Managing Dynamic Memory Allocations in a Cloud through Golondrina

Alexander Pokluda, Gastón Keller and Hanan Lutfiyya
Department of Computer Science
The University of Western Ontario
London, Canada
{gkeller2,hanan}@csd.uwo.ca

***Abstract*—In this paper, we present a policy-based framework that supports automated dynamic resource management in a virtualized environment. This allows for flexibility in how resources are allocated. We show how this framework can be used to support memory management through the use of migration and making local resource adjustments.**

***Keywords*-virtualization; cloud computing; policy-based management; memory management; migration**

I. Introduction

Modern data centers are comprised of tens of thousands of servers, and perform the processing for many Internet business applications. To guarantee that an application will always be able to cope with all demand levels the application is statically allocated enough resources so that peak demand is satisfied. The unit of allocation is typically a physical machine. One estimate is that servers are over-provisioned by over 500% in order to deal with peaks [1]. The result is that the physical machines are often underutilized. However, some physical machines may sometimes become heavily-loaded because of time-varying demand [2].

One approach to increasing utilization is *server consolidation*, which consists of hosting multiple servers in one physical machine. This approach is possible through *virtualization*. Virtualization refers to an abstract layer between the operating system and the hardware. The layer provides an interface to the actual hardware that allows for the support of a number of *virtual machines* where a virtual machine typically has application software installed on it. Virtualization reduces the unit of resource allocation to fractions of a physical machine. This potentially benefits data centers by allowing several applications to make use of the same physical machine. If the virtual machines are placed on a physical machine based on peak demand, then the physical machine can still be highly underutilized. On the other hand if the virtual machines are placed on a physical machine based on the average demand, then this may result in virtual machines competing for the same resources when demand increases. The reason is that demand for an application may increase such that it needs computing resources currently being used by other applications on the same physical machine.

The computing resources needed by an application change over time which suggests that there is a need to dynamically allocate resources to applications as demand changes. This is referred to as *on-demand* resource allocation and we use the term *cloud* to refer to a data center that supports on-demand resource allocation.

Much of the work in dynamic resource provisioning for Internet applications [3] involves constructing a performance model that is used to predict application resource requirements based on previous application behaviour. The predicted resource requirements are used to determine the needed number of application instances (e.g., [4], [5]).

Performance models are also used to periodically determine optimal placements of virtual machines (e.g., [6]–[8]) on physical machines based on a set of criteria that is typically mapped to an optimization model. This may require the migration of virtual machines from one physical machine to another physical machine. Since migration incurs CPU and network overheads and thus may degrade application performance, this work is typically done as part of periodic maintenance. The implication is that most of this work is not intended to respond to relatively sudden workload changes due to a burst of load for an application or a node failure [3].

As noted in [9] most of the existing work does not consider adjusting local resource allocations to virtual machines. The work in [9] shows the advantages of adjusting local resource allocations to a virtual machine by adjusting parameters that control CPU cycles allocated to virtual machines. The paper then proceeds to show an optimization model that not only assumes migrations but also adjustments to allocated CPU cycles. The work in [10] allocates additional memory in response to a service level objective (SLO) that is violated. The primary action in response is migration. Although the paper states that it determines the cause of the SLO violation no details are presented. The key commercial cloud platforms, e.g. Amazon Web Services [11] and Microsoft Azure [12], allow for changes to the number of virtual machines in use for an application but there appears to be little support for adjusting local resources for a virtual machine.

The specific contributions discussed in this paper are the following: memory management using OpenVZ [13] was studied, a simple heuristic for identifying memory stress situations was developed using a black-box approach, a simple heuristic for adjusting memory allocations was developed and experimented with, and a flexible system framework that supports the

separation of decision making from the specific virtualization technology is presented. The latter was of interest since very little work allows for fully automated management. Although [10] does provide such a framework we differ in that we allow for local resource adjustments nor do they show a direct relation between policies and management framework configuration.

