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. Pébay*, D. Roe*, D. Thompson*, and M. Wong°
*Sandia National Laboratories MS *9159 / °9152 P.O. Box 969, Livermore, CA 94551 U.S.A.*
{{brandt, fxchen, vdesap, gentile, jmayo, pppebay, dcroe, dcthomp, mhwong}|{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 components and enabling technologies applicable to creating a system to provide this capability: specifically 1) Scalable fine-grained monitoring and analysis to inform resource allocation decisions, 2) Virtualization to enable dynamic reconfiguration, 3) Resource management for the combined physical and virtual resources and 4) Orchestration of the allocation, evaluation, and balancing of resources in a dynamic environment. We discuss both general and HPC-centric issues that impact the design of such a system. Finally, we present our prototype system, giving both design details and examples of its application in real-world scenarios.**

***Keywords*-virtualization; migration; resource management; IaaS; HPC; KVM;**

I. Introduction

As the compute nodes of high performance compute (HPC) clusters become more complex and powerful, the required complexity of a system for efficiently managing these resources increases dramatically. Making resource allocation 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. However, such simple resource allocation decisions are made at the possible expense of overall platform efficiency and application performance.

In order to make more intelligent resource allocation decisions the system would need more insight into how resources are actually being utilized by the applications running on them. Additionally, in order to dynamically correct for contention or underutilization, there should be a mechanism for migrating processes within a resource pool.

In this paper we explore components and enabling technologies for such a system, specifically addressing the following: 1) Obtaining meaningful low-level resource utilization information through a scalable monitoring and analysis system designed for real-time fine-grained collection and analysis of hardware level information. 2) Utilization of virtualization technology for process level mobility to enable resource rebalancing in response to application requirements, system state, and/or failure prediction. 3) Utilization of a resource manager commonly used in HPC environments, for management and allocation of both physical and virtual resources. 4) Orchestration of the allocation, evaluation, and balancing of resources with respect to applications running on a system and their resource requirements.

The paper is organized as follows: In Section II we discuss some current HPC usage models based on our experience and discussions with system administrators of our capacity compute clusters at Sandia National Laboratories. We use these 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 III. Section IV 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 V. Examples of use of the system for resource management and dynamic process relocation are in Section VI. We close with related work and summary.

II. HPC Usage Models

Typical scientific HPC applications consist of many parallel tightly-coupled MPI processes whose resource demands may change over their potentially long run-times. Modern HPC platforms generally use a per-node static allocation scheme when allocating resources to applications. Given the increasing number of resources available in a compute node (our 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 (e.g. per-CPU, per-core) and there is increased possibility for subsystem contention.

A barrier to higher-granularity resource allocation is lack of insight into the run-time resource demands of applications. 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.

Major factors in allocation requests include memory requirements, CPU utilization, memory bandwidth, and node

These authors were supported by the United States Department of Energy, Office of Defense Programs. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

interconnect bandwidth. While memory footprint presents a hard limit on how many processes can be hosted on a diskless node, we see many application runs on our production systems that utilize less than 20% of the 32GB available. In discussions, some users revealed that they prefer to run a single process per CPU, even when each CPU has multiple cores available, in order to avoid possible contention for L3 cache and main memory. Further, users would like to obtain feedback on the actual utilization of the resources so that they could take better advantage of resources.

Our goal is to enable intelligent and dynamic placement of tasks on resources to minimize single job run-time and maximize system throughput. Information obtained from lightweight, fine-grained, 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 allocation system to orchestrate dynamic intelligent allocation and placement. The resultant capability, then, in addition to providing the obvious benefits of improved resource utilization, could facilitate the quick turnaround of large short-lived jobs, e.g., those of parallel application developers needing to test at scale to verify that the application runs for a few time steps. Processes for such tests could be placed wherever they would fit (memory footprint) within the resource pool alongside running applications with other resource contention ignored for the short run period.

