Applications Know Best: Performance-Driven Memory Overcommit With Ginkgo

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

***Abstract*—Memory overcommitment enables cloud providers to host more virtual machines on a single physical server, exploiting spare CPU and I/O capacity when physical memory becomes the bottleneck for virtual machine deployment. However, overcommiting memory can also cause noticeable application performance degradation. We present Ginkgo, a policy framework for overcomitting memory in an informed and automated fashion. By directly correlating application-level performance to memory, Ginkgo automates the redistribution of scarce memory across all virtual machines, satisfying performance and capacity constraints. Ginkgo also achieves memory gains for traditionally fixed-size Java applications by coordinating the redistribution of available memory with the activities of the Java Virtual Machine heap. When compared to a non-overcommited system, Ginkgo runs the DayTrader 2.0 and SPECWeb 2009 benchmarks with the same number of virtual machines while saving up to 73% (50% omitting free space) of a physical server's memory while keeping application performance degradation within 7%.**

I. Introduction

Cost reduction is a key benefit of cloud computing that continuously attracts more users and applications into consolidated mega data-centers. Therefore, to keep pace with an explosively expanding business, cloud providers must minimize infrastructure and operating costs through efficient, dynamic, and automated resource management. When memory is the primary bottleneck, one way to achieve this goal is through overcommitment, reclaiming unused or duplicated memory to pack more Virtual Machines (VMs) into a physical server.

In this work, we take an application-driven approach to explore how memory can be shared effectively, especially as CPU and I/O capacity increases rapidly due to multicore and high-throughput interconnect technologies. Overcommitting memory remains a significant technical challenge despite its importance [1], [2], [3], [4], [5], [6], [7], [8], due to the fact that applications should be able to perform as well as if each VM had dedicated memory. Furthermore, other management activities (like page caching and swapping) exhibit non-linear relationships between memory and performance, such as when the same memory page is cached or swapped by both the guest operating system (OS) and the hypervisor. The hypervisor needs to distribute physical memory across the virtual machines without causing dramatic application performance degradation. Otherwise, cloud customers would notice they get less resources than they paid for.

In this paper we present Ginkgo, an application-driven memory overcommitment framework that allows cloud providers to run more VMs on a single physical server. This framework redistributes memory across VMs during runtime so that (1) less memory is used overall and (2) application performance is maintained within acceptable Service Levels. This process is comprised of two stages, an off-line profiling stage and a production stage. The profiling stage continuously maintains a performance-to-memory correlation model by sampling application metrics under a variety of memory configurations and loads. We use this model at the production stage to decide the least amount of memory required to provide agreeable application performance.

The main contributions of this work are:

(1) The design and implementation of a hypervisor-independent memory overcommitment framework that provides memory gains with minimal application penalties. (Sections II, III, and IV). Without even using page de-duplication, Ginkgo saved up to 73% (50% omitting free space) of physical memory with less than 7% of performance degradation.

(2) A mechanism to coordinate the heap size growth and shrink of unmodified Java Virtual Machines (JVMs) with the existing VM balloon driver for Java applications. (Section V).

(3) The evaluation of Ginkgo using the benchmarks DayTrader 2.0 (Websphere v7, DB2 v9) and SPECWeb 2009 (Apache), illustrating the non-linear relationship between memory and performance. We demonstrate how this relationship can be inferred and exploited for efficient overcommitment. (Sections VI and VII).

II. The Ginkgo Framework

Ginkgo is designed to grant cloud providers the ability to dynamically redistribute memory among VMs while maintaining application performance within acceptable service levels. Our approach rests on the observation that depending on the incoming load a VM can achieve the agreeable application performance with less physical memory than it is currently using. This observation implies that Ginkgo needs to identify a memory-to-performance "sweet spot", where "just the right amount" of memory is given to reach a certain level of application performance for the current incoming load.

Ginkgo is implemented as a closed control loop with autonomic characteristics. For each VM, Ginkgo builds a performance model by correlating regular samples of application performance, memory usage, and submitted load. (In this work, clients are capable to inform the submitted load, in order to speed up the building of an accurate model.) Then, this model is used to calculate how much memory each VM

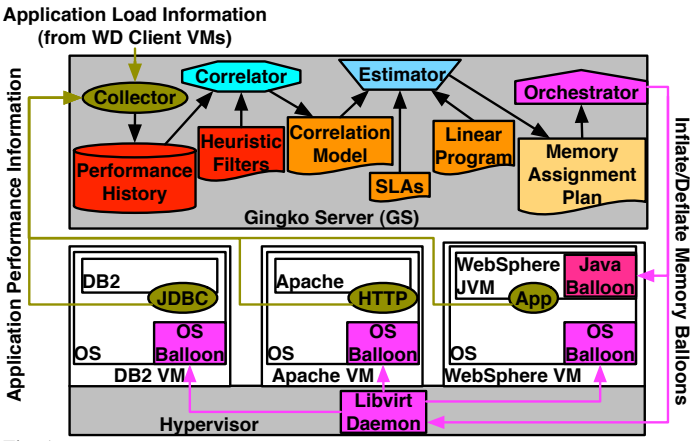

Fig. 1. A view of a sample usage of the Ginkgo framework, including its components and one hypervisor, two Stacks, and three Tiers

needs (memory assignments), also taking into account specific criteria supplied by the cloud provider (e.g., a minimum performance level required by the Service Level Agreement or SLA). Finally, these memory assignments are applied directly to each VM and the process is repeated.

