SAND2010-0019C

Combining Virtualization, Resource Characterization, and Resource Management to Enable Efficient High Performance Compute Platforms Through Intelligent Dynamic Resource Allocation

J. Brandt*, F. Chen°, V. De Sapio*, A. Gentile°,
J. Mayo*, P. Pebay*, D. Roe*, D. Thompson*, and M. Wong°
Sandia National Laboratories MS *9159 / °9152
P.O. Box 969, Livermore, CA 94551 U.S.A.
{ovis}@sandia.gov

Abstract

Improved resource utilization and fault tolerance of large-scale HPC systems can be achieved through fine-grained, intelligent, and dynamic resource (re)allocation. We explore the components and enabling technologies necessary for the creation of an interacting system that could provide this capability: specifically 1) Scalable monitoring and analysis to inform resource allocations decisions, 2) Virtualization to enable dynamic reconfiguration, 3) Resource management for the combined physical and virtual resources and 4) the overall orchestration of the allocation, evaluation, and balancing or resources in a dynamic environment. We discuss both general and HPC-centric issues that impact the design of such as system. Finally, we present our proof-of-concept system, giving both design details and examples of its application in real-world scenarios.

1 Introduction

As the compute nodes of high performance compute (HPC) clusters become more complex and powerful, the required complexity of a resource management system for efficiently utilizing these resources increases dramatically. When there was only a single processing unit per compute node the scheduling choices were relatively simple e.g. allocate a compute node per application process. With the introduction of multiple processing units per compute node (typically 2) the complexity became a bit more complex in that there was now the possibility of contention for both memory and network bandwidth. Contention for network bandwidth was largely ignored and the main decision point was memory footprint. Current systems, however, not only have more processors per node (4 - 8) but each processor currently has four or six cores with more coming in the future. Some systems such as Los Alamos National Lab's Roadrunner system [7] have nodes consisting of a heterogeneous mix of processors. Making scheduling decisions in these environments can still be simple e.g. allocate a set of nodes to a user application based on how many of which type of processing unit they ask for and round up to the node level. However, such simple scheduling decisions are made at the possible expense of overall platform efficiency and perhaps even application performance.

In order to make more intelligent decisions the scheduling system would need more insight into how resources are actually being utilized by the applications running on them. Additionally, in order to correct for contention or underutilization, the scheduling system should have a mechanism for migrating processes from one resource to another in order to minimize contention and maximize utilization.

In this paper we explore the components and enabling technologies for such a system. Specifically we address the following: 1) Obtaining and calculating meaningful low level resource utilization information through a scalable monitoring and analysis system designed for real time periodic collection and analysis of hardware level information for doing failure prediction to enable more fault tolerant HPC systems. 2) Utilization of virtualization technology for providing process level mobility for migration to enable resource rebalancing in response to application requirements, system state, and/or failure prediction. 3) Utilization of a open source resource management system (SLURM) commonly used in HPC environments for management and allocation of both the physical and the virtual resources. 4) Orchestration of the allocation, evaluation, and balancing of resources with respect to the applications running on a system via our proof-of-concept Controller code.

The paper is organized as follows: In Section 2 we discuss current HPC usage models as we know them. This is based on our experience and discussions with system administrators of our capacity compute clusters at Sandia National Laboratories. We use these usage models to motivate applying virtualization and ”Cloud” i.e. Infrastructure as a Service (IaaS) concepts to HPC platforms and discuss considerations in such application in Section 3. Section 4 gives insight into resource allocation, contention, and monitoring issues. Our system design including all of the high level components and how they interact is covered in Section 5 followed by some examples of real use of the system for allocation, balancing, and relocation due to impending failure in Section 6. Finally we discuss related work and summarize in Sections 7 and 8, respectively.

2 HPC Usage Models

Although there are exceptions, modern HPC platforms typically still use a per-node allocation scheme when allocating resources to applications. Given the increasing number of resources available in a compute node (our testbed nodes have 4 CPUs, 24 cores, and 32 GB of memory) this can be very wasteful. As nodes grow in computational resources, allocation will certainly have to be done on a finer granularity whether on a per-CPU, per-core, or some other granularity. With this, however comes increased possibility for contention in some subsystem.

A barrier to higher-granularity resource allocation is a lack of insight into the run-time resource demands of an application. Typically, how the allocated resources are utilized by an application is not monitored or, if it is, the information is only used for gross machine utilization statistics. Some of the major factors in allocation requests are memory requirements, CPU utilization, memory bandwidth, and node interconnect bandwidth. While memory footprint presents a hard limit on how many processes can be hosted on a node, we see many application runs on our production systems that utilize less than 20% of the 32GB available. Discussions with some users revealed that they preferred to run a single process per CPU, even though each CPU had multiple cores available, in order to avoid possible contention for L3 cache and main memory. The same users said that if there was a way to provide feedback on the actual utilization of these resources so that they could take better advantage of them they would. Of course they don’t want such measurements to have any impact on their applications run time.

Our goal, then, is to enable intelligent and dynamic placement of tasks on resources to not only minimize single job run-time where possible, but also maximize system throughput. Information from lightweight, run-time monitoring and analysis of resource utilization can not only provide feedback to the users and their applications but can also inform a resource management system that can orchestrate such intelligent, dynamic allocation though lightweight process migration mechanisms.

