2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

Distributed Virtual Diskless Checkpointing: A Highly Fault Tolerant Scheme for Virtualized Clusters

Ben Eckart[1*], Xubin He[2†], Chentao Wu[2†], Ferrol Aderholdt[3‡], Fang Han[3‡], Stephen Scott[3,4‡]
[1]Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, USA
[2]Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, VA, USA
[3]Department of Computer Science, Tennessee Tech University, Cookeville, TN, USA
[4]Oak Ridge National Lab, Oak Ridge, TN, USA
*eckart@cmu.edu, †{xhe2, wuc4}@vcu.edu, ‡{wfaderhold2, fhan42, sscott}@tntech.edu

***Abstract*—Today's high-end computing systems are facing a crisis of high failure rates due to increased numbers of components. Recent studies have shown that traditional fault tolerant techniques incur overheads that more than double execution times on these highly parallel machines. Thus, future high-end computing must be able to provide adequate fault tolerance at an acceptable cost or the burdens of fault management will severely affect the viability of such systems. Cluster virtualization offers a potentially unique solution for fault management, but brings significant overhead, especially for I/O. In this paper, we propose a novel diskless checkpointing technique on clusters of virtual machines. Our technique splits Virtual Machines into sets of orthogonal RAID systems and distributes parity evenly across the cluster, similar to a RAID-5 configuration, but using VM images as data elements. Our theoretical analysis shows that our technique significantly reduces the overhead associated with checkpointing by removing the disk I/O bottleneck.**

I. Introduction

As high-end computing moves into the petaflops range, we are seeing the number of processing, network, and storage components increase rapidly. Such an increase in components causes extreme concerns in usability since, in the absence of component redundancy, each component adds another point of failure to the system.

As increased parallelism seems to be the dominant trend in the quest for faster clusters, we will continue to see an increasing amount of components at all levels. Currently, at more than 10 Petaflops, the fastest computer in the world is Japan's K Computer, which has about 700,000 cores (November 2011) [10]. As more clusters approach this level and as others surpass it, we will see a much more concentrated effort in fault tolerance and resilience. If we extrapolate trends, we will see that MTBF (mean time between failures) will be on the order of hours or even minutes. At this point, even the most established methods of fault tolerance break down. For instance, Schroeder and Gibson showed that, if trends are extrapolated, in the near future (by 2015) the MTBF of machines will be smaller than the checkpoint time; that is, a cluster cannot avoid data loss even if it does nothing but checkpoint [25]. Theoretically, an increased amount of processors will result in a linear increase in the amount of calculations that can be performed if the problem is able to be parallelized across the system. However, in practice, we see that without proper measures of resilience, the reverse occurs: at some point, adding more processors only serves to *increase* the total execution time, due to the increased number of failures that are more and more likely to occur.

Reports of large-scale clusters show MTBF values as low as 1.2 hours, for Google's servers [12], and a mean of 5-6 hours for modern HPC systems [5]. With these statistics at hand, it should be readily apparent that research into fault tolerant techniques is of utmost importance. Many techniques have been proposed to deal with the looming probability of failure on these high-end machines. Of these, we concentrate on one particular strategy that we find has much promise: system-level virtualization.

While virtualization is not a panacea to the problems facing modern high-end systems, it does provide us with a specialized set of tools that can solve a subset of these problems. One byproduct of virtualizing the cluster is that the concept of "state" becomes more fluid. Indeed, by enumerating the benefits of virtualization related to fault tolerance, we can see that they almost always relate to state manipulation.

1) Saving state: Checkpointing for rollback recovery
2) Moving state: Live migration away from failing nodes
3) Replicating state: VM cloning for redundant execution
4) Debugging state: Fine-grained replay or time traveling for tracing faults
5) Monitoring state: Hypervisor-based diagnostics for fault diagnosis

An immediate drawback to these methods is the overhead cost of the virtualization platform itself. Of these costs, the most salient issue is the high cost of virtualizing I/O, given that virtual I/O requests must be caught and translated into requests on actual physical hardware. Thus, we are primarily motivated to find fault tolerant solutions in the virtual domain that do not exacerbate the I/O problem. One such area within virtualization is the concept of live migration. We plan to explore how live migration can be harnessed by diskless checkpointing to accomplish our goal of high availability without huge hits to performance with respect to I/O. In this paper, we develop a novel diskless checkpointing architecture that supports high availability without burdening the disk subsystem and derive an analytical model to support our methods.

The rest of this paper continues as follows: We begin by reviewing background material in Section II. We then discuss our motivation and goals in Section III, followed by our design of virtual diskless checkpointing in Section IV. We develop a model for virtual diskless checkpointing in Section V. Section VI briefly overviews related work and finally we conclude the paper in Section VII.

978-0-7695-4676-6/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPSW.2012.136

II. Background

Since our work falls in between live migration and checkpointing (specifically diskless), we will briefly discuss the two subjects separately as background material.

