Ginkgo: Automated, Application-Driven Memory Overcommitment for Cloud Computing

Abel Gordon, Michael R. Hines,
Dilma da Silva, Muli Ben-Yehuda,
Marcio Silva
IBM Research
{mrhines,dilmasilva,marcios}@us.ibm.com,
{abelg,muli}@il.ibm.com

Gabriel Lizarraga
Florida International University
gliza002@fiu.edu

Abstract

Continuous advances in multicore and I/O technologies have caused memory to become a very valuable sharable resource that limits the number of virtual machines (VMs) that can be hosted in a single physical server. While today's hypervisors implement a wide range of mechanisms to overcommit memory, they lack memory allocation policies and frameworks capable of guaranteeing levels of quality of service to their applications.

In this short paper we introduce *Ginkgo*, a memory overcommit framework that takes an application-aware approach to the problem. *Ginkgo* dynamically estimates VM memory requirements for applications without user involvement or application changes. *Ginkgo* regularly monitors application progress and incoming load for each VM, using this data to predict application performance under different VM memory sizes. It automates the distribution of memory across VMs during runtime to satisfy performance and capacity constraints while optimizing towards one of several possible goals, such as maximizing overall system performance, minimizing application quality-of-service violations, minimizing memory consumption, or maximizing profit for the cloud provider.

Using this framework to run the benchmarks DayTrader 2.0 and SPECweb2009, our initial experimental results indicate that overcommit ratios of at least 2x can be achieved while maintaining application performance, independently of additional memory savings that can be enabled by techniques such as page coalescing.

Categories and Subject Descriptors Operating Systems [*D*]: 4
General Terms Virtualization, Experimentation, Performance
Keywords Virtual Machines, Operating Systems, KVM, Memory Overcommit, Oversubscription

1. Introduction

The ability to overcommit different physical resources across multiple virtual machines (VMs) enables cloud providers to consolidate more VMs in a single physical host. However, simply triggering existing memory overcommitment (MOC) mechanisms in hypervisors may cause significant performance degradation in VM workloads. This may result in inadvertently reducing the memory sizes of some VMs to a level that hinders their ability to make progress. A good MOC policy should incur as little performance degradation as possible while maintaining existing performance Service Level Agreements (SLAs) between the VM hosting provider and the customer.

Memory is a very valuable, sharable resource and it limits cloud providers' ability to host more VMs in a single physical machine. Almost all modern hypervisors implement MOC mechanisms such as ballooning, page sharing, and host swapping. However, they lack policies to coordinate these mechanisms in order to minimize performance degradation as memory is deduplicated, redistributed across VMs, or swapped out. Without such policies, cloud customers may justifiably complain that they are not getting what they paid for.

When customers run a workload in the cloud, they usually request and pay for an amount of system resources that they know to be sufficient to enable their target application's performance under expected loads. For example, if they are using Amazon's EC2 infrastructure and they request the creation of a VM with the *m1.small* configuration, they expect the VM to be assigned 1.7 GB of memory. They also expect applications to behave such that their target performance indicators are met. Examples of performance indicators include *transactions/requests completed per second*, or *processing time* (not including network delay) for multi-tier applications, and also *instructions per cycle* for HPC-style applications. The growing popularity of non-relational data stores (e.g., NoSQL) might require even more exotic performance indicators. Currently, most cloud providers do not consider this diversified list of metrics as they manage the physical resources backing their cloud services. Some Platform-as-a-Service (PaaS) providers with homogeneous distributions of explicitly supported applications will focus on very specific performance indicators. Our MOC approach addresses Infrastructure-as-a-Service (IaaS) environments where the relevant performance indicators from one VM to the next are likely to differ.

In general, performance indicators for a workload will vary depending on (1) the load being submitted to the VM, i.e., how many requests a VM is required to process and (2) the amount of resources that are allocated to the VM. It is exactly these variations in load during runtime that we want to exploit for opportunities to overcommit resources. At any given time, some VMs may receive lighter loads and be able to achieve their target performance goals using fewer resources. This allows parts of their pre-allocated resources to be reallocated to VMs that require more resources in order to meet performance targets.

To appear at ASPLOS's RESoLVE workshop.

