LA-UR- 09-04995

Approved for public release;
distribution is unlimited.

Title: Overlapped Checkpointing with Hardware Assist

Author(s): Christopher Mitchell
James Nunez
Jun Wang

Intended for: IEEE Cluster 2009 -
Interfaces and Abstractions for Scientific Data Storage
(IASDS) Workshop

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the Los Alamos National Security, LLC for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

Overlapped Checkpointing with Hardware Assist

Christopher Mitchell ^{#1}, James Nunez ^{*2}, Jun Wang ^{#3}

[#] *School of Electrical Engineering and Computer Science, University of Central Florida
4000 Central Florida Blvd., Orlando, Florida 32816-2450, USA*

¹ *mitchell@eecs.ucf.edu*

³ *jwang@eecs.ucf.edu*

^{*} *Los Alamos National Laboratory*

P.O. Box 1663, Los Alamos, NM 87545, USA

² *jnunez@lanl.gov*

Abstract—We present a new approach to handling the demanding I/O workload incurred during checkpoint writes encountered in High Performance Computing. Prior efforts to improve performance have been primarily bound by mechanical limitations of the hard drive. Our research surpasses this limitation by providing a method to: (1) write checkpoint data to a high-speed, non-volatile buffer, and (2) asynchronously write this data to permanent storage while resuming computation. This removes the hard drive from the critical data path because our I/O node based buffers isolate the compute nodes from the storage servers. This solution is feasible because of industry declines in cost for high-capacity, non-volatile storage technologies. Testing was conducted on a small-scale cluster to prove the design, and then scaled at Los Alamos National Laboratory. Results show a definitive speedup factor for select workloads over writing directly to a typical global parallel file system; the Panasas ActiveScale File System.

I. INTRODUCTION

As High Performance Computing (HPC) class systems continue to evolve and grow to the petascale and beyond, the issue of failure mitigation becomes even more important. The Top500 list [7] exemplifies the growing complexity in HPC systems. The release highlights for the November 2008 list show the large push for multi-core processors, with quad-cores dominating the list, and an increase in cores per system (from 4,850 cores in June 2008 to 6,240 cores in November 2008). This increased complexity ultimately leads to a higher probability of failure of any given system due to the sheer number of components. Thus, HPC systems have to contend with failure being conventional rather than exceptional.

Most applications running on HPC systems normally continue to make progress in light of system failures by checkpointing the state of the application to storage. Checkpoints are designed so that at specific intervals or time steps of a calculation, the running application pauses to write the current state of the calculation out to non-volatile media. In the event of a component failure, the application will read back the saved state data from storage and the computation will resume from the saved state rather than from the beginning of the job. According to an analysis performed by Schroeder and Gibson [24] in 2007, it was concluded that “in the case where the number of cores per chip doubles every 30 months, the utilization drops to zero by 2013, meaning the system would spend 100% of its time writing checkpoints or recovering

lost work, a situation that is clearly unacceptable.” While an extreme prediction, this does highlight the disconnect between growth in computational performance and I/O performance gains.

The frequency of writing out checkpoints is normally application specific and, sometimes, user specific: usually taken at intervals on the order of four to eight hours. If the application can restart its computation from the point of the last checkpoint and not from the beginning of the calculation - a few hours of work are lost rather than potentially weeks or months worth.

Unfortunately, checkpointing is an expensive operation because most applications require that all computation wait until the checkpoint writes complete to allow the checkpoint to capture a consistent state of the computation from across all participating nodes. Checkpointing is a very I/O intensive operation which places great demands on the I/O and file systems; a typically weak point in HPC systems [21]. In a worst case scenario, a checkpoint will require the entire contents of all participating nodes’ memory to be written to non-volatile storage. This equates to an enormous amount of storage space when an application uses a moderate to large number of nodes. Attempts at improving the amount of non-productive time spent in checkpointing have centered around either attempting to shrink the checkpoint file size or make the checkpoint I/O process faster. Shrinking the checkpoint file size only works on a per application basis as the data to be saved needs to be examined, optimized, reduced, and possibly compressed. On the other hand, reducing the I/O time is application independent yielding a global optimization and, thus, is the focus of our research.

Other research has yielded a variety of methods to reduce the time to checkpoint an application’s state through expansion of one of the two above general processes. Ross, et. al. [22], experimented with tuning the application I/O patterns within the FLASH astrophysics application in order to improve checkpoint write times. Attempts to improve checkpoint performance on a system-wide scale however are very diverse in their methodology. The Zest project [19] exploits the native spindle bandwidth of the individual drives in an array by removing the RAID controller hardware and building a special purpose, write-only file system. The disadvantage of this technique is that it requires two storage pools to be operational

and maintained (one for Zest and one for an associated Lustre volume). As a result, the user is unable to read back their data until the Zest daemon decides to sync the data to the Lustre volume. The Scalable Checkpoint / Restart (SCR) system [17] is another approach to checkpoint optimization. The SCR system checkpoints to a neighboring node's RAM within the allocation the application is running on and uses the memory on extra nodes to keep data parity. This approach, while fast, reduces the amount of memory available to the application. It also introduces additional jitter at the operating system level [20] due to the management daemons, running in the background on the compute nodes. Finally, a newly introduced system for checkpoint optimization is PLFS [8] which strives to improve performance by converting a performance poor $N \rightarrow 1$ write pattern into a better performing $N \rightarrow N$ write on the parallel file system.