III. Virtualization and IaaS

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 previous application runs. Virtualization seems a likely candidate mechanism for process mobility provided that the overhead doesn't surpass the gains. Though there are other MPI-based process migration mechanisms (e.g., LAM/MPI integrated with BLKR [10] and CHARM++ [16]), virtual machine technologies such as Xen [7] and KVM [3] provide containers that can be migrated within a pool of interconnected resources transparently to an MPI application process running within, though there are transport related dependencies that must be accommodated for technologies other than Ethernet (which is 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 application's needs and to be dynamically located within the pool of physical resources as necessary to maximize performance and resource utilization (IaaS).

A. Overhead

Substantial work (see e.g., [11], [12], [14], [15] and references therein) has been 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 [12] of actual speedups running certain NAS Parallel Benchmarks in a Xen environment. Hardware vendors have built more support into chip sets to maximize performance in these environments. Thus we are taking the approach of building, in anticipation of decreased overhead, the infrastructure for facilitating the efficient application of these mechanisms to traditional HPC platform resource management. Additionally there are potential gains to be had in the area of resiliency by incorporating viable resource health monitoring facilities.

B. Considerations in Provisioning and Mobility for MPI Applications

In this section we discuss the process of launching, running, and migrating an MPI application in a virtualized environment. We address KVM environments as this is what we currently use in our testbed environment. We use numactl to bind each instance to a particular core and the memory region associated with its CPU. Once a KVM is launched and booted, 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.

Live migration is accomplished by first setting up a container on a computational resource to receive the virtual machine and then transferring the KVM state to the container. The resulting KVM 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, an 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. 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 memory page activity during the migration process. Another factor in the migration phase is interconnect speed. We currently use a gigabit Ethernet interconnect and expect the migration to be faster with a higher speed interconnect.

For single process applications, migration can be performed at any time but additional consideration needs to be given when migrating an MPI application as it is possible for in-flight messages to be lost that are destined for a process residing on a migrating KVM 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 (Section V-D) in order to perform the migration at a barrier in such a way that no messages will be lost. This results in some additional overhead and the necessity to re-link an application in order to use this facility. Without this mechanism, however, there is the possibility for an application to hang indefinitely due to lost messages.

IV. Resource Considerations

When making decisions on process placement, it is important to take into consideration the utilization/state of as many resources as possible. For instance, if one were only to consider memory footprint and ignore the CPU load of an application it would not matter on which processing element any VM landed. In this case one might co-locate many VMs on the same core of the same CPU. Though this may be fine from a system memory perspective, it would clearly impact the performance of the VMs as the one processing element would have to time slice between all VMs, each of which could only achieve a fraction of the performance possible were it to reside alone on a processing element. Likewise consideration of 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.

A. *Utilization & Viewpoint*

The information source also has a bearing on the validity of the information and how it should be interpreted. The effect of viewpoint on resource utilization measurements is illustrated in Figure 1. CPU loads were generated for processes running in KVM's with single and multiple KVM's per core. The realized utilization obtained from calculations using `/proc/stat` information from the KVM and the Host are principally in agreement for 1KVM/core, however the processes running in the virtualized environment with multiple KVMs per core (VM view plotted as an average) 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. Our work load generator spends 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 another generator [4] as well with similar results.

In our prototype work (Section VI) we use the host resource view of CPU and memory utilization (metrics our fine-grained resource monitoring and characterization system already collects) to inform placement decisions.