In this paper we present *Ginkgo*, a MOC framework that enables cloud providers to run more VMs in a single physical machine. This is done by redistributing memory across VMs during runtime so that (1) less memory is used overall and (2) VM performance is maintained within acceptable levels. *Ginkgo* unobtrusively monitors VM application performance indicators. Overtime, *Ginkgo* models application behavior for each VM under different loads and memory sizes. These models are used to estimate the performance impact that reducing VM memory will have. *Ginkgo* uses this information to calculate a desirable memory assignment, i.e, the amount of memory to be given to each VM so that performance is maximized while satisfying the memory capacity constraints specified by the provider. *Ginkgo* supports additional optimization objectives or constraints in addition to reduced memory usage. For example, a VM can be priced such that it maximizes cloud provider revenue or in a way that we optimize memory usage while meeting SLA levels between the customer and provider. In all these scenarios, after an objective function is chosen and sufficient amount of historical data is available about application performance, *Ginkgo* uses memory ballooning to dynamically assign memory sizes to each running VM. This results in a closed loop of actions: monitor performance for current load and memory, calibrate performance models, identify VM memory assignments which satisfy those constraints and maximize an objective function, deploy memory assignments, and then repeat. This loop allows *Ginkgo* to regularly readjust memory assignments so that each VM will continue to receive only what it needs as load varies.

The main contributions in this work are:

- The design and implementation of a MOC framework that maximizes application performance while minimizing memory consumption. Using this framework we were able to effectively achieve an overcommit ratio of at least 2:1 with little degradation in performance. These results were obtained without enabling any page coalescing mechanisms, so additional memory savings can still be realized.
- The performance evaluations of two benchmarks, SpecWeb 2009 and DayTrader 2.0, illustrating the non-linear relationship between memory and performance. We demonstrate how this relationship can be inferred and exploited for efficient memory overcommitment.

The rest of the paper is organized as follows. Section 2 provides a short description of the design and implementation of *Ginkgo* and Section 3 presents the result from our initial experimental evaluation. Section 4 reviews related work. Section 5 summarizes our MOC approach and discusses possible paths for improving the proposed framework.

2. Design and Implementation

The *Ginkgo* framework is both a modeling and control system for sizing VM memory in such a way that a cloud provider can maximize a chosen aspect of overall system behavior while maintaining the service level agreements specified for a VM's application. The framework is hypervisor-agnostic in the sense that it only requires hypervisor support to dynamically control the memory size of a VM, for example the ballooning mechanisms available in VMware ESX server [19], Hyper-V [13], Xen [1], and KVM [4]. *Ginkgo* consists of monitoring, modeling, and decision-making components. Figure 1 shows a high-level view of the framework and the interaction between its components.

The framework's components were written to address four different tasks simultaneously:

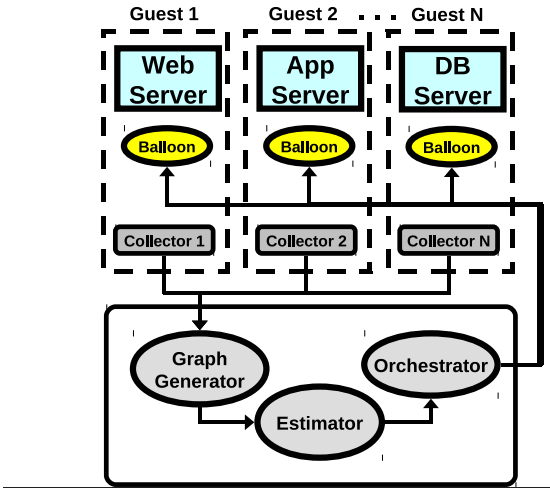

Figure 1. *Ginkgo* System Architecture

1. **Collectors**: transparent, transport-based agents (HTTP, JDBC, or logs) that continuously poll virtual machines for application performance indicators;
2. **Graph Generator**: constructs a memory performance model based on collected values for memory size and performance.
3. **Estimator**: using information from the continuously-updated performance model graphs, the Estimator uses a linear program to make global optimization decisions, i.e., to identify a feasible memory assignment that optimizes an object function, described later;
4. **Orchestrator**: deploys the Estimator's decision, i.e., the identified memory assignments to each VM. This is executed by using the hypervisor's memory ballooning mechanism.

The *Ginkgo* framework can run either in the same server hosting the VMs or on a separate networked machine. It is able to manage any virtualization layer that exposes an API to control memory allocation. In our prototype we used the KVM hypervisor to host the virtual machines and the libvirt API [9] to invoke the balloon driver. Conveniently, libvirt exposes a TCP control channel as well as supports a long list of hypervisors. Thus, *Ginkgo* is able to control any libvirt-supported hypervisor out-of-the-box without even logging into the system under test.

