JPAR – Isolating Java Heap to Protect Runtime of Middleware from Memory Leak

Shu Shi, Ying Li, Kewei Sun, Ying Chen
IBM China Research Laboratory, Beijing, 100094, China
E-mail: sshi@pku.edu.cn, {lying, sunkewei, yingch}@cn.ibm.com

Abstract

Memory leaks are among the most difficult Java application problems, which do various harm to Java middleware-based system, including unpredictable behavior, low performance, and even crash of the whole system. The fatal crash of the whole system may have devastating impact to enterprise applications supported by the system. Traditional approaches to solving memory leak problem are to provide methods and tools to assist finding leak sources in application code, or to diagnose candidate leak objects based on information collected at runtime. However, they can not prevent the whole system from crashing. In this paper, we present a novel approach to protect the whole Java middleware-based system from crashing by proactive isolation, quick detection and effective management of memory leaks. We show how well-isolated Unit of Work, right-sized heap partition, and the heap runtime-monitoring tool can form a framework to protect middleware from memory leak at runtime. A prototype, called JPAR, has been developed to demonstrate that the framework could effectively maintain the whole system continuously operational in the face of memory leaks. The prototype has been optimized from two aspects: appropriate garbage collection strategy for multiple heap partitions, and Resonant Model for setting suitable size of each heap partition. These optimizations effectively reduce the performance overhead caused by multiple heap partitions to 1.5% in our experiments.

1. Introduction

Despite Java’s automatic reclamation of memory, memory management remains a major problem for large scale Java applications. Memory leaks occur frequently and unexpectedly in many Java applications which lead to crash of whole system. Causes of memory leak may be complicated. For example, programmers inadvertently maintain references to objects no longer needed, which prevent these objects from being reclaimed by garbage collection of system, or some applications may have malicious intentions to consume a large amount of memory.

As the design of Java Virtual Machine (JVM) does not consider the isolation issue in memory management, the memory leak logic keeps allocating heap until the memory utilization exceeds the heap capacity. Without available heap memory, the whole system crashes and system administrators even can not run any sophisticated diagnosis [19] to find the root cause of memory leaks. Accounting and controlling memory consumption in JVM [1, 3, 9] can prevent one task with leaking logic from exhausting the whole heap memory. However, it can not prevent the whole system from becoming slow in performance, and then unavailable for service. Since the memory leaking task can not get required memory, the garbage collection (GC) actions have to be triggered to collect more additional memory for its demand. The frequency of GC action keeps growing until most of CPU time has to be spent on collecting garbage. As a result, such excessive GC actions seriously influence the performance of the whole system, or even worse, may cause denial-of-service to all requests. All these phenomena, from users’ perspective, have the same effect as system crash. Therefore, to enable the Java middleware-based system to deal with memory leak, the excessive GC actions triggered by the controlled memory consumption must be managed as well.

In this paper we show how well-isolated UoW (Unit of Work), right-sized heap partition with the heap runtime-monitoring tool can make a Java middleware-based system capable of maintaining operational in the face of memory leak. Here, a UoW is defined as a logical unit of computing or application with different granularity in specific context, such as application server, application, servlet, EJB, session, etc. In the proposed framework, the heap is logically partitioned
for each UoW. Each heap partition is set with suitable size to fit for its corresponding UoW’s demand. During the runtime, the memory leak can be detected and located in a specific partition. Excessive GC caused by isolating memory leak in the partition can also be controlled via the heap runtime-monitoring tool. The proposed framework has been implemented on Jetty, an open-source Java web container and J9VM, the IBM JVM. Our prototype, called JPAR, with the initiative experimental result, demonstrates the effectiveness of our approach to maintain the Jetty-based system continuously operational when memory leak happens. Furthermore, in order to reduce the performance overhead of our framework, we find that the deterioriation of GC performance is the main cause of system performance overhead. The prototype has been optimized from two aspects: to exploit the appropriate GC strategy for multiple partitions in heap, and to set the suitable size of each heap partition. These optimizations effectively enhance the GC performance and reduce the system performance overhead.