B. *Scheduling and Resource Management Issues*

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

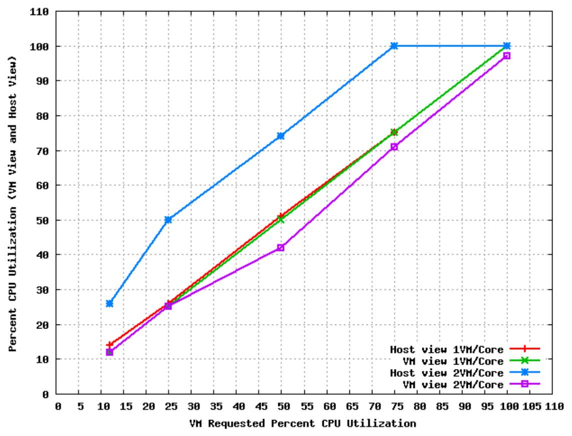


Figure 1. Percent CPU utilization requested in a KVM vs. measured as viewed on the host machine and within the VM. Requested utilization may not be realized due to contention. Host and VM views may be inconsistent.

Resources would initially be allocated according to the users' specifications with the application run in a VM as opposed to directly on the hardware. Utilization measurements could be provided to the user to inform future allocation and also be used to trigger migration during run-time of processes to more appropriately utilize resources especially in the case of contention.

Where contention is observed, processes could be dynamically spread out over more physical resources. Alternatively, if resources were being lightly used, processes could be consolidated onto fewer physical resources. 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 application's 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, it should be done as infrequently as possible. Communication of pertinent information from the application to the system could enhance this process but is not addressed in this work.

The dynamic nature of migration complicates 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 job-to-resource mapping is required. Traditional schedulers do not have the facility for adjusting job allocations at run-time. We have implemented these features in our system (Section V).

V. System Design

In this section we discuss the component parts of our IaaS enabled HPC system shown in Figure 2.

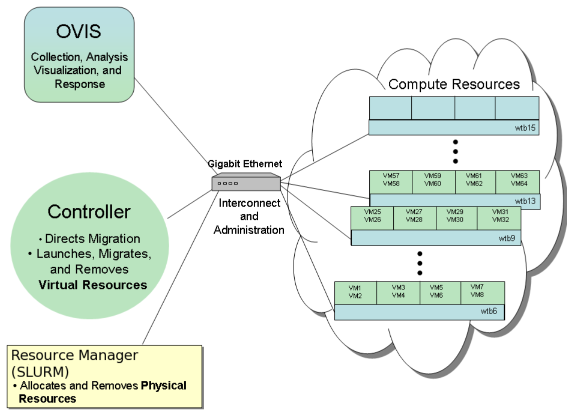

OVIS
Collection, Analysis Visualization, and Response
Controller
• Directs Migration
• Launches, Migrates, and Removes Virtual Resources
Resource Manager (SLURM)
• Allocates and Removes Physical Resources
Gigabit Ethernet
Interconnect and Administration
Compute Resources
wtb15
VM57 VM58 VM59 VM60 VM61 VM62 VM63 VM64
wtb13
VM25 VM26 VM27 VM28 VM29 VM30 VM31 VM32
wtb9
VM1 VM2 VM3 VM4 VM5 VM6 VM7 VM8
wtb6

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

A. *Compute Resources*

As important as the support systems is the 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 processors with 32GB of memory. In the examples shown we use our 1 gigabit Ethernet interconnect.

B. *Fine-Grained Scalable Monitoring and Analysis*

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 utilization.

In order to accomplish this we use OVIS [5], our monitoring and analysis system which has been principally developed to scalably collect and analyze just such data for the purpose of failure prediction. OVIS utilizes a distributed database for data storage and a lightweight daemon running on, or on behalf of, each device for which data is to be collected (compute node and VM in this case but can include any devices of interest) which directly inserts information at regular time intervals into the database. Parallel analysis engines are used to compute models against which individual or ensembles of measurements are compared for detection of either anomalous behavior or signatures indicative of problems. In this study, OVIS utilized data already being collected for the purpose of failure prediction (memory and CPU utilization) in order to compute resource utilization information and inform our Controller subsystem of both non-failure related resource contention and impending failure.

C. *Virtual and Physical Resource Management*

We leverage the SLURM (Simple Linux Utility for Resource Management) [6] resource manager which is commonly used in HPC systems. SLURM provides facilities for maintaining separate resource partitions; allocating resources to jobs and launching batch jobs; running predefined prolog and epilog scripts for setting up and cleaning resources; and storing pertinent information about allocations and the jobs running on them. 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 our managed resource pool which would not be taken into account in the resource utilization calculations.

