RC25475 (WAT1406-017) June 10, 2014
Computer Science

IBM Research Report

Active Control of Memory for Java Virtual Machines and Applications

Norman Bobroff, Peter Westerink, Liana Fong
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

Active Control of Memory for Java Virtual Machines and Applications

Norman Bobroff, Peter Westerink, Liana Fong
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, USA
$\{$*bobroff, peterw, llfong*$\}$*@us.ibm.com*

Abstract

A controller is implemented to manage memory as an elastic resource similar to computing cycles for Java applications. The controller actively arbitrates constrained memory between collocated JVMs in response to demand. A key aspect of the work is that JVM metrics are used as proxies for application KPIs so that application performance instrumentation and modeling are not required. A metric corresponding to the *allocation rate* of memory is derived from the JVM metrics and established as the measure of application performance and is used as the effective feedback mechanism to the controller. The controller is based on a fair share policy in which memory is distributed to equalize the marginal performance value to all JVMs. The design is tested for effectiveness and stability using the suite of SPECjvm2008 and SPECjbb2005 benchmarks.

1 Introduction

Matching real memory and CPU resources to the time varying memory-processor demand footprint of applications is an important element in systems performance management. Active sharing of processors between applications within and across virtual machines (VMs) in response to demand is a mature feature of the operating systems and hypervisors. Active sharing of memory (ASM) is the analogous capability where physical memory pages move seamlessly between applications and across virtual machines to satisfy demand. This improves system wide memory utilization, or alternatively increases the application density or workload intensity hosted on a compute system. ASM is sometimes referred to as logical memory overcommit as it reduces the total amount of memory necessary in a system with time varying workloads from the sum of the maximum demand of each workload to the maximum of the sum of the workloads. ASM is distinguished from paging which requires saving and restoring state in order to reuse pages from processes or VMs. Exploiting ASM requires the ability to identify unused memory in applications and operating systems and (re)map those pages to collocated applications, or move them to another VM on a common hypervisor. This function is widely available at the VM-hypervisor layer in the commercial space. But support at the application layer has been lagging as traditional application design and coding practice has not emphasized the need to dynamically return memory from the process space to the OS.

Widespread use of the Java Virtual Machine (JVM) as a server application platform creates an opportunity to extend the scope of ASM into the application layer. Emerging JVM technologies such as heap ballooning [2, 3] and dynamic heap sizing [4] provide mechanisms to release committed memory from the virtual heap space. Given these advances it is an appropriate time to visit the architecture and control functions required for an automatic ASM solution that focuses on Java applications.

This paper describes two novel aspects of JVM memory management: JVM metrics are shown to be suitable proxies for application based key performance indicators (KPI); and JVM heap memory is actively sized in response to resource changes and workload variability by equalizing the value of memory as indicated by the JVM metrics. Memory intensive benchmarks from the SPECjvm2008 [7] suite and SPECjbb2005 [6] are used to correlatefJVM metrics and application KPIs, and to evaluate the control system.

2 Background and Related Work

Figure 1 shows the platform used to investigate active memory sharing (AMS) in a virtual environment. From a logical perspective, the figure is a tree with application JVMs at the top, and the hypervisor memory pool of a physical machine (PM) at the root. One or more collocated JVMs is hosted by an operating system (OS). Each OS apportions its memory pool to processes (JVMs and other applications), free pools, and system cache. In turn, the operating systems share the common physical platform memory in the hypervisor pool. Memory flows slowly down the tree to the OS and hypervisor pools on the non-critical path, while the flow of memory up the tree is on the critical path. This paper focuses on the upper layer, fairly apportioning memory between collocated JVM based applications in response to workload changes, memory demand, and changes in OS memory.

Commercial methods are becoming available to release unused pages backing the JVM heap memory: VMWare's EM4J [8] applies a balloon mechanism that plugs into the JVM and is leveraged in the memory control work of Ginko [2]. Direct heap resizing is available in the IBM J9 JVM since version 7.0 [4]. Section 3.2 describes in detail how we leverage this JVM control knob, *MaxHeapSize* to actively control heap memory.