The rest of this paper is organized as follows: in section 2 we summarize the related work. Section 3, discusses the overview of the approach. Section 4 presents the design and implementation of the framework. In section 5, we analyze how to optimize the prototype to reduce the performance overhead and we show the experimental evaluation of our prototype in section 6. Section 7 discusses limitations of the prototype and section 8 concludes this paper.

2. Related Work

Researchers explored static and profile-based methods to detect memory leak in source code. For example, [17] presents an algorithm to detect leaks in arrays of objects. The static approach can help to identify leaking sites in application development stage. But due to the limitation of detection rules, the memory leak source that can be found by the static detection approach can not cover the full set of leak source. Diagnosis and profiling tools such as [25, 26, 27, 28] are also developed to analyze the runtime heap information. Following the instructing method like [29], leaking objects of specific patterns are easily identified. LeakBot [15], designed to diagnose memory leak in large java applications, is an impressive one in these memory leak diagnosis tools. It can not only find the problematic data structures automatically, but also refine its diagnosis by tracing application’s running in the on-line mode. Although both static detection and runtime diagnosis are helpful when dealing with memory leak problems, neither of them can prevent the whole running system from crashing when encountering memory leak.

Other researchers’ work [1, 3, 6, 9] explored the resource management in JVM, like CPU, memory, etc. In the multi-tasking virtual machine (MVM) [3], each task has the guaranteed amount of memory and none is allowed to allocate more memory than their guaranteed limit. By accounting and controlling memory consumption within the limit of each task, even the task with leaking logic, cannot monopolize heap memory. While MVM does not consider the high GC frequency caused by the controlled heap consumption, it can not effectively protect the whole system from memory leak. KaffeOS [4] introduces the OS abstraction of process into JVM. It accounts for both CPU and memory on a per-process basis to limit the consumption of each unit. But KaffeOS is not suitable for middleware system. To protect servlets in different processes, KaffeOS needs to start separate servlet engines in each process, which is very high overhead to middleware server.

Recently, J2EE application server is gaining popularity as the platform for hosting enterprise applications written in the Java programming language. Some efforts were initiated to conduct self-management research on application server. For example JAGR [22] can automatically recover from failed EJBs (self-healing). PKUAS [20, 21] can automatically recover from correlated failed services by pre-defined policies. But none of these approaches can enable J2EE application server to recover from memory leaks.

3. Approach Overview

The basic idea of our approach is to combine middleware, JVM and assistant tool together to provide system protection from memory leak. In middleware, the logic of application and middleware itself is divided into different UoWs based on the isolation and protection requirements. UoW, which could be formed with the application server logic, or the critical application servlet, is considered as the heap usage protection unit. In our framework, the runtime heap usage interference of different UoWs is prevented. In JVM, the heap memory is logically partitioned in accordance with the demands of upper UoWs. Each UoW is mapped to dedicated heap partition so that the memory leak occurring in one UoW can be isolated and controlled in its own heap partition, not to exhaust the whole JVM heap. Figure 1 explains the structure. The application logic is divided into three UoWs and middleware logic forms one UoW. Correspondingly, the JVM heap is partitioned into four heap partitions for UoW isolation.
Besides, UoW causing memory leak can be easily found and managed in our framework to prevent worse influence on the execution of whole system. Due to the heap consumption isolation of UoW, the heap partition assigned for leaking UoW would be filled up as memory leak continues. Therefore GC actions in this partition will be triggered frequently to obtain more memory. The runtime characteristics of both full utilization and high GC frequency could be the indication of the leaking heap partition, and furthermore, can help to find the corresponding UoW which causes the memory leak. We provide a runtime-monitoring tool for users, especially the system administrator to observe the runtime status of each heap partition in terms of heap utilization and GC actions. Based on the status information, the user can easily get the indication of leaking heap partition, as well as leaking UoW. Further management actions on leaking heap partition could also be executed, such as dumping the leaking heap partition for more professional memory leak diagnosis, or stopping the leaking UoW directly.

