

# Increasing Reliability through Dynamic Virtual Clustering

Wesley Emenecker, Dan Stanzione  
High Performance Computing Initiative  
Ira A. Fulton School of Engineering  
Arizona State University  
Wesley.Emenecker@asu.edu, dstanzi@asu.edu

## Abstract

*In a scientific community that increasingly relies upon High Performance Computing (HPC) for large scale simulations and analysis, the reliability of hardware and applications devoted to HPC is extremely important. While hardware reliability is not likely to dramatically increase in the coming years, software must be able to provide the reliability required by demanding applications. One way to increase the reliability of HPC systems is to use checkpointing to save the state of an application. If the application fails for some reason (hardware or software errors), the application can be restarted from the most recent checkpoint. This paper presents Dynamic Virtual Clustering as a platform to enable completely transparent parallel checkpointing.*

## 1 Dynamic Virtual Clustering

As High Performance Computing (HPC) systems grow larger and more complex, reliability is an ever more important issue. At ASU, the Dynamic Virtual Clustering (DVC) system has been under development to abstract physical clusters through the use of virtual machines. The primary motivation for the creation of DVC was to increase the throughput and productivity of multi-cluster environments by providing a homogeneous software stack for jobs running across clusters. However, DVC has also become an important tool for increasing reliability and availability in several respects. First, it provides a transparent layer for checkpointing and migration of arbitrary jobs across the cluster, which promotes both failure recovery, and avoidance of job failure when hardware faults can be predicted. Second, by abstracting physical nodes, DVC allows resource management software to continue to schedule jobs in the presence of node faults by using virtualized remote nodes. In this talk, the design of the DVC system will be presented, with an emphasis on performance results in providing a transparent checkpoint/migrate layer to promote fault tolerance.

Virtualization software is rapidly maturing. In particular, Xen uses para-virtualization to provide an abstraction layer for a complete OS. The Xen model is now being supported by the major processor manufacturers; in the next generation of chips, AMD's Pacifica and Intel's VT efforts will provide support to run Xen virtualization at near native speed, reducing the overhead of this approach to near zero. Virtualization will play an important role in increasing the availability of cluster systems. While hardware reliability is not likely to significantly improve in the near future, virtualization can be used to hide faults in the underlying real hardware.

The DVC system uses Xen virtual machines to abstract physical cluster nodes. DVC integrates these virtual nodes with resource management and scheduling software to allow the use of complete "virtual clusters" that can be dynamically allocated and de-allocated on a per job basis. With this system, software environments can be created that are tailored to the specific needs of a job, user, or group. Virtual clusters may map directly to physical clusters, to a subset of a single cluster, or even span multiple clusters. More importantly, this mapping can be adjusted to available resources (a 32 node virtual cluster may run on a particular 32 physical nodes in one instance, and on a completely separate set of physical nodes at the next instantiation). Aside from the obvious simplification of administrative tasks, previous work has demonstrated that a system that can transparently span parallel jobs between multiple clusters will outperform those same clusters acting independently.

An obvious advantage to DVC is that individual hardware faults can be masked at the time of job launch, as long as enough physical nodes are available on the set of physical clusters to satisfy the demands of the virtual cluster. However, modern resource management hardware already provides this advantage, as it will typically schedule around failed nodes, and dynamically provide a host list for jobs to run on (though users often request the full cluster size for a single job, whereas a virtual cluster can be of a smaller size than the physical one, masking this phenomena). More im-

portantly from a reliability standpoint is the virtual clusters' ability to snapshot its state and migrate to new hardware.

Checkpointing and migration are well-known techniques for both dealing with faults and improving throughput in HPC systems. Most successful checkpointing systems rely on code built into the application to provide checkpoint and restart capabilities. Condor provides checkpointing for any application which can be compiled against the Condor run time libraries, but the checkpoint is limited to sequential jobs. Transparent checkpointing of parallel jobs remains even more elusive. The BLCR system allows MPI applications to be checkpointed with a few restrictions, but only if the application is built with BLCR and certain specific MPI implementations. Truly transparent checkpoint/migrate/restart for parallel jobs does not yet exist.