Our proposed method for improving checkpoint performance calls for a novel system that buffers the checkpoint data so that the checkpoint I/O and computation operations can be overlapped. This overlap is what allows for improved checkpoint performance and faster resumption of computation. The process is accomplished by introducing a set of high-speed, non-volatile buffers housed in I/O nodes which are on the same interconnect fabric as the compute nodes. Through these I/O nodes, checkpoint data can be quickly ingested through the cluster interconnect with little congestion from the rest of the HPC center's systems and no potential for bottlenecking at the global parallel file system. Once ingested, the checkpoint data is now the responsibility of the storage subsystem, freeing the compute nodes to resume application execution. Thus the resumption of computation is effectively overlapped with the comparatively slower process of writing the checkpoint data to disk.

This two step process works exceptionally well because it moves the critical data path to compute node and the I/O nodes performing the buffering rather than the compute node and the storage servers. Thus, high returns on performance can be obtained both from the increased write bandwidth to the buffers, as well as from allowing the compute nodes to return to application execution sooner. These returns are substantiated by our testing enumerated in section III, making this solution a competitive offering to combat increasing I/O demands on HPC systems.

Our solution alleviates the issues surrounding other proposed methods by providing a solution that does not require drastic application rework, introduce OS jitter, consume application memory, or require significant equipment investment or redesign.

In section II, we cover the system design with descriptions of each major component as well as an overall discussion of the checkpoint's data flow. In section III, we discuss the testing performed to prove the theory of operation as well as to show the performance gains possible with this architecture over writing to a global parallel file system. Section IV discusses alternate designs that were proposed and the pitfalls associated with each, as well as a look at future progression of this design

that would aid its transition from prototype to production. Finally, we look at related work in section V and make our final conclusions in section VI.

II. SYSTEM DESIGN

A. Overall Data Flow

In most current HPC environments compute and storage resources are separated so that storage can be shared across many clusters [9], [13], [25]. By centralizing storage within the HPC environment though, we sacrifice performance since the storage is physically located farther away from the compute nodes, and no longer accessible via the high-speed cluster interconnect but rather a traditional Ethernet based storage lane [13]. This forces the data through additional network hops and, potentially, slower, more congested networks to reach the central storage services. Thus, when a checkpoint is written out, it must be passed from (1) the compute node, to (2) a cluster interconnect to storage or I/O nodes, and then finally to (3) the global parallel file system or central storage. Steps (2) and (3) introduce a funnel effect where increasing numbers of requests are competing for finite I/O bandwidth thus further increasing checkpoint write times.

In our proposed overlapped version of checkpointing, as shown in Figure 1, this process breaks down in a slightly different manner as follows. The application enters its usual checkpointing routine, but instead of calling `MPI_File_open` and `MPI_File_close`, the application makes a call to `MPI_Chkpt_open` and `MPI_Chkpt_close`, as seen in step (1). These modified calls serve to redirect the file path for the output data from the intended final destination to the intermediary buffers and log the final destination path, seen in step (2). The checkpoint data is forwarded through the PVFS2-client on the compute node to the PVFS2-server on the I/O nodes that house the intermediary buffers in step (3). The data store for PVFS2 on these I/O nodes is composed of RAM-Disks that are striped together by PVFS2 so that it appears as one large storage pool to the compute nodes. This pool forms the buffer location for both the checkpoint data and associated log files as shown in step (3a).

On these I/O nodes, a daemon monitors the log files looking for new checkpoint files that have completed writing and have had their file handles closed by `MPI_Chkpt_close`. Once `MPI_Chkpt_close` finishes, responsibility for the checkpoint file now rests with the storage system, allowing the application to return to computation. Upon detection of a closed file, the checkpoint daemon issues a copy command to relocate the checkpoint data from the buffer space to its final destination on the global parallel file system, as shown in steps (4) and (5).

Looking at a timing diagram of the process with and without the assistance of the intermediary buffers illustrates where savings can be achieved with this system in comparison to traditional methods. In traditional systems, computation progresses until the application barriers for a checkpoint of the current state of the computation. At this point, all computation stops until the write to persistent storage is complete. To

Fig. 1. Data flow of overlapped checkpointing process.

start this checkpoint process, all processes, across all nodes, must wait in a barrier to write the checkpoint file. This initial synchronization serves to ensure that all “in-flight” communications have been drained from the network and no computation is progressing on any node. Once within the checkpoint routine, the processes write out their data to the parallel file system. Once each process completes the checkpointing process, it is free to return to doing work until the next checkpoint. Occasionally, there is a barrier at the end of the checkpoint process before resuming computation.

In contrast, the overlapped checkpointing solution with an intermediary buffer makes the following changes to the timing diagram of the checkpoint process, as seen in Figure 2. In an overlapped system, all processes must still coordinate before, during, and after the checkpointing routine as in the traditional system; but the overall time to write out the checkpoint is reduced due to the presence of a “closer”, faster buffer. This allows the processes to resume computation earlier than in the traditional system. Additionally, the processes do not have to wait for the buffer to migrate the data to the global parallel file system, as this is done in the background by the I/O nodes.

As a result, the checkpoint write to disk can be overlapped with the computation.