Recent work has studied active JVM memory sizing. Ginkgo [2] implements an application driven memory overcommitment system. Salomie [3] designs an application level ballooning controller in Xen-based environment. CRAMM [9] enables dynamically choosing of JVM heap sizes to meet workload demand, while avoiding latency in paging. QoE-JVM [5] uses an economic model for active heap sizing in the Jikes research JVM.

3 Approach

3.1 JVM metrics as proxies for application performance

There are several reasons to use JVM metrics as proxies for application performance as developed in Section 4. Most Java processes and applications do not maintain internal measures of their rate of progress. When available the interpretation of the KPI's often requires domain specific knowledge. Processing services often support a mix of incoming request types each with its own resource requirements. A shift in workload composition can change KPI's in a way that needs to be understood by the controller logic.

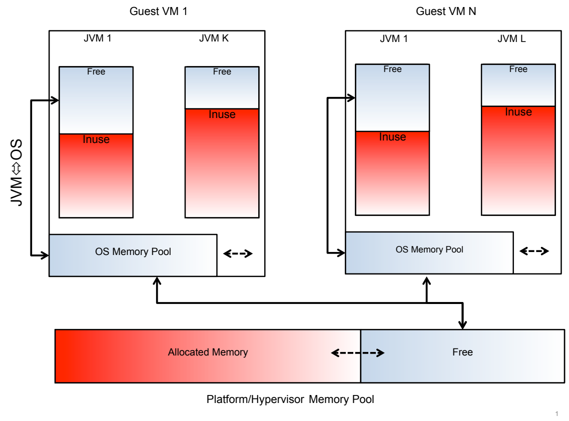

Figure 1: System view of elastic memory

End-to-end application performance depends components other than the JVM. For example, the database tier may be slowed because of insufficient OS system buffers. Note that the effect of a slow database on a Java application tier is manifest in JVM metrics such as rate of object allocation since the application can't make progress. Here, the local controller gives the JVM less memory than if the JVM tier is running at full load. This released memory makes its way to the database server via the flow of Figure 1.

3.2 JVM direct page releasing mechanism

The JVM memory control knob leveraged in this work is the *MaxHeapSize* parameter of IBM's J9 JVM. At startup J9 reserves a contiguous region of virtual process space for its heap sized by the command line argument *-Xmx* which is exposed through JMX as the immutable *MaxHeapSizeLimit* . The J9 JVM maintains a second, soft, heap maximum setting called the *MaxHeapSize* whose operation is described in Sciampacone [4]). Basically, MaxHeapSize can be set via JMX at any time during JVM execution to a value less than MaxHeapSizeLimit. When actual heap used drops below MaxHeapSize the JVM attempts to resize the heap using MaxHeapSize as the new limit.

4 JVM Metrics and Application Performance

This section analyzes the correlation between JVM metrics and workload intrinsic performance (e.g., business operations per second-bops) for memory intensive benchmarks culled from SPECjvm2008 and SPECjbb2005. The goal is to utilize the JVM metrics as proxies for application KPI's to correctly size or arbitrate JVM memory.

4.1 Metrics collected from the JVM

Several JVM metrics are exposed through the JMX API, providing a measure of how the application benefits from memory.

- Mem-freed - Cumulative number of bytes collected by the GC since a JVM startup.
- Heap-inuse - Current amount of the heap memory containing objects with live references.