The construction of models to correlate resources (e.g. VM memory size) to VM performance becomes more complex if the models have to handle scenarios with highly dynamic variations in demand of VM services. We chose to address this issue in *Ginkgo* by characterizing a VM memory-to-performance relationship on a per-load basis. This means that the demand for VM services is monitored and mapped to an application-specific load classification. For example, the load for a web server can be classified in terms of the number of simultaneous sessions.

By continuously profiling and updating memory versus performance graphs for different applications and loads, *Ginkgo* is able to characterize the application performance behavior for different memory allocations. With this information, *Ginkgo* can now pursue its memory overcommitment objectives by determining how much memory each VM actually requires to deliver good performance under its current load.

After *Ginkgo* has calculated a memory assignment for each VM, *Ginkgo* still needs to apply these assignments to each VM. The orchestrator computes the difference between the current memory assignment and the target memory assignment, gradually inflating

and deflating the balloon in incremental steps (a configurable value defined to be 64MB in our experiments) until it achieves the target assignment for every virtual machine. The delay between steps is also a configurable parameter, set to 1 second in our experiments.

3. Evaluation

In this section, we evaluate *Ginkgo*'s ability to overcommit memory without degrading application performance. We use Daytrader 2.0 and SpecWeb 2009 to represent cloud workloads. Our setup consisted of two IBM x3550 M2 machines, configured with 2 quad-core Intel Xeon 2.9Ghz CPUs (16 cores with hyperthreading enabled). One of the machines, configured with 64GB of RAM was used as the system under test, while the other, configured with 24 GB of RAM was used as the load driver. On both machines, we used Redhat Enterprise Linux 6.0 as our host OS and KVM to run the virtual machines. To increase memory pressure, we disabled page coalescing support (KSM) in all our experiments — despite the fact that our linear model already supports it. To avoid any artifacts due to throttling on cores eventually submitted to lighter loads, we also disabled all cpu throttling support (e.g., cpufreq) that comes enabled by default on RedHat linux. The guest images are Ubuntu Linux 9 and were placed in an IBM XIV storage system accessed using the iSCSI protocol over a 1Gb Ethernet interface. Our load driver machine is directly connected to the system under test via a dedicated 10Gb Ethernet interface (crossover-cable). *Ginkgo* itself is written in java and uses the java bindings from the libvirt API. These come standard with a Redhat 6 installation. *Ginkgo* runs on the the load driver machine and is able to operate remotely because our collectors use http and jdbc connections to query the application and also because we submit balloon adjustments using libvirt over TCP/IP.

Profiling Phase

Before we can evaluate the effectiveness of *Ginkgo*, we need information on the expected VM performance for each of the workloads. For this purpose, we start running multiple instances of SPECweb and Daytrader at different loads levels (as much as a single VCPU can handle). Then, we incrementally change the amount of memory assigned to the VMs from 2.0GB to 0.25GB to see how memory variations affect performance. In general, we profile by having *Ginkgo* make balloon adjustments at increments of 128 megabytes and waiting several minutes before making the next adjustment. At the end of the profiling phase, we store the performance measurements into *Ginkgo* and use them later for benchmarking real scenarios. Again, note that as we described previously, data collection and graph generation continue to occur even after the profiling phase has ended.

When the workloads run in a stand-alone configuration they do not share physical resources with other guests running in the same physical host. Previous work [15] has shown degradation in application performance when multiple VMs share CPU and I/O, even when they receive 100% of their pre-allocated quota for these resources. To create memory performance graphs that include the interference of multiple virtual machines running in the same host, we perform the profiling phase running 10 SPECweb stacks (20 VMs) and 10 Daytrader VMs simultaneously.

We use the Banking version of SpecWeb 2009 [18] deployed in two different virtual machines - one running the web server and one running the backend database, configured with 1 VCPU and 2GB of memory each. We profile each 2-VM pair (stack) of SPECweb VMs from loads ranging from 10 to 100 simultaneous sessions. Even though our hardware can handle as many as 900 sessions when a single stack is running, we do not stress SPECweb with this high load to avoid CPU and I/O bottlenecks when running multiple stacks. We use the average server processing time during a 30-second time period as a performance indicator. We have a loadable apache module which simply outputs over HTTP the time taken to process each URL, averaged over 30 seconds. We do not include network delays in the reported values, as they are not directly affected by changes in memory pressure. For this particular experiment, the application — SPECweb front-end (apache) — operates best with 1.5 GB of memory or more, but can still function well with less depending on the incoming load. Processes running inside the web server do not consume all the physical memory - however the remaining memory is used by the guest's Linux kernel to cache I/O operations and thus increase application performance.