A. Live Migration

Live migration is a technique to transplant a virtualized system from one physical machine to another. Usually, the technique is optimized for minimal downtime under the context of machines residing on the same local network. Virtualization provides a solution for such functionality compared to the more traditional process migration technologies. For instance, since virtualization encapsulates hardware resources, it becomes more straightforward to uproot a virtualized OS from its physical connections to a machine than a non-virtualized OS, which may have "residual dependencies" or other physical dependencies [16]. Residual dependencies under traditional process migration are usually handled by some state remaining on the machine to forward requests. With live migration of virtual machines, the old machine can be terminated completely once the transactional state has been fully completed. Thus, the VMM does not need to understand the internals of the OS or the processes running in it.

Live migration can also be very fast: Clark *et al* showed that downtime under live migration can be very low, with the original implementation in Xen experiencing 60 ms of downtime while live migrating a Quake 3 server [7]. Thus, the impact is reasonable, since the total migration time is in minutes and downtime is in milliseconds, and once migration is complete, the source domain is completely free.

An optimized application of live migration opens new roads for automated maintenance and other autonomic features. Virtual machines can be moved away from failing hardware, loads can be optimized for energy or performance, and maintenance downtimes can be carefully controlled. The technologies behind virtualization also lead to flash cloning [29], advanced logging of nondeterministic events [11], and highly available systems [9], including our own proposed architecture.

In general, live migration must preserve both local and global names. Local names include the memory, registers, and disk. Global names include IP addresses. Since most cluster configurations currently run diskless [18] using a shared NAS, our problem of copying the disk state is somewhat simplified. However, in the case of local disks, there have been techniques proposed to deal with the problem of shared disk state, including mirroring [7] and stackable file systems [28]. Other local names, including memory and registers can be passed over the network using the relatively narrow VMM interface and resumed on the other side. Global names can be dealt with by cluster management software, or, in the case of IP, by sending ARP packets or tracking MAC addresses.

B. Checkpointing

The most common and simplest form of fault tolerance in HPC systems is to checkpoint work. Checkpointing amounts to saving the state of computation at discrete intervals so that if any failure occurs, execution may resume from the last previously saved checkpoint. The idea is undeniably simple, yet it has garnered over 30 years of rich research. Seemingly innocuous questions sometimes have deep mathematical answers: How often should one checkpoint? What parts of the system should be checkpointed? What or who should be in charge of the checkpointing?

The question of when one should checkpoint is a function of the length of the computation, the failure rate, and variables concerning the checkpoint overhead and latency. We should wish to minimize the expected execution time in the presence of failure. Given estimates of the failure distribution of the system (typically Poisson), we can then calculate optimal checkpointing intervals that minimize the expected time to completion [14].

The questions of what to checkpoint and where to checkpoint are a matter of efficiency and usability. At the application-level, the programmer is burdened with providing checkpointing in the code itself. Yet, this level may provide the most lightweight and efficient checkpoints, if done correctly, since the programmer knows exactly how to save the state of the program. A simpler approach would be to use a library, such as *libckpt*, but this requires relinking code [22]. Another common approach is at the kernel-level, but residual hardware dependencies and network state make creating and maintaining checkpoints difficult (the same is true of both application and user-level or library-based checkpoints). The last level is at the hypervisor for virtualized systems.

1) Incremental checkpointing: Incremental checkpointing can be thought of as a type of temporal compression. Since we know that the principle of locality dictates that certain regions of memory be "hot" or "cold" during most types of computation, we can infer that successive checkpoints will share many of the same pages. In fact, in many cases, the working set is so comparatively small that saving only the changed state during checkpointing becomes a huge advantage [24]. In the classic scheme, only pages modified since the last checkpoint are written out to disk.

Compressing the increments adds some overhead: write the compressed pages to a disk, copy and compress pages in the buffer, and perform page handling.

Other methods have advanced the Plank's original proposal to include adaptive checkpointing [32]. Since optimal checkpointing intervals are usually calculated with a constant cost for the checkpoint, one can construct an online algorithm to calculate the most beneficial times to checkpoint during incremental checkpointing (where the checkpointing cost is not constant, but depends on dirty pages).

For adaptive or runtime checkpointing, a cost-benefit calculation can be derived as follows: If you skip a checkpoint, your cost is a "long rollback," and if you take a checkpoint, your cost is a "short rollback" to the checkpoint. So, at any time, we have the choice of taking a checkpoint or skipping a checkpoint. We can calculate the expected recovery time for both, subtract them and look at the differential. Intuitively, we know that the longer a process goes on without checkpointing,

the more time it is in danger of losing. Also, the higher the checkpointing costs, the more dirty pages are. At some point in this time interval, it will make more sense to checkpoint than to not checkpoint (our expected recovery, given the overhead of checkpointing, will be less).