The rest of the paper is organized as follows: Section II discusses the architecture of the system, Section III introduces the memory management parameters, Section IV shows how a memory stress situation is detected, Section V describes how the system can react to memory stress situations, Section VI describes the system's prototype implementation, Section VII discusses the experiments, and Section VIII concludes.

II. Architecture

An important aspect of a management framework is the ability to determine the appropriate action in response to workload changes. The appropriate action depends on strategies and those strategies may change over time [14]. Strategies can be represented through policies [14]. This section presents a policy-based management framework which is graphically depicted in Figure 1. More details can be found in [15] and [16].

A. Policies

A policy associates an event with one or more rules of the form *if* condition **then** *actions*. The rules are evaluated when the event occurs. One type of policy is used to specify a required *operational condition* expected at run-time and the actions to be taken if the condition is violated. Operational conditions represent run-time requirements of system behaviour (e.g., expected CPU load, memory consumption).

An example policy is presented, using Ponder-like formalism [17], in Example 1.

Example 1:

```
oblig NotifyMemoryStressViolation{
  target DepartmentContainers
  subject Manager
  on memoryStress(containerID,
                  hostNameID,
                  freePhysicalPages);
  do IncreaseMemory(containerID, hostNameID)
   when freePhysicalPages > N;
  do t = ChooseTarget()
   when freePhysicalpages < N ->
    MigrateContainer(containerID,hostNameID,t);
}
```

The `on` clause specifies an event. The name of the event is *memoryStress*. The event expression associated with this name, which is assumed to be previously defined, is the following: *memoryStressScore* > 0.80, where *memoryStressScore* is the attribute that represents the memory stress score (this is discussed in detail in Sections III and IV). This indicates that the event occurs if the container's memory stress score exceeds 0.80. The event specification is parameterized by attributes (e.g. *containerID*, *hostNameID*, *freePhysicalPages*). Essentially the event is generated if an operational condition is violated. The information associated with the generated event are the attributes that parameterize it.

The entity specified in the `subject` clause is a manager process that determines the actions to be carried out when the event specified in the `on` clause is true. The `target` clause specifies the group (or domain) to which the policy applies - `HRDepartmentContainers` in this case. This group refers to a set of containers (a type of virtual machine) in a specific department.

The actions are determined by the rules. The condition in the rule is used to determine if the actions specified in the rule should be executed. In Example 1, each `do` clause specifies a rule to be evaluated when the event occurs. The first rule states that the memory allocation to the container should be adjusted if there is sufficient memory on the hardware node that the container is on. There may not be sufficient memory to accommodate a container's resource needs. Migrating a container is another possible action. The implementation of *ChooseTarget()* may simply be to pick the hardware node with the most free pages and that the number of free pages exceeds some threshold value. There could be other criteria used for choosing a target physical node. A *configuration policy* is used to choose a particular implementation of *ChooseTarget()*. An example of a configuration policy is seen in Example 2.

Example 2:

```
oblig ConfigurePolicy{
  subject Manager
  on ConfigurePluginRequest(k,p);
  do ConfigureChoseTargetPlugin(p)
   when k = ``ChooseTargetLocation";
}
```

This policy determines the action to be taken upon receipt of a configuration request. In this example `t` is the keyword and `p` is the reference to the appropriate plugin. If the keyword is "ChooseTargetLocation" then the plugin for `p` is used as the plugin for `ChooseTarget()`.

B. Sensors and Actuators

Monitoring requires some form of instrumentation, which we refer to as a *sensor*, specific to the managed object. Information about a managed object is provided, in practice, by a variety of mechanisms, including SNMP agents, JMX, reading accounting files in the *proc* filesystem. Each sensor has an interface, but these interfaces are different for different types of sensors. Sensors often have a limited ability to add or delete conditions to be evaluated. Furthermore it may be necessary to start and stop sensors. The lack of a common and expressive interface to sensors makes it challenging to automate the configuration of management entities to support policies. We address this challenge with the *sensor agent*. Sensor agents interface with sensors.

