2013 IEEE 27th International Symposium on Parallel & Distributed Processing

Optimizing Checkpoints Using NVM as Virtual Memory

Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan
College of Computing
Georgia Institute of Technology, Atlanta, Georgia, USA
sudarsun@gatech.edu,{ada, schwan}@cc.gatech.edu

Dejan Milojicic
HP Labs, Palo Alto, USA
dejan.milojicic@hp.com

***Abstract*—Rapid checkpointing will remain key functionality for next generation high end machines. This paper explores the use of node-local nonvolatile memories (NVM) such as phase-change memory, to provide frequent, low overhead checkpoints. By adapting existing multi-level checkpoint techniques, we devise new methods, termed NVM-checkpoints, that efficiently store checkpoints on both local and remote node NVM. The checkpoint frequencies are guided by failure models that capture the expected accessibility of such data after failure. To lower overheads, NVM-checkpoints reduce the NVM and interconnect bandwidth used with a novel pre-copy mechanism, which incrementally moves checkpoint data from DRAM to NVM before a local checkpoint is started. This reduces local checkpoint cost by limiting the instantaneous data volume moved at checkpoint time, thereby freeing bandwidth for use by applications. In fact, the pre-copy method can reduce peak interconnect usage up to 46%. Since our approach treats NVM as memory rather than as 'Ramdisk', pre-copying can be generalized to directly move data to remote NVMs. This results in 40% faster application execution times compared to asynchronous approaches not using pre-copying.**

***Keywords*-Non volatile memory (NVM); PCM; Checkpointing; Memory bandwidth, Pre-Copy;**

I. Introduction

Moving toward the exascale, the failure rate of applications is expected to increase to the order of tens of minutes. To overcome such lack of availability, there is an urgent need for robust fault tolerance mechanisms with low impact on application performance. Checkpoint/restart is a well-known and widely used fault tolerance technique used in a majority of current HPC applications. A checkpoint is a snapshot of application state stored in persistent devices that can be used for restarting execution after failure. However, with increasing problem sizes and failure rates, checkpoint sizes per processor are increasing at a higher rate than available I/O bandwidth [1], leading recent studies to conclude that for exascale machines, $checkpointsize/IObandwidth$ ratio must be drastically reduced to make such systems usable [2].

Scalability for checkpointing cannot be obtained from traditional methods using parallel file systems (PFS), caused by problems that include limited I/O bandwidth and contention at the I/O subsystem level, and already apparent on current petascale machines [3]. New I/O methods using intermediate staging nodes for buffering checkpoints can insulate applications from slow disk latencies [4], [5], but researchers have already begun to look for additional ways to reduce I/O data volumes and reduce necessary data movements [4], [6], [7].

This paper explores a promising alternative to disk-based checkpointing by adapting and improving multi-level checkpointing methods that have already been shown to exhibit 30-40% benefits over traditional PFS-based checkpointing methods [8]. These methods combine frequent local storage checkpoints with less frequent remote checkpoints (e.g., to neighboring or to I/O staging nodes). Local checkpoints can reduce the resources consumed by checkpoint data movement (in terms of data volume and compute node CPU), as well as the contention for resources such movements may impose on application processes. Another argument in favor of these methods is that a substantial portion of application failures is due to soft errors, recoverable via local node reboot or application process restart, vs. hard errors that render the entire compute node unusable. In fact, recent results [9] show that in the ASCI Q machine at LANL, about 64% of failures are due to soft errors, and hence can be handled with frequent local checkpoints, and a study by Intel [10] further points out that, with moving towards 16nm from the 90nm process, such soft error rates can increase by up to 32x. Considering a 64-128 core per node configuration and multiple sockets (say, 5 sockets), this would result in a soft error rate increase of up to 100-120%. Using node local checkpoints permits each node to restart from its local state without the need to access the distributed storage infrastructure.

The NVM-checkpoint facility described in this paper uses future byte addressable nonvolatile memories (NVM) like PCM (Phase change memory) [11] as low-cost, low-power solutions to storing local checkpoints. While most prior work uses NVMs as fast I/O devices, hidden under standard file system interfaces, NVM-checkpointing gains substantial performance benefits from using NVM as memory. Using NVM as memory utilizes several key NVM hardware features, including byte addressability, hardware-supported virtual addressing, processor caches, and the ability to use NVM as an extension to DRAM when this is a scarce resource. To overcome the bandwidth limitations of future NVMs, it exploits hardware paging support and page protection techniques, as well as a novel pre-copy checkpoint method.

1530-2075/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPS.2013.69

This pre-copy method starts moving checkpoint data even before a checkpoint is initiated, thereby reducing the total data volume to be moved at checkpoint time. Further, by providing a remote memory access interface to applications, NVM-checkpoints extend this pre-copy scheme to remote checkpointing, as well, leveraging the RDMA nature of the interconnects used in high end machines. Here, pre-copying reduces peak bandwidth usage, which then reduces the likelihood of interconnect contention.

In summary, the contributions of this paper are as follows:

- Nonvolatile memories are used for fast multilevel (node local and remote) memory checkpointing. NVM is treated as slow, persistent memory as opposed to fast disk, thereby leveraging NVM features that include byte-addressability, hardware support for memory management, and caching. The resulting application-level checkpoint solution hides associated checkpoint latencies and permits efficient use of NVM resources.
- Novel checkpoint 'pre-copy' methods leverage the 'memory' nature of NVMs, to further reduce the performance impact due to NVM bandwidth limitations.
- Remote extensions of pre-copying via a remote memory access interface to NVM serve to reduce the peak interconnect bandwidth used for checkpointing, thereby reducing the potential interconnect contention experienced by applications.
- NVM-checkpointing is fully implemented on Linux-based nodes and is evaluated with three well known HPC applications. Experimental results demonstrate 15% reduction in local checkpoint time compared to ramdisk of which 8-10% is due to pre-copy, up to 46% reduced peak interconnect usage using the remote pre-copy mechanism, and 40% improvement in application execution time compared to an asynchronous (non-blocking) approach without pre-copy.

The remainder of this paper is organized as follows. Additional background and motivation for using NVM with multi-level checkpointing are discussed first. Next, we describe the performance models used to assess NVM impact on application execution time and to guide the choice of optimization opportunities. Section IV describes the design of our solution, followed by implementation details in Section V. Experimental analyses appear in Section VI, followed by concluding remarks and future work.

II. Background and Motivation

Application-initiated Checkpointing. Checkpointing mechanisms differ based on the level of application transparency, synchronization, and storage hierarchy. Regarding transparency, they can be broadly classified into application-initiated vs. transparent checkpoints. Transparent checkpoints do not require applications to explicitly handle failures, by saving an entire process address space, whereas application-initiated checkpoints store only those data structures identified by application developers. When the application footprint is large, transparent mechanisms incur high storage cost and space. Studies show that large scale HPC applications mostly use application-initiated mechanisms [12] and require less storage space. This paper's scope, therefore, is restricted to application-initiated checkpoints.