2) Diskless checkpointing: Classic examples of fault tolerance are in the use of RAID storage systems [20] [6], which employ parity codes to store redundant information in such a way that some number of hard drive failures can occur without data loss. Since the advent of RAID, many other types of computer systems have made use of the underlying techniques for gains in reliability. RAID uses the principle that it is often cheaper to combine many low cost components and "build in" the cost of failure than it is to design a large monolithic system to meet all performance and reliability needs.

It has been noted that the disk is the main component that contributes to checkpointing overhead and performance degradation [23]. Thus, it may seem intuitively beneficial to try to remove the disk from the checkpointing system. Diskless checkpointing is a technique that attempts to utilize the RAID principle to rely on memory, rather than disks, to store checkpoints. Just as with RAID, where many smaller disks suffer from poor reliability in aggregate, so does memory. Thus, parity is introduced to counteract the innate unreliability of volatile memory such that some amount of memory failure can be tolerated without loss of checkpoints.

The diskless method as presented by Plank doesn't actually improve overhead much, but it vastly (by factor of 34 in [23]) improves latency. Overhead is the amount of time execution is suspended by the checkpointing process. Latency is the amount of time it takes before the checkpoint is usable. An example given by Koren [14] is when a checkpoint is stored in a temporary buffer, and execution in the process is resumed, another process will then dump the buffer onto disk or over the network (and onto a disk). During the dump process, the checkpoint will be unusable. Thus, latency is always at least as much as overhead. Also, during the dump, the other network traffic will be severely degraded, in some cases by up to 87 percent [21]. Thus, lowering latency is a crucial step for high-performance checkpointing.

In the original work, three variants are discussed: normal, fork, and incremental. Normal is the case when one needs three times the memory of the process to hold the process itself, the current and previous checkpoints. Incremental checkpointing tries to compress this space by write-protecting all pages after the first checkpoint is made, catching exceptions, adding the page number to a list, and then storing the old page in the checkpoint buffer and the new page where it is supposed to be. When it is time to make a checkpoint, the old pages can be discarded, and the process is set read-only again. Thus, whenever a checkpoint is needed, we can simply look up the old pages in the buffer, and merge them into the current page data of the process to make the previous checkpoint. This observation means that only the changed pages are needed. Forked checkpointing implements copy-on-write by forking (cloning) the process. Optimized copy-on-write means that each process only contains the changed bytes. We still need the current and previous checkpoint during checkpointing, so if I is consumed, $2I$ is needed during checkpointing. In general, this will require vastly less space than the "naive" implementation.

Others have since taken diskless checkpointing and implemented more advanced codes than simple RAID5-like parity. Wang *et al* recently implemented RDP codes [8], which tolerate up to two simultaneous failures, and found favorable results [30]. Diskless checkpointing has been somewhat slow to catch on for production use, but it is being used successfully at Lawrence Livermore National Laboratory (LLNL) at the time of writing [18].

III. Goals & Motivation

By merging the fault tolerant benefits of virtual checkpointing with the benefits of virtualized distributed management frameworks, we arrive at a vision of future computing where clusters are virtualized, and checkpointing is completely transparent and parallel.

Fault tolerance is critical in today's large distributed systems. Traditional schemes such as checkpointing offer limited solutions that may not scale to tomorrow's machines. Thus, it is of great importance to develop new fault tolerant systems for distributed systems. We are motivated by the promising aspects of leveraging virtualization as an avenue through which we can apply a fault tolerant infrastructure for large virtualized systems.

We wish to apply the techniques of diskless checkpointing to the virtual cluster domain in order to alleviate one of the largest drawbacks of traditional checkpointing: the disk bottleneck. Virtualization allows us to checkpoint beneath the kernel, live migrate VMs across the cluster, and share redundant memory. Our objective is to apply the state-of-the-art diskless checkpointing techniques in the virtual domain and develop new fault tolerant architectures that are both high-performing and resilient.

IV. Design of Virtual Diskless Checkpointing

HPC systems are suffering a reliability crisis, with no definite solution in sight. Virtualization offers promising benefits of fault tolerance and optimization, but with critical drawbacks of inefficient I/O performance. Thus, it is the primary goal of this work to provide fault tolerant techniques on virtualized clusters that attempt to minimize or circumvent the inherent inadequacies of I/O performance.

One such way to attack this problem is to borrow a result from the ongoing work in diskless checkpointing. Diskless checkpointing shares similar concerns: it is a method invented to remove the I/O bottleneck from the process of checkpointing to secondary storage, an expensive operation.

A. Virtual Diskless Checkpointing

In this section, we present a novel virtual diskless checkpointing architecture. We start with a simplified platform, and

then extend our technique to what we call Distributed Virtual Diskless Checkpointing (DVDC).

Virtual checkpointing confers many benefits over traditional checkpointing. Since the process of virtualizing hardware resources abstracts connections into software, it is easier to maintain state when saving and resuming from checkpoints. Furthermore, the state of the system can be saved without libraries, changes in code, or any effort on the part of the programmer. Thus, the system is completely transparent, taking the burden of fault tolerance out of the algorithm or program itself and into the virtualization platform.

