Managing Dynamic Memory Allocations in a Cloud through Golondrina

4th International DMTF Academic Alliance Workshop on Systems and Virtualization Management: Standards and the Cloud

Alexander Pokluda, Gastón Keller and Hanan Lutfiyya

Department of Computer Science
University of Western Ontario

October 29, 2010

Outline

Outline

Outline

Outline

Outline

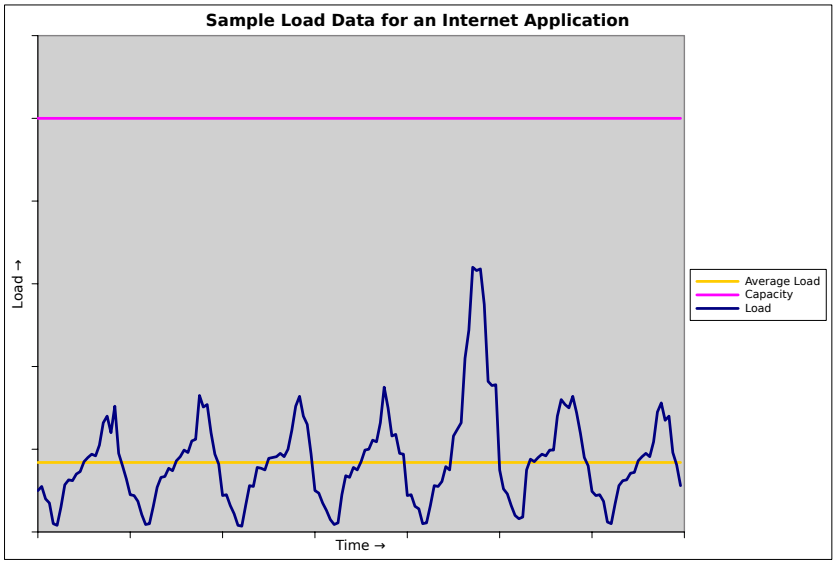
Introduction

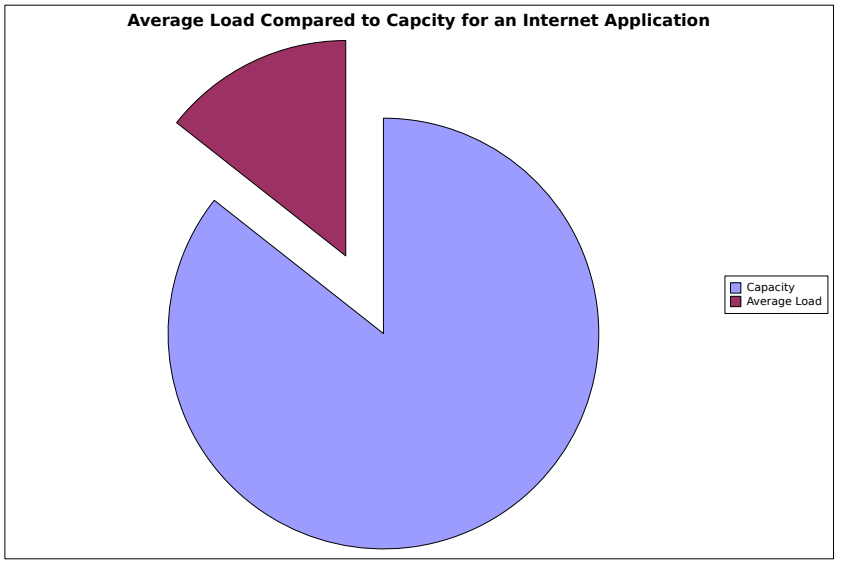
Abstract

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.

Accommodating Peak Demand with Over-Provisioning

- Modern data centres are comprised of tens of thousands of servers, and perform the processing for many Internet business applications
- Unit of allocation is typically one physical machine
- One estimate is that servers are over-provisioned by more than 500% in order to deal with peaks in demand





Over-Provisioning Leads To Underutilization

- Over-provisioning means underutilization
- One approach to increasing utilization is *server consolidation*, which consists of hosting multiple servers on one physical machine
- Server consolidation is possible through *virtualization*
- Virtualization provides an interface to the actual hardware that can support a number of *virtual machines* that have application software installed on them