It should be noted that checkpoints are written at a set frequency, determined by the application developer. The determination of checkpoint frequency is usually based upon estimated Mean Time To Interrupt (MTTI) for the machine the application is executed on and how long each simulation step is estimated to take. Thus, developers will decide to checkpoint after a set number of simulation time steps that should take less time to complete than the estimated MTTI rating for the machine. An attempt at quantifying this process of selecting a checkpoint interval was refined by Daly in [11].

B. I/O Node Setup

The central components of this system are the I/O nodes that house the buffers on behalf of the compute nodes they service. These I/O nodes currently generate their buffer space out of RAM Disks in an effort to maximize available write bandwidth to the compute nodes and exceed the bandwidth available from the hard drives that compose the global parallel file system. An additional advantage of using RAM Disks is the fact that seek

Fig. 2. Timing diagram of checkpointing process using the proposed overlapped checkpointing process.

requests from the applications associated with non-contiguous access patterns (such as $N \rightarrow 1$ strided [15]) are absorbed by the RAM Disks with no performance penalty, further improving performance. The setup of the RAM Disks must be carefully planned because responsibility for safe handling of the checkpoint data is transferred to these I/O nodes and is no longer overseen nor detected by the application. The servers running RAM Disks acting as buffers for this system must be UPS protected to protect against power failures. To protect against the data loss due to reboots or prolonged power failures (where UPS reserves become depleted), an `init.d` module is installed on the I/O nodes which detects transitions between Linux run levels. When a system exits to run level 6 or 0 (reboot or halt), the module proceeds to unmount the RAM Disk and use the `dd` operation to generate an image of the RAM Disk on persistent storage (local hard drive if available or network attached storage). Upon restart and entrance into run levels 3 or 5 (full multi-user mode with or without a graphical interface respectively) the same module initializes a new RAM Disk and then proceeds to restore the saved image. This process ensures that as long as a run level transition occurs, the contents of the RAM Disk are protected. In cases where a run level change does not occur prior to failure, a more serious error has occurred (ex. Kernel panic) where data integrity is unlikely to be maintained anyway. Methods to handle this scenario are discussed in section V.

Since a single node is unlikely to be able to house enough RAM to build a buffer large enough to satisfy checkpoint demands (x86-64 based machines typically can physically handle between 32 and 128 GB of RAM depending on the limitations of the chipset and the operating system [1]), a management system must be put in place to aggregate RAM Disks across multiple I/O nodes. To accomplish this task, PVFS2 is used to enable the striping of RAM Disks across multiple nodes in order to aggregate the available space into a single buffer

pool [6]. This also serves to further improve performance as multiple independent RAM Disks can cooperate to ingest application data. Finally, in using PVFS2, parallel semantics are gained for the RAM Disk buffer pool. This is important, as interaction with the buffer is transparent to the user who is expecting to write to a parallel file system. By supporting parallel semantics, applications can continue to use existing MPI-IO calls without modification, within the limits of what PVFS2 supports, and, in turn, eases the implementation of this system.

It should be noted that while RAM Disks are being used for this iteration of the proposed system, any form of non-volatile storage media would be applicable. The idea being to use a media that is faster than traditional hard drives in terms of bandwidth while also minimizing or eliminating seek penalties. Thus media such as NV-RAM or Solid State Disks (SSD) would be appropriate substitutes for RAM Disks while also providing the added benefit of true non-volatility that is only currently being emulated by UPS backing the RAM Disks.

PVFS2 alone is not capable of handling the data migration process from the buffer to the data's final location on the global parallel file system. To accomplish this goal, a management daemon is installed on the primary I/O node. This daemon scans the log files generated by the `MPI_Chkpt_open` and `MPI_Chkpt_close` commands looking for confirmation that a checkpoint written to the buffer is complete and its file handle has been closed by the application. With confirmation of the checkpoint's completeness, the daemon proceeds to use the original destination path contained within the log file to migrate the file from the buffer to its final destination location on the global parallel file system.

To facilitate portability across various storage systems, the primary I/O node (where the management daemon is installed) requires the client modules to be installed for the file system

in use within the HPC installation (ex. Panasas's DirectFlow client, PVFS2's client kernel module, Lustre client, or GPFS client). The client modules allow the various parallel file systems to be mounted directly on the I/O node and made accessible without the need for special tools. The PVFS2-client kernel module is installed on the primary storage node to provide visibility of the buffer system from the file system hierarchy. Effectively, this allows the management daemon to treat the file migrations as simple file copies between mount points in the file system hierarchy. Future iterations of the management daemon software will leverage a parallel copy to derive further improved bandwidth to the parallel file system.

While this setup works perfectly well when a large enough buffer is available on the I/O nodes to ingest the checkpoint file, this solution must also handle situations where the checkpoint file is larger than the buffer size. In the current implementation tested, the system administrator is asked to size the buffer space based on their worst case runtime situation. So, in other words, what is the largest checkpoint file their applications generate multiplied by how many instances of this application can run at once. The next iteration of this system will proceed to stream the buffered data to the parallel file system once the buffer reaches a defined watermark rather than waiting for the checkpoint file to be completed and closed. In doing this, the buffers would no longer have to be grown to an unreasonable level while still providing a performance boost to the application.