As the proposed framework involves the mechanisms in middleware, we expect that the implementation of the whole framework does not impose significant performance overhead on the Java middleware-based system. By analyzing experiment results of different system configurations, especially the configured GC strategy, we find that the performance of GC is affected after the heap is partitioned into multiple heap partitions. The efficiency of each GC action deteriorates badly as the number of heap partitions increases and leads to high GC overhead. Besides, we also find that the size of heap partition impacts the performance result as well. If the heap partition is too small, the memory consumption demands of the corresponding UoW can’t be well satisfied. As a result, high frequent GC actions will be triggered, which occupies CPU time and then depresses the system performance. The appropriate GC strategy for multiple partitions in heap and the suitable size of each heap partition are two main aspects to optimize the framework.

4. Implementation

We implement the proposed framework on Jetty and J9VM. Jetty [11] is a small and efficient Java HTTP Server and Servlet Container. It is an open source project which has been optimized by commercial and experimental use. J9VM [12] is one of IBM’s production Java virtual machines. The performance of J9VM is competitive with other leading product virtual machines.

4.1. Heap Partition & Isolation

The configuration of isolated UoW is based on a basic assumption that the middleware is considered as “trusted” in terms of heap memory consumption, while upper applications are “un-trusted” which may cause memory leaks. Therefore, in the implemented prototype, the most important, trusted middleware is configured as one UoW, while for the upper applications, their components, like servlets, can be configured as UoWs that should be isolated. In the present prototype implementation, TPC-W [13], the web application benchmark has been used as the application whose servlets are selected as the UoWs to isolate.

The memory space mechanism in J9VM has been leveraged to partition the heap. The memory space, as the unit of heap memory assignment, manages physical memory directly to cope with the requests of object allocation. One memory space can’t use any physical memory of other memory spaces. If the maximum size limit of one memory space is reached, GC will be triggered to collect garbage in the memory space. JclRM, which is the class library of J9VM, exposes APIs of memory space for user to do some basic management operations, including creation, destruction, etc.

The memory space can be assigned to one particular thread. All objects created by the thread will be allocated to assigned memory space. In J9VM, each running thread has an attribute called current-memory-space and all the allocation requests of the thread will be sent to the current-memory-space. In typical web application, servlet is naturally bound with a thread. Each servlet is assigned to a thread from middleware’s thread pool for execution and returns the thread back
when the execution is over. To isolate the servlet UoW, the memory space is created for the servlet in the first run and changes the current-memory-space to servlet’s thread at the point when the thread is taken out from the thread pool for servlet’s execution. Then all servlet objects will be allocated in its own memory space. The UoW of middleware is attached with J9VM’s default memory space. All objects that are not allocated to specific memory spaces for servlets will be considered as objects of middleware and allocated in J9VM’s default memory space.

In our prototype, a mapping table is kept in Jetty to record the mapping relationships between memory spaces and UoWs. Such mapping could be many-to-many according to the isolation requirements. Several UoWs could be isolated into one memory space by setting the current-memory-space of all these UoW’s thread to the same one. Several memory spaces could also serve for one UoW simultaneously.

One special concern in isolation of servlet UoW is that the current-memory-spaces of the servlet thread has to be changed back to original one after servlet’s execution, no matter whether it exits normally or abnormally. Otherwise, the thread would remain attached with the specific memory spaces for servlet. If the thread would be allocated for any further use of middleware, all the objects of this middleware logic will be wrongly allocated into the servlet memory spaces. Such object misplacement ruins the heap isolation for UoW and will cause problems in the future leak protection scenario.

What size should each memory space have is another issue to consider. The size of each memory space is set fixed when it is created and can not be dynamically adjusted. So the size of memory spaces for each UoW should be counted and reserved before running the system. Inappropriate heap reservation may seriously affect the performance of system. Section 5 discusses this issue in detail.