Benchmark	Commit(MB)	Alloc Rate(MB/s)
SPECjbb2005	2000	2400
compiler.compiler	5000	400
derby	2000	1300
scimark.lu.large	2000	20
scimark.sort.large	1000	7
scimark.sparse.large	2000	20
scimark.fft.large	1800	20
crypto.aes	400	300
xml.validation	150	600
xml.xform	200	600
serial	420	300

Table 1: Max memory and allocation rate in benchmarks

- Heap-committed (hpCom) - Physical memory mapped to the virtual heap.
- GC CPU - The fraction of the system CPU cycles spent in GC. A decrease in GC CPU often provides an indication of whether adding memory is benefiting the application.
- Collection rate (coll-rate) - The number of GCs reported by the JVM over the sampling interval. The algorithm that determines when GC occurs is internal to the JVM.
- Allocation rate (alloc-rate) - This measure is is derived from the inuse-heap and mem-freed metrics. It is the rate of memory allocated during an interval and is computed in the sample interval $[t_1, t_2]$ using the allocated bytes;
  alloc-rate = [heap-inuse(t_2) - heap-inuse(t_1) + mem-freed(t_2) - mem-freed(t_1)] / [($t_2 - t_1$]

Allocation rate depends jointly on application demand, and the ability to satisfy the demand. Furthermore, it is a complementary measure to the GC CPU and GC collection rate metrics. If the JVM heap allocator is slowed down because of low memory, that latency translates at the application code to time spent in the Java ’new’ memory allocation operator.

4.2 Memory intensive workloads

Two benchmark groups are used to establish the correlation of the JMX metrics and workload performance. SPECjvm2008 contains over 20 individual benchmarks that cover a wide range of applications. Of these, the 10 which use more than 128MB of committed heap are considered memory intensive. The excluded set in this group use less than 50MB opt committed heap. SPECjbb2005 is representative of a traditional transactional workload.

The benchmarks selected are summarized in Table 1. They cover a range of committed heap size from 128MB to 5GB, and allocation rates from 10MB/s to over 1GB/s. All benchmarks are CPU intensive and multithreaded.

4.3 Results

The correlation between the SPECjvm2008 benchmark KPIs and the JVM metrics is explored as a function of the MaxHeapSize (MB) parameter as the JVM control knob. Figures 2, 3, 4, and 5, are representative data sets the workloads. In each figure, the SPECjvm2008 performance number (bops) recorded during the runs is shown in (a). The corresponding JVM metric averages are displayed in subplots: (b) - GC CPU (%); (c) - collection-rate (ct/s); (d) - allocation rate (MB/s); and (e) - committed physical memory (MB). These data also indicate the open loop response the MaxHeapSize input control.

The *derby* database benchmark of Figure 2(a) typifies workloads exhibiting the *threshold* memory pattern. Here, most of the gain in application performance occurs within a critical heap size, after which the value of adding memory is low.

Derby also illustrates an interesting behavior with respect to committed memory within the threshold pattern. The region at the right hand side of Figure 2(e) shows that as the heap maximum is increased beyond the critical region, the JVM continues to commit real memory and grow the heap well into the low benefit region. Doubling the memory by incrementing the MaxHeapSize control value 1GB provides less than 2% performance improvement. This *memory greedy* pattern is also observed in the scientific benchmarks grouped together in Table 1. For example, in the large FFT benchmark of Figure 3, the JVM commits about 1GB of memory beyond the point of improving performance. Systems executing this workload pattern benefit from limiting MaxHeapSize to avoid consuming physical memory.

The *xml.validation* benchmark (Figure 4(a)) also typifies the *threshold pattern*, but is not memory greedy. Figure 4 shows that the JVM only committed 150MB.

In contrast, the compiler benchmark of Figure 5 benefits proportionally to the maximum heap memory control parameter. The behavior is monotonic, but there is

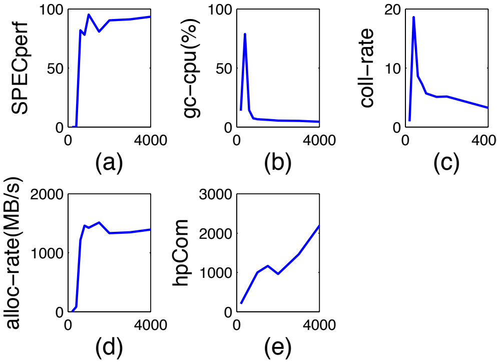

Figure 2: The SPECjvm derby workload typifies the threshold pattern and is memory greedy

