Research Journal of

Information Technology

ISSN 1815-7432

Academic
Journals Inc.

www.academicjournals.com

OPEN ACCESS Research Journal of Information Technology

ISSN 1815-7432
DOI: 10.3923/rjit.2017.18.24

Research Article
Adaptive Calling of Garbage Collector to Decrease Memory Leaks Based on Methods Ranking

Mohamed S. Farag, M.M. Mohie El-Din and O.M. Hassan

Department of Mathematics, Facility of Science, Al-Azhar University, Cairo, Egypt

Abstract

Background: Memory management with dynamic allocation one of the most complex problems for software managed environment. Memory leak problem usually appear with long running information server applications. **Materials and Methods:** Garbage collectors is an effective tool to manage memory leak but only when the amount of lost storage is bounded. It has been shown that application performance with garbage collector is highly dependent on the application behavior and resource availability. Meanwhile, the PageRank algorithm is a widely used scoring function of networks in general and of the World Wide Web graph in particular. Current approaches rarely considered the application specific methodology to handle the memory management problem. **Results:** In this study, we focused on decreasing the memory leak for large-scale information systems running on servers. An approach proposed to apply adaptive calling of garbage collector within the methods relevant to memory leak during the application running, where PageRank algorithm is adjusted to estimate the importance of each method. **Conclusion:** Java benchmark tool "DaCapo" has been used to test and compare the memory and processing time before and after applying the adjusted algorithm, which showed the memory leak ration is successfully decreased with minor effect on processing time.

Key words: Memory leaks, garbage collector, PageRank algorithm, memory management

Received: August 03, 2016 Accepted: October 27, 2017 Published: December 15, 2016

Citation: Mohamed S. Farag, M.M. Mohie El-Din and O.M. Hassan, 2017. Adaptive calling of garbage collector to decrease memory leaks based on methods ranking. Res. J. Inform. Technol., 9: 18-24.

Corresponding Author: Mohamed S. Farag, Department of Mathematics, Facility of Science, Al-Azhar University, Nasr City, 11884 Cairo, Egypt
Tel: 20-1006-574-243

Copyright: © 2017 Mohamed S. Farag *et al*. This is an open access article distributed under the terms of the creative commons attribution License, which permits unrestricted use, distribution and reproduction in any medium, provided the original author and source are credited.

Competing Interest: The authors have declared that no competing interest exists.

Data Availability: All relevant data are within the paper and its supporting information files.

INTRODUCTION

Memory leaks with large-scale information systems running on servers devoid to be belongs to operational configuration, it can easily lead to degradation of system performance with high potentials for system failure[1].

Managed environments enables portable and secure execution of applications written in high-level programming languages. Improving and enhancing the performance of these environment is the insurance to the continued success of these environments and the programs that they execute. Managed environments use garbage collection to enable automatic memory reclamation and memory safety. The garbage collection must recycle heap memory effectively without adding significant overhead on the executing application[2].

This study will focus on decreasing the memory leak for large-scale information systems running on servers. Keeping these systems running and avoiding memory problem is very important especially at the peak of the request hits time. We will studying the most relevant points at the systems that could cause the failure and decrease this probability.

Prior work, explained that application performance with garbage collected is highly dependent upon the application behavior with considering the resource availability. So man and Krintz[2] showed that given a wide range of diverse garbage collection algorithms, no single system performs best across applicationsconfigured with different heap sizes.

The goal of most of prior work has been to provide general-purpose mechanisms that enable high performance execution across all applications. However, many researchers find that the performance of a memory management system (the allocator and the garbage collector) is dependent upon application behavior and available resources.

That is, no single garbage collection enables the best performance for all applications with different heap sizes and the difference in performance can be significant. Currently, managed environments enable application tospecify heap and GC configuration with different configurations of the execution environment[2]. Recent researches focus on studying the performance of heap allocation and collection techniques[3-7]. Memory leak detection techniques could be classified as online detection, offline detection and hybrid methods[8].

Bond and McKinley[9] an online approach has been introduced for leak detection, instead of monitoring object staleness, wholedata structure staleness is identified and insteadof swapping out the data structure it is reclaimed (pruned) altogether. Sor *et al.*[10] presented an idea for a statistical approach tomemory leak detection based on growth analysis. Maxwell *et al.*[11] an offline approach presented to analysis the heap dumpsfully or partially to detect memory leaks. The LeakChaser method that introduced in Xu *et al.*[12] used hybrid approach to detect memory by three-step iterative profiling methodology to find causes of memory leaks in Java applications.

