Operating System Support for Java Virtual Machine: A New Methodology of Boosting Java Runtime Performance

Shui Fu
Hainan Telcom Corp.
Hainan, China

Steve Zhang
Korea University
Seoul, South Korea

***Abstract*—Java virtual machine (JVM) provides a virtualized execution environment for Java applications. To maintain safety and efficiency of the execution environment, JVM employs some major runtime sub-systems such as garbage collection, just-in-time compilation and lock management. Recent work found that the runtime overheads increase significantly when workload of Java applications increase. To solve this problem, researchers of academia and industry have developed various approaches of utilizing VM-OS-hardware interactions to improve Java runtime performance. In this article, I discussed recent representative works in this research area.**

Keywords-Java Virtual Machine; Garbage Collection

I. INTRODUCTION

Software systems are becoming more and more complex due to multiple software layers that include application servers, shared libraries and virtual machines. Moreover, the widespread adoption of chip-multiprocessor systems also promotes the use of thread-level parallelism in today's software development. For example, the execution of an EJB application needs to go through several layers from Java application server, Java virtual machine to operating system. As a middle layer, Java virtual machine needs several major runtime sub-systems to maintain efficiency and safeness of the execution environment, such as JIT (Just-In-Compiler), GC (Garbage Collector), lock management (or thread synchronization), etc. For large EJB applications, it is common for them to employ hundreds to thousands of threads. Xian et al. studied SPECjAppServer2004 (as shown in Figure 1) [11][24][25]and found that the overhead of JVM runtime system increases significantly when workload and number of threads increase.

To reduce overhead of Java runtime systems, some techniques have been implemented inside Java virtual machines to tune performance through utilizing runtime information. For example, method invocation information has been used by the VMs to select the complexity of dynamic compilation optimizations. Heap usage information can be used for dynamically resizing heap to reduce the frequency of garbage collections.

However, the above traditional approaches have limited power of improving virtual machine performance. To support highly scalable and efficient Java application servers, *a new technology trend has emerged which makes Java virtual machine interact with operating systems and hardware supports to improve efficiency of runtime systems.* Recently, Azul Systems Inc. developed Azul Virtual Machine (AVM) [1] which removes the limitations of general-purposed Java virtual machines. AVM runs on top of customized operating system and Azul Vegas processors and it can scales up to 864 processor cores and 670GB memory heap. Similarly, there are some other research efforts to improve Java virtual machine performance by making operating systems aware of virtual machine runtime activities. In the following sections, I will present recent works on utilizing OS and architecutal support to reduce Java runtime overheads.

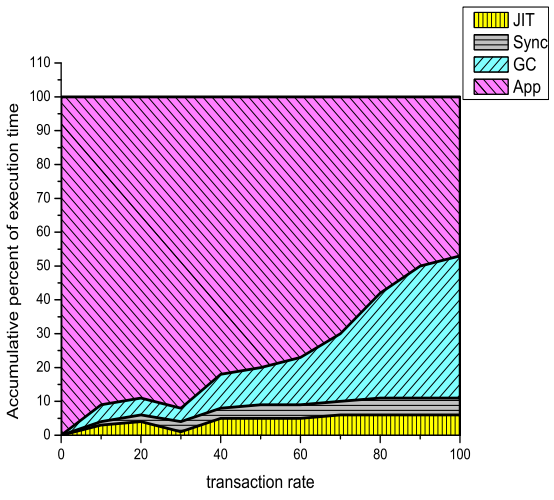

Figure 1. Accumulative time spent in major runtime functions[25]

978-1-4244-8950-3/10/$26.00 © 2010 IEEE

II. INTERACTING WITH VIRTUAL MEMORY MANAGER (VMM)

Some researchers extended virtual memory manager to interact with Java virtual machine to improve garbage collection performance. The GC performance is dependent on the heap size: if heap size is too large, it could incur paging overhead; if heap size is too small, garbage collection will be invoked frequently which causes long pause time. Since page information can be retrieved from VMM, recent GC systems introduce ways to better coordinate with VMM to guide heap resizing or GC triggering decisions.