Application performance monitoring is used as the fulcrum of Ginkgo's memory assignment decisions. Because of this, the framework is hypervisor-agnostic in the sense that it only requires hypervisor support for dynamic VM memory resizing (e.g. ballooning [3]), which is available in all modern hypervisor architectures, including VMware ESX Server [3], Xen [9], Hyper-V [10], and KVM [11]. For our prototype, we use the KVM hypervisor on Red Hat Enterprise Linux 6 to host the VMs.

We group cloud elements in the following categories, as depicted in Figure 1:

1. **Stacks**: The logical grouping of VMs servicing client's requests (e.g., 1 WebSphere VM + 1 DB2 VM form a DayTrader Stack);
2. **Tiers**: A key or tag that uniquely identifies the hardware and software on a VM that is part of a Stack. Our framework creates performance models on a per-Tier basis to track divergence in hardware and software versions which might spawn new models.
3. **Workload Driver**: The set of clients that generate a stream of requests directed to a given Stack;
4. **Hypervisors**: The set of physical hosts that house one or more provisioned Stacks;

The Ginkgo framework itself has different components interacting with the aforementioned elements:

1. **Collectors**: threads that poll each Tier of a Stack for performance data;
2. ***Correlator***: analyzes the collected data and constructs a set of models correlating observed application performance to a given memory size and submitted load level;
3. ***Estimator***: uses the *Correlator*'s generated models to compute a set of feasible per-VM memory assignments that optimize an objective function (e.g. maximum performance or minimum memory usage);
4. ***Orchestrator***: an actuator that establishes a connection to a cloud hypervisor and applies the new memory assignments through existing memory ballooning mechanisms;

The quality of the memory assignments from the *Estimator* is highly contingent upon an accurate and extensive memory-to-application performance correlation model. Before a production hypervisor is enabled for overcommitment, Ginkgo first does an initial phase of experimental profiling. In a larger, production cloud, one could generate these models dynamically by only ballooning free space and page cache space as workload conditions change over a long period of time. In this work, to explore the potential of fully populated models for a wide range of memory sizes and produce an optimal model, we generate them offline. In the future, a "library" of such models could be maintained by the cloud provider, with the appropriate model assigned to a new VM at provisioning time. This online model selection could be revised over a long period of time.

III. Correlating Performance and Memory

Ginkgo creates and maintains models to correlate between performance and memory. The experiment space for these models is populated by multiple memory sizes for each load level, with a fixed number of VMs per hypervisor. We generate models for each target application stack. For instance, the DayTrader Stack is modeled by recording its performance while submitting load levels up to 8 simultaneous request streams and varying the memory sizes from 2 GB to 250 MB (in decrements of 32 MB). The entire Stack is submitted to various load levels at each memory size for long periods of time (at least an hour per memory size), with application performance sampled at short intervals (30-60 seconds) - producing hundreds to thousands of measurements.

Since the behavior patterns for individual Tiers in a Stack have markedly different characteristics, we have defined three types of Collectors:

1. **Web Collectors**: a wget operation that queries a small loadable Apache module over HTTP. The module reports the average time needed by the server to process URLs (not including network delay);
2. **DB Collectors**: a query that uses the Java Database Connectivity (JDBC) interface to obtain the number of transactions completed within a certain time;
3. **App Collectors**: an XML-based wget parser (also over HTTP) that extracts response times from an application server. In the case of WebSphere, this XML comes from the built-in Performance Monitoring Infrastructure (PMI).

As a concrete example, load submitted to the DayTrader Stack results in two different models populated respectively by two different Collectors. Client requests (i.e., load) always arrive at the Stack through the topmost Tier (e.g., WebSphere VM). We assume that increasing (or decreasing) the load on the topmost Tier will likewise do the same on the other Tiers. This architecture neither requires Tiers to be aware of Ginkgo itself nor make any other Ginkgo-specific changes

for monitoring. As long as there is some method or interface for periodically querying application performance, Collectors can be independently invoked to provide performance values. Enterprise applications typically already have such interfaces. Collectors are also resource-agnostic, in the sense that the same Collector can be used to monitor performance regardless of which specific kind of resource (e.g., memory, CPU, I/O) is most relevant for a given Tier.

With this data, the first step in the production of a correlation model for each Tier is the summarization of the performance samples obtained for each pair of submitted load level and memory size. The *Correlator* currently employs two methods for summarizing data. The first method produces weighted moving averages, commonly used with values reported as throughput measurements, such as those produced by the DB Collector. The second method uses the 95^{th} percentile function, which is useful for application performance values reported as response times, which is the case for the Web Collector.