Though the present prototype implementation focuses on isolating servlet as UoW, we argue that the isolation and partition method is the same with other types of UoW. Session and EJB are two types of UoW we will handle in our next steps.

4.2. Memory Leak Isolation and Detection

When memory leak occurs due to one servlet, since each servlet’s object allocation has been limited in its dedicated memory space, the leaking servlet can not deprive heap memory from any other servlets. Therefore, in the face of memory leaks, the whole system, except the leaking servlet, can get required memory for continuous operation. As for the leaking servlet, frequent GC actions will be triggered to reclaim garbage objects in its memory space and further allocation requests will be rejected by throwing OutOfMemoryError when overall memory allocation requests exceed the capacity of memory space.

As discussed in section 1, if without management of excessive GC actions, only controlling the heap usage of leaking servlet can not protect system operational from memory leak. Therefore, JPAR Isolation Monitor (JIM), a heap runtime-monitoring tool, is provided to enable system administrator to declaratively monitor and control memory spaces and GC actions in J9VM.

JIM’s appearance is similar to TOP which is a tool in UNIX/Linux system to monitor runtime process information. It has the Client-Server structure that an agent called JIM server is built inside J9VM for gathering runtime information of memory space and GC. The JIM client is separately installed at administrator’s side to receive and display the information obtained from the JIM server. JIM provides the following main features:

- As for each memory space partitioned in heap, its information of heap memory and GC, as well as the name of corresponding servlet are all displayed in a single line of JIM’s view.
- GC information, including total count, type, time cost, garbage collected, efficiency and overhead ratio in last sample period are accounted and displayed. In particular, GC count, type and efficiency are specially accounted and displayed for each memory space.
- JIM server and JIM client can be connected and disconnected at runtime.

The graphic user interface of JIM is shown in Figure 2, with the sample view of the prototype. In accordance with UoWs to isolate, there are six memory spaces correspondingly. The first memory space, BASE, is J9VM’s default memory space assigned for the middleware UoW. The second memory space, NULL, is called “quarantine”, which is also created by J9VM in default, as a virtual memory space to separate system default memory space with those customized for application. The other four memory spaces are created for isolating four servlets of TPC-W running on Jetty. For example, the memory space with ID 3 is used to isolate
TPCW_best_sellers_servlet which is displayed at the end of line.

By observing the free size and GC information of memory spaces in JIM, it can be easily observed which memory space is occurring memory leak. For example, in Figure 2, memory space with ID 3 is triggering too many GCs and its memory space is almost full. It could be inferred that TPCW_best_sellers_servlet might be encountering memory leak.

JIM also supports administrator to dump and analyze the suspected memory space. To input the command “dump” in JIM will trigger the heap dump of J9VM. User could also specify the memory space ID to dump the heap content of only specific memory space. JIM is capable of analyzing heap dump file and breaking down objects according to the type and the size of the reference tree. Interface to using other diagnosis tools is also provided by JIM.

MID: the logic ID of memory space; ALLGC: total count of GC triggered by this memory space; LOCAL/GLOBAL: separate count of two types of GC; FREE: the free heap size of the memory space; SIZE: the total heap size of the memory space; EFF: GC efficiency of in this memory space; NAME: the name of UoW corresponding to the memory space.

Figure 2 JIM GUI

4.3. Memory Leak Control

After having the judgment of probable memory leak problem in one memory space, the system administrator could choose to stop the excessive GC actions of this abnormal memory space to free the CPU time. The interactive control interface is also provided by JIM. System administrator specifies the ID of memory space which is judged to stop and then JIM will send the identifier of the “stopped” memory space to Jetty (the communication protocol with JIM is currently implemented in Jetty for our prototype and it could also be simply added to other middleware servers). In present prototype, the memory leak control mechanism is simple. According to the mapping table, Jetty could distinguish the subsequent servlet requests which are bound to the “stopped” memory space and directly reject them by throwing exceptions. Therefore, no GC actions in the “stopped” memory space will be triggered to occupy CPU time any more.