Attributes	DRAM	PCM
Write Bandwidth	~8 GB/Sec	~2 GB/Sec
Page Write latency	~20-50 ns	~1 us
Page Read latency	~20-50 ns	~50 ns

Table I
NVM vs. DRAM H/W performance [11]

Multilevel Checkpoints. Considering the storage hierarchy, checkpoint data can be stored in global distributed storage using some PFS (e.g., Lustre), to an intermediate staging I/O node (e.g., using I/O methods like ADIOS [4]), or using a hybrid multilevel checkpoint approach. Multilevel checkpoint mechanisms are intended to overcome limitations with traditional approaches based on PFS, the main idea being to (temporarily) store checkpoint data at multiple locations, from local scratch memory, to storage resources (i.e., ramdisk/SSD/disk) at remote neighbors (i.e., peers) or designated (i.e., I/O) nodes, and finally to the PFS. The rationale is that application reliability will improve with increased levels of data redundancy. Studies have shown both local and remote node checkpointing to scale well, reducing data movement cost. An issue is that they rely on local memory to achieve performance, but future machines are already predicted to be memory scarce, and alternatives in which compute nodes are outfitted with HDD or SSD are not likely viable. NVM-checkpointing uses NVMs as node-local stores, and it overcomes the NVMs' potential bandwidth limitation (around 2 GB/sec, see Table I) with novel pre-copy based methods that alleviate bandwidth pressure. Section IV describes these mechanisms in more detail.

NVM for Checkpoints. Future NVMs like PCM [11] and Memristors [13], offer hardware-based persistence with 100x faster access rates compared to existing HDDs and SSDs. Better access rates combined with byte addressability, virtual memory paging support, and higher storage density make NVMs a promising candidate for overcoming the DRAM scalability barrier in sub-45nm technology nodes [14], and to provide persistence for fault tolerance. However, NVMs have certain hardware limitations, including slow writes, low die bandwidth, and high write costs. As seen in Table I, write latencies are 10x higher and overall bandwidth is 4x lower compared to DRAM, and other major limitations include 10^8 write durability compared to 10^{16} for DRAM and 40 times higher write energy/ bit.

Prior work using NVMs primarily differs in the placement of NVMs in the I/O stack. It can be broadly categorized into (1) NVMs as fast disks hidden behind DRAM acting like an intermediate cache [15], or (2) hybrid memory models in which NVMs are placed in parallel with DRAM connected by a memory bus as an access interface [9], [14]. When NVMs are treated like fast disks, applications access them using the file system interface for reading and writing persistent data. The use of the file system interface for applications is beneficial from the compatibility point of view, since legacy applications need not be modified. Further, this decouples the application from the underlying storage device. However, it fails to exploit NVM's byte addressability benefits, and every access to NVM requires user-to-kernel transitions along with additional data serialization cost. Moreover, current multi-layered, hierarchical file system designs using the virtual file system interface (VFS) suffers from high access latencies and synchronization costs for kernel metadata structures.

In contrast, when treating NVMs as an extension of current DRAMs, with support for virtual memory paging, NVM access does not imply kernel level transitions. Further, from the software interface point of view, hybrid memory systems can be (1) application transparent NVMs where the OS/HW manages data placement with no application involvement [16], [9], or (2) exposed to applications for direct access (loads and stores), supporting application controlled data placement similar to DRAM [17], [18]. Transparent checkpointing uses the first approach, where the entire process address space is replicated (copied) to NVM. For the second approach, applications use explicit interfaces to mark and allocate data structures that require persistence [17], [18], and these are the only ones placed into persistent NVM.

The NVM-checkpointing approach described in this paper is based on the hybrid memory model. In contrast to earlier work, however, NVM is not directly exposed to applications for checkpoint data writes. Instead, applications use specialized NVM interfaces, in a similar manner as with explicit application-initiated checkpoint operations in HPC codes, to provide information regarding the data to be placed in NVM, i.e., checkpointed data. During computation, application data remains in DRAM, thereby avoiding potentially substantial application slowdowns (shown to be up to 25% [19] for certain classes of write-intensive HPC codes). In addition, there is OS-level support for efficient NVM management and data movement across the DRAM/NVM boundary. In fact, these mechanisms are sufficiently general that they can also be used to support transparent checkpointing; this is not, however, the target of our current work, in part due to the possibly prohibitive checkpoint sizes seen in the HPC applications with which we are working. NVM checkpointing specifically takes advantage of NVM byte addressability and hardware support for management. To deal with limitations such as limited bandwidth and slow writes to NVM, it uses mechanisms like pre-copy and shadow buffering.

T_{total}	total execution time
$T_{compute}$	total compute only time
$T_{restart}$	total checkpoint fetch and restart time
T_{lcl}	total local checkpoint time
O_{rmt}	overhead due to remote checkpoint
I	Checkpoint interval time
t_{lcl}	local checkpoint time
t_{rmt}	remote checkpoint time
R_{lcl}	local checkpoint fetch time
R_{rmt}	remote checkpoint fetch time
$MTBF$	Mean time between failures$(1/\lambda)$
$MTBF_{lcl}$	failures recoverable from local nodes
$MTBF_{rmt}$	failures requiring remote node recovery
F_{lcl}	no. of failures recoverable from local node
F_{rmt}	no. of failures requiring remote node data
$NVMBW_{core}$	effective NVM bandwidth per core

Table II
CHECKPOINT MODEL NOTATIONS

III. PERFORMANCE MODEL AND GOAL

The performance of a checkpoint mechanism depends on several factors like MTBF (mean time between failures), checkpoint data size of application, storage hierarchy and the hardware devices used. The goal of our work is to understand the implications of NVM on checkpointing performance, and we use a simple model by extending the existing 2-level checkpoint model [20], [21] to suit our NVM based study.

Figure 1 shows a basic timing diagram of a multilevel checkpoint, with remote checkpoint overlapped with next compute and local checkpoint step, as commonly done with asynchronous non-blocking I/O operations. The total application execution time is directly impacted by the performance of the local checkpoint, remote checkpoint and the restart/recovery performance. The total runtime of an application can be denoted by

$$T_{total} = T_{compute} + T_{lcl} + O_{rmt} + T_{restart} + T_{recomp} \quad (1)$$

where T_{recomp} is the computation after the last checkpoint before application failure. This is typically wasted computation that needs to be re-executed due to failure.