Strategic Placement of Virtual Machines Can Increase Utilization

- If virtual machines are placed on physical machines based on peak demand, physical machines may still be highly underutilized
- If virtual machines are placed on physical machines based on average demand, the virtual machines may compete for the same resources when demand increases
- Much of the work in dynamic resource provisioning for Internet applications involves constructing a performance model
- Performance models are used to periodically determine optimal placements of virtual machines as part of periodic maintenance

Contributions

Contributions

1. Memory management using the OpenVZ virtualization platform was studied
2. A simple heuristic for identifying memory stress situations was developed using a black-box approach
3. A simple heuristic for adjusting memory allocations was developed and experimented with
4. A flexible framework that supports the separation of decision-making from the specific virtualization technology was designed

Outline

Our Solution

An important aspect of a management framework is the ability to determine the appropriate action in response to workload fluctuations.

The appropriate action depends on strategies that can be represented through policies.

We present a policy-based management framework that is depicted in the figure to the right.

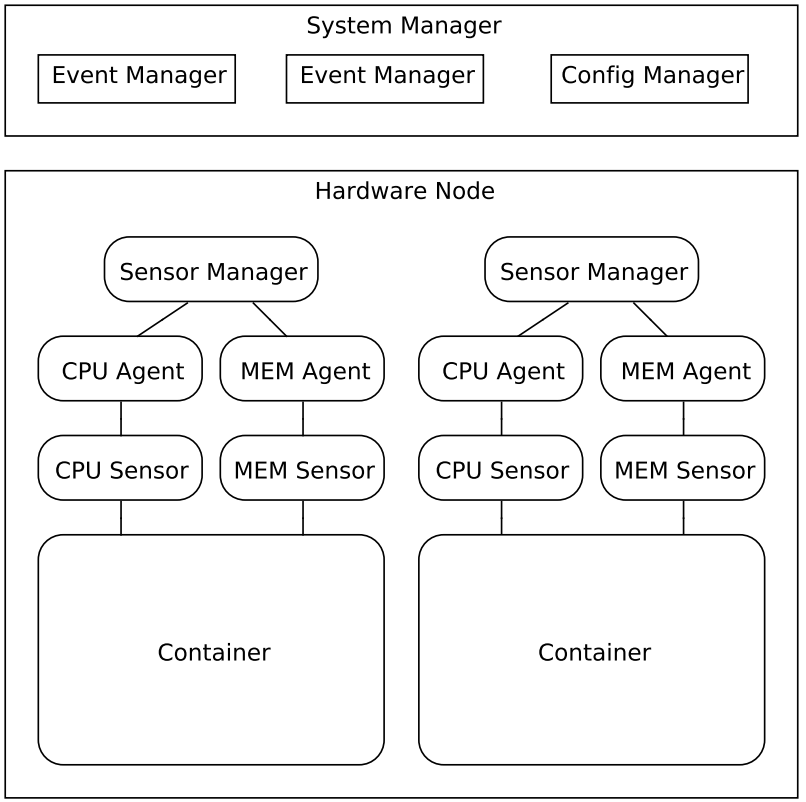


Figure: Golondrina's architecture

System Behaviour is Configured Through Policies

- A policy associates an event with one or more rules of the form **if** *conditions* **then** *actions*

```
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);
}
```

Example: A sample memory-stress violation policy

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

Example: A sample configuration policy

System Behaviour is Configured Through Policies

- A policy associates an event with one or more rules of the form **if** *conditions* **then** *actions*

```
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);
}
```

Example: A sample memory-stress violation policy

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

Example: A sample configuration policy

Collecting Resource Usage Data

- Monitoring requires some form of instrumentation, which we refer to as a *Sensor*, specific to each managed resource
- A *Sensor Agent* provides a standard interface to sensors
- The set of sensors on a hardware node is managed by a *Sensor Manager*
- An entity that changes the system is referred to as an *Actuator*; similar to sensors, there is an *Actuator Manager* and *Actuator Agent* (not shown)