Frequently, I/O nodes are shared by jobs running on the same machine. Thus, the design of these nodes needs to address the issue of data security. Traditional security in the I/O stack within an I/O system has relied on user permissions applied to files to prevent different users from accessing different data sets. With the introduction of this intermediary buffering layer, this concept of separation between users must still remain in tact. On systems such as Sandia's Red Storm or the BlueGene series of machines, I/O or service nodes are provisioned with the compute nodes for a job ensuring that users have a dedicated private I/O path and no chance for cross contamination between users [9], [25]. For the first version of our system, we will assume a trusted environment in addition to dedicated I/O servers for a set of compute nodes during the job allocation. Subsequent versions of our system will consider integrating tighter security by introducing secure erasure procedures between jobs, active checking of permissions and user credentials.

C. MPI-I/O Modified Calls

The I/O nodes alone are not enough to make this system work transparently. A bridge mechanism must be provided to the application to allow the use of the intermediary buffer. The future work section describes a version that requires very little modification for the application developer and does away with the need for MPI-IO. For the initial prototype, two modifications are required to the MPI-IO library to augment the MPI_File_open and MPI_File_close routines leaving all other I/O routines completely unchanged [14]. These new

functions are called MPI_Chkpt_open and MPI_Chkpt_close and have the following parameter list:

```
int MPI_Chkpt_open(MPI_Comm comm, char
*fName, int amode, MPI_Info info, MPI_File
*fh)
and
int MPI_Chkpt_close(MPI_File *fh, char
*fName, MPI_Comm comm)
```

As long as the developer uses the modified function calls, supplied via a separate header file, use of the I/O nodes and their associated buffers will occur, rather than directly writing to the parallel file system. Transparently, the MPI_Chkpt_open command uses a configuration file called "chkptPath.cfg" within the individual compute nodes to locate the proper mount point of the I/O nodes and then uses this information to perform a path substitution, thus redirecting the checkpoint data to the I/O nodes. This process is illustrated in Figure 3 which shows the control flow of the MPI_Chkpt_open function. Additionally, there is no modification to the parameter list passed allowing for a direct substitution of the function name. Likewise the MPI_Chkpt_close function also utilizes "chkptPath.cfg" to locate the mount point to the storage servers and uses the additional parameters passed to determine which file log to process (the char *fName parameter) as well as which MPI communication group is responsible for this I/O operation. Only the root of the communication group manipulates the log files. MPI_Chkpt_close is illustrated in flow chart format in Figure 4.

Fig. 3. Flowchart of MPI_Chkpt_open function operation.

These functions make the critical assumption that checkpointing is a write intensive operation with little to no reading being performed while the writes are taking place. While there is nothing preventing the developer from performing a read operation, the developer must be aware of the fact that the target of the returned file handle is really the I/O node's buffers and thus must use the modified calls strictly within the bounds of the modified open and close calls and then must reopen the file with the standard API calls afterwards for continued access to the file. Thus, the following rules apply to use of these functions: (1) the application is permitted to use these

Fig. 4. Flowchart of MPI_Chkpt_close function operation.

functions to generate a new checkpoint file. (2) reading and writing to this file is permitted as long as MPI_Chkpt_close has not been called on the file, (3) once the file has been closed traditional MPI_File_open and MPI_File_close must be used to access the data as it is now on the parallel file system. Overall, the developer can make an MPI-I/O aware application compatible with this overlapped checkpoint optimization with little additional effort and few changes to the code.

III. PROOF OF CONCEPT & PERFORMANCE TESTING

A. Test Software

Testing of the effectiveness of the Overlapped Checkpointing system was conducted by employing a standardized test system that exercises the MPI-IO subsystem and reports both timing and bandwidth information. While many such tests were available for such purposes, MPI-IO Test [4] from Los Alamos National Laboratory was chosen. MPI-IO Test was designed from the start with both parallel I/O and scale in mind. This test suite is capable of reporting various statistics relating to the timing of the I/O operations performed as well as the resulting bandwidth achieved during these tests. Additionally, this suite is capable of testing all three I/O access patterns seen in HPC applications: $N \rightarrow 1$, $N \rightarrow N$ strided, and $N \rightarrow N$ [4], [15].

For thorough test coverage of our new system, all three access patterns were tested against both the Overlapped Checkpointing system as well as directly writing to Panasas. Tests were also run both with 47,001 byte (un-aligned write) and 1 MB write (aligned write) sizes. This variation in write size allows examination of the system in what is typically a worst case situation and a best case situation respectively due to alignment in relation to disk blocks. Total checkpoint sizes were varied such that the following were tested: 3.5 GB, 7.0 GB, 17.5 GB, and 24.5 GB. This range was selected to get adequate coverage from small checkpoints to checkpoints as large as our Overlapped Checkpointing Buffers could handle as discussed below in the system setup for Cadillac. Finally, all testing was repeated three times in round robin fashion (each access pattern was run and then the same sequence repeated three times) to ensure result consistency as well as to eliminate the possibility of caching skewing the test results.

B. Scalability Testing - LANL Cadillac Cluster

1) *System Setup*: To test the overlapped checkpointing method at larger scale, testing was conducted using Los Alamos National Laboratory's Cadillac Cluster. Cadillac is a 128 node dual processor (Intel Xeon 2.20 GHz.) test bed cluster for storage system research. The nodes each have 4 GB of DDR-266 RAM. The compute nodes were loaded with the cluster's default Fedora Core 4 install (Kernel 2.6.14-3) while the PVFS servers used for the overlapped checkpointing testing were upgraded to Fedora Core 9 (Kernel 2.6.25-14).

