IJISET - International Journal of Innovative Science, Engineering & Technology, Vol. 2 Issue 11, November 2015.
www.ijiset.com
ISSN 2348 – 7968

A Software defects detection and prevention through virtualization.

Jean Paul TURIKUMWE[1], Dr. Cheruiyot W.K[2] and Dr. Anthony luvanda[3]

Abstract

Software defects detection and prevention is an important practice which needs to be adopted by software developers as it is an important indicator for software quality and efficiency.
In this thesis, we have investigated the existing approaches for memory leak detection, both automatic and manual, and proposes a novel lightweight approach for automatic memory leak detection, utilizing monitoring capabilities and programming interfaces of modern Java Virtual Machines using application virtualization technology. The results of test on multiple applications showed that a considerable part of them presented different level of memory leak problems.

Our work has extended to analyzing the root cause of the memory leaks. In the taken sample projects, we had discovered the causes which was the root problem of memory leak. These cause was related to loaded Drivers which kept in memory unused open connection, DriverManager which was not unloaded on redeploys and other unnecessary resource references. Using the developed pattern, we came on eliminating the memory leak problem and the application was smoothly running with the minimal assigned heap.

Keywords: *Software defects, memory leak, lightweight, application virtualization.*

1. Introduction

Software bug can arise in software and their results affect the software quality, incur additional cost if the software is not rejected at its early stage. Bug also called defect is an error, flaw, failure, or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. Corrective measures need to be identified to prevent these bug for avoiding them to happen in future. Several prevention technics has been developed and their strength are proportional to software development stage or the nature of the software.

Application virtualization is an umbrella term that describes software technologies that improve portability, manageability and compatibility of applications by capsulizing them from the underlying operating system on which they are executed.

A fully virtualized application is not installed in the traditional sense, although it is still executed as if it were. The application is fooled at runtime into believing that it is directly interfacing with the original operating system and all the resources managed by it, however in reality it is not [1].

Thomas Zimmermann on Predicting Bugs from History shown that, the history of successes and failures is provided by the bug database: systematic mining uncovers which modules are most prone to defects and failures. Correlating defects with complexity metrics or the problem domain is useful in predicting problems for new or evolved components. Likewise, code that changes a lot is more prone to failures than code that is un-changed [2].

Edgar H. Sibley, after analyzing two software module one modified due to that it contained errors and a new developed module. He found that Modified and new modules were shown to behave similarly except for the types of errors prevalent in each and the amount of effort required to correct an error. Both had a high percentage of interface errors [3].

Garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed [4].

1. Motivation

The approach of detecting the leaks related defect developed, will help in improving application efficiency before their deployment and their upgrade as well. Developers who doesn't consider the software defects which are related to memory and their respective heap size, will benefit this approach and an improved awareness on these defects is considerable.

2. JVM Heap size

According to JDK6U18, In the Client JVM, the default Java heap configuration has been modified to improve the performance of today's rich client applications. Initial and maximum heap sizes are larger and settings related to generational garbage collection are better tuned [5].

The default maximum heap size is half of the physical memory up to a physical memory size of 192 megabytes and otherwise one fourth of the physical memory up to a physical memory size of 1 gigabyte.

The maximum heap size is not actually used by the JVM unless the program creates enough objects to require it. A much smaller amount, termed the initial heap size, is allocated during JVM initialization. This amount is at least 8 megabytes and otherwise 1/64 of physical memory up to a physical memory size of 1 gigabyte.

3. Experimental leak analysis

To illustrate different memory leak case, I have used Plumbr which an online tool for memory leak detection. We have used this commercial tool because it has many features which allows to monitor a running application.

That tool detect memory usage during the execution and same technique was used as during this case studies. The application initialization time was recorded and was set to 2 minutes fall all application in the case study. We have set the heap size to 100 MB for all application during the test case. Memory usage graphs cover all 14 selected java application, whereas execution time was analyzed separately from all runs.

Used memory with their respective heap size are summarized in **Table 1**. The constant heap size of **sunflow** is due to that it was not presenting the memory leak problem up to the testing time. The large Heap size of Bonita BPM, is due to that it was presenting a high increasing speed of used memory size. To allow the application to keep running, we forced the JVM to reallocate the memory. The application may not show the memory leak problem due that the codes of the performed operation is was highly controlled to not cause the memory leaks. The results obtained are summarized as below with **S** and **H** representing used memory size and allocated heap size respectively