As with normal non-virtualized checkpointing, the disk remains the bottleneck: Large VM images sent to a shared network store (NAS or SAN) can tie up resources for a critically long time. Our idea is to apply some of the techniques of diskless checkpointing to alleviate the burden of the disk. We start with the simplest implementation that simply stores a checkpoint of each VM and assigns a checkpointing node to hold all VM parity.

Plank's diskless checkpointing strategy treats processors as if they were disks, and uses RAID-like techniques to maintain N+M state redundancy [23]. Thus, if any processor fails, enough redundancy is kept among the remaining processors so that its state can be fully recovered. Plank's strategy used *libckpt*, a checkpointing library that one could link their application to in order to receive checkpointing capabilities [22]. Virtualization, however, provides an interface that, since it rests below the kernel, can checkpoint without library intervention. Applications, user-level libraries, and even the kernel itself need not be aware that it is being checkpointed.

The question naturally arises then if it is possible to marry the technique of diskless checkpointing with system-level virtualization. A naive implementation might simply employ the technique in the most straightforward way possible, creating an in-memory checkpoint per VM and then assigning m dummy VM's to hold the parity of the checkpoint. This process is what is done in diskless checkpointing. Immediately, we see that virtualization both complicates and provides opportunities for improvement on this method. First, VM's are not necessarily one-to-one with a physical machine. In fact, in most cases, we would like to run as many VM's per physical node as possible to best utilize its resources. Thus, we cannot simply apply the diskless checkpointing technique as given, since VM's residing on the same physical node would be subject to the same hardware faults, and thus be perfectly correlated in these types of errors. In N+1 redundancy, having more than two virtual machines per physical node would mean that data loss would occur any time the physical node experienced a failure. A second observation is that virtual machines can migrate from node to node, using live migration [7], mixing up the distribution of VM's per physical node. The final observation is that we should not need any VM's that simply store parity. In traditional diskless checkpointing, the redundant processors simply provide a safe place to store the parity. That is, we are protecting against hardware faults that are assumed to be mostly uncorrelated at the machine level. Therefore, we cannot store the parity on a machine that is doing actual work. However, in the virtual domain, it seems wasteful to allocate a VM for sole purpose of storing parity.

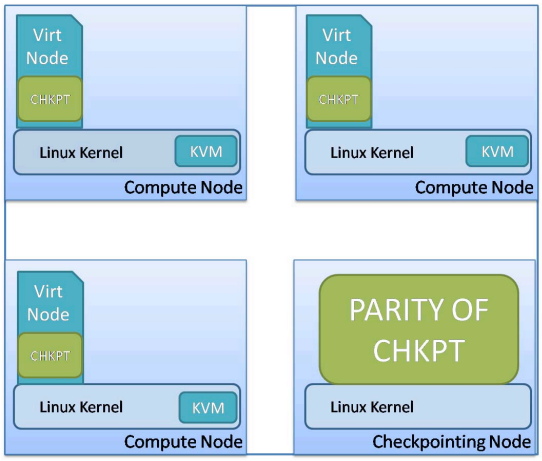

Fig. 1. A first-shot implementation of diskless checkpointing on a simple virtualized cluster.

A first-shot solution to these problems is the following architecture presented in Figure 1: We restrict ourselves to one VM per physical node, where we have N+1 physical nodes (in the case where we protect against single failures). We coordinate a consistent distributed checkpoint (using the techniques of Section II) at each VM. Each VM then sends its checkpoint data, fan-in, to another node, until the redundant physical machine has calculated parity on all the machines. At this point, the parity machine notifies the other machines, and execution resumes. If any physical machine goes down, its VM will go with it, and the checkpoint of the VM can be constructed from the remaining checkpoints and parity on the parity machine. We should note several things here: first, we have restricted ourselves to one VM per node; second, we have applied Plank's method, only instead of linking applications to the libckpt library, we handle checkpointing at the hypervisor level. The former observation is an unreasonable proposal in most cases, since one of the main benefits of running VM's is that entire systems can be multiplexed on hardware. The latter observation is only a modest benefit beyond Plank's method, without really utilizing any of the special characteristics and opportunities that virtualization offers, especially in a cluster or grid environment.

B. Distributed Virtual Diskless Checkpointing

Removing the restriction of one VM per node, it turns out, actually opens up new possibilities for optimization and improvement over traditional diskless checkpointing. Suppose we have three nodes: each with 1 VM. In the N+1 case, two nodes will contain normal VMs and the last will be a checkpointing or parity VM. If we want to remove the one VM per node restriction, we can simply add more VMs to the nodes, as long as we don't include them in the parity calculation. If we want the same fault tolerance as the first

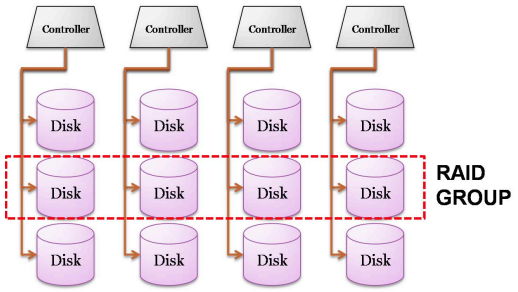

Fig. 2. Orthogonal RAID that can survive controller failure.