A sensor agent's interface provides methods that include (i) starting and stopping sensors; (ii) testing whether a sensor is already running and if it is not starting the sensor; and

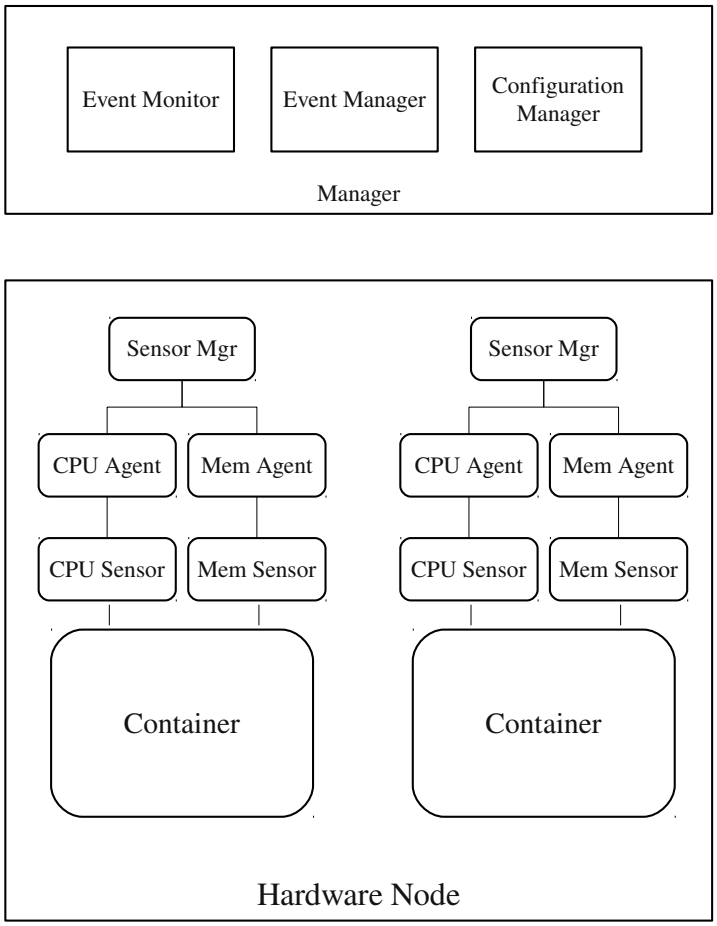

Fig. 1. Golondrina's architecture

(iii) getting data from a sensor. The implementation of these methods is specific to the sensor.

The set of sensors on a hardware node is managed by the *sensor manager*. A sensor manager maintains information on sensors on the hardware node. The interface of a sensor manager provides methods that allow the sensor manager to be informed of new sensors being added or removed from the hardware node. Information maintained for a sensor includes the frequency with which sensor data should be collected and the recipient to which the sensor manager should forward that data to. Separating sensors from sensor managers allows for the sensor manager to be independent of the virtualization technology used. In other words, the implementation of the sensor manager does not change if the virtualization technology changes from OpenVZ [13] to Xen [18].

An entity that changes the system is referred to as an *actuator*. Similar to sensors there is an *actuator manager* and *actuator agent*.

C. Manager

The *Manager* includes an *Event Monitor* that receives sensor data which it analyzes to determine if an event has occurred. The occurrence of an event is based on the evaluation of the condition associated with the event, e.g. $memoryStressScore > 80$. For each operational condition policy, the Event Detection server maintains a set of tuples in the form of (*e,p*), where p represents a policy identifier and e is the event expression. We note that evaluation of operational conditions can also be done by sensor agents which can also maintain the set of tuples. The experiments in this paper

assume that it is the Manager's Event Monitor that evaluates operational conditions.