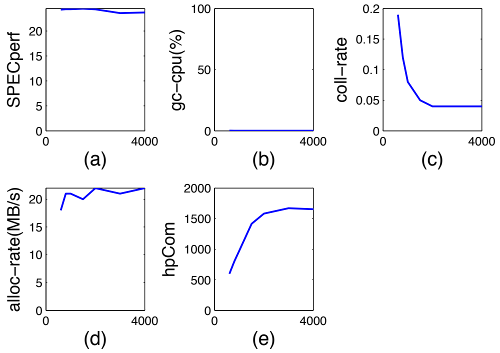

Figure 3: Scientific benchmarks such as this FFT tend to fit the memory greedy pattern.

a region of less performance gain between around 1GB and 3GB sandwiched between larger slopes of the performance - heap memory relation. The compiler performance plateaus at about 4.8GB from our experiments, just beyond the right end of the plot.

Additional insight is obtained by correlating the JVM metrics with the SPECjvm2005 and SPECjvm2008 KPI (e.g.bops). The correlation coefficients are shown in Figure 6 for the alloc-rate, gc-cpu, and coll-rate measures. The weakest correlations for all three JVM metrics are observed for the scientific benchmarks typified in the FFT benchmark of Figure 3. The lower correlation does not imply that the JVM metrics are not suited as input data to the active memory control system. In the case of the *scimark.fft.large* benchmark, the data of Figure 3 are flat and so the jitter contributes significantly to the correlation calculation.

These experiments suggest that decisions about the benefit to the application of additional memory be made on the basis of the observed change in JVM metrics as memory is added to the JVM, rather than on the metric values themselves. Consider the *threshold pattern* of Figure 2. Adding memory clearly benefits the application, as indicated by the concurrent improvement in allocation rate.

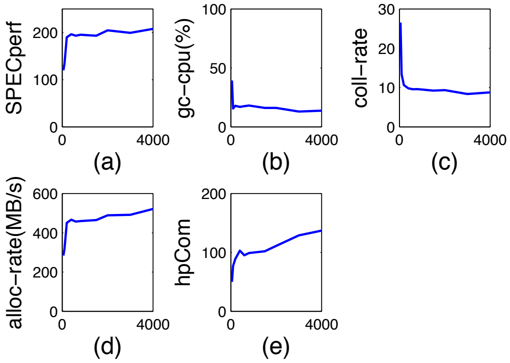

Figure 4: The SPECjvm2008 xml.validation benchmark exhibits a threshold pattern, but is not memory greedy

Figure 6 suggests the *allocation rate* is the single most consistent indicator of workload performance. The only benchmark where its correlation is significantly depressed relative to the other metrics is for the sci.lu.lg benchmark. Closed loop studies in Section 5 further validate the choice of *allocation rate* as the best single proxy for application performance.

5 Active JVM Memory Control

The objective of the memory controller is to leverage the JVM performance metrics of the prior section (esp. allocation-rate) to *fair share* the available memory between collocated JVMs. The *fair sharing* condition defines the distribution of memory between JVMs at a given workload so that: equal changes in the MaxHeapSize of each JVM result in equal changes in the relative performance of each JVM. The *relative performance slope* (S) for each JVM (j) is defined as the slope of the curve of the application performance (P_j) against MaxHeapSize, normalized by the performance value:

$$S_j = \frac{\Delta P_j}{\Delta MaxHeapSize_j} \times \frac{1}{P_j}.$$

The controller attempts to actively set $MaxHeapSize_j$ such that S_j is the same for all j.