Daytrader is an online stock trading system benchmarking application. It is a multi-tier system, with a web front end, a J2EE application layer and a DB layer. We deployed Daytrader 2.0 in a single virtual machine with 2 VCPUs and 2GB of memory, running on top of Apache Geronimo Application Server using the Apache Derby DB, populated with 40,000 quotes and 2,000 customers. We configured Daytrader's parameters to use Session EJB3 and asynchronous 2-phase order-processing. The java maximum heap size was configured not to exceed the initial allocation of memory for the VM. To measure Daytrader performance, we profile each instance from loads ranging from 10 to 20 simultaneous requests per second. The collector parses tomcat logs and calculates the average processing time of the requests served in the latest 30 seconds.

Cross-workload Comparisons

As we described in the previous section, each collector can use a different metric to represent application performance. These metrics can have different meanings and limits. For example, when *Ginkgo* monitors processing time, smaller values are better. In contrast, if *Ginkgo* monitors requests per second, bigger values are better. Absolute values for different workloads might have different implications. While for DayTrader, 700ms processing time might represent good performance, for SPECweb similar values could be catastrophic.

Ginkgo needs to compare different workloads and identify what is good and what is bad to make a global memory allocation decision. To solve this problem we normalize the memory performance graphs, where 100% always represents the best performance indicator ever seen for each load level. In addition, for each workload we define a 0% performance indicator. Based on the absolute values we observed, we considered any processing time value higher than 750ms, 2500ms and 7500ms as 0% performance for the SpecWeb front-end, back-end and Daytrader respectively.

3.0.1 Baseline Load Generation

Coming up with some baseline degree of load variation is important for testing real virtual machines in a controlled, overcommitted environment. Note that it is not our intent to try to characterize the meaning of a load "spike" or the effect of different loads on CPU performance. Thus, in order to simplify the problem and still be able to assess *Ginkgo*'s effectiveness, we generate what we call a "load plan", consisting of a set of time steps for all of the VMs running on the server such that each virtual machine receives a specific load from our testing infrastructure. Variation at each time step is done in the following manner:

1. First we take the range of loads ever seen for each workload (from our profiling phase) and statically assign to the guests

running such workload a virtual CPU capacity (i.e., number of vCPUs) that is sufficient to avoid a processing bottleneck.

2. For each timestep, we randomly choose one of the loads in the observed range and one of the running VMs on the system under test, assigning that load to the chosen VM iteratively until one of two conditions happens: either we run out of VMs to assign or we run out of available host CPU capacity, whichever comes first.

This simplistic assignment of loads to VMs gives us enough load variation over time for testing the *Ginkgo* system. To our surprise, as we will see later, there were occasionally some spikes in the generated load plan, but most of the time the aggregate load on the system remained steady while loads for individual virtual machines went up and down dynamically. We generated this type of load plan consisting of 20 SPECweb virtual machines (10 for each tier) and 10 virtual machines for DayTrader - 30 virtual machines total. Recall that each VM has 2GB of memory and the total memory is 64 GB, so no memory pressure is permitted in the baseline load generator.

Finally, before the load plan is actually generated, we "prime" the *Ginkgo* database with a baseline of CPU performance data from which to start. So, we run our benchmarks on a system with 64GB of memory all available and vary the load until there is no more CPU capacity available. At that point, we stop the baseline experiment and activate *Ginkgo* as described in the next section.

3.0.2 Memory overcommit with *Ginkgo*

To measure *Ginkgo*'s effectiveness at maximizing performance over the baseline (which has a constant performance of 100% for the whole load range), we now limited our host to use only 32GB of memory by setting the kernel to boot 32768M of RAM. We also configured the Estimator component's linear model to limit the total allocatable memory to 28GB. Thus, for both the baseline and the real experimental scenario, 4GB of memory were left to be used by the host. Then we simultaneously ran all 30 virtual machine instances with the aforementioned loadplan, both with only half the physical memory (w/ *Ginkgo* running and re-allocating physical memory) and with all the physical memory available (without *Ginkgo*).