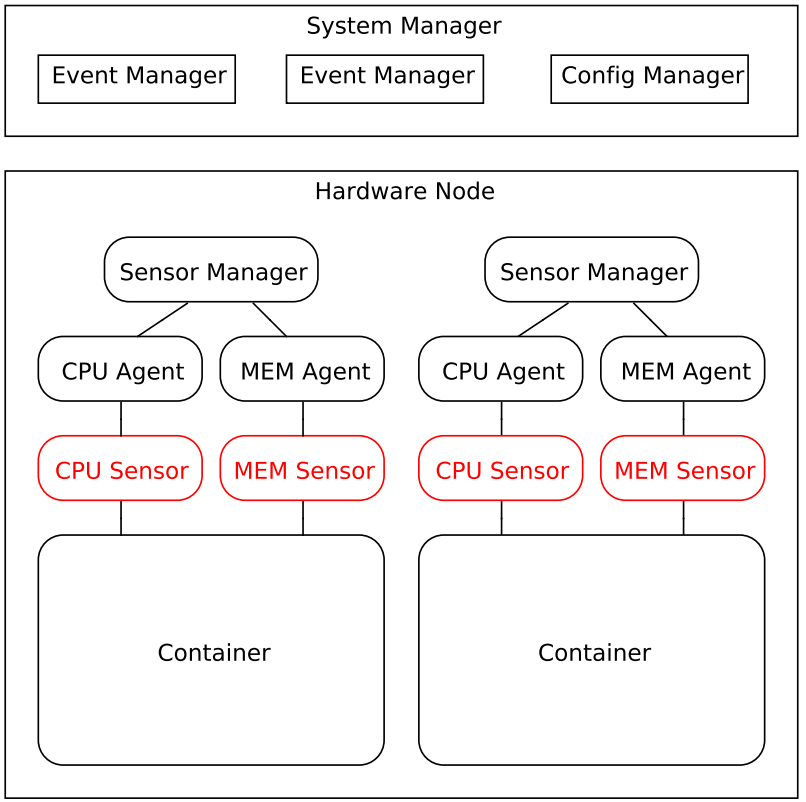


Figure: Golondrina's architecture

Collecting Resource Usage Data

- Monitoring requires some form of instrumentation, which we refer to as a *Sensor*, specific to each managed resource
- A *Sensor Agent* provides a standard interface to sensors
- The set of sensors on a hardware node is managed by a *Sensor Manager*
- An entity that changes the system is referred to as an *Actuator*; similar to sensors, there is an *Actuator Manager* and *Actuator Agent* (not shown)

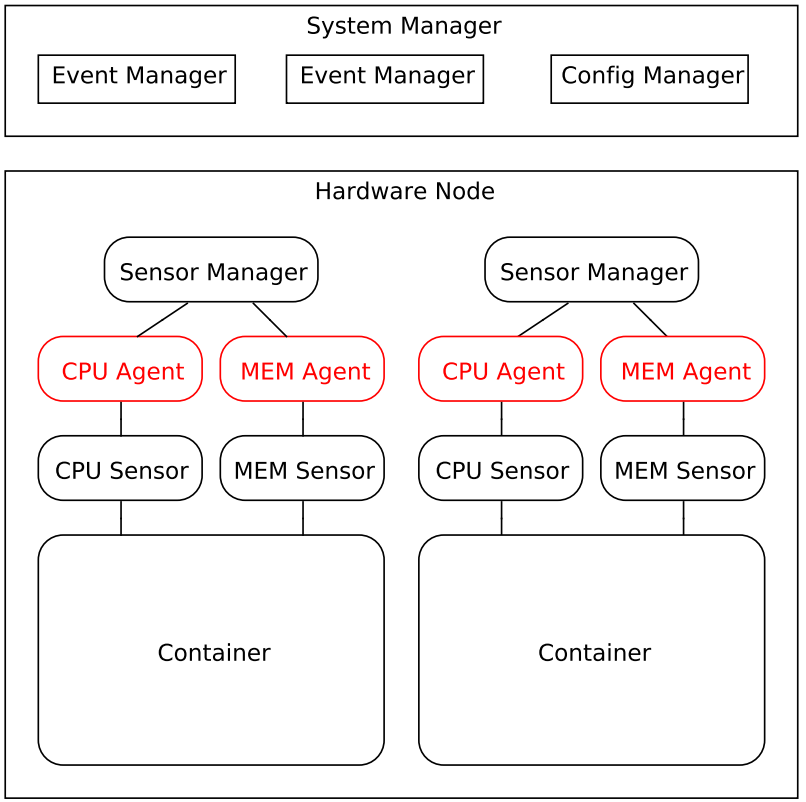


Figure: Golondrina's architecture

Collecting Resource Usage Data

- Monitoring requires some form of instrumentation, which we refer to as a *Sensor*, specific to each managed resource
- A *Sensor Agent* provides a standard interface to sensors
- The set of sensors on a hardware node is managed by a *Sensor Manager*
- An entity that changes the system is referred to as an *Actuator*; similar to sensors, there is an *Actuator Manager* and *Actuator Agent* (not shown)