Meanwhile, the PageRank algorithm is a widely used scoring function of networks in general and of the World Wide Web graph in particular. PageRank considered a link structure-based algorithm, which gives a rank of importance of all the pages crawled in the internet by the Googles's web crawler. To calculate a PageRank is actually to calculate the distribution of a transition matrix which is based on the web graph structure[13-15].

The PageRank of a Web page A, denoted by (A), defined by using the following equation[16]:

$$PR(A) = (1 - d) + d.\sum_i \frac{PR(T_i)}{C(T_i)}$$

where, $PR(T_i)$ is the PageRank of page T_i which has connection with page A, $C(T_i)$ is the number of outbound links on page T_i and d is a damping factor which can be set between 0 and 1.

In this study, an approach introduced to apply adaptive calling of garbage collector within the methods relevant to memory leak during the application running, where PageRank algorithm has been adjusted to estimate the importance of each method.

MATERIALS AND METHODS

Ranking mthods for adaptive GC calling: This study describes the approach of adaptive GC calling, starting by the profiling problem for collecting the methods memory related information which was solved by enhancing one of the existing profiles (JP2) to allow collecting and extracting methods memory information. Moving forward to the manipulation of the extract memory data to methods calling matrix and applying the PageRank-like algorithm.

Finally, after reaching the relevant points to apply the informed calling GC, system should apply this informed calling adaptively to minimize the effect on the processing time, how this calling was handled.

Memory aware profiling: The process of automatic collection and presentation of data that is representing the dynamic

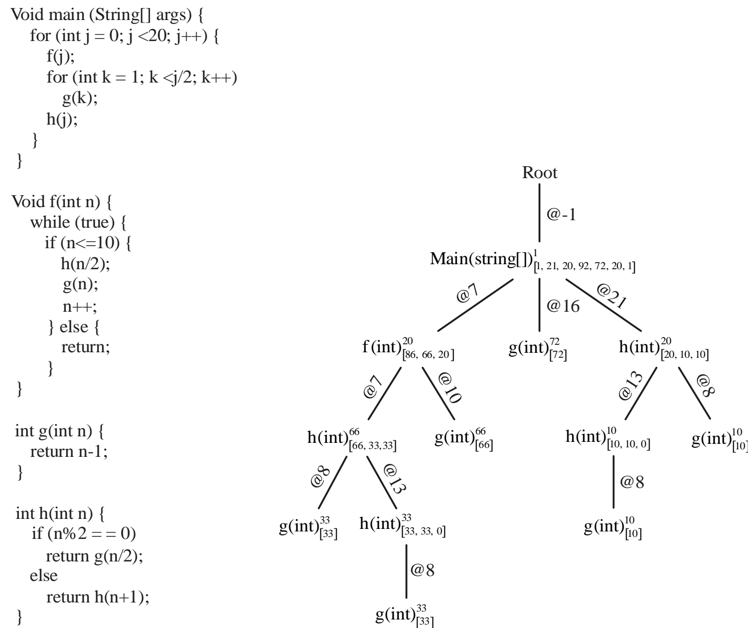

Fig. 1: Sample for Java code and the represented CCT

behavior of the program is called profiling[17]. After profilers collect and analyze the data, it can be either automatically feedback to the compiler or present it for the developers. Each case has different requirements in designing the profiler[18]. The JVM uses a program dependence graph as the intermediate data structure when compiling Java bytecodes to machine code. When using the compiler in debug mode, it is providing a textual output of the graph[19,20].

Meanwhile, calling context provides a complete picture about program control flow which represents important profiling perspective. A calling context represents a sequence of methods that have been called but have not yet completed. Calling context provides dynamic metric for profiling like the number of times a method has been invoked or the CPU time spent within a method[21,22].

Calling Context Tree (CCT) is a well-known data structure commonly used to represented calling context. Each node in the CCT corresponds to unique calling context and any dynamic metric can be supported (e.g., CPU time, number of cache misses). The parent of a CCT node corresponds to the caller's context, while the children nodes represent the callees[21]. Figure 1 as listed by Sarimbekov *et al.*[23], shows sample for Java code and the CCT generated after one execution of method "Main", each CCT node stores the number of method invocations (m) and dynamic execution counts for each basic block ([c1,..]) and the bytecode index (@), where the method was invoked in the corresponding calling context.