In Figure **??** we show the results obtained when the host used only 32GB and *Ginkgo* managed the memory allocations. The x-axis represents run time, divided in steps of 10 minutes. For each step, the y-axis shows the normalized values for different metrics, including load, performance, memory, and CPU activity. We can see that *Ginkgo* was able to maintain near maximum performance while overcommitting memory by a factor of 2.14x (60GB/28GB). Figures **??**, **??**, and **??** show a similar timeline for individual VMs running different workloads. One can see how *Ginkgo* adjusts the memory allocation for each VM according to the incoming load.

4. Related Work

Issues related to overcommitment of resources in virtualized environments have been investigated from several angles, building up on the extensive list of results from resource management research.

Hypervisor-level enhancements to memory management have been proposed to allow for a better understanding of how VMs use their memory: the Geiger project [8] introduces techniques for the hypervisor to infer guest OS buffer cache behavior; Lu and Shen propose an efficient method for tracing VM memory access that can be used to predict the VM miss rate for a given memory size assignment [10]; Magenheimer *et al* proposed the idea of a

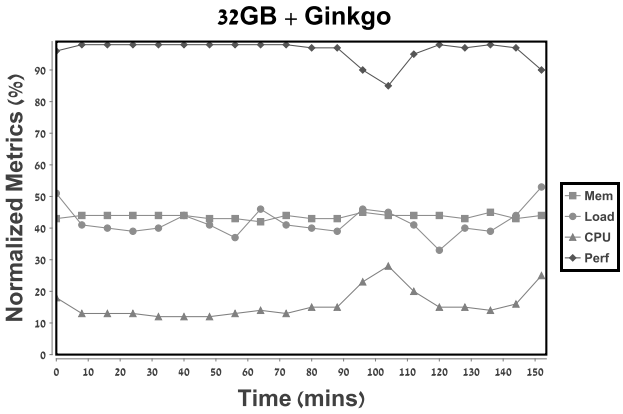

Figure 2. Aggregate 2.14x overcommitment performance with 32GB host memory and *Ginkgo* enabled.

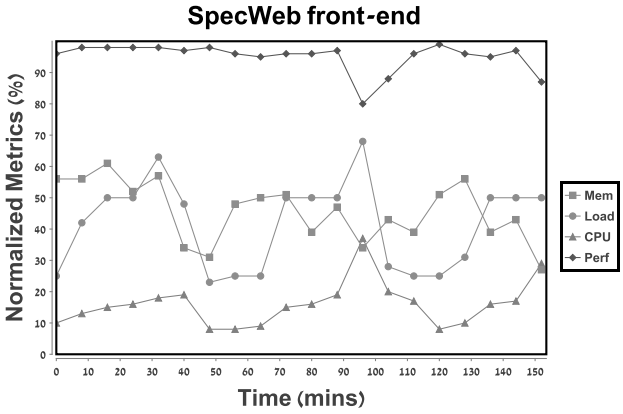

Figure 3. Individual SpecWeb VM front-end *Ginkgo* performance with 32GB host memory.

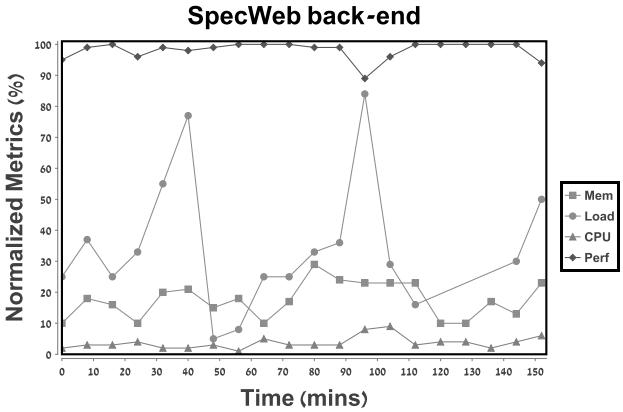

Figure 4. Individual SpecWeb VM back-end *Ginkgo* performance with 32GB host memory.

hypervisor-based cache [11]; Antfarm [7] introduces techniques to enable the hypervisor to gain knowledge about processes within the hosts. *Ginkgo* takes a different approach to model VM memory usage: it targets environments in which application-level performance indicators for the VMs are available, thereby directly correlating memory size and application behavior.