JVM metrics are used to measure the application performance P_j. Section 4 identified *allocation rate* as a strong candidate for an application performance proxy. The open loop and offline data are now used to evaluate the JVM metrics in our slope equalizing algorithm. For example, the Specjvm2008 benchmark KPI data in Figures 5(a) and 4(a) is compared against the JVM metrics of Figures 5(b-d) and 4(b-d) in the algorithm to establish which single metric compares best to the fair sharing point given by the actual benchmark KPI numbers.

Figure 7 illustrates the equalization of relative performance slopes using data from the SPECjvm2008 *xml.validation* and *compiler.compiler* benchmarks (Figures 4 and 5). The horizontal and vertical axes are the

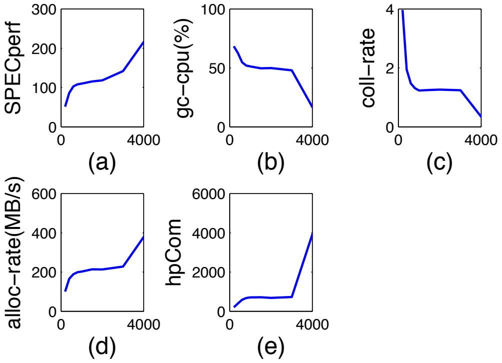

Figure 5: SPECjvm2008 compiler.compiler benefits up to about 5GB of memory

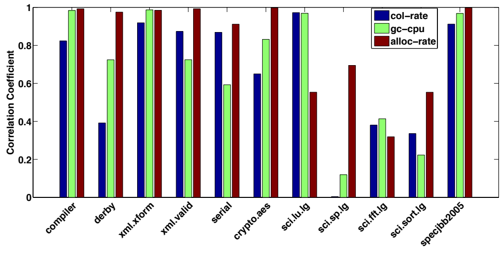

Figure 6: Correlation between SPEC performance and

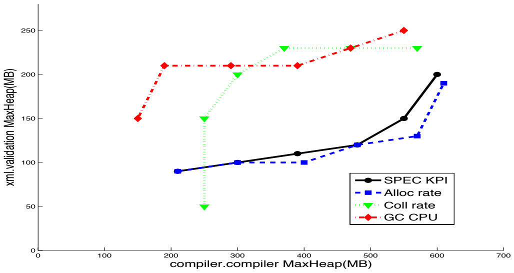

Figure 7: Memory balancing results using measured data for 3 JVM metrics and the SPECjvm performance values.

target value of the MaxHeapSize control parameter computed by the algorithm for each application JVM. Each data point represents the MaxHeapSizes that equalize the S_j for a given total memory. The total memory varies from 200MB on the left to 800MB on the right. The solid line labeled *SPEC KPI* is the reference curve showing the ideal apportionment using the application (i.e.SPECjvm2008) KPIs. The three other lines correspond to the GC-CPU, collection rate, and allocation rate metrics. The data show that the allocation-rate metric produces the closest agreement with the SPEC KPIs. This result supports the correlation analysis of Figure 6. Similar results are achieved using other workloads of Table 1.

5.1 Controller architecture

Figure 8 is a component diagram of the measure-analyze-control cycle that tracks workload memory demand and actively sets the *MaxHeapSize* parameter of each JVM. On the right of the figure is the *data collector* which uses JMX to poll the data from the JVM. The typical polling interval is 5 seconds.

The JVM metrics are fed into the control module on the left which has three logical components: the slope evaluator; the *Compute Next MaxheapSize* module that estimates the next MaxHeapSize value based on the current state; and the dither function. The data collector and controllers for collocated JVMs run in a single lightweight JVM process use less than 0.1% CPU and 20MB of memory.

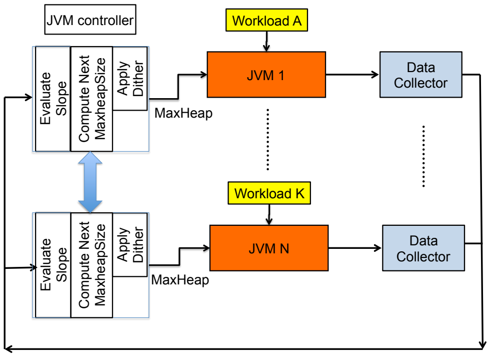

Figure 8: Main components of the memory controller.