The *Event Manager* is used to associate an event received by the Manager with one or more rules. When an event is detected the Event Manager finds the rules associated with the event and uses those rules to determine the appropriate action.

Currently there are three primary mechanisms used to dissipate resource stress situations: migration, replication and adjustment of parameters that allocate local computing resources to a container. Earlier we discussed that migration requires a target physical node to be chosen and the choice of a target depends on specific criteria. The *Configuration Manager* associates plugins with specific functions.

D. Interactions

One approach to supporting the policy described in Example 1 is briefly described. Each sensor manager is requested to send memory utilization data in fixed time intervals. The sensor manager does so by contacting the memory utilization sensor. When the Event Monitor receives the data it evaluates the condition specified in the policy. If the condition evaluates to true then it calls the Event Manager to extract the rules to be evaluated. At start up or when a new policy is added, the policy is read to configure the different architectural entities.

III. Memory Management Parameters

Today most operating systems assume that memory is allocated as *pages*. The amount of memory actually used depends on application behaviour. For example, a *malloc()* is used by programs to make a request for memory of a specific size. If successful a pointer to a block of memory is returned. Another need arises when the stack needs more memory to accommodate function and system calls. Insufficient memory means that the *malloc()* or that function/system calls fail since the process stack is not allowed to expand.

In this section we describe the memory management parameters used. This is based on OpenVZ. OpenVZ is a Linux kernel modified to run multiple, isolated *containers* (i.e. virtual user-space environments). The primary advantage of OpenVZ compared to other virtualization technologies (e.g., Xen [18], VMWare [19]) is its memory management capabilities. Essentially an OpenVZ container is guaranteed a certain amount of memory. If, however, there is memory not being used then a container that needs more memory can get more than what is guaranteed.

Each container has its own set of parameters, that allows for different memory allocation schemes for each container. The system parameters can be mapped one on one with a subset of OpenVZ's system resource control parameters (also known as *user beancounters*).

In this discussion we use the term *alloc* to refer to the current number of pages allocated to a container. The term *allocUsed* is used to refer to the number of pages actually used by the container. The pages are assumed to include user, kernel and swap spaces.

The first parameter to consider is *allocationGuarantee*, which represents the number of pages that a container is guaranteed to be granted if requested. The next parameter to consider is *allocationLimit*, which determines the maximum number of pages that can be allocated to a container. Between the previous two parameters exists *allocationBarrier* - a soft limit so to speak. These parameters are responsible for determining the success of a memory allocation request.

A memory allocation request for *n* pages is guaranteed to succeed if the following is true:

$$n + alloc \leq allocationGuarantee$$

All memory requests will succeed if n is less than the number of free physical pages and if the following is true:

$$n + alloc \leq allocationBarrier$$

If n+$alloc$ is greater than $allocationBarrier$ then only a high priority request (e.g., process stack expansion) will succeed, but only if n is less than the number of free physical pages and if the following is true:

$$n + alloc \leq allocationLimit$$

Beyond *allocationLimit* every request fails, independently of the availability of unallocated memory pages.

It is possible for the sum of the *allocationLimit* values of all the containers hosted in a hardware node to exceed the total amount of physical memory available. Thus, effectively overselling memory.

The last parameter to consider is *minMemoryGuarantee*, which serves as a safety limit. When a hardware node runs out of memory (RAM plus swap space), the kernel starts searching for processes to be killed so as to free memory. Those containers whose *allocUsed* value is lesser than *minMemoryGuarantee* will not have processes killed. The kernel will kill processes from those containers that are using the largest amounts of memory in excess of *minMemoryGuarantee*.

IV. Memory Stress Detection Mechanism

CPU and memory utilization statistics are collected periodically for each container. The memory utilization statistics serve to calculate a *memory stress score* for each container. This score indicates whether a container needs additional memory.