Next, the *Correlator* takes each per-Tier model and undergoes a non-linear regression. The curve-fitted tables of this non-linear regression are then used later by the *Estimator* to create memory assignments. In situations where Ginkgo does not have enough data points, be it due to the need for filtering (as discussed below) or lack of experimental history, the *Correlator* uses the non-linear regression to interpolate or extrapolate coordinates to calculate missing performance values.

In addition to these methods, other statistical transformations are applied to create *heuristic filters*. The first filter computes the coefficient of variation to filter out outliers. (Outliers are common when a VM is thrashing from a "swap storm"). The second filter is implemented by taking note of the fact the performance should, for a given load level, decrease monotonically with the decrease of memory assigned. If it doesn't happen, we throw out such measurements and we also instruct the framework to stop reducing the memory size. These filters were added out of experimental observation. The analysis of the performance data sampled under corner situations (e.g., low memory and moderate to high loads) showed it to be unreliable, mainly due to the artifacts caused by swapping.

Application performance might also be affected by factors not related to memory, such as CPU contention, I/O contention or cache interference between VMs running in the same host. Since the *Correlator* constantly adjusts the model by taking into account the latest measurements recorded by the Collector, Ginkgo will detect and react to new conditions even without knowing the cause of the degradation. We noticed this in our own experiments when we deployed Ginkgo on a new environment that had considerably worse storage performance; the model was reused but later updated to reflect the measurements resulting from the slower I/O path. As long as the source of (non-memory-related) performance interference is kept constant for each load level and memory assignment, the framework will gracefully re-adjust itself.

IV. Determining Memory Assignments

The *Correlator*'s performance model, the heart of this framework, is continuously used by the *Estimator*, the brain of this framework, to infer the memory needs of running VMs. The *Estimator*, triggered at regular time intervals (or proactively during SLA violations), processes the performance model to make memory decisions by determining how much memory each VM actually requires to deliver good application performance under its current load. To do this, the *Estimator* uses a Linear Program (LP) which takes as input the performance model, a set of SLAs, and a configurable optimization goal. For our evaluation we focused on maximizing performance, however, the LP can also be configured to maximize revenue, minimize cost or minimize memory consumption.

First, we list the different inputs of the LP:

1. A list of running virtual machine instances;
2. A list of all possible target memory size configurations;
3. A data matrix of normalized application performance for each possible memory size configuration;

For the data matrix, each row represents a VM, and each column a memory size. The value in each cell represents the normalized performance a given VM is expected to achieve for the given memory size. The LP uses normalized performance values because different types of Tiers might use different performance metrics (e.g., response time versus throughput), and must be compared to make a decision. The 100% normalization upper limit always represents the highest (summarized) performance value for each load level. The 0% lower limit represents a per-Tier cutoff. For example, response-time based performance metrics have a specified value of 1000ms as the 0% cutoff, while throughput-based metrics have a cutoff at zero. The LP defines a matrix of binary variables (output) where each row corresponds to a VM instance and each column corresponds to a possible memory assignment size. The LP includes the following constraints:

1. **Binary Matrix Constraint**: Every VM instance should receive at most one memory configuration size from the matrix.
2. **Constraint per Hypervisor**: The sum of memory configuration sizes assigned to all VM instances should not exceed the total capacity of the hypervisor. The total capacity may or may not include page coalescing activity.
3. **Configuration Constraint**: Do not assign more memory than was configured for each VM instance at provisioning time in the cloud.

Now that we have introduced the constraints and the inputs used by the LP, we define the objective function. The *Estimator* supports several desirable functions, including maximizing performance and profit or minimizing memory consumption. Certain objective functions may require additional constraints. In this work we optimize for performance and this requires an extra constraint to define the minimum performance each VM should achieve in the form of an SLA. For an alternative objective of profit maximization would similarly require that

the LP use an extra data matrix specifying revenue per performance indicator unit for each VM.

In our experimental evaluation the *Estimator* maximizes performance. After the LP has completed, the *Orchestrator* component has the responsibility of applying the necessary memory assignments to each hypervisor. For each running VM, the *Orchestrator* first calculates the difference between the current and the assigned amount of memory. Then, to reach the desired memory allocation, the *Orchestrator* gradually sends fixed-size balloon inflate/deflate requests at regular intervals. In our experiments, we used 32MB for the size of the memory adjustment requests.

V. Overcommitting Java Application Stacks

Java-based application performance puts significant trust into the JVM memory management system, which in turn relies on the OS memory management sub-system. As long as the OS allocates enough physical memory to hold the entire JVM heap, the application will perform as expected. However, this assumption may no longer hold true in an overcommitted virtual environment. If the hypervisor takes away memory from a VM, it is possible that part of the active JVM heap will be swapped to disk. On the other hand, if the JVM is made virtualization-aware, it can proactively reduce its heap size. While doing so can increase the number of garbage collections and decrease application performance [12], this degradation is an order of magnitude smaller than the degradation caused by swapping to disk when the JVM does not have enough physical memory [13]. Without a virtualization-aware JVM capable of re-adjusting its heap size during runtime, Ginkgo's decisions to overcommit memory were being based on the current JVM heap size instead of attained performance as we intended.