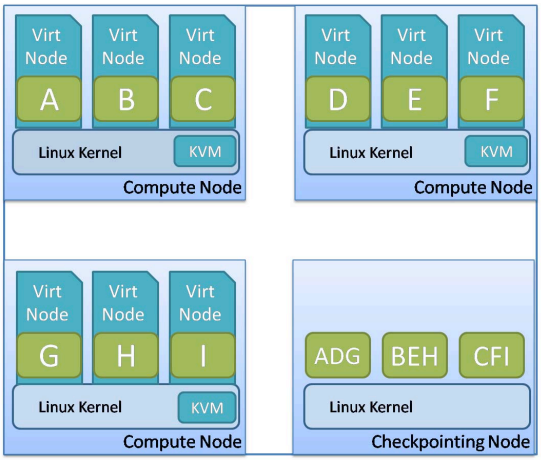

Fig. 3. A virtualized cluster using diskless checkpointing and orthogonal RAID. In the checkpointing node, the three-letter checkpoints correspond to parity taken from each checkpoint (e.g. A XOR B XOR C for ABC).

three, we can make one of the three a parity VM as well. Now we have two parity VMs and four normal VMs. We can continue to add VMs in this way with the new restriction that for every two VMs, we must create a third parity VM and store the group of three on different nodes.

This construction should feel very familiar to the common practice of gridding RAID groups of disks across different controllers. The idea is that a controller failure could bring down the entire RAID, but if only one disk per RAID group is assigned to each controller, any controller failure will not destroy a single RAID group. A configuration is depicted in Figure 2. The cluster configuration is shown in Figure 3.

By stacking RAID groups, it is possible to achieve more than one VM per node. Furthermore, we can see that we can distribute the responsibility of parity upkeep among the nodes in a RAID5 fashion. Thus, instead of having "checkpointing processors" that can do no real work (or else they would have to be checkpointed as well), we can distribute the parity and allow all physical machines to host working VMs that contribute to the overall execution of a job. Figure 4 shows a sample configuration that accomplishes Distributed Virtual Diskless Checkpointing (DVDC) with all compute nodes.

Additionally, the process of calculating parity at checkpoint time is also simplified. Instead of m processors doing all

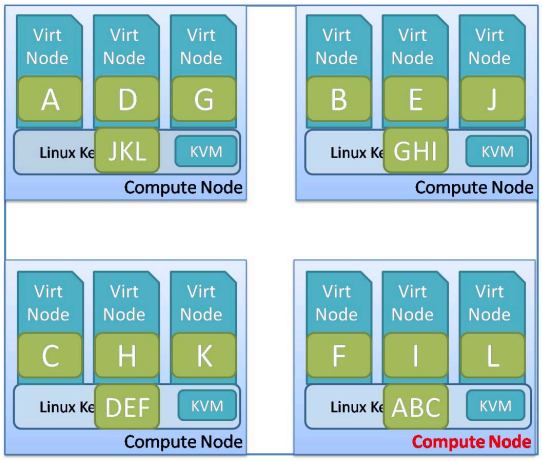

Fig. 4. A virtualized cluster using diskless checkpointing and orthogonal RAID with no checkpoint node.The three-letter checkpoints correspond to parity from each individual checkpoint (e.g. A XOR D XOR G for ADG).

the parity work, the parity calculation is evenly distributed automatically by having each node contribute equally to parity checkpointing. The parallelization of the parity calculation should relieve the CPU burden by a factor linear in the amount of machines in the cluster.

C. Utilizing Live Migration

Remus, as explained in Section VI, extends the concept of live migration to rapidly ferry dirtied pages to a backup node, where many checkpoints are maintained in the case of a failure. Cully et al. [9] were able to achieve checkpoints on the order of 40 times a second using this technique, although at that rate there was a significant impact to the system. As shown in many checkpointing studies [23], the disk remains the largest bottleneck, contributing to high checkpoint latencies. Diskless checkpointing is primarily a method not for reducing overhead, but latency. That is, due to the parity calculation process, we may not see huge gains in overhead, but the point at which the checkpoint is usable should be drastically reduced since we do not have to flush anything to disk. The effect is subtle, but can greatly enhance the overall estimated time to completion for a task in the presence of errors. This is not to say that using a disk is inefficient; indeed, the simplicity and reliability of secondary storage has kept traditional disk-based checkpointing as the mainstream method for implementing checkpoint/restart fault tolerance.

We can use the same process Remus uses for fast checkpointing and apply it to the realm of diskless checkpointing. This realization comes from the fact that Remus is simply using live migration as a convenient method through which to implement efficient incremental checkpointing. As with normal diskless checkpointing, we can compress the information that must be stored and passed over the network by employing copy-on-write or incremental checkpointing and suitably compressing the differences of the last checkpoint when sending information over the network. Thus, the amount of information

we must keep in-memory becomes a function of how fast and how many pages get dirtied, and, for compression, what percent of each page is changed.