Bookmarking Collector (BC)[5], Cooperative Robust Automatic Memory Management (CRAMM)[21][20], Yield Predictor (YP)[13] are representative works which extended virtual memory manager of operating system for GC runtimes.

The Bookmarking Collector (BC) proposed by Hertz *et al.* is a garbage collection approach which is tightly coupled with VMM of the operating system to minimize paging overhead when memory pressure is very high. The BC enables interaction between the Java virtual machine and operating system to guide heap sizing decisions. Furthermore, the BC records summary information of evicted pages and performs GC only on pages that are in physical memory without touching already evicted pages. In addition, BC guides operating system to make better decision on which pages need to be evicted.

The CRAMM proposed by Yang *et al.* [20][21][22] is a framework which dynamically adjusts heap size through coordination between VMM and Java virtual machine. In CRAMM, a new VMM is implemented to compute working-set size information of each process. The information is passed to a heap sizing analytical model [20] to predict an appropriate heap size that can improve performance as well as minimizing page faults. The heap sizing decisions from VMM is passed through to Java virtual machine and guides it to dynamically adjust heap size.

The Yield Predictor (YP) developed by Wegiel *et al.* provides a simple solution to improve GC efficiency by estimating GC yield (i.e., number of dead objects) using hardware page reference bits used by OS virtual memory manager. The YP exploits the prior empirical result that objects with similar life spans tend to be clustered in the heap and tend to die together[2][3][4][6][7][8]. These dead objects are likely in one or multiple virtual pages. Therefore, pages that have not been recently referenced by the application are likely to be dead. The YP uses a dedicated polling thread to consult the OS kernel to obtain reference bits of each page. This information is fed into a simple analytical model to estimate total number of dead pages in the heap. When the number of dead pages is small, an impending garbage collection is likely to be ineffective (i.e., unable to reclaim sufficient space) and YP will skip the ineffective collection. By reducing number of ineffictive collections, YP can reduce GC pause time.

III. MAKING OS SCHEDULER AWARE OF JVM ACTIVITIES

Modern OS schedulers are designed to maintain responsiveness and fairness, and thus, they cannot distinguish whether a thread to be scheduled is executing the application logic or performing JVM functions (GC, JIT and lock management). So, they simply treat them equally. However, JVM functions often have much different resource requirements and execution phases that often require different scheduling strategies to gain optimal performance. To solve the mismatch between resource scheduling of the OS and resource requirements of JVM functions, some researchers have built an specific OS scheduler framework which is aware of JVM activities.

A. Allocation-aware scheduler: improving GC performance

Xian *et al.* presented an allocation-phase aware scheduler [10], which exploits object allocation information to guide thread scheduling decisions. The principle of the scheduler is based on the following two insights[23][26]:

- *Objects are created and then die in phases.* Invoking garbage collection in the middle of an allocation phase usually results in ineffective collection performance. Furthermore, a long allocation pause often indicates that an allocation phase is ending and a mortality phase (i.e., a phase that most objects get used and die) is starting.
- *Commonly used thread scheduling policies degrade garbage collection performance.* In large multithreaded Java applications, existing schedulers tend to schedule threads in such a way that leaves many incomplete or partial allocation phases in the heap. Because most collectors trigger garbage collection when the heap is full, having a large number partial phases mean that garbage collection is ineffective because these objects are still alive.

In allocation-phase aware scheduler, time-based round robin is replaced with memory-based round robin, a policy that regulates the amount of memory each thread can allocate during its turn on the CPU. For example, if the memory quantum is set to 200 KB, a thread can stay on the processor until it allocates 200 KB of heap memory. At that point, the thread is suspended, and the next successive thread is scheduled. If the memory quantum is tuned to be slightly larger than the most common heap working set of a thread, it can ensure that in most cases, a thread has enough time on the processor to allocate its current working set and then executes the subsequent object mortality phase.