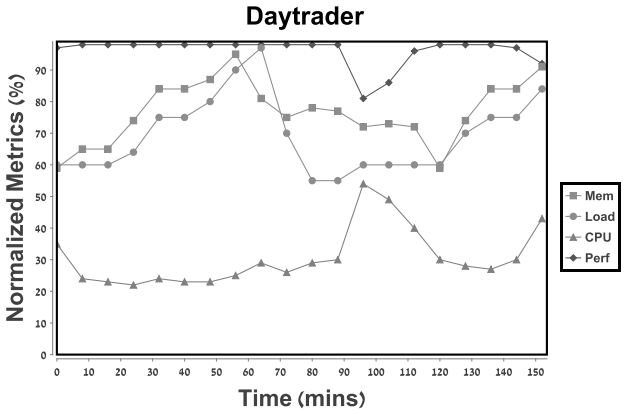

Figure 5. Individual Daytrader VM *Ginkgo* performance with 32GB host memory.

In [2, 3, 6, 12, 14, 17, 19], methods are proposed for reducing memory usage by leveraging redundancy of memory among VMs. Wood *et al* propose a method for VM placement that collocates VMs in servers based on their page sharing potential [20]. These memory saving techniques are orthogonal to *Ginkgo*. In some cloud environments, the runtime page content commonality of VM images may be large enough to enable a reasonable memory overcommitment ratio, but many data-intensive workloads (such as database servers) may limit the impact of page coalescing techniques.

In order to dynamically adjust the amount of memory given to a specific VM, researchers have looked at the behavior of applications running inside the VM. Waldspurger presents a page sampling technique to infer the memory needs of a VM [19]. Zhao and Wang proposed a scheme for dynamic VM memory balancing [21] where VM memory needs are estimated by monitoring swap space and intercepting VM memory accesses to maintain a LRU histogram. Padala *et Al* propose *AutoControl*, an automated method to dynamically control and manage resources in a virtual environment by estimating the relationship between applications and resources. It combines an online model estimator and a multi-input, multi-output (MIMO) resource controller [16]. *Q-Clouds* [15] also uses online feedback to construct a MIMO model to perform CPU resource management to mitigate the performance interference effects from a virtualized environment. Neither *AutoControl* nor *Q-Clouds* directly address memory.

Heo *et al* present a dynamic memory controller that uses feedback control to dynamically adjust the allocation of CPU and memory while maintaining service level objectives [5]. This approach differs from *Ginkgo*'s in that it does not attempt to characterize incoming load so that load-specific controllers could be deployed. Also, their experimental evaluation was based on synthetic workloads and real world traces, while *Ginkgo* deploys actual workloads. In terms of scale, Heo's approach was demonstrated in a scenario with 4 VMs, while *Ginkgo*'s results presented in Section 3 involve dozens of VMs. *Ginkgo* has been designed to scale to hundreds of VMs; so far we have run *Ginkgo* experiments involving 64 VMs.

5. Conclusions and Future Work

Managing memory overcommitment mechanisms is a complex task and must be done carefully to avoid performance degradation. As we shown in this paper, application performance metrics can be used to infer memory needs and calculate efficient memory assignments.

In *Ginkgo* we used a white-box approach to monitor the application, assuming we have some limited monitoring support from virtual machines images. While this kind of solution easily fits Platform-as-a-Service clouds, Infrastructure-as-a-Service clouds may require a different and non-intrusive solution. To address this problem, we plan to infer application performance indicators using monitors running at the hypervisor level. For example, the numbers of requests and response time of a web server can be estimated by observing incoming and outgoing TCP connections/packets.

Albeit treated as white-box regarding the performance metrics, the application is still seem as a black-box regarding service correctness under extreme conditions. In our experiments we were able to detect a small minority of corner cases where extremely low memory under high load turned the performance metrics reported by the application unrealiable due to silent errors. We plan to implement a "sanity checking" mechanism for the values reported (e.g., the performance cannot improve suddenly as we take away memory for a given load) in order to discard spurious data collected.

Furthermore, it may be the case that some cloud customers *are* willing to actively participate with the Ginkgo system, given the ease of its use. The need to export a performance metric in a standardized way is a useful feature for many modeling systems, not just ours. If, perhaps, a number of customers could choose a common transport protocol which is simply and allows for polling over, say, HTTP - that would be all that is required for this framework to function correctly.

In this work we showed how to overcommit memory based on application performance. In the future, we plan to extend *Ginkgo* to allocate CPU and I/O by creating multi-dimensional performance model graphs and extending the linear model to consider assignments for each sharable resource.