To solve this problem, we used the Java Native Interface (JNI) to implement a mechanism to dynamically adjust the JVM heap size during runtime, called JavaBalloon. Using JNI allowed us to deploy this without recompiling the JVM at all. The JavaBalloon is loaded as a background thread when the JVM starts. Similar to the balloon used by VMs, the JavaBalloon thread waits for external requests to reduce or increase the heap size. For decrease requests, the JavaBalloon first allocates Java objects to remove them from the scope of the garbage collector. Finally, the objects are pinned in the heap and their corresponding native memory is released to the guest kernel using the madvise(MADV_DONTNEED) system call. This makes the memory available for other usage in the guest, allowing the normal hypervisor ballooning event to reclaim that memory. Similarly, when a request to increase the heap size arrives, the JavaBalloon reverses that process.

As we show in section VI, using the JavaBalloon Ginkgo was able to achieve higher memory savings. Figure 1 depicts Ginkgo's integrated view, including its JavaBalloon support inside one of the WebSphere virtual machines. Ginkgo synchronizes between the VM balloon size and the JavaBalloon size. When the *Orchestrator* needs to take memory from a VM running a Java workload, it first inflates the JavaBalloon and then the VM balloon by exactly the same size. On the reverse side, the *Orchestrator* first deflates the VM balloon and then deflates the JavaBalloon.

VI. Generating the Correlation Model

Before we can evaluate the effectiveness of Ginkgo, we need an understanding of the expected performance of each Tier.

A. Configuration

Our experiments use an 8-core IBM x3655 machine, with 32GB of memory and a hardware RAID-5 SATA disk array. The VMs all have 2GB of memory and run Ubuntu 9. The hypervisor runs KVM on Redhat Enterprise 6. KSM (page de-duplication) is disabled. Our Workload Driver machine is identical to the first machine and sits on the same 1GbE switch. Ginkgo itself runs on this machine and uses the libvirt API to deliver memory assignments.

Profiling is performed by running multiple VMs at different loads while varying the memory size to record variations in application performance. As Ginkgo makes each 32MB memory balloon adjustment, this data runs through the *Correlator* to plot the graphs we see in this section. We generate profiles using a highly-loaded *non-overcommited* hypervisor with 15 virtual machines: 30GB total, with 2GB available for the hypervisor itself. To date, we have been able to run Ginkgo experiments on larger hypervisors involving as many as 64 VMs. The experiments we show here use 15.

To generate load, we generate "load plans", consisting of different load levels at different time steps. In order to vary the load, we first determine the highest and lowest loads than an individual Stack can handle; then we take a subset of those loads and determine the empirical guest CPU usage required to achieve maximum performance with a single Stack; Finally, we generate time steps by randomly choosing one of the loads in the observed range and assigning it to a 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 procedure gives us sufficient variation over time for testing.

B. Daytrader 2.0

We first profile Daytrader, a stock trading benchmark, on WebSphere (WAS) v7 and DB2 v9, both with and without a JavaBalloon-enabled WAS. WAS VMs receive two vCPUs and DB2 VMs receive one. We set the JVM maximum heap size to 1.8GB, just shy of the 2GB maximum for the VM, leaving room for kernel caches. The hypervisor also has 2GB out of 32GB reserved for other daemon processes. The database is really the only entity with variable free space as it is more I/O intensive.

We profile each Daytrader Stack with loads up to 8 simultaneous request streams. In Figure 2, we show the WAS memory-to-performance correlation results. We can see that performance is degraded when the amount of memory assigned to the VM is near or less than the JVM heap size (1.8GB), which is expected when we over-inflate the OS balloon mechanism, giving the VM less memory than it needs

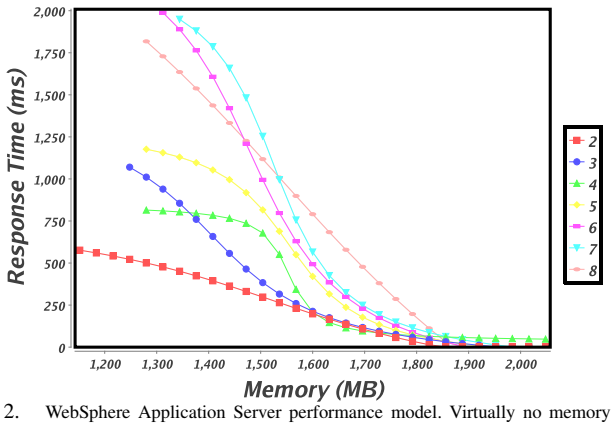

Fig. 2. WebSphere Application Server performance model. Virtually no memory is released during profiling because of the unmalleable nature of the Java heap. Transient swapping causes unpredictable performance variations.