As of this writing, however, SLURM (v2.0) does not provide the ability for determining and tracking the dynamic virtual to physical resource mapping we require. In our prototype system, then, we maintain separate virtual and physical partitions using SLURM, with virtual to physical mappings maintained by our Controller. Since SLURM does not allow revision of allocations after job launch the Controller also tracks associations of new allocations with existing allocations. Relatedly, though release of partial allocations (e.g., those vacated post-migration) is not supported by SLURM, such retained resources can be used by the Controller for future allocations as appropriate.

D. *Orchestration by the Controller*

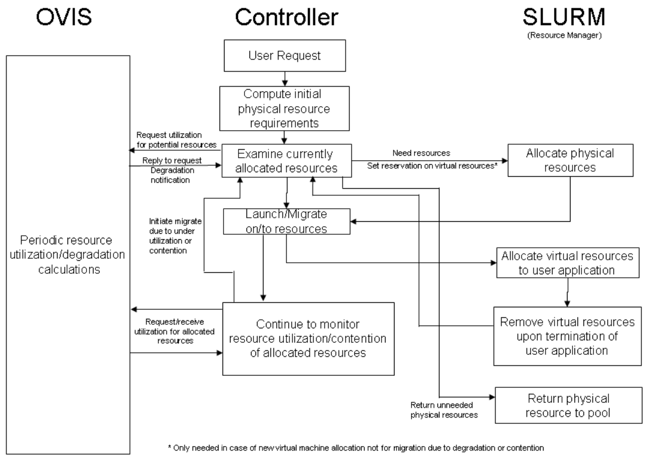

OVIS
Controller
SLURM
(Resource Manager)
User Request
Compute initial physical resource requirements
Request utilization for potential resources
Reply to request Degradation notification
Examine currently allocated resources
Need resources
Set reservation on virtual resources*
Allocate physical resources
Initiate migrate due to under utilization or contention
Launch/Migrate on/to resources
Periodic resource utilization/degradation calculations
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. Interactions of the Controller.

At the heart of our system is our Controller whose interactions with the monitoring and analysis system and the resource management system are diagrammed in Figure 3. In this diagram it is understood 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 for predefined conditions (e.g. resource contention, impending failure).

An allocation cycle begins with a user request which is sent to the Controller rather than SLURM but using generally the same syntax. Additional optional arguments allow the user to specify how much memory he/she expects each process to consume, a maximum packing density (cores/CPU), and expected process to process bandwidth. These arguments can be used by the Controller to make the physical allocation request based on its evaluation of the resources required to satisfy the request. For example, if, in our system, the user requests 16 processors and has specified an upper bound on memory of 1.8GB per process, the Controller would request, based on how the physical resources are interconnected within the actual compute nodes, an allocation from SLURM that would allow placement of 4 processes per CPU (one per core) thus allowing it to satisfy the user request while utilizing at most 8 GB per CPU including host OS overhead.

Resource allocations from SLURM are 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. This enables the higher-granularity allocation that we seek while still enabling control over shared node-level resources. It also enables the Controller to potentially fulfill full or partial resource requests from the pool of resources under its control rather than requiring a full new allocation from SLURM. For instance, in the previous example, if the Controller had 4 free CPUs on separate nodes in its resource pool it would allocate these to satisfy the request unless there were further constraints on this or previous allocations, such as inter process bandwidth, that could not be met under such an allocation. Under such circumstances a new node would be requested from SLURM.

As part of the allocation step the Controller also contacts OVIS with a request for actual resource utilization information on the potential target resources in its pool. This informs 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, thus in our prototype extreme generality of all requests and (re-)allocation scenarios is not supported, but rather, a few common rulesets are applied.