In addition to the obvious general benefits of improved resource utilization, such a capability would be of particular use in several scenarios in the HPC environment. For example, users doing development need to test their applications at scale, thus requiring a large number of resources but for a very short period of time. In our current HPC production systems, such users have to wait in a queue for for up to a week to run such a test. The situation would be vastly improved by the ability to place short jobs on resources already running long-lived jobs in a such a way as to have minimum impact on the long running jobs. The impact (e.g., time and memory footprint constraints) would be monitored and bounded by the resource management system. Also of note is that the users of a HPC platform typically all have to build their application codes for the platform on which they are running. Use of full virtualization would allow a user to utilize their system image and environment on any such enabled system. Additionally even if the host operating system were upgraded the user could use their image until such a time as they deemed it valuable to them to port to a later OS. This would decouple the system administrators upgrading a platform from the application programmers need to port to a new software platform. Currently one can not happen without the other which requires a lengthy process of upgrading a small section on which the application programmers must ensure their codes run before a whole system upgrade can be completed.

3 Virtualization and IaaS

As alluded to above, even with understanding of resource usage provided by an appropriate monitoring system, unless there is a mechanism to provide process mobility within the pool of platform resources, users would ultimately be responsible at launch time for appropriate use of resources based on feedback from their last similar application run. Virtualization then seems a likely candidate mechanism for such process mobility with the provision that the overhead doesn’t surpass the gains. Though there are other MPI based process migration mechanisms (e.g. LA/MPI integrated with BLKR [3] and CHARM++ [16]) virtual machine technologies such as Xen [15] and KVM [6] provide nice containers that can be migrated within a pool of interconnected resources transparently to an MPI application process running within though it should be noted that there are transport related dependencies that must be accommodated for technologies other than ethernet (used in this work), i.e. Infiniband and Myrinet, but these are beyond the scope of this paper. Use of such virtualization technologies allows the infrastructure presented to the user application to be tailored to the applications needs and to be dynamically located within the pool of physical resources as necessary to maximize performance and resource utilization (IaaS).

3.1 Overhead

There has been substantial work done to quantify and identify sources of overhead associated with running HPC applications in virtualized environments. While we have seen from 20% to over 100% overhead in running a particular HPC application in a Kernel Virtual Machine (KVM) environment on our testbed system when scaling from 1 to 240 processes, there are claims [9] of actual speedups running certain NAS Parallel Benchmarks in a Xen environment. Hardware vendors we have spoken with claim that more support is being built into the chip sets to maximize performance in these environments. Thus we are taking the approach of building the infrastructure for facilitating the efficient application of these resources to traditional HPC platform resource management. Additionally there are also potential gains to be had in the area of resiliency using a combination of such a management system together with a viable resource health monitoring facility. In that scenario, in order for virtualization to be a viable technology in HPC the cost of the overhead only has to be lower than the combination of the benefits in platform efficiency and the benefits from less frequent checkpoints.

3.2 Considerations in Provisioning and Mobility for MPI Applications

In this section we discuss the process of launching, migrating, and running an MPI application in a virtualized environment. In particular we will be discussing this in the context of KVM environments as this is what we are currently using in our testbed environment. We use oneSIS [10] to create the diskless NFSroot image used by all our KVMs. We use a Linux operating system with a 2.6.27 kernel which is what also is run on our compute nodes. When launching a particular instance, the boot image and maximum memory allocation need to be specified and the proper virtual networking set up. Since we are doing diskless booting we also require a properly configured dhcp server. Additionally we use numactl to bind each instance to a particular core and the memory region associated with its CPU. Once this KVM is launched on a machine it takes approximately 90 seconds to boot completely. At this time it is a fully functional linux host and can support an application with a memory footprint equal to the container size less the operating system and container itself.

Live migration is accomplished by first setting up a container to receive the virtual machine and then transferring the KVM state to the container. Again the host, core, and memory region are specified for a particular host. Note that the new container must be the same size as the original. The new container after state is transferred has the same identity as the original and processes running inside don't undergo any change (the migration is transparent). Though static migration is an option the advantage of live migration is minimal downtime as most state is transferred while the machine is still running and only when there is a minimal amount left is operation frozen and the remainder transferred. The time for these two phases is quite different. Our container setup takes about 0.8 seconds and is independent of specified container size. The migration phase takes substantially longer (10 - 13 seconds in our testbed for a 500MB KVM) but can vary depending on how much memory pages are changing during the migration process. Another factor in the migration phase is interconnect speed. In our testbed we use a gigabit ethernet interconnect and would expect the migration to be faster with a higher speed interconnect.

For single process applications migration can be performed at the whim of the entity performing migrations but some consideration needs to be given when migrating an MPI application. This is because it is possible for in flight messages to be lost that are destined for a process residing on a migrating KVM during the final phase of migration while its state is frozen. In order to preclude such loss we have written a `MPI_Barrier` wrapper that performs checks and cooperates with our migration coordination entity (described in section 5.4) in order to perform the migration, if requested, at a barrier in such a way that no such messages will be lost. This results in some additional overhead, the necessity to re-link an application in order to use this facility, and the loss of ability to migrate an MPI process at an arbitrary time. Without this mechanism, however, there is the possibility for the application to hang indefinitely due to a lost message.

4 Resource Considerations

When making decisions on process placement, which in this work we do by placing KVMs, it is important to take into consideration as many resources as possible. For instance if one were only to take into consideration memory footprint and ignore the CPU load of an application it would not matter on which processing element any KVM landed. In this case one might co-locate many KVMs on the same core of the same CPU. Though this would be fine from a system memory perspective it would clearly impact the performance of the KVMs as the one processing element would have to time slice between all KVMs each of which could only achieve a fraction of the performance possible if it resided alone on a processing element. Likewise taking into account memory activity and the load that will be put on the memory bus, network activity and both the internal and external network loads, and all other shared resources will be required in order to make sound decisions about which resources should host which processes.

Furthermore, *where* information comes from has a bearing on the validity of the information and how it should be interpreted. In this work we used the host resource view of both cpu and memory utilization to inform migration placement decisions. The reasons for not using all of the above information is that we don't currently have the tools to derive good approximations of memory bus utilization and our test MPI application is not network intensive.