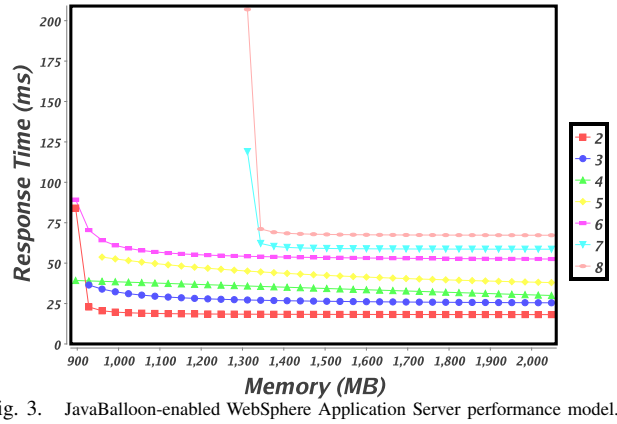

Fig. 3. JavaBalloon-enabled WebSphere Application Server performance model.

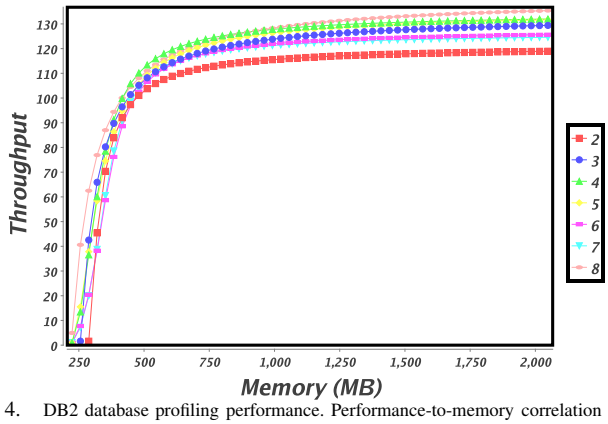

Fig. 4. DB2 database profiling performance. Performance-to-memory correlation has a smooth curve due to DB2's self-adjusting capabilities.

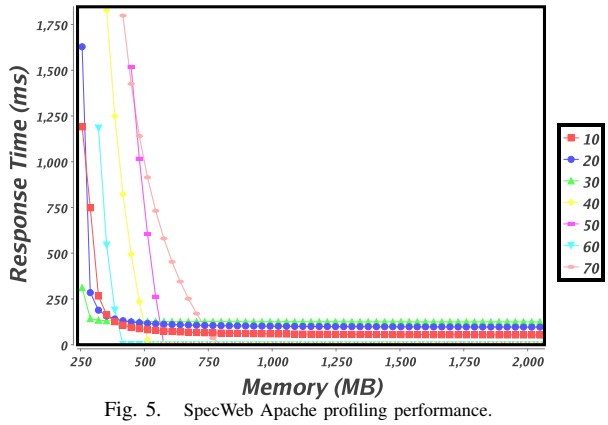

Fig. 5. SpecWeb Apache profiling performance.

to hold the entire JVM heap, hence forcing the guest OS to swap. Ginkgo can avoid entering this situation because it knows there is a significant drop in application performance under a given memory size by checking with the *Correlator*.

Figure 3 shows results with a JavaBalloon-enabled WebSphere, as described in section V. Here, we modify Ginkgo's profiling infrastructure to inflate the JavaBalloon *before* the OS balloon mechanism is inflated. This allows us to steal *as much as 1GB* of memory - almost 800MB more than what would have been available without it. Notice that performance remains stable, even with hundreds of megabytes of reduced memory.

DB2 has the capability to self-adjust the I/O cache size within the database, allowing it to release memory from its internal cache by monitoring the amount of available memory that was stolen during profiling. As a result, there is a clear, smooth tradeoff between database transactions per second performance and memory as shown in Figure 4.

C. SPECweb2009

We use the Banking version of SpecWeb 2009 [14] with Apache - also with 2 vCPUs per VM. The SPECWeb backend VMs, which perform database emulation, do not reside on the main hypervisor and are not used in the evaluation of Ginkgo. We generate profiles with loads ranging from 10 to 70 simultaneous request streams. For performance, we use the Web Collector which calculates the time taken to process each URL. The model for Apache is illustrated in Figure 5, showing the curve for each load. For this particular profile, Apache operates best with 750MB of memory, but can still function well with less depending on the incoming load. Apache will simply spawn fewer processes to serve fewer requests and does not suffer from the dynamic heap problems of Java. Also, processes running inside the web server do not consume all the physical memory, allowing the remaining memory to be used by the guest's Linux kernel to cache I/O operations, thus increasing application performance.

VII. Memory Efficiency Gains

Here we evaluate Ginkgo's overcommitment abilities. We will use the offline-generated profiles (presented in previous section) to estimate memory assignments online.