V. An Analytical Model on Virtual Diskless Checkpointing

As a starting point, we can start with a model with no checkpointing system. In the presence of failure, the system must restart from the beginning. If we assume that we know the fault-free execution length of the program and the failure distribution, then we can create a simple model to derive the probability model for the program's time to completion.

In a checkpoint-free system, we may imagine a "progress bar" that randomly resets according to a given probability distribution. Our question of execution length then becomes, "how long will it take until the progress bar reaches 100% without randomly starting over?" Checkpointing modifies this thinking only slightly by keeping the progress bar from completely starting over and instead going back only to the end of the last checkpoint time.

Events that have a constant rate of occurrence over all fixed time intervals are said to follow a Poisson process. Though we can imagine cases where the Poisson assumption may not hold even on single computers (cf. the "bathtub curve" model for failures with its infant mortality and end-of-life scenarios), it is often used as a basis for fundamental design decisions due to its mathematical tractability.

A. Theory

In the case of the restarting progress bar, we find the probability of completion as a function of:

- T - the total execution length of the program (fault-free)
- T_{fail} - an expontial random variable describing the time before failure
- T_{nochk} - a random variable for the total execution length given no checkpointing system
- λ - the parameter for the exponential variable (1/MTBF)
- F - a Poisson random variable denoting the number of failures that occur over the total duration

The expected time to completion can be construed as a product of the expected time before failure given that the failure occurred before time T with the expected number of failures that occur before a complete run. That is, the time penalty is the average amount into the job before failure times the average number of failures before completion.

$$E[T_{nochk}] = E[F]E[T_{fail}|T_{fail} < T] + T$$

Using the assumed distributions, this expression is as follows,

$$E[T_{nochk}] = \frac{e^{\lambda T} - 1}{1 - e^{-\lambda T}} \times \frac{1 - (\lambda T + 1)e^{-\lambda T}}{\lambda} + T \qquad (1)$$

We cover the case with no checkpointing because this particular formulation makes it easy to see how checkpointing affects the expected value. With checkpoints, our rollback does not go all the way back to the beginning of execution, but back to the previous checkpoint. Thus, by looking at the product as discussed before, we can see that checkpointing simply breaks the job into many smaller sub-jobs, each with effectively no checkpointing system. To show these changes, we need to introduce a couple new terms to express the expected running time under a checkpointing system,

- N - the length between checkpoints
- $\frac{T}{N}$ - the number of checkpoints
- T_{chk} - a random variable for the total execution length given a checkpointing system

Thus, with N being the time between checkpoints, our formula, assuming no overhead momentarily, becomes:

$$E[T_{chk}] = (E[F]E[T_{fail}|T_{fail} < N] + N) \times \frac{T}{N} \qquad (2)$$

$$= (\frac{e^{\lambda T} - 1}{1 - e^{-\lambda T}} \times \frac{1 - (\lambda T + 1)e^{-\lambda T}}{\lambda} + N) \times \frac{T}{N} \qquad (3)$$

Finally, we can give the expressions for the case with non-negligible checkpointing overhead costs. We introduce the following terms,

- T_{ov} - overhead introduced by checkpointing
- T_r - time to repair given a failure
- $T_{chk;ov}$ - a random variable for the total execution length given a checkpointing system with non-negligible checkpointing overhead

The expected time to completion for such a system is similar to the previous case, but failures can occur during the checkpointing process and repair costs are paid per failure.

$$E[T_{chk;ov}] = (E[F](E[T_{fail}|T_{fail} < N+T_{ov}]+T_r)+N+T_{ov}) \times \frac{T}{N}$$

Where,

$$E[F] = e^{-\lambda(N+T_{ov})} - 1$$

and,

$$E[T_{fail}|T_{fail} < N+T_{ov}] = \frac{1 - e^{-\lambda(N+T_{ov})}(\lambda(N + T_{ov}) + 1)}{\lambda - \lambda e^{-\lambda(N+T_{ov})}}$$

B. Analysis

To gain a better understanding of the behavior of the proposed system, we can plug in the parameters of our model with previously published performance statistics. As mentioned before, published MTBFs of high-end clusters can be as low as 3 hours MTBF, giving a failure rate (λ) of 9.26e-5 failures/sec. We set our execution time to 2 days (typical of long-running HPC application), and the baseline overhead is 40 ms, which conforms to figures given commonly in many Live Migration papers [9] [7]. We look at the expected time to completion and compare the ratio of this time with the time to completion under no faults. We also compare our method with a baseline disk-full checkpointing method.