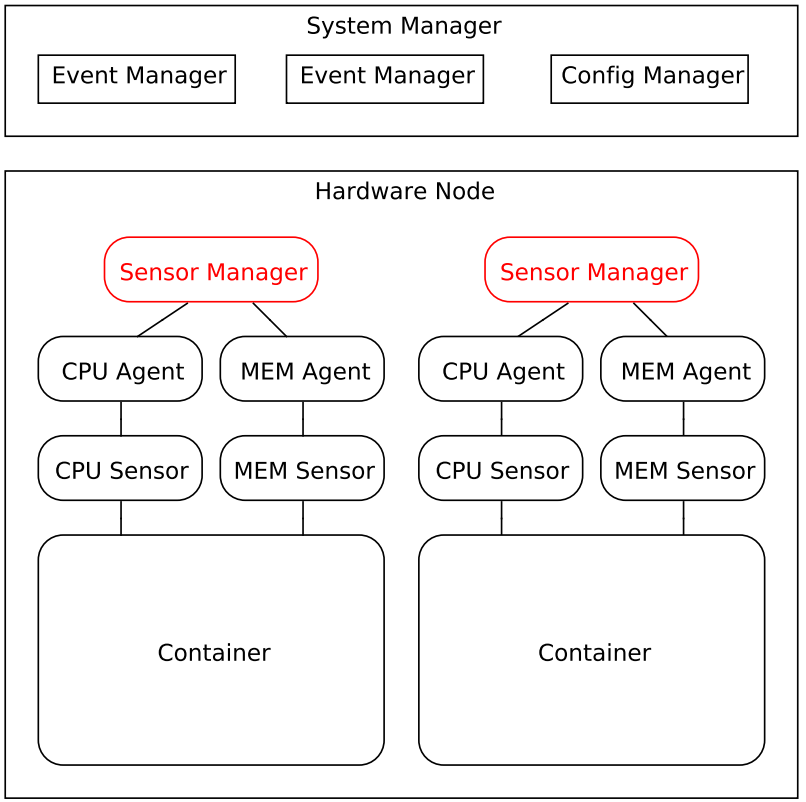


Figure: Golondrina's architecture

Collecting Resource Usage Data

- Monitoring requires some form of instrumentation, which we refer to as a *Sensor*, specific to each managed resource
- A *Sensor Agent* provides a standard interface to sensors
- The set of sensors on a hardware node is managed by a *Sensor Manager*
- An entity that changes the system is referred to as an *Actuator*; similar to sensors, there is an *Actuator Manager* and *Actuator Agent* (not shown)