Improving Local Checkpoint. The local checkpoint performance is dependent on the total number of local checkpoints (N_{lcl}) and the time taken for each local checkpoint (see Table II). The interval between each local checkpoint is dependent on the number of failures from which application is able to recover from local NVM. More formally,

$$N_{lcl} = T_{compute}/MTBF_{lcl}$$
$$T_{lcl} = N_{lcl} * t_{lcl}$$
$$t_{lcl} = chkpt.datasize/NVMBW_{core}$$

The time per each local checkpoint is dependent on the per process checkpoint size and the effective NVM bandwidth. As the number of cores per node increases, the effective bandwidth per core can substantially reduce. *Hence to improve the performance of local checkpoint, we need methods* *that reduce the impact of this limited bandwidth, thereby improving the local checkpoint overhead and performance.*

Reducing Remote Checkpoint Overhead. The remote checkpoint overhead for a synchronous coordinated checkpoint depends primarily on the effective interconnect/network bandwidth (i.e., InfiniBand in our case) to move data to the remote node.
$T_{rmt} = N_{rmt} * t_{rmt}$; N_{rmt}- no. of remote checkpoints
$t_{rmt} = chkpt.datasize/datamovementcost(G/sec)$
However, for asynchronous checkpoints, the remote checkpoint can be overlapped with computation, and its overhead is primarily the noise factor it imposes on the application execution. Prior studies have shown that, overlapping checkpoint data movement with communication intensive application can cause substantial overhead (e.g., 25% in [20], [4]). The communication overhead is primarily due to bandwidth contention between checkpoint data movement and application communications. Other relatively minor noise factors include CPU and memory. Hence the equation of remote checkpoint can be rewritten as,
$o_{rmt}(sec) = \alpha_{comm} + \alpha_{others}$
where o_{rmt} represents the remote checkpoint overhead, α_{comm} the application communication overheads due to asynchronous remote checkpoint, and α_{others} other overheads like memory and CPU.

Therefore, we require mechanisms which leverage NVM to reduce the bandwidth contention between application and remote asynchronous checkpoint without affecting the remote checkpoint interval.

Restart. With increasing failures, the restart and recomputation time of an application directly impacts the performance of an application. The recomputation and restart time depends on the local node and remote node failure rates. Using a node local NVM that can survive software failures (including system reboot) can substantially improve restart time performance.
$T_{restart} + T_{recov} = T_{lclrstart} + T_{lclrecomp} + T_{rmtrstart} + T_{rmtrecomp}$
The total restart time depends on the local and remote restart and recovery time.

Local failures: We assume that on average a computation fails half way between compute interval (I), as a result after fetching a checkpoint from a local node (R_{lcl}), I/2 of the computation must be re-executed
$F_{lcl} = T_{compute}/MTBF_{lcl}$
$T_{lclrstart} + T_{lclrecomp} = F_{lcl}(R_{lcl} + (I + t_{lcl})/2)$
where F_{lcl} denotes the no.of local recoverable soft failures

Remote failures: Since, we use node local NVM, our model assumes that the number of total failures that can be recovered from local NVM is higher than the remote NVM recovery. As a result, the interval between each remote checkpoint can contain several local checkpoints denoted by K. We assume that on average half of the segment (see Figure 1), containing one or more compute and local checkpoints is completed when a non recoverable node failure occurs. As a result, during restart, these compute and local checkpoints should be redone.

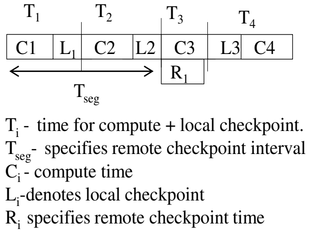
T1
T2
T3
T4
C1
L1
C2
L2
C3
L3
C4
R1
Tseg
Ti - time for compute + local checkpoint.
Tseg- specifies remote checkpoint interval
Ci - compute time
Li-denotes local checkpoint
Ri specifies remote checkpoint time

Figure 1. Multilevel checkpoint timing diagram

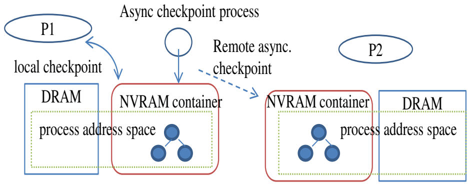

Figure 2. NVM-checkpoint Architecture

$F_{rmt} = T_{total}/MTBF_{rmt}$; F_{rmt} - denotes no.of hard errors
K = no. of local checkpoints in a remote checkpoint interval
$T_{rmtrstart} = F_{rmt} * R_{rmt}$; total time spent on remote fetch
$T_{rmtrecomp} = F_{rmt} * K(I + t_{lcl})/2$; wasted computation which includes compute and local checkpoint
In this work, we do not focus on improving the restart time performance, but we assume the local and remote restart time proportional to local and remote checkpoint time. Similar assumptions have been made in prior work [9].

IV. NVM-Checkpoints: Key Mechanisms

We next present the rationale for the key mechanisms and design decisions of NVM-checkpoints, including the use of NVM as 'memory' to leverage NVM advantages like byte addressability, and hardware support for VM management, protection and caching. Additional system methods like shadow buffers and pre-copy, deal with NVM access latency and bandwidth limitations.

NVM-checkpoints are based on the use of NVM for both local and remote checkpoint, where remote nodes can be peer compute or dedicated I/O nodes. Unlike prior work [20], [21] which is dependent on DRAM memory either for local or remote checkpoint, we reduce the DRAM memory usage by using NVM for local, remote, as well as intermediate buffers for data movement. With memory size predicted to be a bottleneck for exascale machines, using NVM can be highly beneficial. As discussed in Section III above, remote node checkpoints are less frequent compared to local checkpoints and the selection of the checkpoint interval depends on hard error failure probability, i.e., failures that make a compute node unusable and no data can be recovered.

Using NVM as Virtual Memory. Future NVMs can be used as fast disks with a file system interface or as virtual memory (VM). Arguments in favor of using NVMs as virtual memory include i) ability to exploit VM paging and protection, ii) hiding high write latencies using processor cache, iii) byte addressability and hence avoiding serialization and iv) ability to use NVM not only for storage, but also as heap for processing. Further, current file systems are not optimized for NVMs and require redesign.

To experimentally verify this, we use an I/O benchmark MADBench2 [22], to compare the performance of a ramdisk based checkpoint with a in-memory (DRAM) checkpoint. For memory checkpoint, we replace I/O calls (open/read/write/seek) with allocation and memcpy calls. The checkpoint data size is varied from 50 to 300 MB per core. In all the cases, memory checkpoint performs better compared to the ramdisk method, although both store data in DRAM. With increasing data size, the difference in checkpoint time widens and for 300MB, the ramdisk approach is 46% slower compared to in-memory checkpointing. Further analysis and profiling of both approaches shows that in the case of the ramdisk approach, the application executes 3x more kernel synchronization calls and spends 31% more time waiting for kernel locks, compared to the memory checkpointing approach. Additional costs like serialization and user-kernel transition for I/O calls add to checkpoint time. Using NVM as a fast disk requires filesystem redesign or architectural enhancements like BPFS [23].

This motivates us to use *NVM as virtual memory* for checkpointing in the first place. As a result, a process address space is extended with a per-process *NVM container*, as shown in Figure 2, maintaining persistent versions of the process' checkpoint data. In addition to eliminating file system serialization overheads, the use of NVM as memory permits us to reap other important advantages like using memory protection mechanisms for pre-copying.