The Xen virtual machine provides the ability to pause, save, and restart the virtual OS, including the state of all processes running within that OS. DVC builds on this capability to allow a snapshot to be taken of the entire virtual cluster. In the case of a virtual cluster running only single node computations, this is a largely trivial task, requiring only a reliable storage system to save the state of each OS, and an image management capability to track the correct staging and restart of images. The more interesting case is when MPI-based parallel jobs are running, which use the cluster's interconnection network to communicate. With DVC, the capability to transparently checkpoint parallel jobs has been demonstrated, through the use of coordinated checkpoint commands to all running virtual nodes, combined with a reliable network protocol which will retransmit any packets lost "on the wire". In this paper, the design of DVC will be presented, along with measurements of the overhead required for virtual clusters running both sequential and parallel jobs, and a measure of the efficiency of DVC checkpoints vs. application specific checkpoints for common applications.

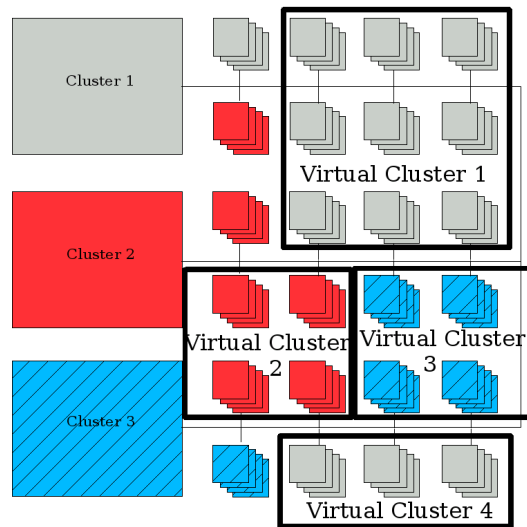
Dynamic Virtual Clustering (DVC) was designed to give three major abilities:

1. The ability to run any job within a cluster that would otherwise require modifications to the cluster's software environment.
2. The ability to run a job submitted to one cluster unmodified on a second cluster, even if the second cluster has a different software stack.
3. The ability to transparently run a single job that spans multiple clusters.

Our approach to DVC uses virtual machines to accomplish these goals, and the result is illustrated in figure 1.

In addition to the three main goals DVC, many virtual machines also have the additional capability to save and restore a guest environment. Extending this capability to

Figure 1. Dynamic Virtual Clusters



the cluster realm enables the transparent checkpointing of parallel applications. With the capabilities gained by using temporary virtual clusters on top of real clusters, we can increase the reliability of clusters and their applications with the transparency given by DVC. If a single physical node dies, we can restart a checkpoint of the entire virtual cluster on a different set of physical nodes. Instead of being constrained to a real set of nodes that may be broken, virtual nodes cannot be broken (at least not in the same manner).

This argument for reliability hinges on the ability of virtual machines to checkpoint guest environments and hence guest applications. If a set of virtual machines can transparently and easily checkpoint a parallel application, we can increase the reliability of the system as a whole.

## 2 Checkpointing

Checkpointing is one of the most useful ways to recover from an application failure. By saving the state of an application at periodic intervals, an application can be resumed from the last checkpoint instead of completely starting over. There are three main ways of checkpointing:

1. Application level
2. User level
3. Kernel level

Application level checkpointing is the fastest method since it requires only necessary data to be saved for a restart. Unfortunately, it places a heavy burden on the programmer since the programmer must explicitly write support for checkpointing. Adding this support to an application is not

required, and because of the extra time and effort required to add the capability, checkpointing is often not added.

User level checkpointing is able to save an application’s state without requiring the application to explicitly have support for this ability. This solution generally requires the application to be linked with a checkpointing library like libckpt [7]. Checkpointing in this manner is slower and more difficult to implement than application level checkpointing since many other factors must be taken into account. Open files, sockets, memory state, application code, etc. must all be taken into account when saving the state of an application [10, 2]. This is much more information to save than the application must deal with, because the library doesn’t know which data is necessary for a restart. Given this lack of knowledge, this implementation must save the state of the entire application.