B. Contention-aware scheduler: reducing synchronization overhead

The emerging popularity of chip-multiprocessor systems has instigated a widespread adoption of thread-level

parallelism. In applications exploiting thread level parallelism, locks are frequently used as a mechanism to prevent multiple threads from having concurrent accesses to a shared resource and to synchronize the execution order of threads. With locks, whenever a thread attempts to acquire a lock held by another thread (i.e., lock contention), it either retries to obtain the lock (spin-locking) or is suspended until the lock becomes available.

In JVM, locks are maintained through the *monitor* mechanism. A monitor is a mutual exclusive lock. In language level, Java provides two semantics to identify monitor regions: synchronized statements and synchronized methods. These two mechanisms are supported by two special bytecode instructions (*monitor enter* and *monitor exit*) in the Java virtual machine's instruction set.

The OS scheduler does not know monitor activities in JVM. This leads to early preemption of threads in critical sections. If a thread is preempted during a critical section, other contending threads would fail acquiring the lock associated with the critical section and relinquish their time quantums. This phenomena is referred as lock convoy. The consequence of lock convoy is the overhead increase of lock contentions and context switches.

To reduce such overhead, Xian *et al.* developed a Contention-Aware Scheduler (CA-Scheduler) [12] which exploits object synchronization information to reduce the occurrences of lock convoys in large application servers running in multiprocessor systems. CA-Scheduler relies on collaborations between operating system kernels and JVM to improve the efficiency of Java runtime environments. The basic notion of CA-Scheduler is (i) to collect the necessary information as part of JVM execution, and (ii) to pass on that information to the operating system to guide its scheduling decisions.

C. JIT-aware scheduler: improving Just-In-Compilation

In Java, there is a separate compilation thread which compiles "hot" (or frequently-used) Java methods. Kulkarni et al. [17] found that round-robin scheduling affects compilation performance significantly. In round-robin scheduling, each thread including the compilation thread has a fixed amount of timeslice. If the timeslice is too small, the compilation thread cannot complete its task in one timeslice. Then the execution of that method has to be delayed until the compiler thread is done. This can take several timeslices. Even worse, as the number of threads in the application increases, the resources given to the compilation thread are reduced and the performance of applications degrades.

To solve this problem, they modified scheduler to give higher priority and a longer timeslice to compiler thread so that it can have enough time to complete the compilation, avoiding extensive execution delay. This solution can significantly reduce start-up time and improve application's performance.

IV. HARDWARE/OS SUPPORT

The Azul VM (AVM) [1] of Azul Systems Inc. is the first JVM which utilizes a synergistic hardware and software approach to improve scalability and performance of Java runtime systems. The AVM runs on the Vegas processor, a new general purpose processor designed for running virtual machines. The AVM utilizes the new architecture support to implement a highly scalable concurrent garbage collector and lock management.

Due to the strong correlation between GC pause time and the heap size, Java application servers cannot scale well in large memory footprint. Azul Pauseless Garbage Collector overcomes this by decoupling the pause time from the heap size. It is achieved by the following architectural support.

- Dedicated processor cores for garbage collection.
- Hardware-assisted read barrier instructions

The AVM implements a lock management scheme called Optimistic Thread Concurrency (OTC) which improves throughput by letting multiple threads execute synchronized code concurrently, detecting data contentions and rolling back any conflicted threads. The OTC uses similar technology as optimistic locking which is widely used in database systems and transactional memory. Since in most cases threads contending locks do not contend data that locks protect, the OTC has rare roll-back events. The OTC uses hardware-assisted write/read barrier instructions to monitor write/read accesses to data. With barrier instructions, AVM is able to detect possible data contention. If a thread is writing data and another thread has read the data, or a thread is reading data and another thread has written the data, data contention is deemed to have occurred. If a data contention has been detected for a thread, the execution of this thread will be rolled back to the state of lock acquisition point.