Overcoming slow NVM write using shadow buffering. Table I shows the five year projection for PCM load/store latency and bandwidth [11]. The NVM write (store) operation is almost 10x slower compared to DRAM, while read latencies are comparable. Considering the high write latencies, exposing NVM directly to applications (e.g., as heap memory used during the computation) will depend on the load-store ratio of the workload. There are several research proposals to overcome slow writes by hiding NVM behind large intermediate cache [16]. But considering the scale of HPC applications, exposing NVMs directly to the application result in a severe application slowdown. A recent analysis by Li et al. [19] for write intensive HPC applications showed 25% slowdown when directly using NVM. For frequent checkpointing, writing data directly to NVM will be inefficient.

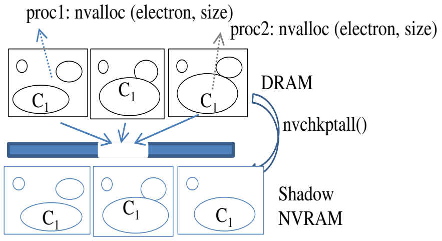
proc1: nvalloc (electron, size)
proc2: nvalloc (electron, size)
C1
C1
C1
DRAM
nvchkptall()
C1
C1
C1
Shadow
NVRAM

Figure 3. Shadow buffering for NVM

Therefore, to overcome this limitation, we use a *shadow buffering* mechanism shown in Figure 3. The shadow buffering method involves explicitly handling heap allocation for all checkpoint data structures (heap variables), and creating a DRAM and a shadow NVM memory chunk for checkpoint variables. Applications directly use a DRAM memory chunk pointer, and during a local checkpoint, all checkpoint data structures are moved to NVM chunks. The memory chunks can be of any granularity. When reading back the data, the application can directly access write protected NVM, and an attempt to modify the data would move the data back to DRAM for further writes. Shadow buffering is an effective technique when write sizes are large (in MBs), as in case of HPC checkpointing [24].

Dealing with limited NVM bandwidth using pre-copy. Future exascale systems are predicted to have at least 128 cores, and with increasing core count, the sustainable memory bandwidth per core (including NVM bandwidth) can reduce significantly. Figure 4 shows the effect of increased core count on effective per-core DRAM memory copy bandwidth using the LANL parallel memcpy benchmark [25]. As seen from the figure, with increasing core count, the per core bandwidth reduces by 67% even for data size of 33 MB. For NVMs with 2 GB peak device bandwidth and DDR based interface, the effective per core bandwidth can be as low as 400 MB/Sec in a 12 core/node configuration. Hence, using a shadow buffering technique alone may not be sufficient for an application with gigabytes of per node checkpoint data to overcome this NVM limitation. Dong et al. [21] were first to discuss the DRAM-NVM memory contention issue for transparent checkpoints and proposed a thread level serialization approach of using a dedicated checkpoint core for background copy. However, using serialization for checkpointing would incur high locking overheads and lead to slower checkpoints when the total checkpoint data size is less than the effective per core bandwidth.

To reduce the NVM bandwidth contention for a local checkpoint, we propose a *chunk level pre-copy* mechanism. The idea is to move the application allocated chunks (data structures) asynchronously to NVM before a coordinated checkpoint is started by partially overlapping computation with checkpoint. Three important questions need to be addressed for verifying the feasibility of this approach: 1) When to start chunk-level pre-copy to NVM?, 2) What happens if chunks are modified after pre-copy?, 3) What

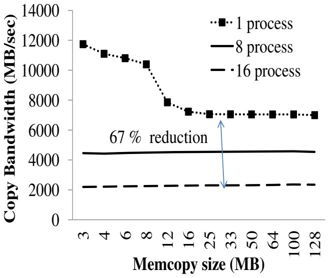

Figure 4. Memcopy bandwidth for parallel process

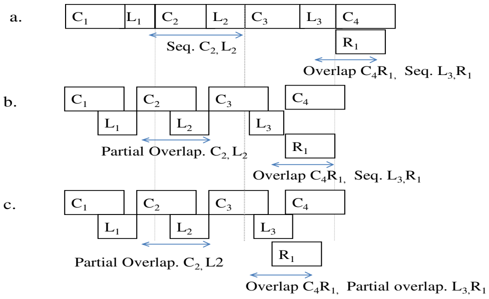
a.
C_1
L_1
C_2
L_2
C_3
L_3
C_4
R_1
Seq. $C_{2,} L_2$
Overlap $C_4R_{1,}$ Seq. $L_{3,}R_1$
b.
C_1
C_2
C_3
C_4
L_1
L_2
L_3
R_1
Partial Overlap. $C_{2,} L_2$
Overlap $C_4R_{1,}$ Seq. $L_{3,}R_1$
c.
C_1
C_2
C_3
C_4
L_1
L_2
L_3
R_1
Partial Overlap. $C_{2,}$ L2
Overlap $C_4R_{1,}$ Partial overlap. $L_{3,}R_1$

Figure 5. Checkpoint pre-copy timing diagram. C, L, R denotes compute, local and remote checkpoint step respectively. Figure a. shows basic sequential local and non-blocking remote checkpoint. Figure b. shows a pre-copy local checkpoint with compute and local checkpoint steps partially overlapped. Figure c. shows pre-copy remote checkpoint with all compute, local and remote checkpoints partially overlapped

happens if a checkpoint fails before completion?.

To answer these questions, we adopt virtual machine migration techniques where memory pages are moved to a destination incrementally even before the virtual machine is halted by using page protection methods to capture page protection and move them. The incremental movement reduces the migration time and peak I/O (network) bandwidth usage compared to moving all data at once. In order to deal with a crash and recovery, we maintain two chunk versions – a most recent completed checkpoint and a one that's currently 'in progress', i.e., being modified under pre-copy. By using NVM as memory as opposed to I/O device, we leverage hardware-supported protection mechanisms, and therefore can adopt this type of incremental technique.

Chunk-based pre-copy (CPC). A key aspect of our incremental approach is to move data in memory chunks as opposed to a memory page. Applications allocate data structures to heap which include variables that need to be checkpointed. The checkpoint variables – corresponding to chunks – are allocated using NVM specific interfaces and may vary in size from a byte to hundreds of megabytes. When all compute cores in a node write to NVM during a coordinated local checkpoint, the bandwidth becomes a bottleneck and such limitation increases with increase in the number of processors. Incremental data movement is a widely accepted mechanism for reducing such bottlenecks, for instance, most of the transparent checkpoint mechanisms 'pre-copy' process pages incrementally with write protection enabled and capture further modifications by handling protection faults. One prominent issue with the incremental technique is that, when most of the data frequently changes, protection fault overheads can be high, negating the benefits of page pre-copy. For instance, handling a page protection fault can take 6-12 μsec, and 3 sec. for 1 GB of data. Specifically, for application-initiated checkpoints in HPC applications, since most checkpoint data structures fully change, using page level pre-copy will not be beneficial.