The final solution to checkpointing is kernel level checkpointing. This is the most transparent solution to checkpointing since it does not require any application involvement (linked libraries are considered involvement) in order to checkpoint the application, and it has the same overhead as user level checkpointing since it saves the state of the entire application. One example of this kind of checkpointing is CRAK [11].

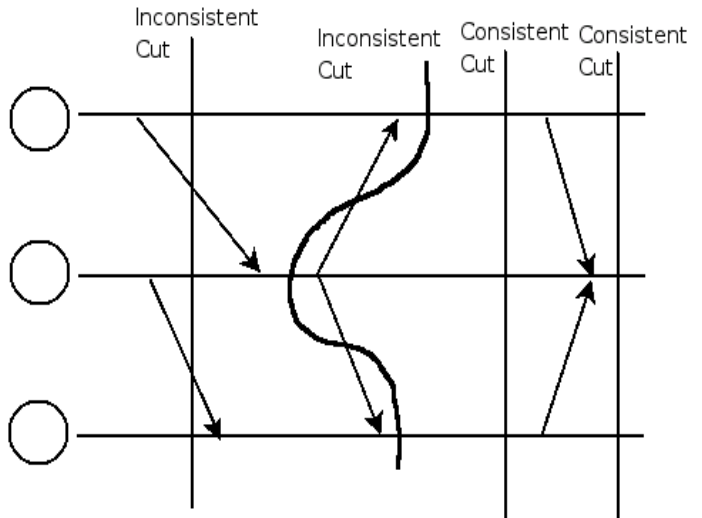
The DVC system introduced above uses virtual machines to run parallel applications. One major advantage to virtual machines is that many support the ability to save and restore guest environments. This approach has even more overhead than user level checkpointing since the state of the entire guest environment is saved (all memory available to the guest including the guest kernel), but in many ways is simpler to deal with since all guest kernel state is saved.

### 2.1 Parallel Checkpointing

The ability to checkpoint parallel applications is a powerful way to mask failures. In the event of a software or hardware failure, the application can be rolled back to a recent saved state in order to prevent the loss of all computation. For applications that use hundreds of processors for weeks, this ability is critical since the failure of a single node is often capable of crashing the entire application. Traditionally, application level checkpointing is the most successful and fastest. While this approach is the fastest since only needed data is saved, it also puts the largest burden on the programmer. Relying on the application to checkpoint itself is a suboptimal solution since not all applications provide this capability.

The second approach to parallel checkpointing saves the state of the application transparently. While implementing transparent checkpointing is much more difficult, if the application can be saved and restarted without being aware of the checkpoint, then all applications can be checkpointed.

Figure 2. A sample of network cuts



The largest difficulty arises from the fact that most parallel applications are co-dependent. In other words, each process in a parallel application depends on another process. If one process fails or somehow loses its synchrony with the rest of the distributed processes, the application will fail. This gives rise to the necessity of a consistent cut of the network state of the parallel application. Figure 2 shows some possibilities for consistent and inconsistent cuts of the network state. If an application is checkpointed with an inconsistent state, when restarted it will be unable to continue because information has been lost. If an application is checkpointed with a consistent state, the application will not lose any information, and thus will be able to continue from a checkpoint.

There have been several largely successful efforts to accomplish parallel checkpointing by consistently cutting the network, most notably BLCR[3, 8] and CoCheck[8]. These approaches rely on user level libraries to be linked with the parallel application as well as kernel support so that the library can intercept communication and provide the necessary consistent “cut” of the network [2, 9].

The disadvantage to this approach is that applications must be re-linked in order to provide this support.

In a Dynamic Virtual Cluster, we are able to checkpoint parallel applications completely transparently by coordinating the checkpoint of multiple virtual machines.