Hypervisor support for memory compression was presented in Difference Engine [2] and was recently incorporated to VMware vShpere 4.1. While this technology can increase memory overcommit ratios, it also impacts application performance. We can extend *Ginkgo* to profile and model 2-dimensional memory assignments, where the first dimension represents the amount of physical memory each VM needs and the second dimension the portion reserved for compressed memory.

We can also use *Ginkgo* to improve and accelerate the deployment of new virtual machines. At any given point in time, the estimator can consider the new VM and simulate a memory assignment, even when the VM is not actually running in the host. By looking at optimization results, we can estimate performance and memory consumption or profit in cases where the host absorbs the VM. Management software can then use this information to decide whether or not start running the VM.

On the other hand, *Ginkgo* is already capable of modeling situations where VM migration features are enabled in the cloud. If the estimator component is configured with a linear model that allows VMs to be exempt from receiving a memory assignment and defines a corresponding penalty for this case, it can then identify a more flexible subset of VMs that should run on the host which would still improve the object function value (overall performance or profit).

Although our evaluation of *Ginkgo* chose to maximize performance, in some scenarios we don't need to achieve maximum performance to pass quality of service goals. We can define "acceptable" performance for each workload and give the minimum amount of memory required to satisfy this constraint. This definition for acceptable performance can be provided by the customer or the provider. However, without previous profiling and run-time modeling, the cloud provider would not be able to estimate the memory requirements needed to host a workload without noticeable violations. Alternatively, providers might offer service

category levels, such as bronze, silver, and gold to avoid defining these values in absolute terms. Based on expected profit or maximum allowed resources, *Ginkgo* can be used to estimate what acceptable performance values might be for each category. These values can even be updated on a daily or weekly basis, according to runtime/history measurements. By publishing these updated values, the provider can give customers the opportunity to upgrade/downgrade the QoS category assigned to a VM or to further optimize their own workloads.

Acknowledgments

The author Gabriel Lizarraga appreciates the support provided by the National Science Foundation under grants HRD-0833093, CNS-0837556, CNS-0540592, and CNS-0426125.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[2] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *8th USENIX Symposium on Operating System Design and Implementation (OSDI 2008)*. USENIX, December 2008.

[3] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Commun. ACM*, 53(10):85–93, 2010.

[4] I. Habib. Virtualization with kvm. *Linux J.*, 2008(166):8, 2008.

[5] J. Heo, X. Zhu, P. Padala, and Z. Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *IM'09: Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, pages 630–637, Piscataway, NJ, USA, 2009. IEEE Press.

[6] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–12, New York, NY, USA, 2009. ACM.

[7] S. T. Jones, Andrea, and Remzi. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 1–14, Boston, MA, USA, 2006.

[8] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. *SIGOPS Oper. Syst. Rev.*, 40(5):14–24, 2006.

[9] Libvirt Team. libvirt: The virtualization api. http://libvirt.org.

[10] P. Lu and K. Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–15, Berkeley, CA, USA, 2007. USENIX Association.

[11] D. J. Magenheimer, C. Mason, D. McCracken, and K. Hackel. Paravirtualized paging. In *Workshop on I/O Virtualization*, 2008.

[12] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis. Efficient resource provisioning in compute clouds via vm multiplexing. In *ICAC '10: Proceeding of the 7th international conference on Autonomic computing*, pages 11–20, New York, NY, USA, 2010. ACM.

[13] Microsoft Technet. Hyper-V dynamic memory evaluation guide. http://technet.microsoft.com/en-us/library/ff817651)WS.1.0).aspx, July 2010.

[14] G. Milos, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened page sharing. In *USENIX 'ATC*, 2009.

[15] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 237–250, New York, NY, USA, 2010. ACM.

[16] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26, New York, NY, USA, 2009. ACM.

[17] M. Schwidefsky, H. Franke, R. Mansell, H. Raj, D. Osisek, and J. Choi. Collaborative memory management in hosted linux environments. In *OLS '06: 2006 Ottawa Linux Symposium*, 2006.

[18] Standard Performance Evaluation Corporation. SPECweb2009. http://www.spec.org/web2009/.

[19] C. A. Waldspurger. Memory resource management in vmware esx server. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 181–194, 2002.

[20] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 31–40, New York, NY, USA, 2009. ACM.

[21] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 21–30, New York, NY, USA, 2009. ACM.