Time(Sec)	**120**		**160**		**200**		**240**	
Application	**S**	**H**	**S**	**H**	**S**	**H**	**S**	**H**
EasyChurch	10	100	175	200	440	490	510	610
Bonita BPM	40	100	320	420	630	670	650	705
MATSim	25	100	230	300	520	610	510	640
Convertigo	70	100	125	205	310	350	450	490
Subsonic	90	100	190	290	340	390	350	420
Biogenesis	25	100	165	210	270	320	320	430
sunflow	40	100	45	100	40	100	40	100
sashimi	25	100	180	230	290	320	340	455
MindRaider	35	100	60	90	230	300	220	320
Jena	15	100	330	420	435	515	610	690

Table 1. Java application memory usage with Plumbr

As per the this result, some of these application behaviours are illustrate with the following charts

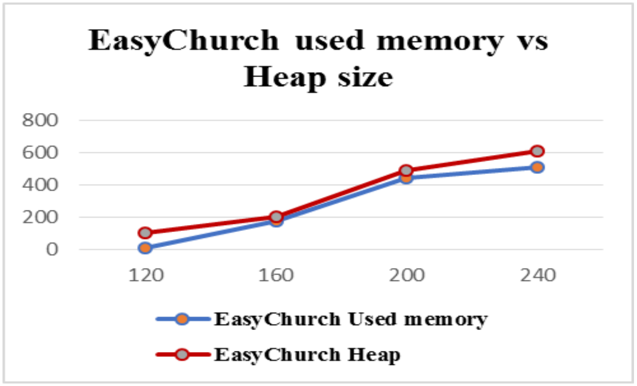

Fig 1 EasyChurch used memory vs Heap size

As per this chart, the case of EasyChurch shows that it has using the maximum allocated heap size. After the 240 seconds, the application frozen throwing OOM message and the application crashed.

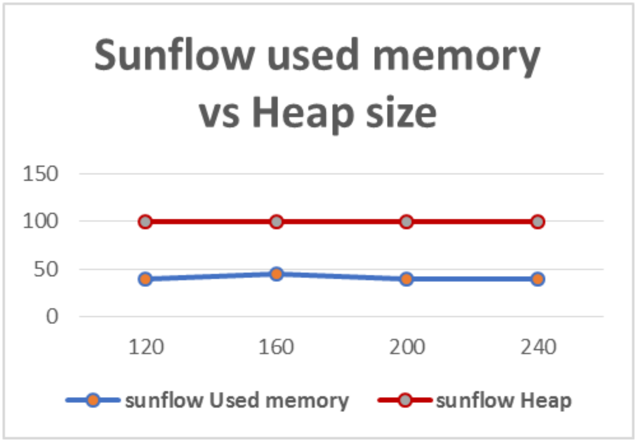

Fig 2 Sunflow used memory vs Heap size

This case of sunflow, shows that the application was running smoothly with the allocated heap memory. Normally the memory usage of an application depends on the process in execution.

4. Memory leak diagnostic

To investigate the memory problem for the case of EasyChurch application, we used NetBeans Profiler to determine the root cause of the issue. NetBeans has a built in tool called Profiler which has several options to monitor the resource usage. This Profiler provides a view called the Runtime Heap Summary that shows the amount of heap memory in use over time as the Java application is running. It also provides a toolbar button to force the JVM to perform garbage collection when desired. This capability turned out to be very useful when trying to see if a given instance of a class would be garbage collected when it was no longer needed by the Java application. The **fig 3** shows the performance of EasyChurch application during profiling

In the Heap Usage Chart, the Purple color indicates the used memory and the red one indicate the amount of heap space that has been allocated. After lunching the profile module, the application was running without any problem and the used memory was varying between 0 Megabyte and 80 Megabyte which is very normal compared to the application size after performing some operations. The memory leak manifested on creating income operation which was not well controlled on the coding time.

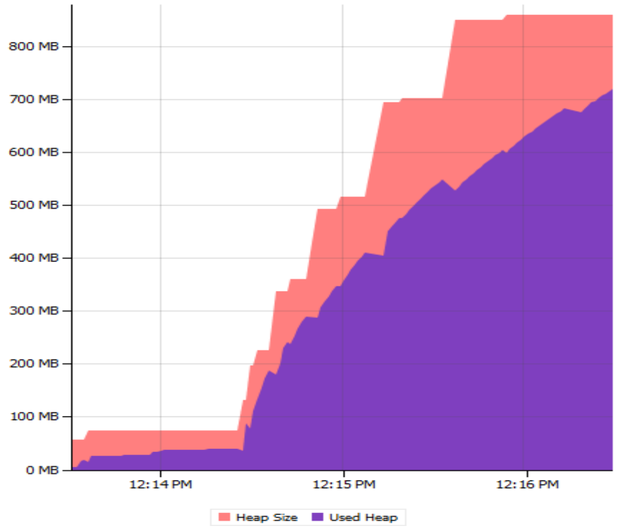

Fig 3 Profiling EasyChurch application

The memory leak situation occurred when the programmer after opening the JDBC connection and started querying the database. The JDBC query code was completely correct according to the logic flow. The cause of the memory leak was due to the thrown exception and active JDBC objects which left unclosed that led the application on being unreliable, overloading the database server and using more memory than it needs. The other cause of the memory leak was due to that the programmer wanted to put extracted record to a HashMap with checking that there was fetched records while the fetch record returns a Boolean which is true always there are some records.