The JP2[23,24] is a platform-independent tool for the Java Virtual Machine to create CCTs with dynamic metrics, such as the number of method invocations and the execution statistics at the level of individual basic blocks of code. Also, JP2 is able to distinguish between multiple call sites in a method and supports selective profiling, i.e., profiling some calling contexts only. But it could only extract the data without analysis or visualize.

The JP2 collect efficiently the calling information between methods but it ignore the time stamp of the calling and the memory stat before and after each call. These problem were solved by modifying the architecture of JP2 to be time and memory aware to be as in Fig. 2. This modification extends the produced calling context tree to list for each method the time stamp of start/end calling. In addition to, the memory states at method start and end and memory states means the amount of available, consumed and total memory amounts.

Ranking methods approach: To model the activity of the random web surfer, the PageRank algorithm represents the link structure of the web as a directed graph. Webpages are nodes of the graph and links from webpages to other webpages are edges that show direction of movement. Although the directed web graph is very large, the PageRank algorithm can be applied to a directed graph of any size.

The core of the PageRank algorithm involves repeatedly iterating over the graph structure until a stable assignment of importance estimates is obtained. Same concept will use to

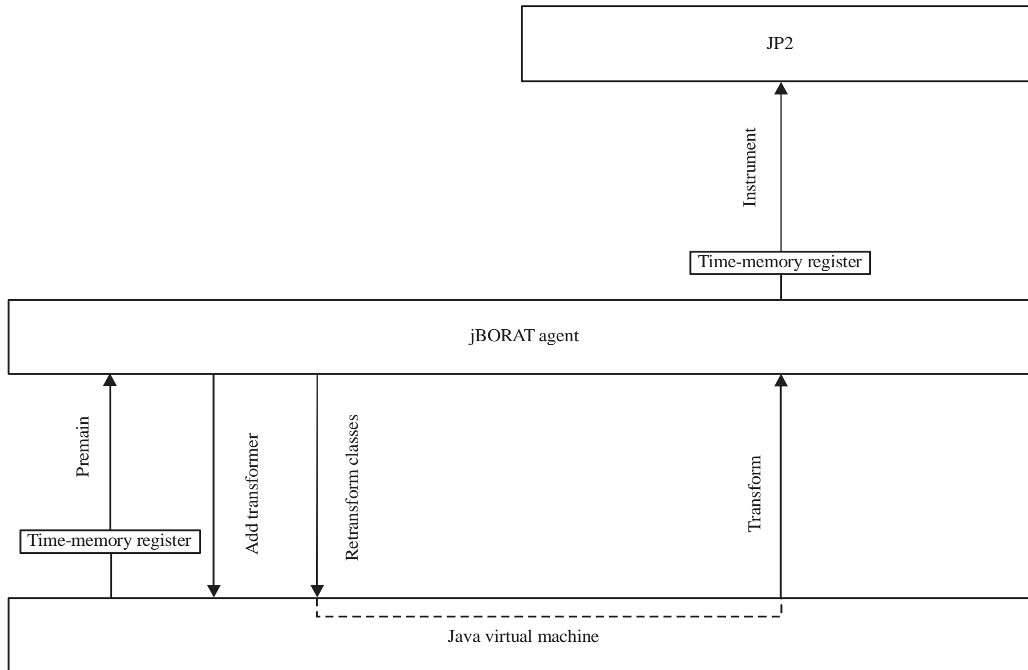

Fig. 2: Modified architecture of JP2 to be time-memory aware

select method importance estimates. The complexity of the software methods graph and size required an effective algorithm while working on computations. PageRank can be computed for very large graph with limited memory resources. Transition matrix computed during PageRank algorithm calculations can be updated partially with no need to start all the analysis over and that fit with the natural of software changes.

In this study, the PageRank algorithm adjusted to solve the methods memory relevance as was described in the previous section, the methods calling can be represented in directed graph form. The methods represents the nodes of that graph and the calling operation represents the edge. For each edge the consumed memory and time recorded.

Based on the idea of PageRank algorithm, the method will be relevance to memory leaks if other relevance methods linked to it.

For n methods M_i, I = 1, 2, ..., n the correspondence method rank is set to r_i, I = 1, 2, ..., n. The mathematical formulation for the recursively defined method rank as in the following equation:

$$r_i = \frac{\sum_{j \in L_i} (r_j . c_{ij})}{N_i}, \; i = 1, 2, ..., n., N_i = \sum_j c_{ij}$$

Where, r_i is the rank of method M_i, N_i is the total consumed memory within M_i, C_{ij} is the consumed memory for linking method M_i with M_j and L_i is the set of methods linked to M_i.

