces in heap.

3. Resonant Model – Sizing Multiple Spaces in Heap

After selecting the suitable GC method, the second challenge is to select correct size of each memory space in heap since the size has serious impact on performance of application that use this memory space. The objective is to minimize the performance overhead caused by managing multiple spaces in heap. We build a analytical model, called Resonant Model, to characteristic the relationship between memory space size and GC frequency. And based on this, the memory space sizing algorithm can predict and select correct size for each memory space in heap to achieve minimized performance overhead.

To build analytical model for sizing memory spaces it is important firstly to understand the relationship between GC frequency and size. Typically, GC contains mark and sweep phases. Our empirical evaluation shows that the mark phase occupies the dominating portion of GC time. Therefore, given a certain heap size, we can expect that the time consumed by each GC is almost the same. As a result, GC overhead is in proportion with the GC frequency. The experiment results of TPC-W on Jetty and IBM JVM illustrated in Figure 1 strongly supports the assumption.

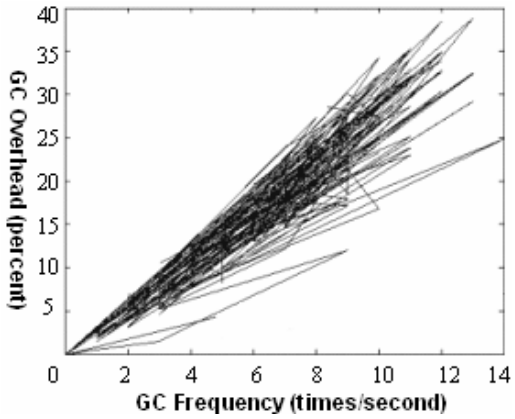

Figure 1: Relationship between GC overhead and frequency

In the partitioned heap, global GC performs global mark and global sweep, while it is triggers by each memory space independently. As a result, there might be a lot of memory spaces, who are compelled to perform garbage collection when they are far from running out of memory. That is to say, when the Java heap is partitioned into multiple memory spaces and global GC is used, the GC frequency of the entire Java heap follows the memory space with highest GC frequency.

When the application and its typical workload are determined, the most efficient way to reduce the GC frequency is by assigning more memory, and vice versa. However, assigning more memory for one memory space means to reduce the total memory shared by other memory spaces. This will result in higher GC frequency of others. To achieve the lowest GC overhead, we need to make the GC frequency in every memory almost the same. That is, to make their GC pace "resonantly". This is the essence idea of the resonant model.

We define the workload as the requests to the application arrived and processed by the application. GC frequency of a memory space is determined by its memory space size and workload. Formula (I) represents the relationship among memory space size, workload and GC frequency. The aim of the *Resonant Algorithm* is to achieve the lowest GC frequency of the heap. Therefore, we need to make full use of memory resources in the heap by allocating them to different memory spaces. As a result, heap size would be the summary of the size of each memory space (Formula II). In order to achieve the lowest GC frequency, the resonant condition must be satisfied. That is, for any memory spaces, the GC frequency should be the same (Formula III). Related function is illustrated as follow:

$$\begin{cases} f_{MS_i}(MSsize_i, workload) = GCfrequency_i & \text{I} \\ \sum_i g^{-1}_{MS_i}(GCfrequency_i, workload) = HeapSize & \text{II} \\ (\forall i, j)(GCfrequency_i = GCfrequency_j) & \text{III} \end{cases}$$

When the resonant condition is satisfied, the lowest GC frequency and overhead would be achieved by assigning a suitable memory space size for each memory space.

4. Experiment Result and Future Work Plan

In order to evaluate the effect of the resonant model, we have done preliminary experiments on TPC-W benchmark on Jetty and Trade6 benchmark with WAS6.1. Experiment results are similar. Taking Trade6 on WAS6.1 as example, each servlet of Trade6 is treated as an application and assigned a memory space in the heap. Firstly, we select average equal size to each memory space, which results in 30/second GC frequency, and 0.96% GC overhead ratio. After applying resonant model and selecting correct size of each memory space, the GC frequency reduced to 14/second, and GC overhead becomes to 0.59%, reduced by 39%. Correspondingly, the performance overhead caused by managing multi-spaces in heap can be reduced from 8.38% to 4.25% when CPU utilization of server is 40% and from 42.42% to 3.52% when CPU utilization of the server is about 70%.

Our future work would be focused on improve the resonant model algorithm, which will make a rapid response when the workload of the application switches. Leverage the memory space resizing mechanism in the revised Java heap and make sure the stable performance of the middleware-based applications.

References

[1] S. Borman, et al. A Serially Reusable Java Virtual Machine Implementation for High Volume, Highly Reliable, Transaction Processing, In *IBM Technical Report TR 29.3406,* IBM Hursley, UK.

[2] G Czajkowski. Application Isolation in the Java Virtual Machine. In ACM SIGPLAN Proceedings of the 15th conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota, Oct, 2000

[3] D Menasce. TPC-W A Benchmark for E-Commerce. IEEE Internet Computing, 6(3):83– 87, 2002.

[4] Jetty. http://jetty.mortbay.org/jetty/

[5] JProfiler. http://www.ej-technologies.com/products/jprofiler/overview.html