A. Configuration

First, the baseline overcommitment experiments consist of a mix of virtual machines across all Tiers: 4 WebSphere VMs, 4 DB2 VMs and 7 Apache VMs running simultaneously. After the baseline performance is established, we vary the maximum allocatable hypervisor memory from 30GB all the way down to 8GB with the same mix of VMs. Then we see what the performance is when memory is taken away at different granularities according to assignments determined by the Ginkgo system. For every 2GB we remove from the hypervisor, we allow load to vary to the VMs running on

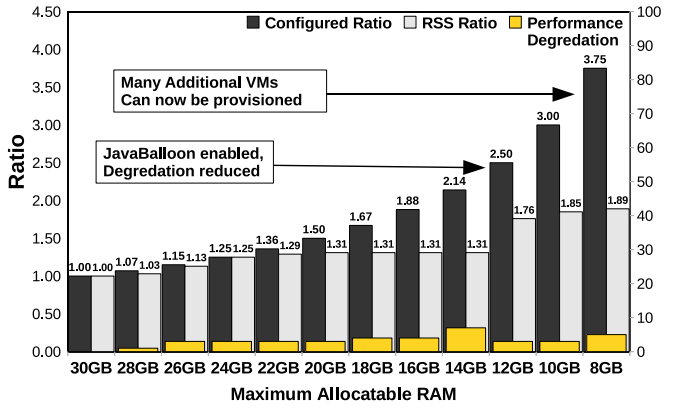

Fig. 6. Comparison of different overcommit ratios achievable with Ginkgo. Enabling the JavaBalloon at 12GB restores performance by increasing application malleability.

the hypervisor for over an hour before removing the next 2GB. Furthermore, to ensure that the extra memory is not used for hypervisor-level page caching, we fork a host-level 2GB process to allocate, touch, and mlock() that memory into DRAM so that it is not swapped and not available for use by the hypervisor, allowing for faster experimentation.

By employing this experimental procedure we can show that is possible to run the same number of VMs on a much smaller (memory sized) hypervisor. In a situation where the memory size is the only obstacle to achieve higher VM count (i.e., there is enough CPU and I/O bandwidth) it could be also said that the hypervisor used on our experiments can effectively run more VMs.

B. Results

Figure 6 shows a high-level comparison of the different overcommit ratios and their corresponding performance degradation achieved with Ginkgo. Configured Overcommit Ratio is computed as maximum memory amount that Ginkgo is allowing to be allocated in the experiment (e.g. 30GB, 28GB, ..., 8 GB) divided by the available memory in the hypervisor (30GB). This ratio includes free space, page cache space, and resident memory. RSS Overcommit ratio calculation is similar to Configured ratio, except that guest OS free space is subtracted from the total allocatable memory in the hypervisor. We enabled the JavaBalloon only when the allocatable hypervisor memory was equal or less than 12GB.

Figure 7 to 10 depict two groups of plots: the baseline plots and the results from the *"best"* memory savings we were able to achieve before one or more VMs began thrashing. If even one VM thrashes, we stop the experiment.

Figure 7 shows the normalized, aggregate results of using Ginkgo over the baseline configuration of the hypervisor. The x-axis represents time steps, and each data point represents a step of 10 minutes. For each time step, the left y-axis shows the normalized values for performance and the right y-axis shows the maximum memory allowed by Ginkgo. As we start to take away memory, the system is still able to function with as little as 14GB of memory while only suffering a 2 to 7% performance loss (shown by the line labeled **Ginkgo-Perf**). The performance loss at each overcommit ratio can be

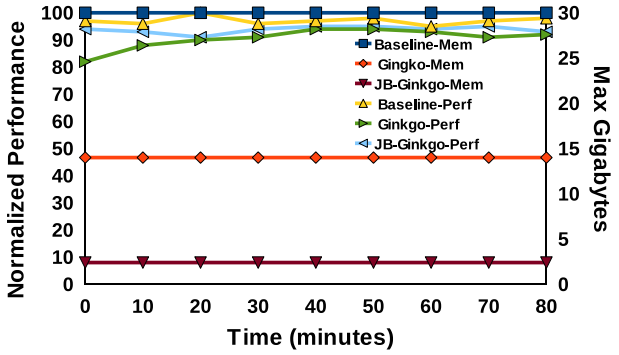

Fig. 7. Aggregate performance for 3 different overcommit levels: Ginkgo is set to allocate a maximum of 30GB (baseline), 14GB (ginkgo), and 8GB (ginkgo with JavaBalloon). The performance loss is small - no more than 7% across all timesteps.

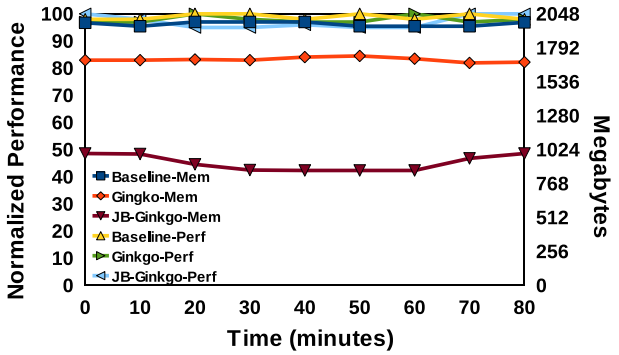

Fig. 8. WebSphere performance with Ginkgo. JavaBalloon results are included, showing more memory savings.

seen in Figure 6. In terms of VM memory configuration, this constitutes a configured ratio of 2.14x. However, in terms of actual memory used by the applications — the RSS Ratio —, this constitutes a ratio of 1.31x. With the aid of the JavaBalloon, the Configured Ratio increased to 3.75x, where as the actual RSS Ratio increased 1.89x.