By isolating the heap usage of leaking UoW, detecting the leaking memory space and controlling leaking servlet’s execution, we achieve our goal to maintain the whole system continuously operational when memory leak happens. However, the present memory leak control mechanism, to directly reject the requests to leaking servlet, may interfere with execution of other normal servlets if they are functionally correlated. For example, in TPC-W, if the “Registration” servlet has memory leak and has been stopped by administrator, no more users could log into the system and submit any buy request. The “Buy request” and “Buy confirm” servlets are seriously interfered because they are functionally correlated with “Registration”. To alleviate or avoid such interference asks for the recovery of leaking servlet, which will be studied further in next step.

5. Optimization

The previously built prototype implemented the runtime memory leak protection, but also brought the unexpected high performance overhead. We define the performance overhead as deterioration of the servlet execution in normal environment, without any memory leak problem. The deterioration of normal servlets’ execution when memory leak happens is considered as the protection overhead, not the performance overhead we are going to discuss in this section.

The performance overhead can be evaluated by comparing overall throughput of the whole system. The following is the quantitative definition of the performance overhead:

$$\text{Performance Overhead} = 100\% - \frac{\text{Throughput}_{\text{with JPAR}}}{\text{Throughput}_{\text{without JPAR}}}$$

We notice, as the number of memory space increases, the performance overhead as well as the GC overhead get both increased. GC overhead is defined as following:

$$\text{GC Overhead} = \frac{\text{Time}_{\text{GC}}}{\text{Time}_{\text{execution}}}$$

In this section we will focus on the study of GC performance in multiple memory spaces and optimizations to reduce GC overhead.
5.1. GC Overhead Reduction

The default GC strategy of J9VM when multiple memory spaces are created is scavenger GC, a type of generational GC. According to the strategy, each memory space is divided into three sub-spaces: tenure, survivor, and allocate. All the objects are allocated in allocate sub-space. As soon as allocate is filled up, a scavenger GC is triggered and all live objects are copied into survivor sub-space. Then the role of allocate and survivor will exchange. The objects in survivor sub-space are neglected during scan phase of scavenger GC and are all flushed when live objects are copied from allocate to survivor. Long lived objects will be moved to tenure sub-space during the scavenger GC. However, only scanning the full memory space is not enough to identify all live objects. Because the cross memory space reference is not limited in the implementation of memory space, any object in one memory space could be referenced by live objects of other memory spaces. The whole heap has to be scanned for each scavenger GC action. The process is shown in Figure 3. The scan phase is relative longer than collect phase because GC does the heap wide scan but only collects in one memory space.

We define GC efficiency for elaboration as following:

$$\text{GC Efficiency} = \frac{\text{Average Garbage Collected per GC}}{\text{Average Execution Time per GC}}$$

For multiple memory spaces, scavenger GC is not efficient. It spends most time in scanning the whole heap leading to the high average GC execution time, while it only collects the garbage in one survivor sub-space leading to little garbage to collect in each GC.

GC efficiency and GC overhead is tightly related. If we keep the experiment environment as well as application workload unchanged, and we set the same time limit for each experiment execution, we could assume the amount of collected garbage objects in each experiment execution is invariant. Based on this invariant garbage assumption, we can get the following equations:

$$\text{GC Overhead} = \frac{\text{Total Objects Collected} \times \text{Time}_{\text{gc}}}{\text{Total Time}_{\text{Execution}}}$$

From the equation, it can be inferred that the low efficiency of “global scan, local collect” scavenger GC will cause high GC overhead and affect the performance of whole system. In the experiment shown in section 6, it is proved that scavenger GC could degrade the GC efficiency to almost 100 times and bring about 15% overhead to the system’s overall throughput when 10 memory spaces are created.