5. Memory leak fix

The found error related to memory leak can be summarised in the following categories:

- **MySQL drivers** launching background threads
- **java.sql.DriverManager** not unloaded on redeploys
- **HashMap** when checking that there was fetched records while the fetch record returns a Boolean which is true always there are some records

The first step need to stop the MySQL loaded drivers. Apparently the most common MySQL drivers launches a thread in the background cleaning up unused and unclosed connections.

After stopping the loaded drivers, the catch is that the context ClassLoader of this newly created thread in the application ClassLoader. Which means that while this thread is running and the application was trying to run other thread threads, its ClassLoader is left dangling behind with all the classes loaded in it.

As that step is not enough, we need to proceed with implementing the *cleanLoaded()* method, which the developer should know to invoke before closing the process under execution. There is a good place for such cleanLoaded hooks in Java web application, which are namely the *ServletContextListener* class *contextDestroyed()* method.

This such function can be invoked every time the servlet context is destroyed on the server, which most often happens during redeploys. This approach is a practice of most java developer, but a number of them don't care on cleaning up this particular hook and their application remains unstable.

Another approach to discover the issue can be related to **datasource** and context **classloaders**. Normally, once the com.jdbc.myslq.Driver registers itself as a driver in java.sql.DriverManager class. This needs to be done with clear focus. With all this done, the application will be able to figure out how to choose the right driver for each query when connecting to the database URL.

When the DriverManager is loaded in bootstrap classLoader instead of from web classloader application, the developer needs to unload the driver manually from the application. In that context, the catch is required with *cleanLoaded()* call and it will be redeployed by the application itself.

For developer, there isn't a perfect general method to unregister the loaded driver. The class which references the one which want to unregister the drivers is hidden from the developer. In this context, developer needs to know all registered JDBC drivers with DriverManager in order to decide the ones to unregister.

For the case of HashMap, normally A HashMap object occupy 32 * SIZE + 4 * CAPACITY bytes with 32(12 bytes header + 16 bytes data + 4 bytes padding), while the theoretical map size limit could be equal to 8 * SIZE bytes. Trove THashMap is a replacement implementation for HashMap. Internally THashMap contains 2 arrays – one for keys, another for values. It means that THashMap needs 8 * CAPACITY bytes for storage [6].

After applying these steps, we needed to run the application following the same exact procedure was performed previously. At this time, the application was deployed in virtual environment provided by Cameyo. Its technology aims to virtualize applications so that they can run on other machines or in HTML5 browsers. The new performance are illustrated on the **figure 4**

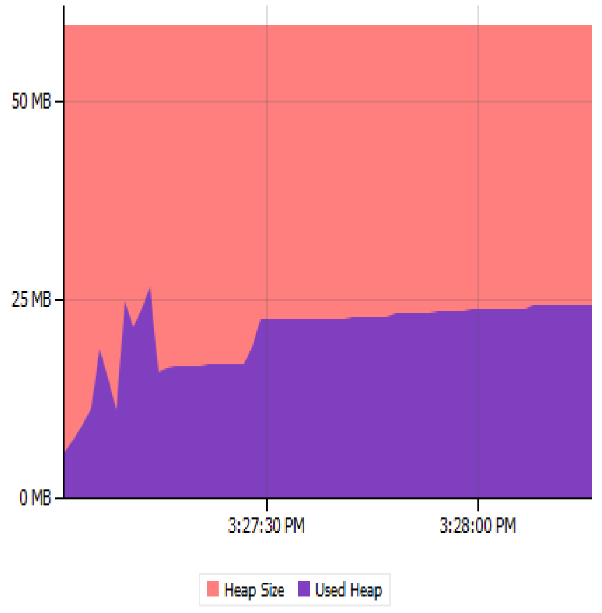

Fig 4 EasyChurch memory usage after applying the fix

As per this figure, there is aconsiderable difference consumed memory in comparison with the assigned memory heap.

6. Conclusions

In this thesis, we have investigated different Java open source projects. In the sample we have tested a part of them had presented different level of memory leak problems. Our work has extended to analysing the root cause of the memory leaks. In the taken sample project, we had discovered the causes which was the root problem of memory leak. These cause was related to loaded Drivers which kept in memory unused open connection, DriverManager which was not unloaded on redeploys and other unnecessary resource references and the HashMap oject which was not properly managed. Using the developed pattern, we came on eliminating the memory leak problem and the application was smoothly running with the minimal assigned heap.