4.1 Utilization & Viewpoint

Why viewpoint needs to be taken into account for proper resource utilization measurements is illustrated in Figure 1. CPU loads were generated for processes running in KVM's with single and multiple VM's per core. The realized utilization obtained from calculations from `/proc/stat` from the VM view and the Host view (i.e., reading from the VM's file system vs the Host's) are principally in agreement for 1VM/core, however the processes running in the virtualized environment when there are multiple KVMs per core (plotted as a single line as the results were the same) not only do not achieve their requested utilization levels (except at levels below 25% in this case) but their sums don't equal the utilization reported by the host for the core on which they are running. We believe this is because their apparent utilization depends on where their sleep takes place relative to when they are swapped out. The way our work load generator works is to spend a user specified fraction of a time interval in computation after which a `usleep` is performed for the duration of the interval. (We performed this test using our own load generator as well as another [8].)

4.2 Scheduling and Resource Management Issues

In this section we discuss some of the issues with managing both physical and virtual resources in a HPC environment with a variety of workloads and how we propose to address them.

In order to minimize adverse impact to user applications our proposed approach is to initially allocate resources according to the users specifications with the only difference being that the application would be run in a VM as opposed to directly on the hardware. Monitoring resource utilization under these conditions can then give insight as to how resources are being utilized. The results would not only provide feedback to the user for future allocation but would also be used to initiate migration of VMs to more appropriately utilize resources.

In the case where contention was observed, this dynamic reconfiguration would mean spreading out over more physical resources. Alternatively, if resources were being lightly used it could mean consolidation onto fewer physical resources thus freeing up more for other applications or co-locating another application to share resources rather than allocate dedicated resources to it. This would require either previous knowledge of the other application's operating characteristics or continuous monitoring and evaluation of the resource utilization. As mentioned previously, there are many factors to be taken into account when calculating resource utilization and decisions to migrate running processes should not be made too frequently or with too little information. For instance since the operational characteristics of an application may vary dramatically over phases of execution making a decision to perform a consolidation too soon in an applications execution sequence may just result in having to make the decision to spread it back out later on. Given the relatively high cost of performing a migration these should be done as infrequently as possible.

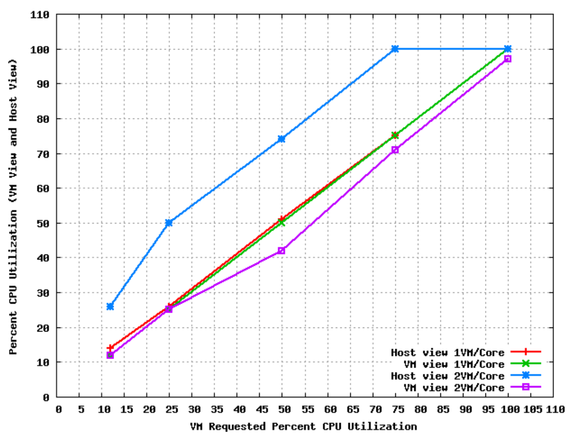


Figure 1. Percent CPU utilization requested in a VM vs. measured as viewed on the host machine and within the VM. Requested utilization may not be realized due to contention. Host and VM views may be inconsistent.

Another scenario is a user who wants to test his latest code at scale but doesn't want to have to wait in the queue for a long period of time. In this case (memory footprints permitting) the application could be co-located with any number of other applications which could potentially be adversely impacted but only for the short duration of the test.

In a final scenario, a high priority user needs to run and normally other jobs would be terminated so the high priority job could run. In such a case these lower priority but well-balanced jobs could be temporarily consolidated or frozen and saved to disk in order to free up enough resources for the high priority job and continue making progress but with degraded performance until such a time as they either finished or appropriate resources were freed and they were redistributed to them.

Finally, the dynamic nature of migration adds complexity to resource monitoring and management. The continuous evaluation of resources of both physical and virtual entities with respect to their job state means that dynamic tracking of the fluid job-to-resource mapping must take place. Further, traditional schedulers do not have the facility for adjusting job allocations at run-time. We have had to implement such features in our system described in Section 5.

5 System Design

In this section we first discuss the component parts of our IaaS enabled HPC system shown in Figure 2. We follow this section with some real world examples of the use of such a system in our testbed environment.

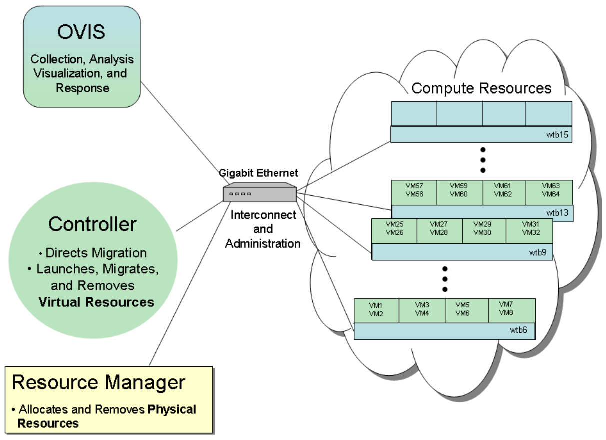


Figure 2. High level view of our proof of concept IaaS enabled HPC system.

5.1 Compute Resources

Just as important as all of the support systems is the actual hardware of which the compute resources are comprised. In order for virtualization to be viable in a HPC setting there must be hardware support. Older systems lacking such support could still take advantage of mobility but the lack of performance due to computational overhead and lack of hardware I/O support for advanced networking technologies, i.e. Infiniband, would render such systems ineffective for HPC applications. Our particular testbed environment is comprised of ten nodes each with 4 AMD 2.2 Ghz Istanbul Opteron processors with 32GB of memory. In the examples shown we use our 1 gigabit ethernet interconnect as we just recently upgraded from the Barcelona processor which didn't provide virtualization support for Infiniband and currently we have not done the necessary reconfigurations.