At time t_i the following information is collected for each container: (i) the number of failed allocation attempts since start of the current accounting period (denoted by $failcntAllocations_i$), (ii) the number of processes killed since the start of the current accounting period (denoted by $failcntKills_i$), (iii) the number of memory pages allocated (denoted by $alloc_i$), and (iv) the number of allocated memory pages actually used (denoted by $allocUsed_i$).

The following algorithm shows how the collected information is used to calculate the *memory stress score* for a container. We note that a similar approach can be used for calculating a memory stress score for a physical node.

The first and second lines calculate the number of process kills and failed allocations in the time period $(t_{i-1}, t_i]$. s_0 is

1: x = $failcntKills_i$ - $failcntKills_{i-1}$
2: y = $failcntAllocations_i$ - $failcntAllocations_{i-1}$
3: **if** x $>$ 0 **then**
4: s_0 = 1;
5: **end if**
6: **if** y $>$ 0 **then**
7: s_1 = 1;
8: **end if**
9: s_2 = min (1, $allocUsed_i/minMemoryGuarantee$);
10: s_3 = min (1, $alloc_i/allocationBarrier$);
11: $stressScore_i$ = max (s_0, s_1, s_2, s_3);

the strongest indicator of a memory stress since processes are being killed to free up memory. s_1 is also a strong indicator of stress since memory allocation requests are being rejected.

s_2 indicates the ratio of memory in use to the safety limit mentioned in Section III. If s_2 is equal to one, then the memory is use exceeds the safety limit and thus the container's processes are at risk of being killed in the event of an overall memory shortage on the hardware node.

s_3 indicates the ratio of allocated memory to the soft limit *allocationBarrier*. If s_3 is equal to one, then the container has been allocated memory in excess of the soft limit and therefore future memory allocation requests will be denied unless they are high priority requests.

V. Memory Stress Resolution

The policy used in this work for resolving memory stress situations is briefly described as follows. If a container is memory stressed (defined as the memory stress score being greater than 0.80) and the hosting hardware node has memory available, then the memory limits defined by the memory management parameters are increased by a fixed amount for the container, granting the container access to more memory. All four memory parameters described in Section III are increased.

If a container is memory stressed, but the hosting hardware node has no memory available, then the stressed container is migrated to another hardware node with sufficient available memory to allow the memory limits to be increased. The latter may not always be possible, so in future work we will consider migrating a different container on the same hardware node, so as to free up memory that could be allocated to the stressed container.

When a hardware node is memory stressed, an attempt is made to migrate the hosted container that is using the largest amount of memory on that hardware node. If this cannot be done without causing a memory stress on another hardware node, then the container with the next largest memory utilization is chosen and the process is repeated.

VI. Prototype Implementation

The prototype developed, *Golondrina*, assumes an OpenVZ virtualization environment which provides operating system-level virtualization.

The prototype was developed in Python version 2.4.3. The communications between system components are handled through the event-driven networking engine Twisted version 8.20. We have implemented sensors that read accounting files in the *proc* filesystem in order to collect CPU and memory utilization data from containers and hardware nodes. Functions that implement some sort of control algorithm such as *ChooseTarget()* are implemented as a plugin.

There are three actuators. The migration actuator relocates a container. The replication actuator creates a new instance of a container hosted in one hardware node into another. The memory adjustment actuator modifies the memory management parameters (see Section III). The implementation of migration and replication is described in more detail in [20].

VII. Experiments

This section presents four experiments. The test environment is described first, followed by the experiments, and finally a discussion of the results and implications.

A. Test Environment

The test environment consisted of three identical hardware nodes, named *bravo01*, *bravo02* and *bravo03*. Each hardware node contained a 3.40 GHz dual-core Intel Pentium D CPU with a 2 MiB cache, 2 GiB of RAM and 2 GiB of swap space. *bravo01* and *bravo03* ran CentOS version 5.3 while *bravo02* ran CentOS version 5.2. Each hardware node ran an OpenVZ kernel based on Linux version 2.6.18.