For testing, the cluster was split into 25 I/O nodes each with a 1 GB RAM Disk enabled (for a RAM Disk size of 25 GB total across all nodes) and 80 compute nodes. The Panasas system was configured such that the volume used was distributed over the six shelves of SB2000 blades [5]. RAID-5 was used on the Panasas volume and the resulting file system was mounted on all Cadillac nodes. The network was setup such that a bank of 24 switches (5 nodes per switch) were used as a front end for all the Cadillac nodes with each node being plugged into a switch in round-robin fashion. Each of these switches was aggregated (via a 4 port gigabit Ethernet trunk) on a central Extreme 8808 switch that also had the Panasas shelves connected to it. Each of the six Panasas shelves had four trunked gigabit Ethernet connections back to the central switch providing a total of 24 gigabits of raw bandwidth to the storage system.

2) *Test Results*: The testing process focused on two key areas of the systems in question. First, testing was conducted to determine the bandwidth achievable when writing the various checkpoint files from the application to either our Overlapped Checkpointing buffers (on the I/O nodes) or directly to Panasas. Second, testing was conducted to determine if the buffers could be offloaded to Panasas within a reasonable time frame.

Testing of the write bandwidth to the overlapped checkpointing system compared to the Panasas storage system showed that our buffering system does improve checkpoint write performance when used with certain access patterns as seen in Figure 5. When looking at the $N \rightarrow 1$ access pattern, Overlapped Checkpointing displays an average of 6.14 times better performance than Panasas (between 545 and 559 MB/sec compared to between 85 and 91 MB/sec respectively) with un-aligned writes. However, aligned writes favor Panasas by a factor of 1.27 times (1,298 to 1,546 MB/sec versus 1,048 to 1,174 MB/sec). The $N \rightarrow 1$ strided writes, on the contrary, favor the Overlapped Checkpointing system by a factor of 3.71 times for un-aligned (238 to 267 MB/sec versus 69 to 77 MB/sec respectively) and 1.97 times for aligned writes (772 to 824 MB/sec as compared to 323 to 514 MB/sec respectively). Finally, the $N \rightarrow N$ writes favored Panasas for both the aligned and un-aligned tests. The un-aligned writes generated a 5.14 times improvement in favor of Panasas (1,113 to 1,416 MB/sec compared to 224 to 257 MB/sec respectively) while the aligned writes showed a 2.39 times improvement (1,033 to 1,530 MB/sec versus 433 to 631 MB/sec). It should

Fig. 5. Results of transfer testing from application to both the Overlapped Checkpointing and to Panasas systems, respectively, for each access pattern.

be noted that on most of the tests, Panasas's write bandwidth spiked much higher than above for the smallest checkpoint file (3.5 GB) creating an unrealistic outlier that was not factored into the ranges and improvement factors above to avoid skew for the average case.

Looking at the transfer time between the Overlapped Checkpoint buffers and Panasas, the goal was to determine if the data could be transferred within a reasonable time frame such that it would be unlikely that the next checkpoint would come in before the first one was completely written out. Enumerations of checkpoint intervals, as mentioned earlier, is highly application dependent so ideally if the transfer can take less than an hour and preferably less than thirty minutes then this condition will be considered satisfied. Testing showed that transfers for the single file checkpoints (produced by the $N \rightarrow 1$ and $N \rightarrow 1$ strided writes) took between 2.5 and a little under 19.5 minutes for the file sizes tested. The N file checkpoints (produced by the $N \rightarrow N$ writes) transferred in between 2.5 and a little over 18 minutes depending on the aggregate size of the N files. Since the system is currently using a sequential copy, these numbers are considered a worst case but still meet the criteria. An advanced version of this system will employ a parallel copy to perform the transfers thus further improving the transfer time.

Therefore, it can be concluded that Overlapped Checkpointing is an ideal solution for the problematic access patterns detailed above ($N \rightarrow 1$ strided - aligned and un-aligned as well as $N \rightarrow 1$ un-aligned writes). These patterns typically lend themselves to poor performance due to the high number of seeks and/or the poor alignment with the physical disk

block sizes. By providing our system's buffering capability, these problematic access patterns are transformed into a contiguous write with minimal seeking and little mismatch with the disk block size when they are transferred to disk. However, as this system does not provide a benefit to all access patterns, the application developer is advised to not blindly utilize this system much as how the system administrator is advised to consider both the cluster size and performance impact when evaluating this option for their system. Overall, the Overlapped Checkpointing system provides commendable performance improvements for select problematic file access patterns commonly seen in HPC workloads.

IV. LESSONS LEARNED & PROPOSED ENHANCEMENTS

A. Initial Design and Drawbacks

The design that was detailed in section II and tested in section III is the result of an iterative process that refined the idea of overlapping the checkpoint and computation. Each successive iteration addressed key design pitfalls that would have made this process substantially less desirable for production consideration.

The first design proposed called for non-volatile "storage-class memory" devices to be installed in every compute node. These devices could be true NV-RAM, Solid State Disks (SSD), or even prototype Phase Change Memory (also known as PC-RAM) [26]. The idea being that if there was a non-volatile location within each compute node that could be accessed at bus speeds, the checkpoint could be "written out" fast enough to notice a significant speed improvement over the traditional practice of sending the data to a parallel file

system on the network. The non-volatile memory would be managed by either an FPGA or a programmable NIC such that this became a plug-and-play solution with little to no intervention in the compute node's software stack. The pitfall in this design was the complexity of the firmware and drivers and the inherent cost of installing a dedicated, specialty card on every compute node. Also, if the nodes are in blade form factor they will not have the space to accept an additional card.