After determining which physical resources will be allocated to an application, the virtual environment is set up. 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. That 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 significant 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 live 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 (Section VI-A). For example, upon launch using our OS 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.

Once the VMs have been booted and are recognized by SLURM the Controller submits the original job request to SLURM for those virtual resources on behalf of the original user and maintains the virtual-physical mapping.

Periodic data collection is used during run-time 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 to maximize resource utilization and system throughput. Currently this is in the experimental phase and is performed for a subset of the potential resources as described in Section VI-A. The detection by OVIS of failure indicators is communicated to the Controller without request and is acted upon as described in Section VI-B.

If, upon resource analysis, dynamic re-configuration of resources is required, the Controller is responsible for arranging resources with the Resource Manager, launching new containers, and performing the migration. For the case of MPI-based applications, orchestration of the migration is performed in concert with the `MPI_Barrier` wrapper (Section III-B). New virtual-to-physical mappings are maintained by the Controller both for its resource tracking and to enable continuous application-centric monitoring of the dynamic environment.

Once a job completes or the time expires 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.

VI. Examples

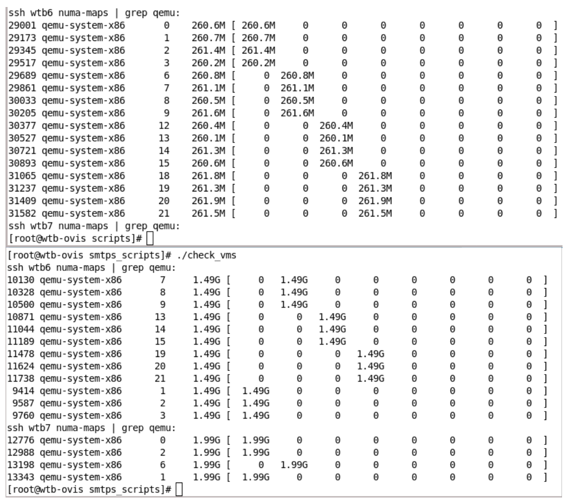
We demonstrate our system in two scenarios. The first is a case in which an application's memory consumption grows over time. Currently an application must wait in the queue until resources are available to satisfy the total required allocation. We demonstrate how our system can utilize some aspects of KVM virtualization and migration to give such an application earlier access to compute resources than would currently be possible. The second uses OVIS's failure prediction based on a known failure mechanism to migrate processes from an unhealthy to a healthy resource.

A. Speculative Oversubscription

In this example we describe the use of a characteristic of the KVMs previously mentioned, i.e., that upon initial launch they can present a smaller footprint to the host node than their maximum allocated size. We use this feature to speculatively launch a job that will oversubscribe the physical resources if the processes running in the KVMs utilize all allocated memory. This provides the application with resources on which to run until either additional resources come available or it requires additional resources to continue. We demonstrate this by launching a sixteen process job with each process requiring a maximum of 2GB but with the known characteristic of needing that over time and not requiring it all up front. Numa-maps output showing the per-process memory in use is shown in Figure 4.