For the experiments, a container hosted a web application running on top of a LAMP (Linux-Apache-MySQL-PHP) software stack. The web application was Joomla!, a free and open source web content management system. The software stack consisted of Apache HTTP Server version 2.2.3, MySQL database server 5.0.45 and PHP version 5.1.6. The containers were running CentOS version 5.3 for the Linux component of the software stack.

Load generation for the web application running in the container was carried out using Apache Jakarta JMeter version 2.3.4. JMeter threads were used to simulate web browsers belonging to different users. The sample website that comes with Joomla! was the target of the HTTP requests generated with JMeter.

B. Experimental Runs

The goal of the experiments was to study the impact of the policies specified in Section II on memory utilization.

For the experiments, we leveraged JMeter's capacity to specify the number of test threads to run, the loop count and the ramp-up period. The loop count determines the number of times each thread loops. The ramp-up period, measured in seconds, is used to set the rate at which test threads are started.

The results presented in Table I are based on the number of threads, ramp-up period, loop count and frequency that memory utilization information is sent to the *Manager* set to 9, 2, 5, and 10 respectively for experiments 0, 1, 2. The results seen for Experiment 3 assume a ramp-up period of 36, the frequency of sending memory usage data is set to 2 (the explanation is provided later). The results in Table I

assume that the memory stress score threshold value is 0.80 for Experiments 0,1,2 but is 0.70 for Experiment 3 (more discussion later).

The experiments consisted of generating HTTP request for eight different web pages of the Joomla! sample website. Each iteration of a JMeter test thread would load these eight pages in sequence and then repeat the process until it had been repeated as many times as the loop count value. The time between iterations followed a Gaussian distribution.

For each experimental run we collected the minimum and maximum response times observed for each of the eight web pages, the number of HTTP responses that did not contain a valid HTML page (as a percentage), the throughput (rate of completed requests), the average bandwidth, and the average number of bytes of each HTTP response. The throughput value was computed using the elapsed time between the moment when the first HTTP request was sent and the moment when the last HTTP response was received. Each experiment ran four times. The average was taken for each of the metrics described in the previous paragraph. This is reflected in Table I.

The three hardware nodes were used as follows: *bravo01* ran the Golondrina Manager component while *bravo02* and *bravo03* ran the Golondrina client component (consisting of the sensors and actuators); *bravo01* was also used to run JMeter, the load generator and performance monitor.

1) Experiment 0: This experiment is the base case. The container that receives HTTP requests is given enough memory so that no memory stress situation occurs. This establishes upper-bound (best case) reference statistics. Essentially it represents the results obtained when sufficient resources are assigned to a container. In these experiments we assume that *allocationGuarantee* is set to 128MiB, *minMemoryGuarantee* is set to 100MiB, *allocationLimit* is initialized to 132MiB and the value of *allocBarrier* is 128MiB.

2) Experiment 1: In this experiment the container starts with the minimum number of resources assigned to it, i.e. *allocationGuarantee* is set to 64MiB, *minMemoryGuarantee* is set to 60MiB, *allocationLimit* is initialized to 66MiB and the value of *allocationBarrier* is set to 64MiB. In this experiment policies are disabled.

3) Experiment 2: This experiment is similar to that of Experiment 1, but it uses a modified version of the policy specified in Example 1. The modification is that the second rule is not used.

4) Experiment 3: This experiment is similar to that of Experiment 1, but uses the policies specified in Examples 1 and 2.

C. Discussion of Results

By comparing the minimum response times of HTTP requests for experiments 0-3, it can be seen that the minimum response time for runs without errors was about 166 milliseconds, whereas experimental runs with higher error rates had lower response times. This is likely due to responses being generated quickly. The low minimum response times likely occurred early in experimental runs when only the first experimental thread was running–before the container became memory stressed. This likely contributed to the high standard deviation seen in the experimental runs in which the container experienced a memory stress.