The solution is to substitute the scavenger GC with more efficient GC strategy. Global GC is picked out. When global GC strategy is selected, each memory space remains a flat space. When the memory space is full, GC is triggered and the whole heap is scanned to identify all live objects. The most important difference between global GC and scavenger GC is that in global GC, garbage of all the memory spaces will be collected during the collect phase. Figure 4 illustrates the whole process. The experiments in section 6 also show that GC efficiency of “global scan, global collect” global GC overwhelms that of scavenger GC and the system.
performance overhead as well as GC overhead are effectively reduced.

As we can see from Figure 5, we assume that there are two memory spaces in the heap. UoW A fills its memory spaces quickly, while UoW B fills its memory in a lower speed. After both UoWs start to run, UoW A will fill its memory space in a short time and triggers a global GC. As a result, the memory space for UoW B is compelled to perform GC accordingly, when it does not need to. Therefore, GC frequency complies with the memory space with the highest GC frequency and the related overall GC efficiency is low.

Resonant Model

Global GC still has deficiencies. Global GC would be triggered when any of the memory spaces is full, and all memory spaces are compelled to perform GC. If different memory spaces are filled up in different speed, then GC activity would be almost always triggered by the memory space with the highest speed. If most memory spaces are far from full but forced to perform GC, such passive, pre-mature GC will result in low GC efficiency and high GC overhead.

Resonant Model is proposed to solve this problem by properly reserving the memory space size to make the GC frequency of each memory space almost the same. When the size is not set properly, the GC frequencies of different memory spaces have wide differences. The memory space, with the highest GC frequency, will trigger GC from time to time and other memory spaces are forced to perform GC as well. However, most of them do not really need to because they are far from full. This circumstance is illustrated in Figure 5:

![Figure 5 Global GC for multiple memory spaces](image)

We can make the following assumptions:

**Assumption 1:** When the total heap size is determined, because global GC will scan and collect garbage in the heap wide, the execution time of each global GC action is almost the same.

**Assumption 2 (Invariant Garbage Assumption):** Under the same experiment environment, workload and execution time, the total garbage collected is invariant.

Together with the previous deduction: GC overhead is in inverse proportion to the GC efficiency, we can get the further conclusion:

**Conclusion:** When total heap size is set to a certain value, and global GC strategy is selected, the smallest GC overhead is achieved when the lowest GC frequency is achieved. Figure 6 illustrates the relationship between GC overhead and GC frequency. The experiment is done within a fixed size of memory heap. As we can see from Figure 6, the points stand for the relationship between GC overhead and GC frequency in a certain time slot. The adjacent points are connected with lines to give a clear view. All of the points of the
figure are restricted into a small angle. That is to say, the GC overhead and the GC frequency are in direct ratio. That is, when the GC frequency is low, the GC overhead is small, which validates our conclusion.

![Garbage Collection Overhead](image)

Figure 6 GC overhead and GC frequency

For each UoW, we use MSsize to represent the memory space size of the UoW and GCfrequency to represent the GC frequency of the UoW. The quantity of requests to different UoWs within a time unit forms the workload of the application. Under a certain workload, the GC frequency of the UoW could be represented as following:

\[
 f_{UoW}(MSsize, workflow) = GCfrequency \\
 f_{UoW}^{-1}(GCfrequency, workflow) = MSsize
\]

To keep the overall GC frequency as low as possible, the following conditions should be satisfied

\[
 \sum_i f_{UoW}^{-1}(GCfrequency_i, workflow_i) = HeapSize \\
 \forall (i, j)(GCfrequency_i = GCfrequency_j) \\
 \min(GCfrequency_i)
\]

6. Experiments

In order to evaluate the efficiency of JPAR prototype, we build experiments with two main parts: a) malicious memory leak logic is added into the scenario to test whether memory space could effectively protect the normal servlet’s execution when memory leak happens; b) performance of primitive un-modified framework and JPAR implementation with different configurations are compared in the scenario without memory leak to show the performance overhead brought by multiple memory spaces and our efforts on reducing it.