### 3 Lazy Synchronous Checkpointing

Checkpointing the entire state of a guest environment is able to avoid many of the problems associated with user level and kernel level checkpointing. By saving the state of

the guest kernel, open files, memory state, and even sockets can be saved along with the application so that upon a restart, all data used by the guest application will still exist. The largest problem with checkpointing virtual machines in parallel is synchronizing the saves of the guests. Similar to the difficulties encountered by BLCR and CoCheck, saving virtual machines in parallel must keep the network state consistent. However, unlike BLCR and CoCheck, the synchrony required by parallel virtual machine checkpoint is much less stringent. The reason for this is that if the parallel application uses a reliable protocol for communication, TCP for example, then we simply have to save the guest machines before any TCP timeouts can occur. There are several scenarios that must be examined to determine whether this solution will work. The major assumption in both cases is that all virtual machines involved are checkpointed at the same time.

Scenario 1: A message from a sending application is lost in the network before it can be delivered to the receiving process that was checkpointed.

Solution: Since the receiving process was checkpointed before it could be delivered, the receiving kernel never ACKed the message (assuming TCP). Because the receiving kernel never ACKed the message, the sending kernel should will assume the message was lost and resend the message. However, since both the sending and receiving guests were checkpointed, no messages can be sent between them until they are restarted. After a restart, the sender will send any unACKed messages, but now the receiver will be able to receive and ACK the messages. Thus the state of the network is consistent.

Scenario 2: The message is delivered to the receiver who ACKs the message. Unfortunately, the ACK is lost before it can be delivered to the sender before both are checkpointed.

Solution: Upon restart, the sender will resend the message (it hasn't received an ACK for it). The receiver has already gotten the message (and presumably delivered it to the application), and so will resend the ACK for the message. The sender will receive the ACK, and continue computation, thus no messages are lost.

These two scenarios solve both inconsistent cuts that are shown in figure 2.

The greatest difficulty to this approach is the coordination of virtual machine checkpoints to prevent any network timeouts. Reliable network protocols (again TCP) will not retry sending forever. There is a finite amount of time to save all virtual machines participating in the parallel computation before a network timeout occurs and causes the application to crash.

### 3.1 Lazy Synchronous Checkpointing Implementation

Two implementations of Lazy Synchronous Checkpointing (LSC) are examined here, a naive approach, and the current working prototype.

**Naive approach:** The naive approach to LSC requires a single program to open a terminal connection to each node running a virtual machine that will be checkpointed. Once all terminal connections are open, the program executes a "vm save" command in parallel on all nodes. This should cause each virtual machine to save itself within the period of time required to prevent any network timeouts.

**Evaluation:** This naive approach was simple to implement, but unreliable at best. The attempts at synchronizing the execution of a save command did not scale beyond 8 nodes, with 10 nodes failing 50% of the time and 12 nodes failing 90% of the time. Testing beyond 12 nodes was not performed.

**Current prototype:** The second implementation of LSC relies on the synchronization of host clocks with NTP (Network Time Protocol). Although it is well known that network time synchronization cannot completely synchronize host clocks [6], network time protocols can synchronize time to within a few milliseconds. While this jitter may be significant for some applications, for LSC it is sufficient to allow a completely transparent implementation of parallel checkpointing. To parallel checkpoint applications with this approach to LSC, all nodes involved in the checkpoint must be synchronized with a network time protocol (NTP in this case). Once all nodes are synchronized, a program on each node is set to execute a "vm save" at a certain time in the future. This program must have access to a microsecond precision timer, as well as a sleep timer capable of the same. Since all applications are set to save the virtual machine at the same time, a coherent network state with no timeouts is obtained.

There are drawbacks to this approach that have not been addressed yet, but this prototype has been shown to work with the standard HPCC benchmark. This implementation does not take into account a heavily loaded server which may not be able to service a checkpoint request immediately, and it does not check neighboring processes to make certain that the sleeping checkpoint process is still executing. These are a few of the issues that must be examined in depth in order to make certain that this approach will always work.