5.2 Scalable Monitoring

As mentioned previously, understanding how system resources are being utilized both individually and collectively is of paramount importance when making resource allocation decisions. In a real system this may mean real time monitoring of tens to hundreds of thousands of computational units and their associated computational load, memory usage and bandwidth, network utilization, etc. as well as similar metrics for the virtual machines running on them. Such monitoring must be of high enough fidelity to allow timely decisions to be made when resources are being severely oversubscribed or failure is predicted but at the same time be lightweight enough to not be a significant contributor to resource contention. In order to accomplish this we use OVIS [11], our monitoring and analysis system which has been principally developed to scalably collect and analyze just such data for the purpose of failure prediction. We have previously proposed [1] that OVIS's monitoring and analysis capabilities could enable intelligent resource utilization decisions necessary for HPC in Cloud computing environments.

OVIS utilizes a distributed database for data storage and a lightweight daemon running on each device for which data is to be collected (compute node and VM in this case but can include network and storage devices also) which directly inserts information at regular time intervals into the database. Parallel analysis engines are used to compute models against which individual ensembles of measurements are compared for detection of either anomalous behavior or signatures indicative of problems. For this proof of concept system we have utilized some of the data being collected for this purpose to also compute resource utilization information – specifically regarding memory and CPU utilization for this study. OVIS is then used to inform our Controller subsystem of resource contention as well as impending failure.

5.3 Resource Management

We leverage the SLURM (Simple Linux Utility for Resource Management) [13] resource management (RM) system which is commonly used in HPC systems. This RM provides facilities for maintaining separate resource partitions, launching batch jobs on resources, running predefined prolog and epilog scripts for setting up and cleaning resources, and can write pertinent information about allocations and the jobs running on them to a MySQL database. SLURM can manage resources on a per-node, per-CPU, and per-core granularity. We are currently utilizing it in a per-node management mode to preclude an application from obtaining resources within the managed virtual environment which would then not be taken into account in the resource utilization calculations.

The current release of SLURM (v2.0) does not provide the ability for determining and tracking the dynamic virtual to physical resource mapping information we require. In our overall system, then we keep both a virtual and physical partition in SLURM for resource allocations and job information, with the additional tracking information as part of the capabilities of the Controller, which is described in the next subsection.

5.4 Orchestration by the Controller

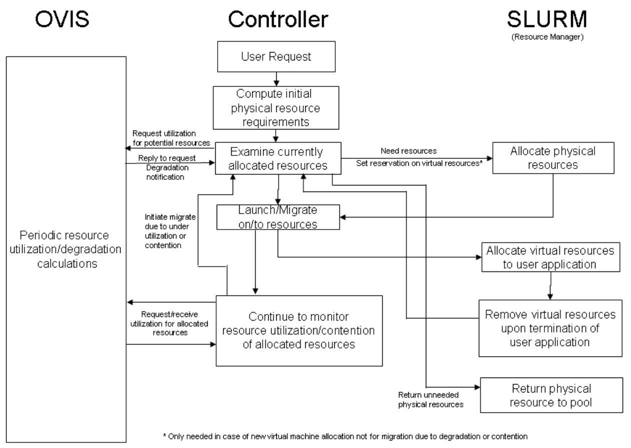

OVIS
Controller
SLURM
(Resource Manager)
User Request
Compute initial physical resource requirements
Request utilization for potential resources
Reply to request Degradation notification
Examine currently allocated resources
Need resources
Set reservation on virtual resources*
Allocate physical resources
Periodic resource utilization/degradation calculations
Initiate migrate due to under utilization or contention
Launch/Migrate on/to resources
Allocate virtual resources to user application
Request/receive utilization for allocated resources
Continue to monitor resource utilization/contention of allocated resources
Remove virtual resources upon termination of user application
Return unneeded physical resources
Return physical resource to pool
* Only needed in case of new virtual machine allocation not for migration due to degradation or contention

Figure 3. High level view of our proof of concept IaaS enabled HPC system.

At the heart of this system is our Controller whose interactions with the monitoring system (OVIS) and the resource management system (SLURM) is diagrammed in Figure 3 and described in detail in this section. There is an assumption in this diagram that OVIS is continuously monitoring all pertinent hardware related measurable attributes on each compute node of the system with some specified periodicity and so has current information with respect to the utilization of resources that can be returned to the Controller upon request, or pushed to the Controller in case of emergency (e.g., impending failures either due of resource oversubscription or predicted failure).

An allocation cycle begins with a user request which in our case is sent to the Controller rather than SLURM but using the same syntax as though it were being sent to SLURM with the exception that it accepts some additional arguments which allow the user to specify how much memory they expect each process to consume, a maximum packing density (cores/CPU), and expected process to process bandwidth. The Controller currently passes the physical allocation request straight through to SLURM, if no Controller specific flags are specified, and the physical resources, if available, are allocated to the Controller. If Controller specific flags are specified the Controller will make the physical allocation request based on its evaluation of the resources required to satisfy the request. For example, if the user is requesting 16 processors and has specified a memory size of less than 1.8GB per process the Controller would know, based on how the physical resources are interconnected within the actual compute nodes, that utilizing the first 16 cores of a compute node would mean sub-optimal placement of some processes with respect to the memory they would be using and hence that it should request an allocation from SLURM that will allow it to satisfy the user request while utilizing at most 8 GB per CPU including host OS overhead.