The design was then iterated to use a RAM Disk setup on each compute node and have a daemon running in the background that would coordinate amongst its peers to reconstruct the checkpoint on the global parallel file system after computation had resumed. Unfortunately, two problems were discovered with this design. First, systems that would benefit most from these enhancements are the capability systems that run "hero-run" science applications [2] and require all available system memory to run these gigantic applications. Due to this limitation, application developers will accept slower I/O rates in order to gain extra fidelity in their simulation by using the memory that would otherwise have been allocated to the RAM Disks. Also, no matter how lightweight the daemon could become, it would still introduce jitter into the system, as the daemons are given CPU time by the OS to schedule the next set of network transfers of the data to storage. This jitter manifests itself in the form of destabilizing tightly synchronized applications due to the daemon taking CPU time at random intervals on any given node, leading to computations no-longer being in lock step with one another. This can also lead to substantially reduced performance due to nodes waiting for long periods at barrier sync points within the code instead of continuing on with productive work [20].

In contrast to installing the buffer on the compute nodes, the idea of placing a buffer at the storage server was considered. This is a technique that is not new in the storage industry as shown by increasingly large buffers on hard drives or as separate modules such as those used by NetApp [16]. The drawback of this approach hinges on the amount of contention at the storage servers. Considering the entire storage stack, the storage servers constitute the location of most contention due to the aggregation of all compute node requests across all attached clusters. This concentration places considerable strain on a buffer located here which drives up the required capacity (and cost) required for it to be effective.

Since it was shown that this idea would not be sound if implemented on the compute nodes themselves or at the storage server, it was proposed that an intermediary location that was still close to the compute nodes but not physically within them would be an optimal location for this optimization framework. Many current supercomputers use some form of an I/O node to bridge compute resources to storage resources [13]. On these clusters, the I/O node may simply be a specialized router that directs data from the cluster interconnect fabric to the Ethernet based storage network. On other systems, such as the IBM BlueGene (L, P, and Q series), a more specialized implementation of the I/O node concept is used which handles more of the I/O process due to the compute nodes running

a light-weight Kernel (Compute Node Kernel) rather than a full OS. If the buffering occurred at the I/O node, network congestion would be reduced due to access being available to the high-speed cluster interconnect and there would be no opportunity for OS jitter at the compute node. These I/O nodes would only need to have additional RAM or other form of non-volatile storage installed. Buffering at the I/O node level also brings the added benefit of reducing cost since several compute nodes worth of data can be aggregated into a relatively small number of I/O nodes rather than providing this capability at each compute node individually.

B. Proposed Enhancements

While the existing implementation properly executes the proposed design and performs respectably compared to existing solutions. There are a few modifications that would benefit a future version of the overlapped checkpointing solution. The first of which being the ability to integrate with the FASTOS I/O Forwarding Scalable Layer (IOFSL) project [3]. The IOFSL project is targeted for large processor count supercomputers and designed to offload I/O operations from the compute nodes by function shipping the requests to a dedicated set of I/O nodes to perform the file system operations. The benefits of this include being able to reduce the system overhead on the compute nodes while shielding the parallel file system from the massive number of clients present in today's premiere clusters. Since the I/O operations have to be shipped to an I/O node in this system, similar to what is being done within the overlapped checkpointing system, a logical extension would be to implement the checkpoint (or any write for that matter) buffering at the I/O nodes. The benefits of this include being able to streamline the implementation by not requiring specialized storage nodes but rather use the existing I/O nodes that are already present in this proposed setup. Since the I/O software stack will already be augmented to push all I/O requests to the I/O nodes, there will be no need to have modified MPI-IO calls specifically for writing out checkpoints. In addition, the associated logging will no longer be needed, as the system will already have the needed information and the ability to designate checkpoint writes with an MPI hint being passed to the environment. In short, all dependence on MPI-IO is removed and no modifications to an application will be needed if we integrate into the IOFSL project.

The ability for our overlapped checkpointing system to handle increased pressure in the scenario of checkpoint data being written faster than it can be migrated to the parallel file system will need to be addressed in a future version. As stated, the current implementation works on the assumption that there is sufficient time between checkpoint writes to perform all needed file transfers to the parallel file system. It is feasible that this may not always be the case and as such the issue could be addressed in one of two ways. The simplest method for handling this scenario is to have the return of the file handle from MPI_Chkpt_open be stalled long enough to finish writing out the previous checkpoint to the parallel file system. While this negates any performance advantage for the current

checkpoint operation, it ensures that the subsequent writes do not fail from a full buffer. A more advanced solution would introduce a streaming-like approach to the buffering operation where as pieces of the file arrive at the I/O servers, transfer to the parallel file system would begin as well instead of waiting for the full file to be received at the I/O server. In this manner, the scenario of dealing with buffer pressure can be alleviated.