To avoid such costly operation, we use a chunk level protection, i.e., all pages of a chunk allocated using the NVM interface are pre-copied in the background, even before a coordinated local checkpoint, and all chunk pages are right protected. After the pre-copy step, when a page belonging to chunk gets modified, the entire chunk is marked dirty (i.e., by unprotecting all pages) and pre-copied again in the background. During a coordinated local checkpoint, only the remaining dirty chunks are copied, thereby reducing the total data movement size to NVM and reducing the peak bandwidth utilization.

Figure 5 shows the timing diagram for both cases, the coordinated local checkpoint without pre-copy on the top graph, and NVM-checkpoint pre-copy scheme where local checkpoint is partially overlapped with compute phase. One issue with the pre-copy approach is that, for applications that randomly modify chunks, there may be significant application overhead due to repeated pre-copy of data from DRAM to NVM. As a side effect, pre-copy increases the protection fault overhead and the amount of data movement across the DRAM-NVM boundary. To address this issue, we derive two variations of the basic pre-copy technique.

Delayed Chunk Pre-Copy (DCPC). Starting local checkpoints at the beginning of a compute iteration is unnecessary, as there are several data structures that repeatedly get modified before a checkpoint, and intuitively it is sufficient to start the pre-copy sometime just before the checkpoint step. As a first step, to determine the checkpoint interval, our method waits for the first checkpoint step to complete and finds the approximate interval (checkpoint time - compute start time) along with per core checkpoint data size. Then, in order for pre-copying to have sufficient time to move all data to NVM before a coordinated checkpoint begins, we determine a pre-copy starting time called pre-copy threshold using the below equation. We continuously adapt the pre-copy threshold to deal with application changes across iterations.

$T_c(sec) = D/NVMBW_{core}$,
$T_p(sec) = I - T_c$,
where T_c is Checkpoint time, T_p is pre-copy threshold time, D is checkpoint data size (MB), I is the checkpoint interval, and $NVMBW_{core}$- Effective NVM BW/core.

Delayed Pre-Copy with Prediction (DCPCP): While pre-copy threshold reduces the need for repeated pre-copy of chunks, in some cases chunks can be modified until the completion of a compute phase (i.e., even after the pre-copy

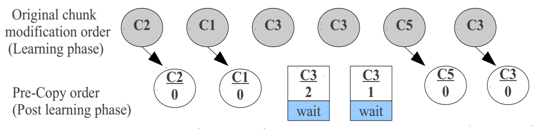

Figure 6. PreCopy with Prediction

threshold). For instance, in a molecular dynamics application (Lammps), we observe that a three dimensional result array with relative molecular positions in a lattice gets modified until the end of a compute iteration. We call these variables *hot chunks*. Applying a pre-copy or delayed pre-copy can increase the work done (i.e., repeated copying) and as a side effect, increases the dirt tracking and data movement cost. The pre-copy overhead for such hot chunks should be reduced, or no pre-copy should be applied. To enable this, we use a simple prediction table mechanism which captures the frequency of chunk modification by maintaining a counter for each chunk and a state machine representing the modification order.

Figure 6 shows the chunk modification state machine for Lammps and for simplicity only three out of the total 31 chunks are shown. During the initial learning phase (first checkpoint), chunks are tracked for changes and the prediction counter is updated. For subsequent iterations, when the processor issues a write fault, the chunk corresponding to the fault address is marked dirty, but not copied to NVM until the modification count is equal to or greater than the value in the prediction table. For instance, the chunk C3 is modified three times during the initial run. In subsequent runs, it is not copied until the counter becomes 0.

These delay-based mechanisms are optimizations and do not affect the data consistency. Hence, if the prediction fails, the data would be copied during the coordinated checkpoint step. Our analysis on three large scale HPC applications, GTC, Lammps, and CM1 code show a fairly constant modification order. This is not surprising, considering the fact that iterations are repeated without change in initial input. Our evaluation section shows the benefits and implications of the pre-copy schemes.

Using PreCopy for remote checkpoint. Remote checkpoints are required for tolerating node and rack failures. Remote checkpoints can be done to another compute node, dedicated I/O node or to the PFS. Zheng et al. [20] recently showed that, just by adding one more level of checkpointing to a buddy compute node in a different rack, the probability of unrecoverable failure can be as low as 0.000977% for an MTBF of 20 years per node, 5000 nodes, checkpoint interval of 6 minutes and 1200 hours of application time. While the checkpoint time is directly proportional to the available bandwidth between the compute node and the remote destination, remote checkpoints can be overlapped with the next phase of the computation, without requiring the application to block.

Overlapping remote checkpoint results in noise which makes the application slower. Noise can be categorized into 1) communication noise caused by interconnect contention between a communication intensive application and asynchronous checkpoint data movement, 2) computation noise caused by an asynchronous checkpoint process interfering with the application (stealing CPU cycles), 3) application blocking, when there is insufficient buffer space for holding data until remote checkpoint. For reducing communication noise, prior work [20] proposed scheduling algorithms to serialize communication and remote checkpoint, or delaying the remote checkpoint until application communication is complete which can increase the remote checkpoint time. Compute noise can be mitigated by utilizing unused cores (‘helper cores’) [1], whereas, blocking can be avoided by using disk or NVM as a buffer. In this paper we focus on reducing the communication noise by using a pre-copy mechanism.

Chunk-based remote checkpoint. Our major goals of remote checkpoint are 1) reducing peak resource usage and 2) making remote checkpoint faster without affecting application performance. Based on these two goals, we propose a remote memory model where applications can allocate, access and copy NVM buffers to local as well as remote destination nodes. The novelty of our approach is that the application need not wait for a local checkpoint to complete before a remote checkpoint is initiated. Instead, chunks are ’pre-copied’ incrementally to remote nodes as soon as they are modified by the application, similar to local checkpoint pre-copy discussed earlier. By overlapping remote checkpoint with local checkpoint and compute phase (see Figure 5), the available time window for remote data transfer is increased, thereby reducing the necessary blocking time for remote checkpoints. Also, moving in granularity of chunks instead of moving all checkpoint data at once reduces peak interconnect bandwidth usage for large checkpoint data. Finally, maintaining separate versions for the most recently completed checkpoint vs. the one currently in progress, helps protect data consistency in the event of crashes.

In order to fully benefit from the memory nature of NVM devices, and with assumption of future support for DMA operations between the interconnect and NVM, the NVM-checkpoint remote pre-copy operations are designed to leverage RDMA capabilities of high-performance fabrics. Specifically, our implementation extends the widely used aggregate remote memory copy (ARMCI) library. A helper asynchronous process on each physical node is responsible for remote checkpoints. The helper process utilizes our shared NVM (similar to shared memory) support to access local checkpoint chunks and pre-copies by tracking dirty NVM chunks. As a design optimization to decouple the helper process from applications, we introduce additional APIs and system calls to query our library about a process’

Method	Description
$genid(varname)$	generate id from varname
$nvalloc(id,size,pflg)$	allocate NVM memory. pflg -whether data should be persistent
$nv2dalloc(..dim1,dim2..)$	2D Fortran allocation wrapper
$nvattach(id,src,size)$	create shadow NVM copy for exisitng DRAM memory
$nvrealloc(id,src,size)$	grow memory
$nvchkptall()$	checkpoint all persistent chunks
$nvchkptid(id)$	checkpoint specific chunks/variables

Table III
NVM Checkpoint Interfaces

NVM structure. Section V discusses this in greater detail. Further, to reduce wasteful bandwidth usage, we use the delayed pre-copy with prediction (DCPCP) mechanism for remote checkpointing. The delay time before a remote pre-copy is dependent on the remote checkpoint interval.