Once the Controller figures out the job resource requirements it next examines the pool of resources it currently controls. Note that when the Controller receives a resource allocation from SLURM it is issued to the Controller and not to the user application. The Controller maintains context on how the pool of resources under its control are being utilized by user applications and makes requests of SLURM on a compute node granularity though it will allocate to user applications on a per processing element granularity. Thus if a user requests a single processing element and the Controller has no free resources that can satisfy the request while continuing to service the running applications, the Controller will request another compute node in order to satisfy the request. If, in the example of the previous paragraph, it had 4 compute nodes each with a free CPU whose directly associated memory was also free it would utilize these rather than request additional resources unless the inter-process bandwidth was specified and would exceed that available over the associated interconnect.

As part of this step the Controller would contact OVIS with a request for actual resource utilization information on the potential target resources in its pool. This step is necessary to inform the Controller about resource usage not bounded by allocations such as network traffic and memory bus utilization (not currently being collected) as this could have significant bearing on what additional resources may or may not be effectively utilized if allocated. As previously mentioned, there are many factors to be taken into account when making resource utilization decisions and so, in practice in our proof-of-concept system, extreme generality of all requests and (re-)allocation scenarios is not supported, and rather, at this point a few common rulesets are applied.

The next step, after determining which physical resources will be allocated to an application, is to set up the virtual environment. This is initiated by the Controller at which time the maximum memory occupancy must be specified. Once the KVM is launched the size cannot be changed without destroying and rebooting it. This of course would require checkpointing the application and restarting it in the new containers which is a very costly operation. The boot time of our image is approximately 90 seconds which is a lot of overhead for short lived applications. Fortunately migration only takes approximately 10 to 20 seconds which provides the opportunity to maintain a pool of idle VMs that can be migrated to appropriate resources when needed. The only issue is that of memory size which we expect can be anticipated over a training period. In order to discover what the true footprint of an application is, however, one must query `/proc/meminfo` from inside the VMs as this is opaque to the host. With KVM in particular we found that upon initial launch the host reports relatively little memory usage with respect to the maximum container size. Upon migration however, the host reports the total KVM container size as being used and migration will fail if the maximum size exceeds available resources on the receiving host even though the memory utilization on the original host may have been relatively small (see Section 6.1). As an example, upon launch using our operating system and specifying a memory size of 2.0GB the original host sees about 250MB being used by the KVM process. Upon migration the receiving host sees the whole 2.0GB being used. Thus, though the original host may be oversubscribed with respect to the sum of the maximum sizes of the original KVMs, target hosts of subsequent migrations may not. Once the KVMs have been booted and are recognized by SLURM the Controller submits the original job request to SLURM on behalf of the original user.

Separate from initial allocation of virtual resources to an application is the periodic data collection to be used in assessment of the state of actual resource utilization and, in particular, contention. We plan such automated assessment in order to minimize wall clock completion time for each job and maximize resource utilization and system throughput. Currently this is in the experimental phase and is performed for at least a subset of the potential resources upon a new job entering or a job exiting the system as described in Section 6.1. The detection by OVIS of failure indicators is communicated to the Controller without request and is acted upon as described in Section 6.2.

In the event that a dynamic configuration of job-resource mappings is deemed advantageous (such scenarios have as been described in previous sections), the Controller is responsible for the launching of new containers, arranging resources with the Resource Manager, and performing the migration. For the case of MPI-based applications, orchestration of the migration is performed in concert with the `MPI_Barrier` wrapper discussed in Section 3.2. New virtual-to-physical mappings are maintained by the controller to enable continuous application-centric monitoring of the dynamic environment. These are also necessary at allocation time as well, as the virtual-to-physical jobs mapping would typically not be one-to-one based on the granularity of resource assignments and dynamic changes.

Once the job runs to completion or the time limit is met SLURM writes out results as normal and informs the Controller which then removes the virtual resources. At this point the Controller, can return completely unused resources to the physical resource pool or allocate them to some other application via the process described above.

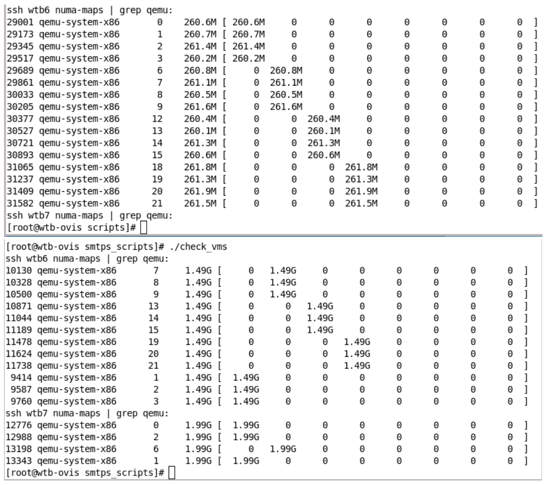
6 Real World Examples

In this section we give some examples and show results of two real world scenarios. The first is a case that is real world in that there are applications which begin with a smaller memory footprint than where they end up. As a result the total memory required by the end of the run is what would need to be allocated for at the beginning. Currently an application must wait in the queue until resources are available to satisfy the total required allocation. We demonstrate a scenario where we can utilize some aspects of KVM virtualization and migration to give such a user earlier access to compute resources than would be currently possible. The second is based on a failure mechanism that we see in one of our production clusters at Sandia. This failure occurs when the system is left, for some currently unknown reason, in a state with abnormally high Active memory usage after having been allocated to and cleaned after completion of an application run. In this case future jobs being allocated these resources have less memory headroom available to them than their peer processes on other resources and hence have a higher probability of failure if they have significant memory usage.