In both cases, we can essentially look at the amount of data and speed of data transmission for each operation to

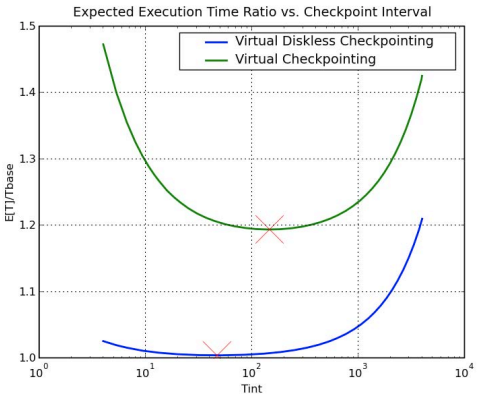

Fig. 5. Diskless Checkpointing vs. Normal Disk-full Checkpointing: We vary the checkpointing interval (Tint) and calculate how the expected time ratio changes. The X marks indicate minima, or optimal checkpoint intervals for each method. In this test case, we use the configuration seen in 4, with four physical machines and 12 virtual machines.

determine overhead times. In both diskless and baseline, the checkpointing itself is commensurable. Both also incur network overhead: the baseline to send to a NAS, and diskless to migrate checkpoints to calculate parity. However, the network step in the baseline is bottlenecked by a single NAS, whereas diskless checkpointing distributes the traffic evenly among nodes. Finally, the last step is to write to the NAS, in baseline, or to calculate parity through a large XOR operation in diskless.

To compare our proposed method with a normal checkpointing system, we ran an analysis, varying the checkpoint interval, to find the optimal checkpoint times in both systems. We then compared the optimal checkpointing times in both systems by looking at the overhead with respect to a perfect, fault-free run of the job. The results of this analysis are shown in Figure 5. Under the sample scenario, diskless checkpointing reduces estimated time to completion by 18% over disk-based checkpointing, with 1% overhead ratio from T_{base}.

There are two important differences here that contribute to performance: First, the network step for DVDC is sped up by a factor roughly linear in the number of machines, since the traffic is distributed equally among all machines. Second, an in-memory XOR operation is going to be orders-of-magnitude faster than a disk write operation of the same size. These two factors taken together show nearly negligible overhead costs over the ideal fault-free execution, while the traditional checkpointing, even at an optimal interval, adds nearly 20% to the total execution time.

VI. Related Work

There is much recent work devoted to implementing checkpointing features in distributed virtual systems. Much of the work either centers around maintaining global consistency [1], file system consistency [28], developing frameworks [19] [33], or highly optimized, low overhead solutions using asynchronous techniques [9] or by exploiting multicore topologies [4]. It should be noted that these techniques also can be applied to virtual machines designed to implement sandboxes around codes, such as the OCAML VM or the Java VM [2]. Also, several frameworks that implement distributed checkpoints can be found in [34] and [15].

Remus is one particular example of a virtual checkpointing system for high fault tolerance [9]. Remus runs servers in pairs, in active/standby mode, where the active node runs speculatively and asynchronously sends checkpoints to the standby node. Because the active node is running "in the future," the checkpoint contained by the standby node will always be the most recent image available to the entire system. This feat is achieved through use of buffers that wait for checkpoint confirmation. These checkpoints can be taken as many as 40 times per second. The end result is a system that can transparently tolerate up to one failure per server pair and incur no downtime in the presence of failure.

Another rich topic related to our methods includes many modifications to and frameworks for live migration [17] [3] [27]. Other strategies from improvement looks at virtualizing device drivers [26] [13] or paravirtualization [31].

Our virtual diskless checkpointing system is most similar to Remus [9], with some significant differences. In virtual diskless checkpointing, each host is both considered "active" and "backup" at the same time. The reason this is possible is due to the partitioning of VMs into different RAID groups. Also, the stored backups are not fully functional VM's but a single parity checkpoint of the entire RAID group. Remus adopts a replication approach using the active/standby paradigm: the authors suggest that Remus can run in an N-to-1 fashion for active and backup hosts, respectively, for additional flexibility. Virtual diskless checkpointing has no such restriction and can accommodate clusters of varying sizes. One trade-off between DVDC and Remus is that Remus runs the active host speculatively and updates the backup asynchronously, so that at any time, the backup contains the "most up-to-date copy" available of the system. In the presence of failure, DVDC requires all nodes to roll back to their previous checkpoints, compute the failed node's checkpoint from parity and data, and then resume. The distinction is somewhat blurred by the fact that Remus essentially "runs in the past" and thus still will lose the speculated portion of execution during failure. Nevertheless, Remus can resume execution upon failure immediately while DVDC must roll back and do parity calculations before resuming.

VII. Conclusions

We have shown different diskless checkpointing architectures for clusters of virtualized systems. We combine ideas from orthogonal RAID, RAID5, and diskless checkpointing to achieve a high-performing virtualized cluster architecture that can withstand physical machine failures. We have derived equations for system performance and models to corroborate our equations and test different configurations. We feel that our novel method of diskless checkpointing using a virtualized cluster is unique in that for a modest memory overhead, we

are able to achieve a low-latency, low-overhead failure tolerant configuration of virtual machines. These findings have spurred us to explore other ways to improve aspects of our design, and we are currently looking at the benefits of using page hashes to speed up live migration when similar VMs reside at the host destination. We have found our algorithms to be relatively easy to implement and understand, and they confer many benefits of reliability and performance.