V. Implementation

We next discuss the implementation of the system- and application-level NVM-checkpoint components.

NVM Kernel. In our hybrid memory design model, the NVM virtual memory support is enabled by an NVM kernel manager which extends the Linux memory (DRAM) manager. We use the NVM kernel manager implementation developed in our previous work [26] and extend it with checkpointing-related support, including for pre-copy and shadow buffering.

The NVM memory manager is responsible for NVM-based paging and in-kernel process-level persistent data structures. To emulate future NVMs like PCM, the kernel manager reserves a fixed physical address range in DRAM during OS boot, and manages these pages along with persistence support. Further, a set of system calls is exposed to allocate, extend or delete NVM pages. Allocations are done using the 'nvmmap' system call similar to the brk/mmap system call for heap. We maintain a metadata structure for each process that keeps track of all NVM pages used by a process. During applications restart, the information in the metadata structure along with the system call arguments are used to load the persistent pages to the process address space. Also, since NVM utilizes the processor cache to reduce the NVM read/write latency, all data in the cache is flushed to NVM before data is marked consistent, using the Linux cache flush kernel method. All data structures/application variables that need to be checkpointed are allocated using NVM user-level interfaces (described below), and maintained on a per-process basis.

To support pre-copy, the asynchronous checkpoint process on each node, should have access to all of the process checkpoint metadata from which the checkpoint data regions can be identified. To enable this, we provide a system interface which loads the entire metadata structure to the process address space. The metadata structure is protected by locks to avoid conflicts between the application and remote checkpoint process. Next, since two versions of checkpoint data (i.e., a committed version and current uncommitted version) can reside as local checkpoint, we add multi-version support for our internal data structures. As an optimization, while our DRAM to NVM local checkpoints uses page protection techniques to identify dirty chunks, in case of remote checkpoints, to avoid frequent protection faults, we add an 'nvdirty' bit for each NVM page supported by a system call to identify dirty NVM pages of a chunk.

NVM user library. Our user library provides NVM allocation, checkpoint and restart interfaces for applications. Table III list few important interfaces. The user library consists of a) an NVM allocation component, b) shadow buffering and checkpoint component and c) a restart component.

a) *NVM allocation component:* All data structures/variables that need to be checkpointed are allocated using the 'nvmalloc' interface and referred to as chunks. Each chunk has metadata structure and a corresponding NVM data region. The allocation component extends the highly scalable Jemalloc allocator to manage allocations to NVM using the 'nvmap' interface and maintains a user level persistent data structures for each such allocation. During chunk allocation, the allocator creates a DRAM chunk and a corresponding NVM chunk and returns the DRAM pointer to application. Each chunk is identified by a unique identifier supplied by the application. All chunk metadata of a process is stored in a separate per process metadata region, not directly accessible by the application.

b) *NVM checkpoint component:* The checkpoint component is responsible for shadow buffering (i.e., moving data from DRAM to NVM), tracking chunk level modification for local and remote chunk level pre-copy, ensuring the consistency of checkpoint and also managing checkpoint versioning. When an application invokes the 'nvchkptall()' interface, all checkpoint variables are moved from DRAM to NVM. Our library uses its metadata structure to identify all chunks and moves them to NVM. All chunks are marked as committed after the library ensures that data is flushed to NVM. For enabling pre-copy, each chunk structure has two dirty bit flags, one for a local checkpoint and another for a remote checkpoint. These dirty bits are used to identify chunks that need to be pre-copied. To deal with failures, we maintain two versions for each chunk, a previously committed chunk and chunk currently being written. When a checkpoint fails, the library reverts to the committed version for recovery. When local NVM space is a constraint, only one version is maintained locally and on checkpoint failure, the application retrieves the chunk data from remote nodes.

c) *Restart component:* During restart, the applications use the same 'nvmalloc' interface with a unique identifier and the persistent flag as the argument (see Table III) to read back data from NVM. The persistent flag specifies a condition whether to read previous checkpoint data if data exists. An optional feature is the checksum capability, where after every checkpoint, a chunk data checksum is calculated and stored along with the chunk metadata. On restart, the checksum is recalculated and verified against the metadata. The restart component first checks if the checkpoint data is available/consistent and if not, fetches the data from the remote peer node. Our current restart mechanism is simplistic and our future plans will consider its in-depth analysis and possible optimizations.

Application Modifications. Our current design requires application level changes for adapting to NVM based checkpointing. Specifically, applications need to use the NVM allocation interface for variables/data structures that need to be checkpointed. Currently, our library provides Fortran and C/C++ interfaces, which are being hardened as we gain experience with more diverse applications. For example, when adding NVM support for Lammps, we noticed that it relies extensively on its custom user level memory management functionalities and modification of such applications can be tedious. Similarly, in some applications, the checkpoint size cannot be statically determined. Such applications can use our 'nvattach' interface to lazily create a checkpoint chunk by attaching the DRAM pointer with a shadow NVM pointer and subsequently use 'nvdelete' method to let the NVM library remove references to the chunk.

VI. Evaluation

The overall goal of NVM-checkpointing is to leverage future NVM present in next-generation exascale machines in reducing checkpointing overheads without compromising application reliability. To understand our ability to achieve this goal, the experimental analysis is based on the performance model described in Section III. We categorize the checkpointing overhead in three components: (1) local checkpoint time, (2) remote checkpoint overhead, and (3) overall resource utilization. Our experiments focus on the key issues affecting the performance of each of these three components and demonstrate the ability of NVM-checkpoint to reduce the checkpointing impact on application performance. Specifically, we evaluate the following:

- Reducing local checkpoint time: We evaluate the effectiveness of NVM-checkpointing to deal with NVM bandwidth limitations and to reduce local checkpoint time.
- Reducing remote checkpoint time: We measure the ability of our remote checkpoint design in improving application efficiency.
- Resource utilization: For both local and remote NVM-checkpointing we evaluate the overall system resource utilization (e.g., interconnect bandwidth, CPU, and memory) of our methods.