6.1 Speculative Oversubscription

In this section we describe the use of a characteristic of the KVMs just mentioned that upon initial launch they present a much smaller footprint to the host node than their maximum allocated size. We use this feature to speculatively launch a job that oversubscribes the physical resources if the processes running in the KVMs expand to fill them. This provides the application with a space to run while it waits for more resources to come available. We illustrate this by launching a sixteen process job each requiring a maximum of 2GB but with the known characteristic of using that over time and not requiring it all up front. A screenshot of the numa-maps output showing the per-core memory in use at allocation is shown in Figure 4.

VM's were initially placed on 4 of the 6 available cores per cpu on a single node. Our system continuously monitors Active memory usage. As the application progresses,



```
ssh wtb6 numa-maps | grep qemu:
29001 qemu-system-x86      0    260.6M [ 260.6M     0      0      0      0      0      0      0  ]
29173 qemu-system-x86      1    260.7M [ 260.7M     0      0      0      0      0      0      0  ]
29345 qemu-system-x86      2    261.4M [ 261.4M     0      0      0      0      0      0      0  ]
29517 qemu-system-x86      3    260.2M [ 260.2M     0      0      0      0      0      0      0  ]
29689 qemu-system-x86      6    260.8M [      0 260.8M      0      0      0      0      0      0  ]
29861 qemu-system-x86      7    261.1M [      0 261.1M      0      0      0      0      0      0  ]
30033 qemu-system-x86      8    260.5M [      0 260.5M      0      0      0      0      0      0  ]
30205 qemu-system-x86      9    261.6M [      0 261.6M      0      0      0      0      0      0  ]
30377 qemu-system-x86     12    260.4M [      0      0 260.4M      0      0      0      0      0  ]
30527 qemu-system-x86     13    260.1M [      0      0 260.1M      0      0      0      0      0  ]
30721 qemu-system-x86     14    261.3M [      0      0 261.3M      0      0      0      0      0  ]
30893 qemu-system-x86     15    260.6M [      0      0 260.6M      0      0      0      0      0  ]
31065 qemu-system-x86     18    261.8M [      0      0      0 261.8M      0      0      0      0  ]
31237 qemu-system-x86     19    261.3M [      0      0      0 261.3M      0      0      0      0  ]
31409 qemu-system-x86     20    261.9M [      0      0      0 261.9M      0      0      0      0  ]
31582 qemu-system-x86     21    261.5M [      0      0      0 261.5M      0      0      0      0  ]
ssh wtb7 numa-maps | grep qemu:
[root@wtb-ovis scripts]# 
```

```
[root@wtb-ovis smtps_scripts]# ./check_vms
ssh wtb6 numa-maps | grep qemu:
10130 qemu-system-x86      7     1.49G [     0   1.49G     0      0      0      0      0      0  ]
10328 qemu-system-x86      8     1.49G [     0   1.49G     0      0      0      0      0      0  ]
10500 qemu-system-x86      9     1.49G [     0   1.49G     0      0      0      0      0      0  ]
10871 qemu-system-x86     13     1.49G [     0       0  1.49G     0      0      0      0      0  ]
11044 qemu-system-x86     14     1.49G [     0       0  1.49G     0      0      0      0      0  ]
11189 qemu-system-x86     15     1.49G [     0       0  1.49G     0      0      0      0      0  ]
11478 qemu-system-x86     19     1.49G [     0       0      0  1.49G     0      0      0      0  ]
11624 qemu-system-x86     20     1.49G [     0       0      0  1.49G     0      0      0      0  ]
11738 qemu-system-x86     21     1.49G [     0       0      0  1.49G     0      0      0      0  ]
 9414 qemu-system-x86      1     1.49G [  1.49G      0      0      0      0      0      0      0  ]
 9587 qemu-system-x86      2     1.49G [  1.49G      0      0      0      0      0      0      0  ]
 9760 qemu-system-x86      3     1.49G [  1.49G      0      0      0      0      0      0      0  ]
ssh wtb7 numa-maps | grep qemu:
12776 qemu-system-x86      0     1.99G [  1.99G      0      0      0      0      0      0      0  ]
12988 qemu-system-x86      2     1.99G [  1.99G      0      0      0      0      0      0      0  ]
13198 qemu-system-x86      6     1.99G [     0   1.99G     0      0      0      0      0      0  ]
13343 qemu-system-x86      1     1.99G [  1.99G      0      0      0      0      0      0      0  ]
[root@wtb-ovis smtps_scripts]# 
```

Figure 4. Numa-maps output of VM size at allocation (t) and after migration (b).

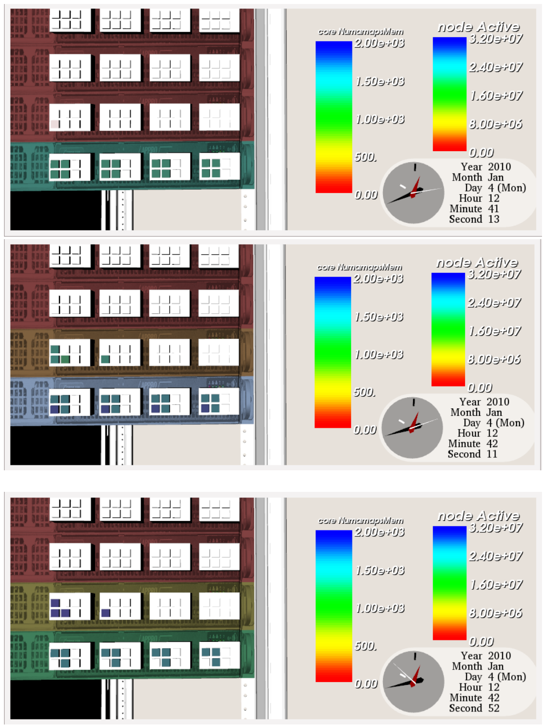


Figure 5. Intentional initial oversubscription of resources is adjusted by migration when contention is detected as the application progress. Screenshots of the OVIS monitoring and analysis system showing memory utilization on the host and VMs. Triggering (t), during (m), and after (b) migration.

it continues to consume memory, increasing in size until OVIS detects that a dangerously high Active memory level is reached for this node (Figure 5(t)), and notifies the Controller. When this occurs, the Controller obtains free resources that will satisfy the application's requirements, launches containers there, and informs the MPI-based application of the need to migrate when a barrier is reached. The application then informs the Controller when it is ready and the Controller migrates a single KVM from each CPU to the new host leaving each CPU hosting 3 KVMs (Figure 5(m and b) with a maximum size of 2GB which will fit within the directly connected memory for each CPU. This is necessary because for performance reasons we use numactl to bind each KVM to the memory directly associated with the CPU on which it resides.