Virtual diskless checkpointing shows many promising benefits over both traditional virtual checkpointing and non-virtualized diskless checkpointing. Diskless virtual checkpointing removes the primary bottleneck when capturing and storing state, and relies on the RAID paradigm to provide extra fault tolerance in the face of unreliable hardware. High-performance computing is suffering a crisis in that hardware reliability issues are consuming more and more usable CPU time. We believe that virtualization could be an effective platform to base tomorrow's high-end systems to meet the demands of both performance and fault tolerance.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. This research is sponsored in part by the U.S. National Science Foundation (NSF) under grant CCF-1102624. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agency.

REFERENCES

[1] S. Agarwal. *Distributed Checkpointing of Virtual Machines in Xen Framework*. PhD thesis, Indian Institute of Technology, 2008.

[2] A. Agbaria and R. Friedman. Virtual-Machine-Based Heterogeneous Checkpointing. *Software Practice and Experience*, 32(12):1175–1192, 2002.

[3] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiobergs. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, San Diego, CA, June 2007.

[4] K. Chanchio, H. Leangsuksun, and V. Ratanasamoot. Enhancing Reliability in Grid Systems with Virtual Machine Checkpointing Mechanism. http://www.hongong.net, 2008.

[5] C. F. Chandler, C. Leangsuksun, and N. DeBardeleben. Towards resilient high performance applications through real time reliability metric generation and autonomous failure correction. In *Resilience '09: Proceedings of the 2009 Workshop on Resiliency in High Performance*, pages 1–6, New York, NY, USA, 2009. ACM.

[6] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–188, June 1994.

[7] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, 2005.

[8] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 1–14, 2004.

[9] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. USENIX Association, 2008.

[10] J. Dongarra. Top 500 supercomputer sites, 2011.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, 2002.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[13] H. Jo, H. Kim, J. Jang, J. Lee, and S. Maeng. Transparent fault tolerance of device drivers for virtual machines. *IEEE Transactions on Computers*, 59(11):1466–1479, November 2010.

[14] I. Koren and C. Krishna. *Fault-Tolerant Systems*. Elsevier/Morgan Kaufmann, 2007.

[15] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical report, Technical Report, 1997.

[16] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[17] M. Nelson, B. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, Anaheim, CA, April 2005.

[18] D. NNSA and D. DARPA. High-End Computing Resilience: Analysis of Issues Facing the HEC Community and Path-Forward for Research and Development. http://www.darpa.mil.

[19] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. *OPERATING SYSTEMS REVIEW*, 36:361–376, 2002.

[20] D. Patterson, G. Gibson, and R. Katz. A case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. of the ACM SIGMOD'88*, Chicago, IL, June 1988.

[21] J. Plank. Improving the performance of coordinated checkpointers on networks of workstations using raid techniques. *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 76–85, 1996.

[22] J. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under UNIX. In *Proceedings of the USENIX 1995 Technical Conference*, pages 213–223. USENIX Association, 1995.

[23] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 9(10):972–986, 1998.

[24] J. S. Plank, J. Xu, J. Xu, R. H. Netzer, and R. H. B. Netzer. Compressed differences: An algorithm for fast incremental checkpointing, 1995.

[25] B. Schroeder and G. Gibson. Understanding Failures in Petascale Computers. In *Journal of Physics: Conference Series*, volume 78, page 012022. Institute of Physics Publishing, 2007.

[26] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.

[27] F. Travostino, P. Daspit, L. Gommans, C. Jog, C. de Laat, J. Mambretti, I. Monga, B. Oudenaarde, S. Raghunath, and P. Wang. Seamless live migration of virtual machines over the Man/Wan. *Future Generation Computer Systems*, 22(8):901–907, October 2006.

[28] G. Vallee, T. Naughton, H. Ong, and S. Scott. Checkpoint/Restart of Virtual Machines Based on Xen. In *High Availability and Performance Computing Workshop (HAPCW 2006)*.

[29] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. *ACM SIGOPS Operating Systems Review*, 39(5):148–162, 2005.

[30] G. Wang, X. Liu, A. Li, and F. Zhang. In-Memory Checkpointing for MPI Programs by XOR-Based Double-Erasure Codes. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 84–93.

[31] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[32] S. Yi, J. Heo, Y. Cho, and J. Hong. Adaptive Page-Level Incremental Checkpointing Based on Expected Recovery Time. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, page 1476. ACM, 2006.

[33] H. Yu, X. Xiang, and J. Shu. A New Global Consistent Checkpoint Based on OS Virtualization.

[34] G. Zheng, L. Shi, and L. Kale. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 93–103. IEEE Computer Society, 2004.