Finally, a cleaner interface to the user and their applications is possible by one of two possible methods which would eliminate the need for a specialized open and close MPI API call. The first possible simplification would involve integrating the functionality directly into the ADIO layer of the MPI runtime. In doing so, it would be possible to have the users simply prefix their file paths with an identifier such as "chkpt:" rather than that of the file system in use. This would allow for the overlapped checkpointing system to interact directly with the internal data structures of the MPI-IO system to get needed information rather than requiring the users to provide it (as currently done with the modified API calls). The other option is to implement a FUSE based implementation that can provide the needed redirection and management leaving the users to simply modify their checkpoint save path. This implementation is similar to what was done with PLFS where a FUSE layer driver was written to transparently manipulate the I/O access pattern to improve performance [8].

With examination and experimentation of these proposed methods, the concept of overlapped checkpointing could easily be transferred to a full production ready package for broad deployment.

V. RELATED WORKS

The concept of using checkpoint/restart as a fault tolerance mechanism is not new. Chandy and Lamport's 1985 paper on Determining Global States in Distributed Systems [10] defined the key principles behind today's barrier synced checkpoint routines. This work on capturing global system states directly influences current works to perform system level checkpointing such as Berkley Lab's Linux Checkpoint Restart (BLCR) [23] and the Transparent, Incremental Checkpointing at Kernel Level (TICK) project [12]. These works attempt to capture the entire system state (including application, data, and OS) with little to no intervention by the application. Unfortunately, this results in larger checkpoints containing redundant data (such as the OS image) and little flexibility for the application developer to customize the checkpoint process to capture only what is really needed.

Ross, Nurmi, Cheng, and Zingale recently showed that checkpoint performance can be improved through careful application profiling and analysis [22]. In examining the FLASH astrophysics code, it was determined that too much time was spent preprocessing the data in the HDF5 library routines prior to sending it to storage. The solution called for packing the data into a large block within memory and writing this contiguous piece to disk, thus bypassing HDF5 and improving performance from 3% to 23% of peak possible performance. This exercise illustrated the application specific nature of

I/O optimizations and the difficulty in obtaining significant performance improvements.

One approach to globally optimizing checkpoint performance is the Zest system by Nowoczynski and Yanovich [19]. Zest calls for a special, write-only file system that exposes all available spindle bandwidth and in turn achieves over 90% storage system efficiency. This is accomplished via removal of the RAID controller, non-deterministically writing the data, and bundling the metadata into the data stream to avoid a separate operation. However, Zest has potentially a significant cost associated with implementing the required two separate disk subsystems. Additionally, the sync process to Lustre so that the data can be read by users adds additional delay in providing access to data.

Scalable Checkpoint / Restart (SCR) [17] also globally optimizes checkpoint performance but uses RAM Disks on neighboring compute nodes to store application and parity data. SCR breaks data into chunks for distribution amongst a node allocation along with the calculated parity value. Like RAID, should a node fail, the data can be reconstructed from the remaining data chunks and parity values. Use of this technique, however, reduces the available memory on the compute node to the application. OS jitter [20] also is introduced due to the management daemons on the compute nodes and the calculation of parity.

I/O Delegate and Caching System (IODC) [18] is a system that improves checkpoint performance by providing a distinct set of I/O nodes to offload file system access tasks to. These nodes reduce lock as well as access contention to the storage resources as compared to allowing all compute nodes direct access. Additionally, a caching system was implemented to provide additional performance gains for frequently accessed data and for write aggregation. Compared to our system though, IODC's default 1 GB cache is not capable of reordering an $N \rightarrow 1$ strided pattern to an $N \rightarrow 1$ pattern which effectively turns the file system's worse case scenario into its base case due to the removal of excessive seeking. Additionally, IODC does not contain any explicit provisions for handling I/O node failure and keeping the data housed in this RAM based cache safe while it is in-route to disk.

Finally, PLFS (Parallel Log-structured File System) [8] is a system to improve checkpoint performance by breaking an $N \rightarrow 1$ checkpoint write, which performs poorly on the file system, to an $N \rightarrow N$ write pattern which can achieve desired write performance. This translation between access patterns is accomplished by a FUSE mounted file system that translates the logical write pattern into the better performing write pattern on the underlying parallel file system. PLFS is a complementary system to overlapped checkpointing as PLFS can still potentially benefit from the buffering mechanism on the I/O nodes. Further exploration would be needed to see if it is possible to combine the benefits of both systems.

VI. CONCLUSION

In this paper we have developed and discussed a novel method of improving checkpoint write performance within

High Performance Computing clusters. By using high-speed, non-volatile buffers close to the compute nodes to quickly offload the data being written to a checkpoint file, applications can resume computation sooner than if left to write directly to the available parallel file system. A management daemon on the I/O servers providing this buffering service allows for the seamless migration of the data to the intended destination after computation has resumed effectively allowing for the overlap of the two operations. A set of modified MPI-I/O calls make this system easily accessible from a developer perspective, which makes adoption easier. Performance was shown to be significantly better when compared to writing directly to the parallel file system for select I/O access patterns. Testing showed that this system works well for both aligned and un-aligned $N \rightarrow 1$ strided writes as well as un-aligned $N \rightarrow 1$ writes, all of which are traditionally hard on existing file systems. Additionally, the transfer of data from the buffer to the parallel file system was shown to occur in sufficient time as to not hold up subsequent checkpoint operations. This system compares favorably with other methods studied to improve checkpoint write performance; including methods to tune individual applications, system level checkpoints, file system level enhancements (Zest), and alternative methods of buffering (SCR). Overall, this system has been proven to be stable and has the possibility to expand to a more full featured, production ready system for use within HPC class systems.