To improve JIT performance, the AVM implements multiple compilation threads to allow compilation to be performed in parallel. The AVM spawns on the order of 50 compilation threads during the ramp-up period of large Java application servers. Since a Vegas processor usually has hundreds of processor cores, using multiple compilation threads is an effective solution to exploit the additional resources available on the multicore architecture..

V. CONCLUSIONS

Java virtual machine provides a virtualized execution environment for Java applications. To maintain safety and efficiency of the execution environment, JVM needs employ runtime systems such as garbage collection, just-in-time compilation and lock management. Recent work found that the runtime overheads increase significantly when workload of Java applications increase. To solve this problem, researchers of academia and industry start studying various approaches which use VM-OS-hardware interactions to improve runtime performance. Table 1 summarizes the toxonamy of recent works on this direction. From the summary, most of recent works are focused on reducing GC

overheads since GC is the major bottleneck which limits the overall performance and scalability of Java applications. Researchers explored many ways of OS support such as extending virtual memory manager, making scheduler aware of JVM activities, and implementing some GC instructions. There are also a few works to reduce lock contention and JIT compilation overhead.

TABLE I. Categorization of technologies using VM-OS interactions to improve runtime performance

VM functions / OS subsystems	Garbage Collection	Just-In-Time Compilation	Lock Management
Virtual Memory Manager	-CRAMM -Bookmarking Collector -Yield Predictor	N/A	N/A
OS scheduler	Allocation-aware scheduler	JIT-aware scheduler	Contention-aware scheduler
Hardware/OS support	Azul Pauseless collector	Azul's Parallel JITs on multi-cores	Azul OTC

REFERENCES

[1] Azul Virtual Machine. http://www.azulsystems.com/technology/avm

[2] M. Hertz, S.M. Blackburn, J. E. B. Moss, K. S. Mckliney, D. Stefanovi'c. "Generatingobject lifetime traces with Merlin". Transactions on Programming Languages And Systems (TOPLAS) 28, 3(May), 476–516.

[3] M. Hertz. Quantifying and Improving the Performance of Garbage Collection. VDM Verlag, Saarbr¨ucken, Germany.

[4] M. Hertz, E. D. Berger. "The performance of automatic vs. explicit memory management". In Proceedings of the 2005 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2005), Volume 40(11) of ACM SIGPLAN Notices (San Diego, CA, Oct. 2005), pp.313–326.

[5] M. Hertz, Y. Feng, E. D. Berger. "Garbage collection without paging". In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005), Volume 40(7) of ACM SIGPLAN Notices (Chicago, IL, June 2005), pp. 143–153.

[6] M. Hertz, N. Immerman, J. E. B. Moss. "Framework for analyzing garbage collection". In R. BAEZA-YATES, U. MONTANARI, AND N. SANTORO Eds., Foundations of Information Technology in the Era of Network and Mobile Computing: IFIP 17th World Computer Congress - TC1 Stream (TCS 2002), Volume 223 of IFIP Conference Proceedings (Montreal, Canada, 2002), pp. 230–241.

[7] M. Hertz, S.M. Blackburn, J. E. B. Moss, K. S. Mckliney, D. Stefanovi'c. "Error free garbage collection traces: How to cheat and not get caught". In Proceedings of the International Conference on Measurement and Modeling of Computer Systems, Volume 30(1) of ACM SIGMETRICS Performance Evaluation Review (Marina Del Rey, CA, June 2002), pp. 140–151.

[8] M. Hertz, J. Bard, S. Kane., E. Keudel, T. Bai, X. Gu, C. Ding." Waste not, want not: Resource-based garbage collection in a shared environment". Technical report TR–951, University of Rochester. 2010.

[9] M. Hertz. Quantifying and Improving the Performance of Garbage Collection. Ph.D Dissertation. University of Massachusetts Amherst.