To examine the non-aggregated source of the small performance loss, Figures 8, 9, and 10 illustrate a per-Application-Tier record at the same 14GB memory gains level for a period of over one hour. While we show only one VM instance for each workload, each VM instance has a different varying load behavior and all of them are able to maintain similar performance indicators.

JavaBalloon Results. Figure 8 (Ginkgo-mem) shows how the WebSphere Tier releases very little memory as determined by our model. When we try to achieve memory gains past 14GB, these VMs simply thrashed, resulting in failures. Thus, we perform an experiment in the same configuration except that the 4 WebSphere VMs are made to be JavaBalloon-enabled. As it can be seen in Figures 7 and 8 (JB-Ginkgo-Perf), we were able to bring the maximum memory down to as low as 8GB.

Also notice that the JavaBalloon-enabled Ginkgo (JB) line even has slightly better aggregate performance at a higher level of memory savings. We observed that this happens because the non-JavaBalloon-enabled WebSphere experiences small amounts of swap I/O since the *Correlator* is not able to gain very much memory without the help of the JavaBalloon.

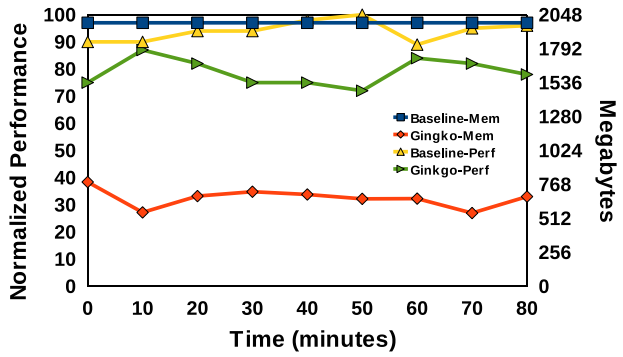

Fig. 9. Daytrader DB2 database performance.

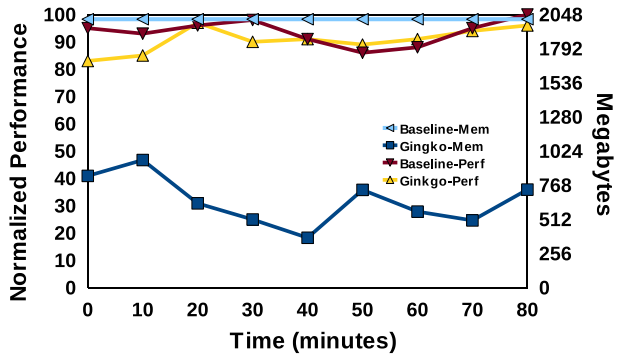

Fig. 10. Apache performance under SPECWeb.

However, when it is enabled with the JavaBalloon, the JVM is actively participating in the memory redistribution actions of Ginkgo and can receive new memory in a timely fashion as the load in the system changes. This reduces the amount of performance loss by a small amount.

VIII. Related Work

The problem of overcomitting virtualized resources has been investigated from several angles. This work presents a complete implementation and further evaluation of the introductory work done in [15], describing the details of our performance correlation model and decision mechanism. We also cover multiple overcommit ratios, Application Stacks spread across multiple VMs and introduce a new hypervisor-aware JVM. Hypervisor-level enhancements to memory management have been proposed to allow for a better understanding of how VMs use their memory. Waldspurger [3] describes how VMWare ESX Server estimates the guest working set during runtime. The Geiger project [6] estimates the working set by inspecting guest OS buffer cache behavior. Lu and Shen propose using VM memory accesses to predict the VM miss rate for a given memory size [7]. Magenheimer *et al* proposed the idea of a hypervisor-based cache [16]. Zhao and Wang proposed a scheme for VM memory balancing [8] by monitoring swap space and memory accesses to maintain a LRU histogram. All these works monitor system metrics to understand the VMs memory demand, however, they are not able to estimate application performance. In cloud-like environments, large-scale multi-tier applications are expected to meet specific SLAs, and performance is rigorously monitored by customers. Additionally, by only monitoring system metrics, hypervisors detect changes in memory demand only after memory pressure has appeared, which might be too late to avoid noticeable performance degradation. Ginkgo tries to go to the source (the application): it starts from the assumption that application performance is already being monitored in the cloud and actively correlates that with memory size and SLAs.