The ranking problem is recursive by definition, so it also can be solve by iteration until find the acceptable solution. This can be solved by the following algorithm:

1 $r_i^{(0)}$, I = 1, 2, ..., n initialized with non-zeros value
2 For k = 1, 2, ... do
  a. $r_i^{(k+1)} = \frac{\sum_{j \in L_i} \left(r_i^{(k)} . c_{ij} \right)}{N_j}, \; i = 1, 2, ..., n$
  b. If $\left\| r_i^{(k)} - r_i^{(k+1)} \right\|$ < tolerance then
    i. Break
  c. End if
3 End for

This representation can be simplified by using matrix model, by defining the Q matrix where, Q_{ij} defined as given equation:

$$Q_{ij} = \begin{Bmatrix} c_{ij}/N_i \;\; \text{if } M_i \text{ links to } M_j \\ 0 \;\; \text{otherwise} \end{Bmatrix}$$

By this definition the Q matrix is stochastic matrix which means $\Sigma_j Q_{ij} = 1$. By finding the eigenvector of matrix Q corresponding to eigen value equals one that will define the ranking vector.

Informed calling of GC: Calling GC within the application in extensive way will definitely affect the processing time. To reduce the memory leak and reserve the time of the processing, calling should be very limited but effective. The first parameter used to control this informed calling is the number of methods that should call the GC. After sorting the ranking vector, only n_{gc} number methods will be used.

As this informed calling is not come free, it was limited to use only at the time that matter. If system has plenty of available memory, reducing the memory leak does not matter. The calling will take effect only when the available memory is less than the specific threshold, m_{gc} used to define the amount of available memory to start calling GC if system reach to it.

The explicit calling of GC in JVM using JVM system API does not necessary force calling[25], the JVM will consider it as hint or recommendation for calling. Force calling of GC can be done by using JVM tool interface[26] or by tricking the JVM using weak reference and checking iterations number t_{gc}. Empirically, the approach of using weak reference was more suitable.

The following algorithm propose how to use the parameters n_{gc}, m_{gc} and t_{gc} to apply the informed calling of GC to reduce the memory leaks with minor effect of processing time:

```
1.    For each n_gc method
          a.  If available memory less than m_gc then
                  i.  Create weak object
                  ii. For k = 1, 2, ..., t_gc do
                          1.  Call GC
                          2.  If weak object removed then
                                  a.   Break For
                          3.  End If
                  iii. End For
          b.  End If
2.    End For
```

RESULTS AND DISCUSSION

Java benchmark “batik” taken from “DaCapo” has been used to test and compare the memory and processing time before and after applying the algorithm. This benchmark are non-trivial real-world open source Java programs under active development[27]. Different data sizes (small, default, large) with different iterations of processing have been used to run this evaluation.

For the following result these parameters has been used: $n_{gc} = 3$ only top 3 methods have been used to call informed GC, $m_{gc} = 15$ informed GC will be called if available memory less that 15% of total memory, $t_{gc} = 3$ only trying to call GC 3 times.

Figure 3 shows the result of running batik benchmark with and without informed GC. Noticing that using informed GC show effect more when system consume more memory. Which means that informed GC decrease the amount of memory leak more efficiently with the enterprise information systems that consume larger amount of memory.

While, Fig. 4 shows the increasing in the processing time effect that informed GC causes. About that, noticing that with larger number of iteration the effect tends to be minor. The algorithm uses m_{gc} the parameter to control the number of calling which causes these effects, by adjusting this parameter empirically can find the best value to do memory leak decreasing with minor processing time effect.

At this study, we focused on decreasing the memory leak for large-scale information systems running on servers. Keeping these systems running and avoiding memory problem is very important especially at the peak of the request hits time.

As shown earlier in this section, by studying the most relevant methods that could cause the failure and explicitly call the GC, that leads to decrease the failure probability that could be happened because of memory leaks comparing to the system without adapted GC calling.

This approach cause minor increasing of the processing time only when the system reach the threshold of starting the adaptive calling of GC. Therefore, system at time of peak of the requests will take more little time but with less consumed memory.

Memory leak detection techniques could be classified as online detection, offline detection and hybrid methods, this approach here is hybrid approach that detect the memory leaks by offline analysis and reduce it during online execution. Unlike the online approaches by Bond and McKinley[9] that actively monitor and interact with the running virtual machine in order to detect leaking objects or collecting the statistics during the execution as by Sor *et al*.[10], we used the offline phase to do the time consuming analysis to avoid imposing overhead on the running application.