The environment of our experiments includes a server (IBM Blade Center, 2-way Xeon 2.4GHz CPU, 2G DDR Memory, running RedHat 9.0) and a client (IBM ThinkCentre PC (Pentium IV 3.2GHz CPU, 2G DDR Memory, running Windows XP Professional), which are connected by 100M LAN. The version of J9VM is 2.3, for Linux x86-32. The version of Jetty is 5.1.4. We choose the Java implementation of TPC-W [14] (version 1.0.1) and Apache Derby as our database. LoadRunner (version 8.0) is used to generate all the experiment requests and record all the performance metrics.

6.1. Memory Leak Protection

In this part, we select four servlets which are not functionally correlated: Bestseller, NewBook, Search and Detail. We also add a new servlet: Bestseller_leak. The Bestseller_leak servlet mimics the function of the Bestseller servlet with malicious logic, which will cause memory leak during execution. We use LoadRunner to simulate the users’ behavior and keep on sending the requests. In order to illustrate how JPAR can help to accomplish the memory leak protection, we divide the experiment scenario into two phases. During the first phase of the scenario, user requests would be sent to the normal servlets to make the systems run smoothly. While in the second phase, user requests which used to be sent to Bestseller servlet would be sent to Bestseller_leak, then we can figure out the results with and without the protection of JPAR. Figure 7 and Figure 8 are to illustrate the experiment results under two different system configurations. The figure’s X axis indicates the elapsed time of scenario and Y axis indicates the response time of requests.

The first experiment is built by using the original framework of un-modified J9VM and Jetty. All of the objects are allocated in a single shared heap, whose size is set to be 128M (experiments of different heap size show similar results). In Figure 7, the bold line stands for the leaking point, when requests start to be sent to Bestseller_leak servlet. The whole JVM crashed soon after leaking point and no servlet could response to the user requests any more because the leak servlet exhausted the entire heap in a short while.

The second experiment is built upon JPAR prototype. We define each servlet as a single UoW and each UoW is assigned a single memory space in order to be separated from one another. The total heap size is also set to
128M. The size of each memory space for servlet UoW is 20M and the remaining is for middleware UoW. As we can see in Figure 8 the whole system keeps running after memory leak happens. That is because with the help of the heap isolation of UoW, Bestseller_leak only eats up the heap of its own memory space, and other UoWs are not interfered. However, due to the excessive GC caused by requests of Bestseller_leak which can't get enough heap memory, the average response time of other normal servlets is driven up distinctly. At the point marked with rectangle in Figure 8, we simulated the system administrator's action to stop the leaking UoW. Jetty will then directly reject all the Bestseller_leak requests and the excessive GC caused by Bestseller_leak will be eliminated. So the average response time of other three normal servlets drop again.

- No memory space/Global GC: primitive unmodified J9VM and Jetty are used with completely default configurations, in which, the heap, with the size of 512M is not partitioned and global GC strategy is selected.
- No memory space/Scavenger GC: primitive unmodified J9VM and Jetty are used. The heap remains un-partitioned, 512M but the GC strategy is set to be scavenger GC.
- Multiple memory space/Average size/Scavenger GC: Each servlet is assigned an average sized memory space of 40M. The total heap size is 512M and the remaining is for middleware. Scavenger GC strategy is selected.
- Multiple memory space/Average size/Global GC: Each servlet is assigned an average sized memory space of 40M. The total heap size is 512M and the remaining is for middleware. Global GC strategy is selected.
- Multiple memory space/Suitable size/Global GC: The total heap size is 512M. The sizes of memory spaces for servlets and the default memory space for middleware are counted by the Resonant Model. Global GC strategy is selected.