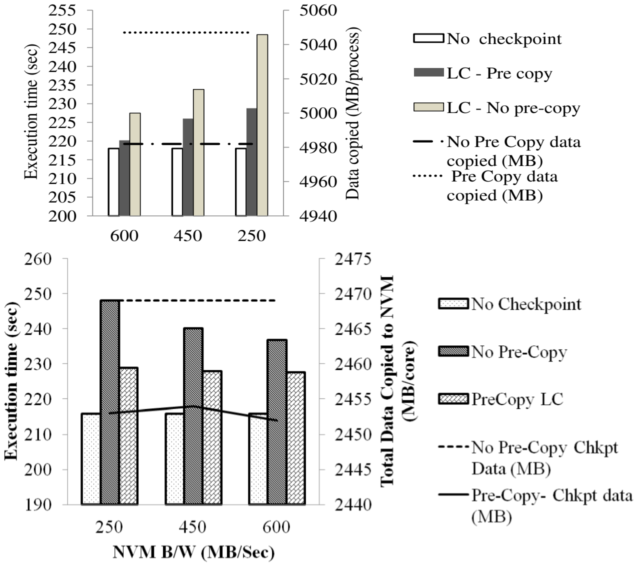

Figure 8. GTC- Local Chckpt Pre-Copy vs. No Pre-Copy

Methodology. All experiments are conducted using a 8 node cluster, with each node consisting of 12-2.8 GHz Intel Xeon cores, 48 GB memory, and 40Gbps InfiniBand. We use the mvapich2 MPI library for all applications, with one process per core. For NVM emulation, we partition half of the DRAM for NVM-related allocations managed by our NVM component. Persistence across application session is provided by locking the DRAM pages from being swapped out or freed. To emulate different NVM bandwidth, we introduce data copy delays derived using the LANL memcpy benchmark similar to [17]. We assume the NVM device bandwidth as 2 GB/sec [11] and vary the effective per core bandwidth.

Applications. In order to understand the impact of NVM-checkpoint on real-world HPC applications and their checkpointing requirements, we study the checkpointing behavior of three large scale widely used HPC applications:

1. Gyrokinetic Toroidal Code (GTC) is a 3-Dimensional Particle-In-Cell code used to study microturbulence in magnetic confinement fusion from first principles plasma theory. The checkpoint data primarily have 2D arrays representing electrons and ions. The application is highly scalable and each core can output two million particles roughly every 120 seconds resulting in 260GB of checkpoint data.

2. LAMMPS is a well known particle dynamics code that supports a wide variety of simulation techniques applicable to biology, chemistry, and material sciences and we use LAMMPS benchmarks with configurations similar to [27] and use Rhodo suite for our analysis.

3. CM1 is a three-dimensional, non-hydrostatic, non-linear, time-dependent numerical model suitable for idealized studies of atmospheric phenomena. For our experiments, we study the 3D hurricane simulation. The CM1 code is an open source Fortran code and uses similar checkpointing mechanism to GTC code.

Application	500K-1MB	10-20MB	50-100MB	above 100MB
CM1	40	0	54	4
GTC	45	9	0	45
LAMMPS	15	0	20	25

Table IV
CHUNK SIZE DISTRIBUTION RANGE (%)

NVM local checkpointing. Figures 7 and 8 compare the local checkpoint performance of a pre-copy checkpoint against a ‘no pre-copy’ case. For ‘no pre-copy’, local checkpoint and compute steps are not overlapped (started only after a compute cycle is finished). The x-axis shows different NVM bandwidth/core estimates, the left y-axis shows the application execution time and the right y-axis – the total data copied to NVM for local checkpoints. For Lammps, the experiments are done with 48 MPI processes using the RhodoSpin benchmark, and checkpoint size/process is ˜ 410 MB. We choose the RhodoSpin benchmark as it checkpoints relatively higher number of chunks that are modified across different application stages. As shown in Figure 7, even with decreasing NVM parallel bandwidth, pre-copy checkpoint adds only 6.5% overhead to application execution time, compared to the 15% in the ‘no pre-copy’ case. Overall, the pre-copy method improves perfromance by 15% compared to the RAMdisk approach. Chunks that are modified just before the checkpoint step cannot benefit from a pre-copy prediction mechanism and result in additional work. As it can be seen, the total data copied by pre-copy is slightly higher (3%). For GTC, we use the same processor count and checkpoint data size. The application shows similar benefits from using the pre-copy approach (see Figure 8). An interesting point to note is the reduction in checkpoint size for the pre-copy case. For GTC, we observe that few large chunks (variables) are modified only once (during application initiation). Similar observations can be made for LAMMPS too, though the aggregate data size of these variables is less significant. By leveraging the memory protection mechanisms in NVM-checkpoint, we can efficiently track chunk-level modifications, and avoid repeating checkpoint for unmodified chunks without more heavy-weight diff computations. The combined use of pre-copy with the reduction of checkpointing data size improves the local checkpoint performance of GTC by 10%, compared to the ‘no pre-copy’ case.

The CM1 application (not shown for brevity) shows less than 5% benefits from the pre-copy approach. To understand the reason for such variation across application, we analyze the chunk size distribution. Table IV shows the checkpoint variable chunk size distribution for the three applications. We analyze the impact of chunk sizes on pre-copy performance for a fixed checkpoint size (400 MB) and checkpoint frequency. Chunk sizes are categorized into different ranges varying from 500KB to over 100MB. In case of CM1, about 40% of the chunks are less than 500K and around 50% of chunks less than 50 MB. In case of GTC and Lammps about 50% and 30%, respectively, of the chunks are larger than 100MB. The NVM bandwidth limitation, which pre-copy attempts to alleviate, causes more significant levels of contention for large chunk sizes. This explains the observed benefits of pre-copy for Lammps and GTC, and the more modest impact of the approach on CM1, where less than 5% of the chunk exceed 100 MB.

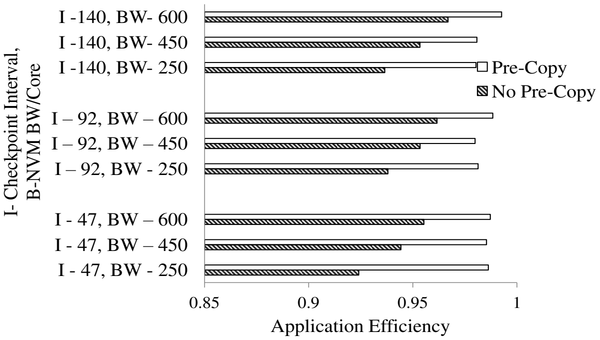

Figure 9. GTC - Efficiency with remote checkpoint

NVM remote checkpointing. We next analyze the impact of the NVM-checkpoint’s remote checkpointing mechanism on the application efficiency. We define efficiency as the ratio of ideal application run time to actual run time. The ideal runtime represents a case when there no failures and application do not checkpoint, whereas the actual time represents the application runtime with local and remote checkpoints. Recent studies have shown a large variation in estimating the MTBF for exascale systems. We use the failure rates (λ) and optimal checkpoint interval (30 to 100sec) estimated by X. Dong et al. [9], and vary the ratio of permanent (λ_{remote}) vs. transient (λ_{local}) failures. Considering our experiments scale, we set the local checkpoint frequency to 40 secs.