The large response times that resulted when the container became memory stressed were due to the fact that the Apache HTTP Server parent process was unable to create additional child processes. In this case, new connections were queued and handled by existing child processes after their current connections were terminated.

In Experiment 1 there was a noticeable increase in the maximum response time and corresponding decrease in the throughput. Each run in this experiment also had a large fail count, meaning that memory allocation requests, such as calls to the C library's malloc or fork functions, were denied. Thus, the container was severely memory stressed. During each run the CPU utilization on the hardware node hosting the stressed container was monitored to ensure it remained below 100 percent. Similarly, the network bandwidth used was monitored during each run and remained below 1.2 megabits per second. Thus, the decrease in performance was due to the memory shortage.

The four runs of Experiment 0 (no memory stress) and Experiment 1 (unresolved memory stress) produced an average throughput of 8.0 and 3.6 respectively. The four experimental runs of Experiment 2 (memory stress resolved locally) produced an average throughput of 4.9. Golondrina detected the memory stress, as expected, and increased the stressed container's memory limits. This resulted in a throughput value that was substantially better than the throughput for Experiment 1 (unresolved memory stress), but not as good as the throughput for Experiment 0 (no memory stress). In this experiment, an average of 314 memory allocation attempts were denied by the kernel. This value is again substantially better than 1002, the average fail count for runs in Experiment 1, but it indicates that the container was still severely stressed for a short period of time. Although Golondrina successfully detected and resolved the memory stress situation, Golondrina did not react quickly enough to prevent memory requests from being denied.

The reaction time could be improved by decreasing the delay in the gathering of resource utilization statistics, decreasing the threshold value for the memory stress score in the policy stated in Example 1 or using some form of prediction.

For Experiment 3 we initially set ramp-up period, frequency of sending data and memory stress score threshold the same as for the other experiments. This caused migration failures since the container ran out of memory before the migration was completed. The reason for this is that OpenVZ starts a process in the container as part of the checkpointing process for a live migration. This process counts towards the container's overall resource utilization. If the container does not have enough memory to accommodate this process, the migration will fail. This suggests the need to take the cost of this process into consideration before attempting to migrate a container.

TABLE I
EXPERIMENTAL RESULTS

Exp	Avg	Min	Max	Std Dev	Err %	Throughput	Fail Count
0	448	166	2275	243.72	0.00	8.0	0
1	1082	124	55576	5817.40	2.22	3.6	1002
2	866	115	49214	4748.84	1.63	4.9	809
3	370	137	4354	419.42	9.24	5.5	1143

We then increased the ramp-up period, decreased the threshold value for the memory stress score and decreased the frequency of sending memory utilization statistics. This was done to minimize the likelihood of the migration failing due to a lack of memory in the container.

The experiments suggest that the threshold values for memory stress scores may need to be low enough to allow for successful migrations. Another possibility is to modify the conditions in the rules to take into account the rate that the memory stress score increases. This will be a topic of further investigation.

There was a significant variance in the error rate between runs for Experiments 1, 2 and 3. One possible explanation for this is the synchronization, or lack thereof, between the moment when a resource stress occurred and the moment when it was first detected by Golondrina. These experiments show a correlation between the HTTP request error rate and the memory allocation request fail count: as long as the fail count remained below approximately one thousand, there were nearly zero failed HTTP requests. Further investigation is needed to confirm this relationship.

VIII. CONCLUSIONS

This paper presents early work in dynamic resource management with an emphasis on memory. The work presented in this paper assumes monitoring of utilization occurs periodically and that this is used to adjust resources in response to workload fluctuations. The intention is that this is done more often than what is needed for optimization models. We see this work as complimentary to work that focusses on virtual machine consolidation. The framework allows for multiple virtualization technologies to be used. Sensors are specific to a virtualization technology. However, it should be possible to determine a measure of resource stress. Once calculated it can be used independently of the specific virtualization technology in place.