5.2 Evaluating relative performance

A key function of the controller is to measure the slope of the allocation rate against the MaxHeapSize parameter at the current workload. To accomplish this, the controller modulates the MaxHeapSize parameter about its current target value. In this *Dithering* [10] technique, the MaxHeapSize is varied faster than the system response through a range about the target MaxHeapSize. The JVM metrics are sampled at limit points of this dithering range. Expanding the dither range too far results in oscillations in JVM performance, while reducing it yields a slower control speed. Empirically, a reasonable trade-off is achieved with the lower point at 80% of the current target MaxHeapSize, and the upper point at 120%.

Figure 9 illustrates operation of the collection of data using dithering when running the *derby* benchmark as memory is removed from the system. Each subfigure is a snapshot showing the allocation rates measured at the ends of the dither range, and at the current MAxHeapSizeTarget. The three dither points are acquired on sequential measurement cycles about 5 seconds apart. This means that in a snapshot the three points are not necessarily at 80%, 100% , and 120% of the target MaxHeapSize as the target may have changed at each measurement cycle. Consequently, the three measured dither-points in the curve window may not lie on a locally convex curve. This situation is improved by relying more on the latest measurements than on older ones.

Figure 9 shows there are critical and noncritical regions of control. In the critical region, at the bottom of the figure, the slope is steep indicating the high value of additional memory to the application. In the noncritical region at the top of the figure, memory is not as valuable. Fortunately, the main difficulties caused by noise and jitter in measuring slope occur in the noncritical region of controller operation where the slope is low.

5.3 Memory balancing methodology

The *'Compute next MaxHeapSize'* module of Figure 8 determines the next set of target MaxHeapSize values to input into the JVMs based on the current system state based on the following procedure:

1. Check the available OS memory - If it has been modified, that memory is apportioned to the JVMs according the principle of equalizing the slopes.
2. Adjust the target MaxHeapSize - The current algorithm uses an iterative, greedy procedure to estimate the new set of HeapSizeMax values that equalize the slopes. At each step, memory is moved from the JVM with the lowest slope to the JVM with the steepest slope. The iterative computation is ended under either of two conditions: i) for any JVM, memory is only changed when within the upper and lower dither points; ii) the deviation from the equal slope condition no longer improves.
3. Select the dither points for each JVM- The direction and value of the dither is chosen for each JVM so that at any time the sum does not exceed the total available memory. Figure 11 shows the phase offset between the dithering pattern two located JVMs.
4. Execute the new MaxHeapSize target for each JVM.

5.4 Experimental results

The controller is evaluated using collocated JVMs running the SPECjvm2008 derby and the SPECjbb2005 transactional benchmarks. Figure 10 shows the allocation rate of each benchmark. The total memory constraint for the two JVMs is 1.5 GB.

The system state is held constant for 580 seconds with SPECjbb2005 using 10 warehouses. The variability in allocation rate during this period is due to different phases in the underlying workload. At 580s, the number of SPECjbb2005 warehouses doubles to 20, introducing a step like change in the demand for memory. An 80s period of exponentially decaying oscillatory behavior in the control system occurs during the transition to the new operating point. The process CPU followed a similar pattern (not shown in the figure), with about a 13% shift from Derby to SPECjbb2005.

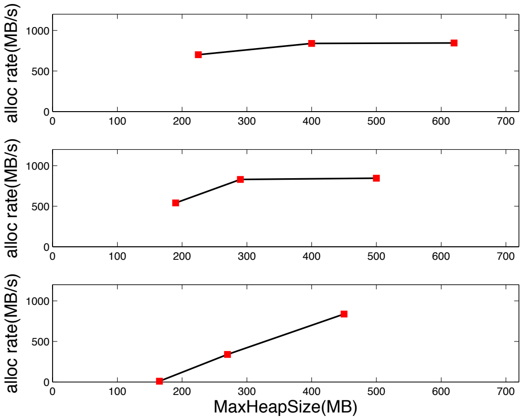

Figure 9: Snapshots of the dithering points.