Time traces of the memory utilization for both VMs and nodes during this process are shown in Figure 6. In the top plot the memory size of the VMs obtained via numa-maps is plotted for 2 representative VMs - one not involved in the migration and one before and after migration. The latter 2 overlap during the migration. The bottom Figure shows the Active memory utilization for both the node being migrated from and the node being migrated to. Triggering of the migration occurs when the Active memory on the node exceeds 75% of its total. The migration time can be seen to be about 50 seconds here which corresponds to the time it takes to transfer the 6GB (4 x 1.5GB) of state from the first host to the second over a one gigabit/sec interconnect. This would be about 5 sec using 10 gigabit ethernet.

6.2 Health Degradation

One of the common failures in one of Sandia's production HPC capacity clusters is a user application having an MPI process on a compute node killed by an "OOM killer" process due to memory utilization being too high. In the case that the user application is consuming too much memory this would be expected, but typically this happens on a compute node that has been left in a state with high Active memory as described above [2] and the rest of the process group on other resources are well behaved. Having one of the MPI processes killed on one such ill-behaved resources kills the whole job.

In this case, we have simulated the failure precursor condition by running an additional process on one of the nodes. Our system continuously monitors the memory utilization not only on a per node basis, as above, but also analyzes it with respect to all nodes involved in the job. Detection of a potential pre-failure condition occurs when 1) a high memory threshold has been reached on a particular node and

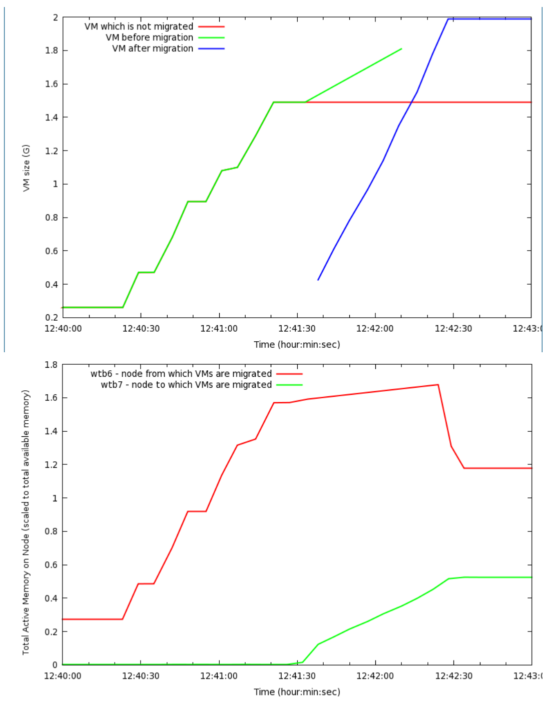


Figure 6. Memory utilization through time during a live migration triggered by excessive memory consumption on the node. VM utilization (top), node utilization (bottom).

2) the node having this condition has an abnormally high memory utilization with respect to the other nodes hosting other processes of the same MPI job as well. Under the assumption of generally well-balanced jobs, this indicates that the problem is one of the node and not of the application. Upon detection the combined condition OVIS sends a message to the Controller which then flags the affected processes (in this case all processes on the node with the problem) to notify them to let the Controller know when they are at their next barrier so that the Controller can migrate their host KVMs to a free node.

Figure 7 shows OVIS screenshots of our testbed running a 64 process MPI job during this example. The shot on the left shows the second node on the bottom to have much higher Active memory (blue is higher, red lower) than those hosting the peer processes (green). The shot on the right shows that after migration the amount of active memory is a little higher in the new host but this is due to a migration causing the total memory allocated to a KVM to be used on the new host node (as described in Section 5.4 while the other nodes have not used memory up to their limit.

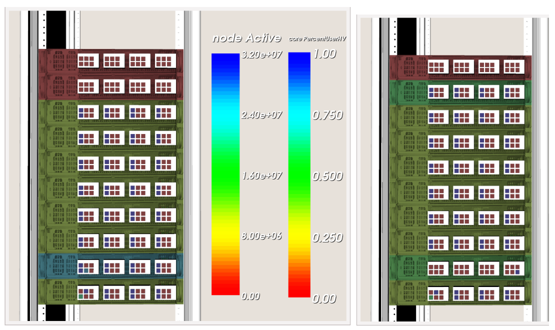


Figure 7. System discovery of abnormal memory utilization on a node triggers migration of the endangered processes (left).Migration is complete (right).