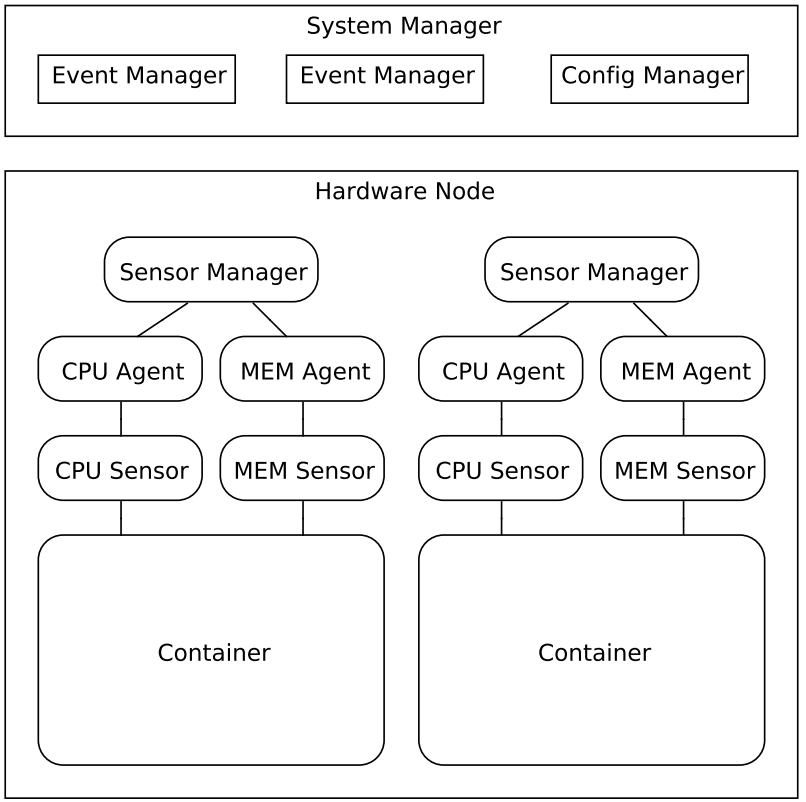


Figure: Golondrina's architecture

Analyzing Resource Usage Data

- The *System Manager* includes an *Event Manager* that receives sensor data that it analyzes to determine whether or not a significant event has occurred
- Significant events are identified by the evaluation of a condition associated with the event, e.g. $memoryStressScore > 0.80$
- The event detection server maintains a set of tuples in the form (e, p), where p represents a policy identifier and e represents the event expression, for each operational condition policy
- The evaluation of operational conditions can also be done by sensor agents

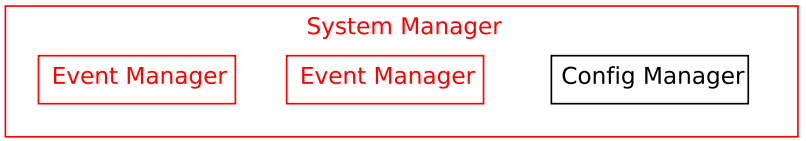
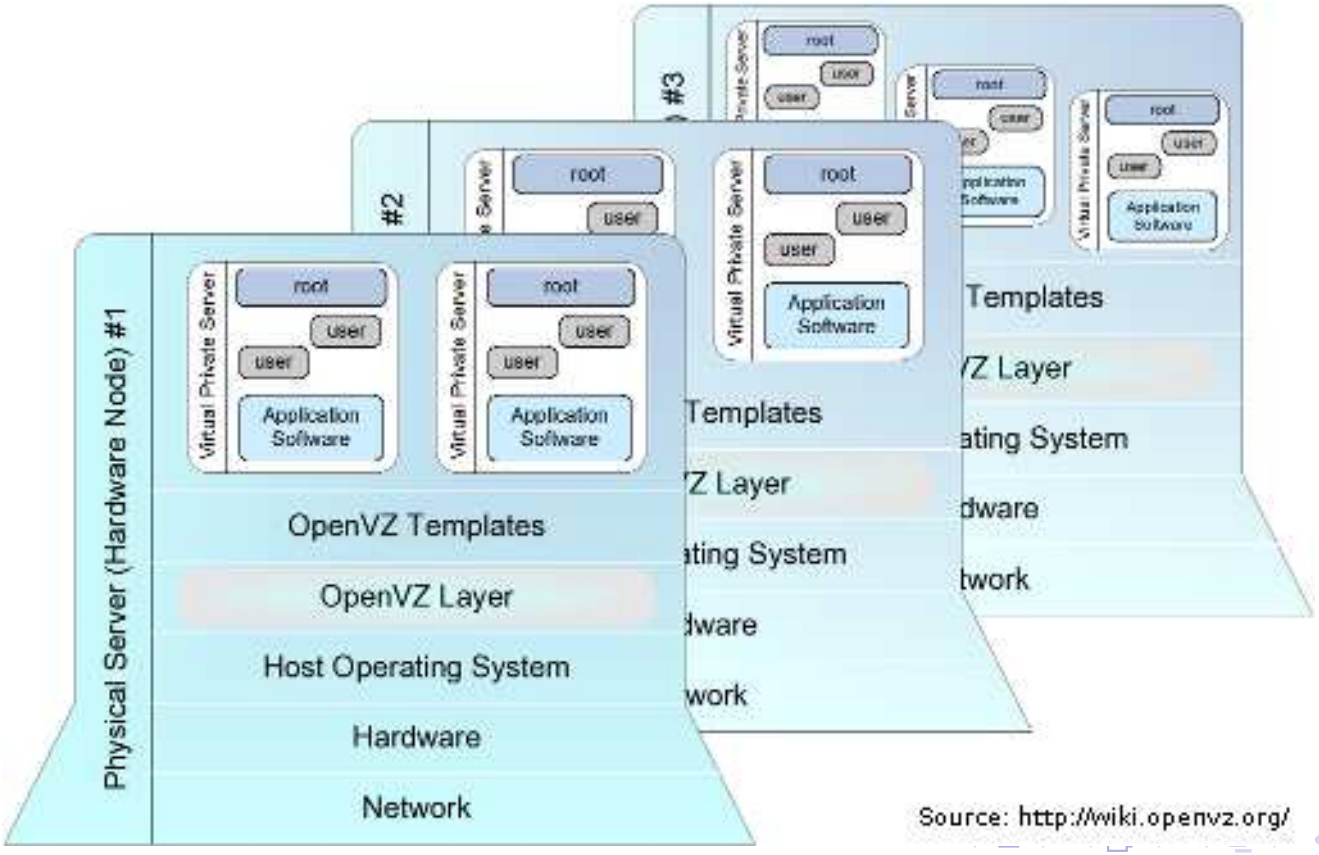