Our approach has the advantage of accessing to run-time information but[11] the analysis the heap dumps as an offline approach loses that which prevent him from providing dynamic solution during the runtime. The LeakChaser method that introduced by Xu *et al*.[12] used hybrid approach to detect memory by three-step iterative profiling methodology to find causes of memory leaks in Java applications, however, the described implementation was applicable only to non-generationalgarbage collectors, because partial heap scans performed by generational collectors may produce both false positives and negatives.

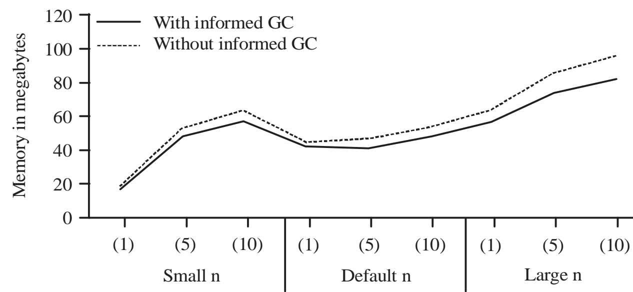

Fig. 3: Memory compare for running batik benchmark with and without informed GC

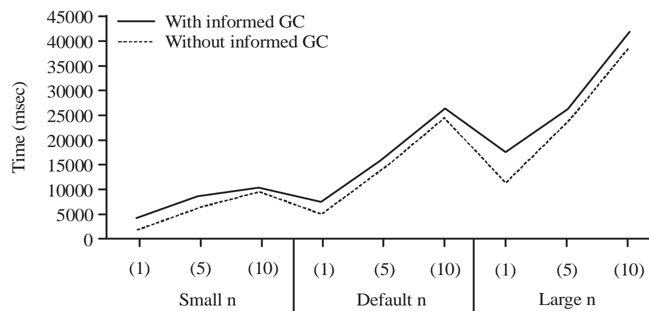

Fig. 4: Time compare for running batik benchmark with and without informed GC

In this approach, using the methods as unit for memory analysis instead of scanning all heap allowed us to go with the explicit calling of GC as change relative to the application with no need to apply modifications of the JVM and without changing any implementation logic within the application. Most of the other approaches tends to detect memory leaks and provide recommendation to apply changes in the application implementation logic or do general modifications on the JVM, which makes it much complex than explicit calling approach.

CONCLUSION

In this study, an approach proposed to apply adaptive calling of garbage collector within the methods relevant to memory leak during the application running, where PageRank algorithm is adjusted to estimate the importance of each method. Current approaches rarely considered the application specific methodology to handle the memory management problem.

By applying our approach, the memory leak ration is successfully decreased with minor effect on processing time. It is an offline approach, which means that the analysis time to find the relevant methods will not affect the processing time. This is application specific technique that allow to maximize the benefits of memory monitoring by limit the conditions of the running application.

Ranking method using PageRank-like algorithm has great benefits for easy calculations of relevant methods with the abilities to cash and update the application result in light processing form. Moreover, the methods calling matrix is generic enough to allow applying other graph analysis algorithms rather than PageRank algorithm.

To continue improving this approach, future work will consider integrating the GC algorithm with the method ranking technique. Automating the process of method analysis by making it part of JVM and adding the autonomous ability to switching from analysis mode to adaptive GC calling will be part of the future investigations.

REFERENCES

1. Willard, B. and O. Frieder, 2000. Autonomous garbage collection: Resolving memory leaks in long-running server applications. Comput. Commun., 23: 887-900.