7 Related Work

There has been substantial work in the area of virtual resource management by cloud providers and others with respect to relatively low performance usage models or where a user is expected to set up and own an environment in which they develop. Amazon's EC2 and Eucalyptus's open source version of EC2 [4, 5] seem more targeted at setting up virtual systems for developers with security, service level agreements, and ease of custom configuration by the user being priorities. The use cases for these systems, though they don't preclude such use, don't appear to lend themselves well to the fluid, high throughput, and relatively open environments typically being used for high performance computing applications. The user needs may be different for each request and a comparatively lightweight virtual infrastructure needs to be set up, used for the duration of an application run that may last from seconds to months, and then torn down. SLAs in these systems are meant to address access to high level resources such as CPU cycles, storage, and network bandwidth but do not address low level issues such as L3 cache contention, memory bus contention, or any other of the data transport and storage mechanisms internal to a compute node that must be shared by co-located processes and can have a dramatic affect on HPC application performance. Also these systems typically provide robustness to failure through redundancy which is a mechanism typically not available in HPC systems which are typically diskless.

There has been work done in the area of process migration both with and without the assistance of virtual machines [14, 16]. The case for doing process level migration is that it can be done by transferring much less state and hence the time involved can be substantially less (subsecond vs. seconds to tens of seconds). Additionally the impact to the running application is much less as overhead associated with such migration is typically only incurred at the time of migration whereas the overhead of running in a virtualized environment is incurred over the lifetime of the application. The downside is that the application wishing to use it must build with a particular MPI implementation. In recent years though, the overhead being reported for running HPC applications in virtualized environments has substantially decreased to the point where it is beginning to look attractive as an environment for these applications. The main benefits being transparency with respect to physical location and the ability to run in an environment quite disjoint from that of the underlying physical resources. Nagarajan et. al. [9] have done proof of concept work with respect to using a virtualized environment to enhance fault tolerance through proactive migration from unhealthy to healthy resources. In this work they also investigate the overhead of running the NAS parallel benchmarks in a virtualized environment using Xen. The results of this study are that the average case overhead is 4.4%. In the best case they actually see a slight speedup which, pending further investigation, they attribute to "memory allocation policies and related activities of the Xen Hypervisor".

Shainer et. al. [12] showed that by proper relative placement of application processes on shared hardware, wall time to completion for a set of applications on a given hardware platform can be decreased when compared to running them disjointly. This is due to the difference in required resources of the two applications. By proper resource requirement analysis and placement, contention for resources is minimized as is wall time. Of course if one of these individual jobs had a high priority and a hard time limit it could have been run faster by having all resources devoted to it.

We rely on all of this background work as motivation for our proof of concept system to tie all of these disparate but complementary pieces together. In particular, the applicability of using virtualized environments for large scale HPC applications hinges on it being/becoming a low overhead technology which studies in this area seem to indicate will become a reality in the near future. Additionally our own work in the area of failure prediction in large scale HPC platforms [11] has provided us with the necessary monitoring and analysis component which we also plan to build on for general purpose resource utilization/contention analysis as discussed in Section 5.2 and as proposed specifically for resource allocation for HPC in Cloud Computing Environments in previous work [1].

8 Summary and Conclusions

In this paper we have described how we believe managing resources in large scale HPC clusters can be made more efficient by the use of the concept of IaaS together with mechanisms for providing it such as virtualization technologies, scalable monitoring and analysis, generalized resource management, and a coordination mechanism to make them all work together. We have described why systems such as Eucalyptus and EC2 aren't just drop in technologies for this job. We have designed and implemented a proof of concept system which we believe is a good basis for such functionality. Finally, though we have not yet achieved full functionality, we have applied our proof-of-concept system to some real world problems (predicted physical resource failure and load spreading) using an MPI application to demonstrate its applicability to the HPC domain.

References

[1] J. Brandt, A. Gentile, J. Mayo, P. Pebay, D. Roe, D. Thompson, and M. Wong. In *Proc. 23nd IEEE Int'l Parallel & Distributed Processing Symposium (5th Workshop on System Management Techniques, Processes, and Services)*, 2009.

[2] J. Brandt, A. Gentile, J. Mayo, P. Pébay, D. Roe, D. Thompson, and M. Wong. Methodologies for advance warning of compute cluster problems via statistical analysis: A case study. In *Proc. 18th ACM Int'l. Symp. on High Performance Dist. Comp. (2009 Workshop on Resiliency in High-Performance Computing)*, 2009.

[3] J. Cao, Y. Li, and M. Guo. Process migration for MPI applications based on coordinated checkpoint. In *Proc. 11th IEEE Int'l Conf. on Parallel and Distributed Systems (ICPADS'05)*, 2005.

[4] EC2. `http://aws.amazon.com/ec2/`.

[5] EUCALYPTUS. `http://www.eucalyptus.com`.

[6] KVM. `http://www.linux-kvm.org`.

[7] LANL ROADRUNNER. `http://www.lanl.gov/orgs/hpc/roadrunner/arch.shtml`.

[8] LOOKBUSY. `http://www.devin.com/lookbusy/`.

[9] Nagarajan, Mueller, Engelmann, and Scott. Proactive fault tolerance for HPC with XEN virtualization. In *Proc. ACM Int'l Conf. on Supercomputing (ICS 07)*, 2007.

[10] ONESIS. `http://onesis.sourceforge.net/index.php`.

[11] OVIS. `http://ovis.ca.sandia.gov`.

[12] G. "Shainer, T. Liu, J. Layton, and J. Mora. Scheduling strategies for HPC as a service HPCAAS. In *Proc. IEEE Int'l Conf. on Cluster Computing and Workshops (CLUSTER09)*, 2009.

[13] SLURM. `https://computing.llnl.gov/linux/slurm`.

[14] C. Wang, F. Mueller, C. Engelmann, and S. Scott. A job pause service under LAM/MPI+BLCR for transparent fault tolerance. In *Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS07)*, 2007.

[15] XEN. `http://www.xen.org/`.

[16] G. Zheng, L. Shi, and L. Kale. FTC-CHARM++: An in-memory checkpoint-based fault tolerant runtime for CHARM++ and MPI. *IEEE Int'l Conf. on Cluster Computing (CLUSTER04)*, 2004.