ACKNOWLEDGMENT

We would like to thank Gary Grider, Andy Nelson, and the HPC-5 team at Los Alamos National Laboratory for their incite into the characteristics of HPC application checkpointing as well as use of their Cadillac cluster. Also, we would like to thank Grant Mackey and Saba Sehrish of the University of Central Florida as well as Rob Ross of Argonne National Laboratory for their assistance in refining the system design. Finally, we would like to thank the NSF for sponsoring this work under grants: CNS-0646910, CNS-0646911, CCF-0621526, CCF-0811413 and the Department of Energy for also sponsoring us under Early Career Principal Investigator Award: DE-FG02-07ER25747.

REFERENCES

- [1] "Dell poweredge 1950 systems: Getting started with your system," September 2008. [Online]. Available: http://support.dell.com/support/edocs/systems/pe1950/multilang/gsg/DAO_BCC/YC585A00MR.pdf
- [2] "Doe office of science, advanced scientific computing research, incite project," March 2009. [Online]. Available: <http://www.sc.doe.gov/ascr/INCITE/index.html>
- [3] "iofs: i/o forwarding scalable layer," March 2009. [Online]. Available: <http://www.mcs.anl.gov/research/projects/iofs/about/index.php>
- [4] "Mpi-io test," July 2009. [Online]. Available: <http://institute.lanl.gov/data/software/>
- [5] "Panasas, inc.," March 2009. [Online]. Available: <http://www.panasas.com>
- [6] "Parallel virtual filesystem, version 2 project," March 2009. [Online]. Available: <http://www.pvfs.org>
- [7] "Top500 supercomputing sites," March 2009. [Online]. Available: <http://www.top500.org>
- [8] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, and M. Wingate, "Pifs: A checkpoint filesystem for parallel applications," November 2009. [Online]. Available: <http://institute.lanl.gov/plfs/plfs.pdf>
- [9] R. Brightwell, "Red storm status," March 2005. [Online]. Available: <http://www.sandia.gov/~rbrigh/slides/invited/redstorm-sos9.pdf>
- [10] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.
- [11] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303 – 312, 2006. [Online]. Available: <http://www.sciencedirect.com/science/article/B6V06-4F490KH-6/2f6ebfa65591e5d0eb09e2ae5ae3b2ed44>
- [12] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 9.
- [13] G. Grider, H. Chen, J. Nunez, S. Poole, R. Wacha, P. Fields, R. Martinez, P. Martinez, S. Khalsa, A. Matthews, and G. Gibson, "Pascal - a new parallel and scalable server io networking infrastructure for supporting global storage/file systems in large-size linux clusters," in *Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International*. April 2006, pp. 10 pp–340.
- [14] W. Gropp, R. Thakur, and E. Lusk, *Using MPI-2: Advanced Features of the Message Passing Interface*. Cambridge, MA, USA: MIT Press, 1999.
- [15] R. Hedges, B. Loewe, T. McLarty, and C. Morrone, "Parallel file system testing for the lunatic fringe: the care and feeding of restless i/o power users," in *Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE / 13th NASA Goddard Conference on*, April 2005, pp. 3–17.
- [16] D. Hitz and A. Bhargava, "A storage networking appliance," <http://media.netapp.com/documents/tr-3001.pdf>, February 2006.
- [17] A. Moody, "The scalable checkpoint / restart (scr) library: Approaching file i/o bandwidth of 1 tbs," in *TeraGrid Fault Tolerance Workshop*, 2009. [Online]. Available: <http://www.teragridforum.org/mediawiki/images/3/38/Moody2009ws.pdf>
- [18] A. Nisar, W.-k. Liao, and A. Choudhary, "Scaling parallel i/o performance through i/o delegate and caching system," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.
- [19] P. Nowoczynski, N. Stone, J. Yanovich, and J. Sommerfield, "Zest: Checkpoint storage system for large supercomputers," in *Supercomputing 2008: 3rd Petascale Data Storage Workshop*. ACM/IEEE, 2008.
- [20] F. Petrini, D. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asc q," in *Supercomputing, 2003 ACM/IEEE Conference*, Nov. 2003, pp. 55–55.
- [21] R. Ross, "Slides from presentation: Meeting the storage needs of computational science at extreme scale," January 2009.
- [22] R. Ross, D. Nurmi, A. Cheng, and M. Zingale, "A case study in application i/o on linux clusters," in *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM, 2001, pp. 11–11.
- [23] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, and A. Lumsdaine, "The lam/mpi checkpoint/restart framework: System-initiated checkpointing," *International Journal of High Performance Computing Applications*, vol. 19, no. 4, 2005.
- [24] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, p. 012022 (11pp), 2007. [Online]. Available: <http://stacks.iop.org/1742-6596/78/012022>
- [25] C. Sosa and B. Knudsen, "Ibm system blue gene solution: Blue gene/p application development," <http://www.redbooks.ibm.com/redbooks/pdfs/sg247287.pdf>, December 2008.
- [26] W. Wilcke, "Flash and storage class memories: Technology overview and systems impact," in *HEC-FSIO Conference*, 2008. [Online]. Available: http://institute.lanl.gov/hec-fsio/conferences/2008/presentations/day3/Wilcke-PanelTalkFlashSCM_fd.pdf