KVM's are initially placed on 4 of the 6 available cores for each CPU on a single node. OVIS continuously monitors Active memory usage. The application continues to consume memory as it progresses until OVIS detects that a dangerously high Active memory level for a node is reached (Figure 5(t)) and notifies the Controller. The Controller then obtains free resources satisfying the application's requirements, launches containers there, and informs the MPI-based application of the need to migrate when a barrier is reached. When the application informs the Controller via the barrier wrapper that it is ready, 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 KVMs and nodes during this process are shown in Figure 6. In the top plot the memory size of the KVMs obtained by OVIS is plotted for 2 representative KVMs: one not migrated and one before and after its migration. During the migration, size is reported on both the initial and final resources, with the final value reported that of the max container size, regardless of the amount in use prior to migration. The bottom plot shows the Active memory utilization for both the node being



```
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 KVM size and location at allocation (t) and after migration (b). Size at allocation is less than the max allowable.

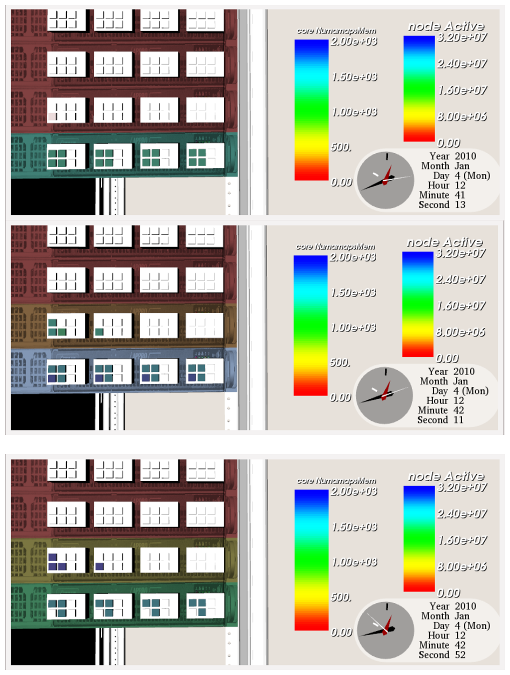


Figure 5. Speculative oversubscription of resources is adjusted by migration when contention is detected as the application progresses. OVIS screenshot of nodes and cores showing memory utilization on the nodes and of the KVMs. Triggering (t), during (m), and after (b) migration.

migrated from and the node being migrated to. (Triggering of the migration was set to occur when the Active memory on the node exceeded 75% of its total). The migration time can be seen to be about 50 seconds 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 GigE.

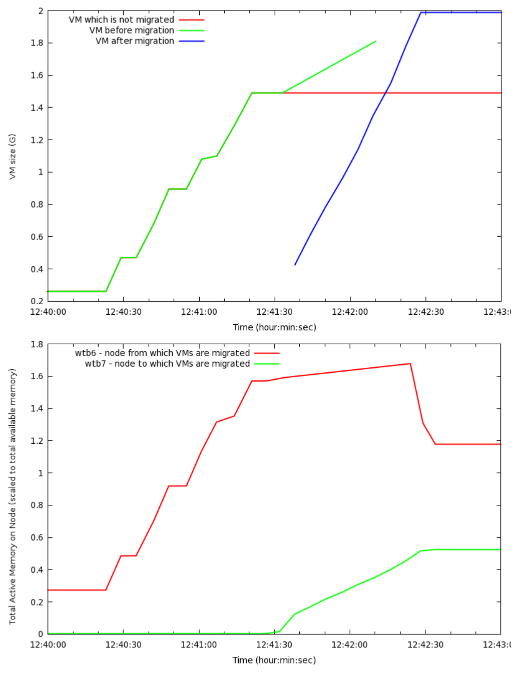


Figure 6. Memory utilization during a live migration triggered by excessive memory consumption on the node. Utilization of KVM (t) node (b).

B. 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 the Linux "OOM killer" process due to memory utilization being too high. In the case that the user application consumes 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 from a previous job [8] and the rest of the process group on other resources are well behaved. Killing of one MPI process on one such ill-behaved resource kills the whole job.

In this example, 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. Upon detection of this pre-failure 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. Prior to migration (l) the 2nd node from the bottom has much higher Active memory (blue is higher, red lower) than those hosting the peer processes (green). After migration (r) the amount of active memory is a little higher in the new host (2nd from top) as migration causes the total memory allocated to a KVM to be used on the new host node (as in VI-A) while the other nodes have not used memory up to their limit.

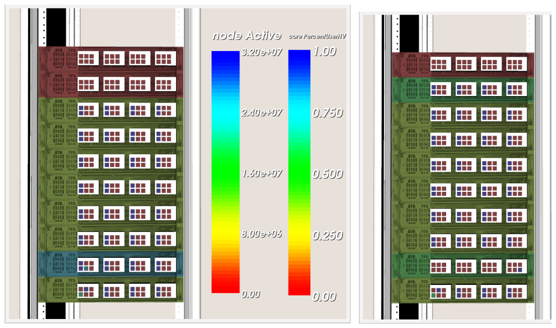


Figure 7. Detection of impending failure condition triggers migration of the endangered processes (l). Migration is complete (r). Host memory utilization and core CPU utilization shown in OVIS display.

VII. 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 [1] and Eucalyptus's open source version of EC2 [2] 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 HPC users' needs may be different for each request and a 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 commercial systems 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.