2. Soman, S. and C. Krintz, 2007. Application-specific garbage collection. J. Syst. Software, 80: 1037-1056.
3. Blackburn, S.M., S. Singhai, M. Hertz, K.S. McKinely and J.E.B. Moss, 2011. Pretenuring for java. Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, October 22-27, 2011, Portland, OR., USA., pp: 342-352.
4. Blackburn, S.M., R. Jones, K.S. McKinley and J.E.B. Moss, 2002. Beltway: Getting around garbage collection gridlock. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 17-19, 2002, Berlin, Germany, pp: 153-164.
5. Arjom, E. and C. Li, 2001. Controlling garbage collection and heap growth to reduce execution time of Java applications. Prceddings of the ACM Conference on Object Oriented Programming, Systems, Languages and Applications, October 13-19, 2001, Tampa, FL., USA.
6. Bacon, D.F., C.R. Attanasio, H.B. Lee, V.T. Rajan and S. Smith, 2001. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 20-22, 2001, Snowbird, UT., USA., pp: 92-103.
7. Hirzel, M., A. Diwan and M. Hertz, 2003. Connectivity-based garbage collection. Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages and Applications, October 26-30, 2003, Anaheim, CA., USA., pp: 359-373.
8. Sor, V. and S.N. Srirama, 2014. Memory leak detection in Java: Taxonomy and classification of approaches. J. Syst. Software, 96: 139-151.
9. Bond, M.D. and K.S. McKinley, 2009. Leak pruning. Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, March 7-11, 2009, Washington, DC., USA., pp: 277-288.
10. Sor, V., P. Ou, T. Treier and S.N. Srirama, 2013. Improving statistical approach for memory leak detection using machine learning. Proceedings of the 29th IEEE International Conference on Software Maintenance, September 22-28, 2013, Eindhoven, The Netherlands, pp: 544-547.
11. Maxwell, E.K., G. Back and N. Ramakrishnan, 2010. Diagnosing memory leaks using graph mining on heap dumps. Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 25-28, 2010, Washington DC., USA., pp: 115-124.
12. Xu, G., M.D. Bond, F. Qin and A. Rountev, 2011. LeakChaser: Helping programmers narrow down causes of memory leaks. Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, June 4-8, 2011, San Jose, CA., USA., pp: 270-282.
13. Brin, S. and L. Page, 1998. The anatomy of a large-scale hypertextual web search engine. Comput. Networks ISDN Syst., 30: 107-117.
14. Grolmusz, V., 2015. A note on the pagerank of undirected graphs. Inform. Process. Lett., 115: 633-634.
15. Sun, H. and Y. Wei, 2006. A note on the PageRank algorithm. Applied Math. Comput., 179: 799-806.
16. Page, L., S. Brin, R. Motwani and T. Winograd, 1999. The pagerank citation ranking: Bringing order to the web. Technical Report Stanford InfoLab. http://ilpubs.stanford.edu:8090/422/
17. Dmitriev, M., 2004. Profiling Java applications using code hotswapping and dynamic call graph revelation. Proceedings of the 4th International Workshop on Software and Performance, January 14-16, 2004, Redwood Shores, CA., USA., pp: 139-150.
18. Liang, S. and D. Viswanathan, 1999. Comprehensive profiling support in the Java virtual machine. Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems, May 3-7, 1999, San Diego, California, USA., pp: 229-242.
19. Ferrante, J., K.J. Ottenstein and J.D. Warren, 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst., 9: 319-349.
20. Paleczny, M., C. Vick and C. Click, 2001. The Java Hotspot™ server compiler. Proceedings of the Symposium on Java™ Virtual Machine Research and Technology Symposium, Volume 1, April 23-24, 2001, Monterey, CA., USA.
21. Ammons, G., T. Ball and J.R. Larus, 1997. Exploiting hardware performance counters with flow and context sensitive profiling. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 16-18, 1997, Las Vegas, NV., USA., pp: 85-96.
22. Binder, W., 2005. A Portable and Customizable Profiling Framework for Java Based on Bytecode Instruction Counting. In: Programming Languages and Systems, Yi, K. (Ed.). Springer, New York, USA., ISBN: 9783540322474, pp: 178-194.
23. Sarimbekov, A., A. Sewe, W. Binder, P. Moret and M. Mezini, 2014. JP2: Call-site aware calling context profiling for the Java virtual machine. Sci. Comput. Programm., 79: 146-157.
24. Sarimbekov, A., P. Moret, W. Binder, A. Sewe and M. Mezini, 2011. Complete and platform-independent calling context profiling for the Java virtual machine. Electron. Notes Theoret. Comput. Sci., 279: 61-74.
25. Shirazi, J., 2003. Java Performance Tuning. 2nd Edn., O'Reilly Media, USA., ISBN: 9780596003777, Pages: 570.
26. O'Hair, K. and J.J. Heiss, 2006. The JVM tool interface (JVM TI): How VM agents work. Web Page, December 2006. http://www.oracle.com/technetwork/articles/javase/index-140680.html
27. Blackburn, S.M., R. Garner, C. Hoffmann, A.M. Khang and K.S. McKinley *et al*., 2006. The DaCapo benchmarks: Java benchmarking development and analysis. Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, October 22-26, 2006, Portland, OR., USA., pp: 169-190.