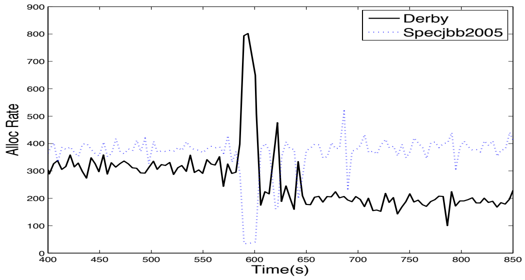

Figure 10: The allocation rate metric for collocated Derby and SPECjbb workloads.

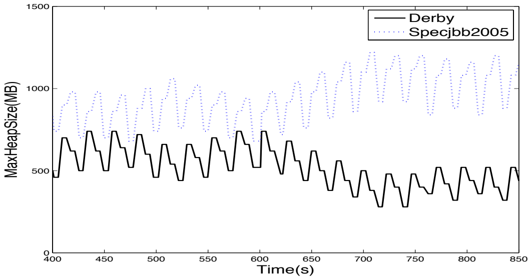

Figure 11: The MaxHeapSize control parameter including dithering for the Derby and SPECjbb workloads.

Figure 11 shows the corresponding control signal of MaxHeapSize sent to JVMs during runs. The dither signal is clearly seen imposed on the average MaxHeapSize control signal. Comparing the strength of the dither to the allocation rate data, Figure 10 indicates the dither does not affect the application performance, as desired. Experiments using the SPECjvm2008 compiler.compiler and SPECjbb2005 yielded comparable results.

6 Conclusion

JVM metrics are shown to work well as proxies for application KPIs so that application performance instrumentation and modeling are not required. This expands the applicability and ease of resource arbitration between collocated Java applications.

The control system of Section 5 is successfully applied in actively apportioning memory between collocated Java applications whose internal functions are largely unknown. Results show the response time to a step in workload intensity is of order of 80 seconds.

References

[1] Cordero, M., Correia, L., and et el. IBM PowerVM Virtualization Introduction and Configuration, June 2013.

[2] Hines, M. R., Gordon, A., Silva, M., Da Silva, D., Ryu, K. D., and Ben-Yehuda, M. Applications know best: Performance-driven memory overcommit with ginkgo. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on* (2011), IEEE, pp. 130–137.

[3] Salomie, T.-I., Alonso, G., Roscoe, T., and Elphinstone, K. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13, ACM, pp. 337–350.

[4] Sciampacone, R., Burka, P., and Micic, A. Garbage collection in WebSphere Application Server V8, Part 2: Balanced garbage collection as a new option. In *IBM WebSphere Developer Technical Journal* (August 2011).

[5] Simao, J., and Viega, L. Qoe-jvm: An adaptive and resource-aware java runtime for cloud computing. *LNCS*, 7566 (2012), 566–583.

[6] Standard Performance Evaluation Corporation. SPECjbb2005 benchmark. http://www.spec.org/jbb2005.

[7] Standard Performance Evaluation Corporation. SPECjvm2008 benchmark. http://www.spec.org/jvm2008.

[8] VMWare vFabric5 Doc Center. Elastic Memory for Java. http://pubs.vmware.com/vfabric5/index.jsp#em4j/about.html.

[9] Yang, T., Berger, E. D., Kaplan, S. F., and Moss, J. E. B. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (2006), OSDI '06, USENIX Association, pp. 103–116.

[10] Zames, G., and Shneydor, N. Dither in nonlinear systems. *IEEE TRANSACTIONS ON AUTONATIC CONTROL AC-21*, 5 (1976), 660–667.