Figure: A subset of Golondrina's architecture

Underlying Resource Management Framework

- The Golondrina prototype currently uses OpenVZ
- OpenVZ is a Linux kernel modified to run isolated *containers* (virtual user-space environments)



How Memory is Allocated

- A process may increase its memory consumption by either explicit requests for more memory (e.g. calls to `malloc`) or stack expansion
- Like most current operating systems, OpenVZ allocates memory in units of *pages*
- We use the term *alloc* to refer to the number of pages currently owned by a container
- We use the term *allocUsed* to refer to the number of currently owned pages that have been written to

Many Per-Container Resource Limits Exist

allocationGuatantee number of pages guaranteed to be granted
allocationBarrier soft limit on number of pages that may be granted
allocationLimit hard limit on number of pages that may be granted

A memory allocation request for *n* pages is guaranteed to succeed if

$$n + alloc <= allocationGuarantee$$

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

$$n + alloc <= allocationBarrier$$

If $n + alloc > allocationBarrier$ then only a high priority request will succeed provided that *n* is less than the number of free physical pages and

$$n + alloc <= allocationLimit$$

Every request beyond *allocationLimit* fails

Outline

How a Memory Stress is Detected

A *memory stress score* is calculated at periodic intervals for each container according to the following algorithm:

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: $stressScore_i = \max(s_0, s_1, s_2, s_3)$;

How a Memory Stress is Resolved

Assume a container's memory stress score exceeds a threshold, identifying it as memory-stressed. The system will attempt to resolve the memory stress by applying the first action below that will not destabilize the system.

1. Increase the container's memory limits, allowing it to access more memory on the current hardware node
2. Migrate the stressed container to another hardware node where it can be given access to more memory
3. Other container-specific actions

If a hardware node is memory stressed but no containers are, then an attempt is made to migrate the container that is using the largest amount of memory on that hardware node.

Outline

The Prototype Implementation and Test Environment

- We have developed a prototype implementation called *Golondrina*
- Golondrina is written in Python and interfaces with OpenVZ
- Golondrina has three types of actuators:
  - *Memory-Adjustment Actuators*
  - *Migration Acuators*
  - *Replication Actuators*
- We tested the system using several containers spread across three identical harware nodes
- Each container hosted a PHP-based website that was sent HTTP requests

Several Experiments Were Used to Validate the Prototype

Experiment 0 The containers were given enough memory so that no memory-stress occurred

Experiment 1 The conatiners started with the minimum amount of memory required and policies were disabled

Experiment 2 Similar to Experiment 1 except local memory adjustments were enabled

Experiment 3 Similar to Experiment 2 excpet that local memory adjustments and migrations were enabled

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

Conclusion

- Early work in dynamic resource management with an emphasis on memory was presented
- The work presented assumes monitoring of utilization occurs periodically
- This information is used to adjust resource allocations in response to workload fluctuations
- This work is complimentary to work that focuses on server consolidation
- The approach to adjusting memory limits was straightforward; future work will focus on adaptive resource control and incorporate information gained from experiments with the prototype