Three measurement metrics, including frequency, efficiency, and overhead, are selected to evaluate the GC performance of five system configurations. For multiple memory spaces, as indicated in Figure 9, the scavenger GC has the extremely high frequency and low efficiency, leading to the highest GC overhead of all five configurations. As analyzed in previous section, this is due to the “global scan, local collect” feature of scavenger GC. The global GC strategy, with the feature of “global scan, global collect”, reduces the GC overhead drastically. It can also be observed that the Multiple Memory Space/Suitable Size/Global GC configuration, which counts suitable size of each memory space based on the Resonant Model, can reduce the GC overhead further to 0.59% because it takes full advantage of “global collect” feature of global GC.

Two measurement metrics, throughput (the throughput of the whole system, in terms of bytes) and hit (average http requests that server processes per second), are used to evaluated the performance of five system configurations. The experiment result set is normalized to the reference value of No Memory Space/Global GC configuration. As shown in Figure 10, when no memory space is created, the scavenger GC performs almost as

6.2. Performance Overhead

Different from the above experiment, ten servlets of TPC-W have been selected to evaluate the performance overhead caused by multiple memory spaces. According to the shopping mix of TPC-W [24], all 10 servlets whose proportion is more than 1% in shopping mix are selected to form the workload. We adjust the amount of user requests to make sure the workload will keep the system busy (average CPU usage more than 80%). The performance overhead is evaluated by comparing five system configurations as the following:
good as global GC. But for multiple memory spaces, Multiple Memory Space/Average Size/Scavenger GC shows about 15% performance overhead. Selecting global GC strategy will reduce this overhead to 7.2% and setting suitable size for each memory space could further reduce the overhead to 1.5%.

7. Discussion

The most fundamental assumption we have made in our design and implementation is that objects created in the thread executing one UoW should be accounted to this UoW. That is, objects that are allocated in servlet A’s thread are assigned to the memory space dedicated to servlet A. This accounting assumption is suitable and reasonable for most typical objects except the long living shared objects, e.g., http session objects. In the current implementation, http session object is allocated to the memory space of the servlet who creates this object firstly. However, in fact the http session object may be shared by multiple servlets for usage so that in theory it should not be accounted to the first servlet only. This kind of misplaced object does not cause any problem in the present implementation with experimenal evaluation. However, to some extent, it violates the basic isolation principle that memory space is sized and isolated in accordance with the heap consumption of its UoW. If the strict compliance with the isolation principle is required in specific scenario, misplaced objects should be moved to the right memory space which they belong to. One efficient way to move misplaced object is to execute the move action during GC so as to save the extra time spending on scanning, finding and moving.

Though objects are assigned to partitioned memory spaces, due to the application logic, they may refer to each other across memory spaces. Reference across memory spaces is another violation of strict isolation among memory spaces. Our current experiment results have already indicated the references across memory space to be the root of low performance of scavenger GC. To record the reference across memory spaces may be a method to improve the efficiency of scavenger GC.

8. Conclusion and Future Work

In this paper we show how well-isolated UoW, right-sized memory space in J9VM with a heap runtime-monitoring tool JIM can make a Java middleware-based system capable of maintaining operational when occurring memory leak. In the proposed framework, the heap is partitioned into right size memory spaces based on the Resonant Model to fit for the memory consumption of its corresponding UoW. At the runtime, the memory leak can be detected and located in a spec-
cific memory space. Excessive GC triggered by the isolated memory leak can be controlled by stopping the execution of leaking UoW via JIM. Our system is effective in isolating and detecting memory leaks and protecting the whole system from crashing. Experiments have been built to demonstrate the effectiveness of the approach, and the optimization approaches.

As the extensions to the present work, we plan to enable our framework to support the isolation of session and EJB, dynamically resize memory space and enhance the JIM tool in memory leak diagnosis and memory space management. Meanwhile, we will try to reduce the performance overhead further in exploring more efficient GC strategy for multiple memory spaces. System recovery and self-healing from memory leak will be another important topic for deep research.

References

[16] B Micklea, Guaranteeing J2EE Performance- a full lifecycle strategy, Quest Software, 2004