## 3.2 Results

Verifying the correctness of the current prototype required much testing. For testing both the PTRANS and HPL benchmarks from the HPCC suite were tested. PTRANS is a communication heavy test that performs a parallel matrix transpose. This was the most important test for verifying that our conclusions about consistent network states were correct. The HPL benchmarks were tested to check the amount of time required by a parallel save and restore. For each scenario, multiple problem sizes were tested with varying times between checkpoints. In more than 2000 tests involving 26 virtual machines on 26 different nodes, no failures to either save or restore all virtual machines occurred. Both PTRANS and HPL reported a decreased speed in execution time due to the checkpoint. Since time was not virtualized in any virtual machine, the jump in wall time due to the checkpoint caused HPL to report a greatly increased execution time.

One interesting point to note is that a software watchdog timer was enabled in all virtual machines. Each save and restoration of a virtual machine caused a watchdog timeout to be reported. Although this did not affect the execution of the environment, it did cause a large number of kernel messages to accumulate.

## 4 Conclusions and Future Work

The results from the current prototype of Lazy Synchronous Checkpointing are encouraging since no test failed. Much work needs to be done to make this solution robust, including more error checking and testing under heavily loaded conditions, as well as integration with resource managers and schedulers like Torque and Moab. This approach scaled from 1 to 26 nodes, and scaling to larger sets of nodes will require the same coordination laid out. The largest issue for scalability is that with more nodes in a checkpoint set, the larger the likelihood of a single VM checkpoint failing. With greater error checking, and a coordinated health check of checkpoint processes, scaling to hundreds or even thousands of nodes should be possible. Extending LSC to enable parallel migration is the next step in the process to increasing cluster reliability with Dynamic Virtual Clusters.

Parallel checkpointing with advanced networks like Infiniband is currently under work [4], but does so with BLCR. Extending DVC's parallel checkpointing to work with Infiniband will require much work developing drivers capable of executing in virtual machines. Finally, testing LSC and migration with Intel VT-R[5] and AMD Pacifica[1] are two critical steps to showing the performance and viability of Dynamic Virtual Clustering in HPC.

## References

- [1] AMD. AMD64 Virtualization: Codename Pacifica Technology, 2005. [http://www.amd.com/us-en/assets/content\\_type/white\\_papers\\_and\\_tech\\_docs/33047.pdf](http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/33047.pdf).
- [2] J. Duell, P. Hargrove, and E Roman. Requirements for Linux Checkpoint/Restart, 2003. <http://ftg.lbl.gov/CheckpointRestart/LBNL-49659.pdf>.
- [3] J. Duell, P. Hargrove, and E Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart, 2003. <http://ftg.lbl.gov/CheckpointRestart/blcr.pdf>.
- [4] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K. Panda. Application-transparent checkpoint/restart for mpi programs over infiniband. *icpp*, 0:471–478, 2006.
- [5] Intel. Intel Virtualization Technology Specification for the IA-32 Intel Architecture, 2005. <ftp://download.intel.com/technology/computing/vptech/C97063-002.pdf>.
- [6] D.L. Mills. Improved algorithms for synchronizing computer network clocks. In *IEEE/ACM Transactions on Networking*, volume 3, pages 245–254, June 1995.
- [7] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of USENIX Winter1995 Technical Conference*, pages 213–224, New Orleans, Louisiana/U.S.A., January 1995.
- [8] E Roman. A Survey of Checkpoint/Restart Implementations, 2003. <http://ftg.lbl.gov/CheckpointRestart/checkpointSurvey-020724b.pdf>.
- [9] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In *In LACSI Symposium*, October 2003.
- [10] Victor C. Zandy and Barton P. Miller. Reliable network connections. In *MobiCom '02: Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 95–106, New York, NY, USA, 2002. ACM Press.
- [11] Hua Zhong and Jason Nieh. CRAK: Linux Checkpoint/Restart As a Kernel Module. <http://www.ncl.cs.columbia.edu/research/migrate/crak.html>.