The approach to increasing memory in this paper is straightforward. We will develop a plugin that supports adaptive resource control using control theory techniques as found in [2]. We will also examine a modification of the policy shown in Example 1 to take into account the results of our initial investigation as described in Section VII.

Dynamically adjusting resources to an application not only should take into account the local resource demands where the container is located but also the resource demands of other application components. We will also look at using the memory stress score to help in diagnosing SLO problems.

ACKNOWLEDGEMENTS

We thank the National Sciences and Engineering Research Council of Canada (NSERC) for their support.

REFERENCES

[1] Youtube vs myspace growth. [08 Aug 2010]. [Online]. Available: http://blog.compete.com/2006/10/18/youtube-vs-myspace-growth-google-charts-metrics/

[2] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem, "Adaptive control of virtualized resources in utility computing environments," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA: ACM, 2007, pp. 289–302.

[3] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, pp. 7–18, 2010. [Online]. Available: http://dx.doi.org/10.1007/s13174-010-0007-6

[4] B. Urgaonkar and A. Chandra, "Dynamic provisioning of multi-tier internet applications," in *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 217–228.

[5] Q. Zhang, L. Cherkasova, and E. Smirni, "A regression-based analytic model for dynamic resource provisioning of multi-tier applications," in *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*. Washington, DC, USA: IEEE Computer Society, 2007, p. 27.

[6] A. Verma, P. Ahuja, and A. Neogi, "pmapper: Power and migration cost aware application placement in virtualized systems," in *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. New York, NY, USA: Springer-Verlag New York, Inc., 2008, pp. 243–264.

[7] N. Bobroff, A. Kochut, and K. Beaty, "Dynamic placement of virtual machines for managing sla violations," 2007, pp. 119–128. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4258528

[8] G. Khanna, K. Beaty, G. Kar, and A. Kochut, "Application performance management in virtualized server environments," in *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, 2006, pp. 373–381. [Online]. Available: http://dx.doi.org/10.1109/NOMS.2006.1687567

[9] M. Cardosa, M. R. Korupolu, and A. Singh, "Shares and utilities based power consolidation in virtualized server 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. 327–334.

[10] M. Schmid, D. Marinescu, and R. Kroeger, "A framework for autonomic performance management of virtual machine-based services," in *Proceedings of the 15th Annual Workshop of HP Software University Association*, 2008.

[11] Amazon elastic compute cloud (ec2). Amazon, Inc. [08 Aug 2010]. [Online]. Available: http://www.amazon.com/ec2

[12] Windows azure platform. Microsoft, Inc. [08 Aug 2010]. [Online]. Available: http://www.microsoft.com/windowsazure/

[13] (2005) OpenVZ user's guide. SWsoft. Herndon, VA. [Online]. Available: http://download.openvz.org/doc/OpenVZ-Users-Guide.pdf

[14] B. Simmons and H. Lutfiyya, "Achieving high-level directives using strategy-trees," in *MACE '09: Proceedings of the 4th IEEE International Workshop on Modelling Autonomic Communications Environments*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 44–57.

[15] G. Keller, "Dynamic resource management in virtualized environments," Master's thesis, University of Western Ontario, 2009.

[16] A. Pokluda, "Dynamic resource management using operating system-level virtualiation, bsc thesis," University of Western Ontario, 2010.

[17] N. Dulay, E. Lupu, M. Sloman, and N. Damianou, "A policy deployment model for the ponder language," in *IM'09: Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, 2001, pp. 529–543.
[18] Xen. XenSource, Inc. [20 May 2010]. [Online]. Available: www.xensource.com
[19] Vmware esx server. VMWare, Inc. [20 May 2010]. [Online]. Available: www.vmware.com/products/esx
[20] G. Keller and H. Lutfiyya, "Replication and migration as resource management mechanisms for virtualized environments," in *ICAS '10: Proceedings of the 2010 Sixth International Conference on Autonomic and Autonomous Systems*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 137–143.