Heo *et al* [17] takes an application-driven approach to present a dynamic memory controller that uses feedback control to dynamically adjust the allocation of CPU and memory while maintaining service level objectives. When using decision mechanisms based on feedback/resource control as presented in these works, changes in the memory requirements are noticed when the application performance starts degrading, and might take some time until the system achieves the desired state. In contrast, Ginkgo tries to avoid this situation by monitoring load and adjusting the system just before the VMs start requiring more memory. Furthermore, the evaluation was based on synthetic workloads and real world traces, while Ginkgo deploys actual Application Stacks. Other approaches [18] provide end-to-end QoS guarantees by reserving CPU and I/O resources. Ginkgo is orthogonal to such approaches and the *Correlator*'s performance models can be used to consider also memory reservations in virtualized cloud environments.

PRESS [19] presents a mechanism to predict resource demand based on signal processing and statistical learning algorithms. The efficiency of CPU prediction was evaluated using representative benchmarks, however, memory prediction was evaluated using real traces. In addition, for real workloads, memory demand is not a simple value as it is for CPU demand. As we show in our evaluation, the memory demand is a function of the target performance. For example, over time, the Linux kernel uses free memory for I/O cache. Thus, additional memory might reduce the number of I/O operations and improve application performance. While prediction algorithms might not require advanced profiling and model calibration, they are suggested to avoid estimation errors. Thus, Ginkgo does not try to predict resource demand, and instead correlates between application performance, incoming load and memory.

CRAMM [13] analyzes how traditional virtual memory managers affect garbage-collected applications, showing how the OS can force trashing if the entire heap does not fit in physical memory. It solves the problem by introducing a cooperative memory management system to predict and adjust the heap size during runtime. While CRAMM does not target virtual environments –ignoring the new level of memory management added by the hypervisor–, it can still be integrated into the guest VM to resize the heap when the balloon driver is inflated or deflated. The effects of this approach would be similar to those we obtained using the JavaBallon, however, Ginkgo's JavaBalloon does not require modifications on the guest's OS memory manager.

IX. Conclusions and Future Work

Managing memory overcommitment is complex and must be done carefully to avoid performance degradation. As we show, performance metrics can be used to infer memory needs and calculate efficient memory assignments. Our results indicate that Ginkgo can allow the cloud provider to achieve significant memory gains with very small performance degradation. We also showed that by adapting the ballooning idea for a JVM, we are able to dynamically change the JVM heap size, allowing Java applications to be malleable.

When changes in cloud hardware or software happen, we expect the *Correlator*'s model to deviate from new environment's actual performance. In such situations, we plan to stop overcommitment or migrate VMs to other hosts until the new model converges to a steady-state. This will be done by freezing the old model and by profiling with a newer model. This would allow for Ginkgo to account for variations in both the environment as well as in system resources other than memory.

In this work, by using performance information, Ginkgo can potentially become resource-agnostic. This can be done by establishing correlations between performance and changes in other resources (e.g., CPU or Network I/O), as long as the hypervisor provides a method for dynamically controlling it. We are interested in extending Ginkgo to allocate CPU and I/O using multi-dimensional performance models and extending the linear model to consider assignments for these resources.

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 sampling and runtime learning, the cloud provider would not be able to estimate the memory requirements needed to host an Application Stack 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.

References

[1] 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*. New York, NY, USA: ACM, 2009, pp. 31–40.

[2] 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.

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

[4] 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. [Online]. Available: http://www.cs.ucsd.edu/~dgupta/papers/osdi08-de.pdf

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

[6] 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.*, vol. 40, no. 5, pp. 14–24, 2006.

[7] 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*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–15. [Online]. Available: http://portal.acm.org/citation.cfm?id=1364388

[8] 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*. New York, NY, USA: ACM, 2009, pp. 21–30.

[9] 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 *Proc. of ACM SOSP 2003*, Oct. 2003.

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

[11] I. Habib, "Virtualization with kvm," *Linux J.*, vol. 2008, no. 166, p. 8, 2008.

[12] S. Soman, C. Krintz, and D. F. Bacon, "Dynamic selection of application-specific garbage collectors," in *Proceedings of the 4th international symposium on Memory management*, ser. ISMM '04. New York, NY, USA: ACM, 2004, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/1029873.1029880

[13] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Cramm: Virtual memory support for garbage-collected applications," in *OSDI*, 2006, pp. 103–116.

[14] Standard Performance Evaluation Corporation, "SPECweb2009," http://www.spec.org/web2009/.

[15] A. Gordon, M. Hines, D. Da Silva, M. Ben-Yehuda, M. Silva, and G. Lizarraga, "Ginkgo: Automated, application-driven memory overcommitment for cloud computing," in *ASPLOS RESoLVE - Workshop on Runtime Environments/Systems, Layering, and Virtualized Environments*, 2011.

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

[17] 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*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 630–637.

[18] D. Xu, K. Nahrstedt, and D. Wichadakul, "Qos and contention-aware multi-resource reservation," *Cluster Computing*, vol. 4, pp. 95–107, 2001, 10.1023/A:1011408729750. [Online]. Available: http://dx.doi.org/10.1023/A:1011408729750

[19] Z. Gong, X. Gu, and J. Wilkes, "Press: Predictive elastic resource scaling for cloud systems," in *International Conference on Network and Service Management (CNSM)*, oct. 2010, pp. 9 –16.