There has been work done in the area of process migration both with [12] and without [16] the assistance of virtual machines. 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 (sub-second 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, the overhead being reported (see [11], [12], [14], [15] and references therein) 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 would be 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. [12] 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. [13] showed that proper relative placement of application processes on shared hardware can decrease wall time to completion for a set of applications on a given hardware platform compared to running them disjointly due to the difference in required resources of the applications. With proper resource requirement analysis and placement, contention for resources is minimized, as is wall time.

We rely on this disparate, but complimentary, background work in our prototype system for enabling intelligent, dynamic resource utilization. Further, our own work in the area of failure prediction in large scale HPC platforms [5] has provided us with the monitoring and analysis component for our system. We had previously proposed such a system for enabling HPC in Cloud Computing Environments [9] and now demonstrate a prototype with this work.

VIII. Summary

We have described why 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 noted that typical HPC production system management models do not provide the continuous fine-grained run-time resource characterizations, nor the ability to dynamically respond to them, that efficient management would require. We have designed and implemented a prototype system which we believe is a good basis for such functionality and have applied our prototype system to some real world scenarios (resource degradation and load balancing) using an MPI application to demonstrate its applicability to the HPC domain.

References

[1] "EC2," http://aws.amazon.com/ec2/.

[2] "Eucalyptus," http://www.eucalyptus.com.

[3] "KVM," http://www.linux-kvm.org.

[4] "lookbusy," http://www.devin.com/lookbusy/.

[5] "OVIS," http://ovis.ca.sandia.gov.

[6] "SLURM," https://computing.llnl.gov/linux/slurm.

[7] "Xen," http://www.xen.org/.

[8] Brandt, Gentile, Mayo, Pébay, Roe, Thompson, and 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 HPC)*, 2009.

[9] ——, "Resource monitoring and management with OVIS to enable HPC in cloud computing," in *Proc. 23rd IEEE Int'l Parallel & Dist. Processing Symp. (5th Workshop on System Management Techniques, Processes, and Services)*, 2009.

[10] Cao, Li, and Guo, "Process migration for MPI applications based on coordinated checkpoint," in *Proc. 11th IEEE Int'l Conf. on Parallel and Distributed Systems*, 2005.

[11] Fenn, Murphy, and Goasguen, "A study of a KVM-based cluster for grid computing," in *Proc. 47th Annual Southeast Regional Conf. (ACM-SE 47)*, 2009.

[12] Nagarajan, Mueller, Engelmann, and Scott, "Proactive fault tolerance for HPC with Xen virtualization," in *Proc. ACM Int'l Conf. on Supercomputing*, 2007.

[13] Shainer, Liu, Layton, and Mora, "Scheduling strategies for HPC as a service (HPCaaS)," in *Proc. IEEE Int'l Conf. on Cluster Computing and Workshops*, 2009.

[14] Tikotekar, Vallée, Naughton, Ong, Engelmann, Scott, and Filippi, "Effects of virtualization on a scientific application running a hyperspectral radiative transfer code on virtual machines," in *Proc. 2nd Workshop on System-level Virtualization for High Performance Computing*, 2008.

[15] Wang, Mueller, Engelmann, and Scott, "Proactive process-level live migration in HPC environments," in *Proc. 2008 ACM/IEEE Conf. on Supercomputing*, 2008.

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