Investigating the memory leak cause is not a direct process and it require powerful debugging or profiling tools. However, once you become familiar with the tools and the patterns to look for in tracing object references, you will be able to track down memory leaks. It is a good practice for every time considers the Debugging tool which consider also the memory consumption for high performance of java application. This also provide insight as to what coding practices to avoid to prevent memory leaks in future projects.

Application Virtualization used in this thesis helped in maintenance and greater portability. The facility to deliver the tools users need quickly and reliably is core to the concept of delivering a flexible, cost-effective and robust workspace. Application virtualization gave facility to deliver applications to devices which do not support these applications. Virtualizing application does not also helps in managing memory efficiently as it uses the assigned memory on their serve and allows applications to run on various version and on any systems.

Using Application Virtualization your applications are protected, since a malicious user wanting to walk away with your applications would not have access. Using the application virtualisation technology in this thesis helped us benefiting from centralised resource and applying different tools that checks the memory leaks related defects.

Future research include, but is not limited to, discovering new types of performance problems, which may be analysed in a similar way by applying the run-time monitoring tool. Engineering research topics also include performance optimizations for the current implementation which would further reduce the runtime overhead that can be very noticeable in applications exhibiting high object allocation rates.

In addition, interpretation of the leak detection results in the context of other dynamic programming languages executing on the JVM, Like Groovy, C#, Python, etc. is yet to be analysed.

References

[1] Bhathal, K. K. (2013). Computer Science and Technology Cloud and Distributed. *Trend and Need of Application Virtualization in Cloud Computing,* 2-3.

[2] Tom Mens, Serge Demeyer (2008). *Software Evolution*. Berlin Heidelberg: Springer. P87-88.

[3] Edgar H. Sibley. (1984). software errors and complexity: an empirical investigation. *Computing practices.* 27 (1), P49-50.

[4] Oracle. (2015, July 24). *Java Garbage Collection Basics*. Retrieved from oracle.com: http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

[5] Oracle. (2014, January). *Java SE 6 Update 18 Release Notes.* Retrieved from oracle.com: http://www.oracle.com/technetwork/java/javase/6u18-142093.html

[6] Java performance. (2013, July 23). *Memory consumption of popular Java data types*. Retrieved from http://java-performance.info/memory-consumption-of-java-data-types-2/

[7] Sakthi Kumaresh and Baskaran Ramachandran. (2012). International Journal of Software Engineering & Applications (IJSEA). *DEFECT PREVENTION BASED ON 5 DIMENSIONS OF DEFECT ORIGIN*. 3 (4), 9

[8] Sommerville, I. (2004). 7th Edition, *Software Engineering*. Dorling Kindersley, India, pp 139-162.

[9] ŠOR, V. (2014). *Statistical approach for memory leak detection in Java applications.* Zurich: University of Partu PRESS.

[10] Pan Tiejun, Zheng Leina, Fang Chengbin, (2008), "*Defect Tracing System Based on Orthogonal Defect Classification*" published in Computer Engineering and Applications, vol 43, PP 9-10, May 2008.

[11] Plumbr. (2014, September 4). *Memory leaks – measuring frequency and severity*. Retrieved from plumbr.eu: https://plumbr.eu/blog/memory-leaks/memory-leaks-measuring-frequency-and-severity

[12] Schwab. (2008, April). *The benefits of application virtualization*. Retrieved from searchitchannel.techtarget.com: http://searchitchannel.techtarget.com/feature/The-benefits-of-application-virtualization

[13] Chris Jackson (30 Apr 2008). *Can You Shim Applications Virtualized in SoftGrid?*. [ONLINE] Available at: http://blogs.msdn.com/b/cjacks/archive/2008/04/30/can-you-shim-applications-virtualized-in-softgrid.aspx. [Last Accessed 31 October 2014].

[14] Stefan Wagner, 2008,"Defect Classification and Defect Type Revisited" Proceedings of the 2008 workshop on Defects in large software systems, (DEFECTS'08) pages 73-83, ACM Press, 2008

[15] Pan Tiejun, Zheng Leina, Fang Chengbin, (2008), "*Defect Tracing System Based on Orthogonal Defect Classification*" published in Computer Engineering and Applications, vol 43, PP 9-10, May 2008.

[16] Fenton, N. E., & Neil, M. (1999). A critique of software defect prediction models. IEEE Transactions on Software Engineering, 25, 675–689.

[17] Rugina, M. O. (2012). *Memory Leak Analysis by Contradiction.* New York: Computer Science Department Cornell University Ithaca.