[10] F. Xian, W. Srisa-an, H. Jiang. "Allocation-phase aware thread scheduling policies to improve garbage collection performance". In *Proceedings of the 6th international Symposium on Memory Management* (Montreal, Quebec, Canada, October 21 - 22, 2007). ACM, New York, NY, 79-90.

[11] F. Xian, W. Srisa-an, H. Jiang. "Garbage collection: Java application servers' Achilles heel". *Sci. Comput. Program*. 70, 2-3 (Feb. 2008), 89-110.

[12] F. Xian, W. Srisa-an, H. Jiang. "Contention-aware scheduler: unlocking execution parallelism in multithreaded java programs". In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (Nashville, TN, USA, October 19 - 23, 2008). ACM, New York, NY, 163-180.

[13] M. Wegiel, C. Krintz. "Dynamic prediction of collection yield for managed runtimes". In *Proceeding of the 14th international Conference on Architectural Support For Programming Languages and Operating Systems* (Washington, DC, USA, March 07 - 11, 2009). ACM, New York, NY, 289-300.

[14] M. Wegiel and C. Krintz, "XMem: Type-Safe, Transparent, Shared Memory for Cross-Runtime Communication and Coordination", ACM Conference Programming Language Design and Implementation (PLDI), Jun, 2008 (PLDI), Mar, 2008.

[15] M. Wegiel and C. Krintz, "The Mapping Collector: Virtual Memory Support for Generational, Parallel, and Concurrent Compaction", ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Mar, 2008.

[16] M. Weigel and C. Krintz, Cross-Language, Type-Safe, and Transparent Object Sharing For Co-Located Managed Runtimes ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2010.

[17] P. Kulkarni, M. Arnold, M. Hind. Dynamic compilation: the benefits of early investing. In *Proceedings of the 3rd international Conference on Virtual Execution Environments* (San Diego, California, USA, June 13 - 15, 2007). ACM, New York, NY, 94-104.

[18] L. Zhang, C. Krintz, and P. Nagpurkar, Supporting Exception Handling for Futures in Java, ACM International Conference on the Principles and Practice on Programming in Java (PPPJ), Sep, 2007.

[19] L. Zhang, C. Krintz, and P. Nagpurkar, Language and Virtual Machine Support for Efficient Fine-Grained Futures in Java, The International Conference on Parallel Architectures and Compilation Techniques, (PACT) Sep, 2007.

[20] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. "Redline: First Class Support of Interactivity in Commodity Operating Systems". In Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation.

[21] T. Yang, E. D. Berger, S. F. Kaplan, J. E. B. Moss. "CRAMM: virtual memory support for garbage-collected applications". In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington, November 06-08, 2006). USENIX Association, Berkeley, CA, 103-116.

[22] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, J. E. B. Moss: Automatic heap sizing: taking real memory into account. In

Proceedings of the 4th international Symposium on Memory Management. Vancouver, BC, Canada, Oct 24-25, 2004. 61-72

[23] F. Xian, W. Srisa-an, H. Jiang. "MicroPhase: Proactively Invoking Garbage Collection for Improved Performance". In Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07), Montreal, Canada. Oct 21-25.

[24] F. Xian, W. Srisa-an, C. Jia, H. Jiang. "AS-GC: An Efficient Generational Garbage Collector for Java Application Servers". In Proceedings of 21st European Conference on Object-Oriented Programming (ECOOP'07), Berlin, Germany. July 30-Aug 03, 2007.

[25] F. Xian, W. Srisa-an, H. Jiang. "Investigating Throughput Degradation Behavior of Java Application Servers: A View from Inside a Virtual Machine". In Proceedings of ACM International Conference on Principles and Practices of Programming In Java (PPPJ'06), Mannheim, Germany, Aug 30-Sep 1, 2006, Page 40-49.

[26] F. Xian, W. Srisa-an, H. Jiang. "Fortune Teller: Improving Garbage Collection Performance in Server Environment using Live Objects Prediction". In Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, San Diego, CA. Oct 17, 2005. ACM Press, Page 246-247, 2005.