Figure 9 compares the remote checkpoint overheads of an asynchronous pre-copy and asynchronous ‘no pre-copy’ for GTC. In our experiments, the average checkpoint data per core is around 433MB, and around 4.7GB per node. We vary the remote checkpoint frequency from 47 to 180 seconds. The horizontal axis shows the application efficiency compared to the baseline case without any checkpointing. From the figure, we observe that even at reduced levels of NVM bandwidth, remote pre-copy checkpointing delivers significant improvements in achieving application efficiency, compared to the ‘no pre-copy’ case. With the increase in available NVM bandwidth, and at increased checkpointing intervals, NVM-checkpoint can achieve application efficiency by 0.98. Similar trends can be observed for the other applications we study. On average, across the three applications, ‘pre-copy’ based remote checkpointing adds 6.2% to the application run time, compared to 10.6% of the ‘no pre-copy’ approach, representing a reduction of nearly 40%. At increasing system scales, this can translate to substantial gains.

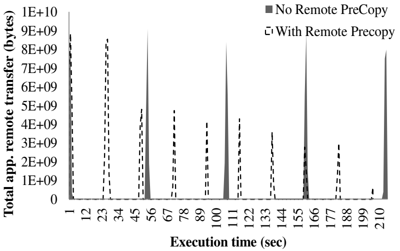

Figure 10. Lammps- Peak interconnect usage

Data/core(MB)	No Pre-Copy- Util%	Pre-Copy- Util%
370	12.85	24.48
472	13.40	25.12
588	14.82	28.31

Table V
CHECKPOINT HELPER CORE AVERAGE CPU UTILIZATION

Several studies in the past have reported substantial communication interference (close to 22% as reported by G. Zheng et al. [20]) when overlapping the asynchronous checkpoint with computation. Such interference is primarily due to higher peak bandwidth utilization of a remote asynchronous process, causing interconnect contention. In case of the 'pre-copy' approach, by increasing the time window of checkpoint, and accessing/moving checkpoint data in chunk granularity, contention is more likely to be avoided, at the cost of potential increase in total checkpointing data volume. Figure 10 compares the peak interconnect usage of the 'no pre-copy' and pre-copy-based remote checkpointing. The y-axis in the figure shows the total checkpoint data transferred across the entire application and the x-axis shows a timeline of the application execution. Clearly, 'no pre-copy' requires moving all data at once, which substantially increases the peak interconnect usage. In case of the pre-copy based approach, the peak resource usage is almost half the 'no pre-copy' case – i.e., by avoiding the need to move all data at once pre-copy substantially reduces the interconnect contention. The high peak resource usage in the initial application stages of the pre-copy approach is due to the learning phase, when our library learns about approximate remote checkpoint interval and checkpoint data transfer time. Thus, the figure also demonstrates the utility of the 'delay-based' optimizations of the pre-copy approach. Finally, the pre-copy methods require use of asynchronous pre-copy threads and additional CPU resources. We find that the average CPU utilization of the dedicated checkpointing core, executing the pre-copying operations, doubles (see Table V), however, it still remains at relatively low levels when compared to the node-wide CPU utilization – at ˜2.5%.

Summary of results. To summarize, our experiments demonstrate that by using NVM as memory, the NVM-checkpoint approach permits the use of pre-copy-based methods which (i) reduce the overheads of local checkpointing by dealing with the NVM bandwidth limitations, and (ii) reduce the overheads of remote checkpointing operations by reducing the peak bandwidth requirements of checkpoint data movements. This, in turn, improves the efficiency of multilevel checkpointing solutions, which have already been established as useful for HPC applications at increasing system scales. We believe that the potential gains in checkpointing performance will only become more pronounced at increased scales of exascale applications and systems.

VII. RELATED WORK

Multilevel checkpoints (node local and remote) have been extensively studied in various forms like diskless [28], [20], [8], [3] or local storage checkpointing. To our knowledge, Plank et al. [28], were the first to propose a diskless checkpointing in distributed applications by using additional processor and memory to replicate checkpoint data without relying on stable storage. To reduce memory usage, several alternatives, like erasure coding technique [29], compute node pairs for in-memory checkpointing [20], or use of a ramdisk-SSD hybrid approach for reducing memory consumption have been proposed. More recently, Moody et al. [8] made an extensive failure modelling of multilevel checkpoint using a Markov model and futher built a multilevel checkpoint library using ramdisk/SSD for local checkpoint. While our work complements some of these prior multilevel checkpointing efforts, our novel contribution is understanding the benefits and implications of NVMs like PCM for multilevel checkpoints. Further, unlike prior work, we show that, replacing HDDs or SSDs with NVM will not be sufficient to exploit NVM hardware capabilities and by using NVM as a virtual memory, checkpoint performance can be substantially improved.

Most prior work on NVM checkpointing has focussed on transparent checkpoint and architectural enhancements. Dong et al. [21], [9] proposed a local and global NVM based transparent checkpoint method. To reduce DRAM to NVM data movement cost and bandwidth, they proposed a stacked 3D DRAM-PCM design. Such techniques are complementary, and at future system scales, NVM-checkpoints can further benefit from both software pre-copy methods and architectural support for improving the effective NVM bandwidth. In addition, we show that application-initiated checkpoints can be impacted by NVM bandwidth limitations and use virtual memory paging techniques to solve them. Finally, Li et al. [19] studied the opportunities of directly exposing NVMs for simulation and checkpointing and concluded that, while some applications may not be affected by slow writes and bandwidth of NVM, applications with high write-to-read ratio (such as checkpoint operations) may suffer slowdown upto 25%. Our initial analysis using binary instrumentation showed similar results, and hence we use a shadow buffering mechanism.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we first analyze the role of future NVMs like PCM and Memristors in reducing the checkpoint cost, critical for application speedup. We specifically adapt the state-of-art "multilevel" checkpointing approach and discuss the importance of a hybrid, i.e., node local and remote checkpoint approach. We next discuss the need for treating NVM as virtual memory to deal with high write latency cost and bandwidth limitations of NVM. We propose three novel pre-copy techniques supported by a shadow buffering mechanism to overcome such hardware limitations. Further, we extend our NVM system design by enabling remote memory operations supported by our pre-copy schemes, and demonstrate reduction in non-blocking remote checkpoint overhead and in peak interconnect utilization.

To reduce checkpoint cost, our entire multilevel checkpoint design has been influenced by using NVMs as a slow virtual memory. While our current checkpoint mechanism has been designed for application initiated checkpoints, we believe our virtual memory based design can be generalized to transparent checkpoint mechanisms in the future. Further, our design requires few application changes like modifying the heap allocation calls and the checkpoint commit interface, but it may not be easy to modify all legacy applications. We plan to look into compiler techniques that can reduce the application changes. Currently our NVM library optimizes the local and remote checkpointing, but an efficient restart/recovery mechanism is important for overall application speedup. Considering the fact that, read speeds of NVMs are comparable to DRAM, we plan to further optimize our recovery mechanism as well.

Acknowledgment. This research was partially supported by the Department of Energy under Award Number DE -SC0005026. See http://www.hpl.hp.com/DoE-Disclaimer.html for additional information.

References

[1] B. Keren et al., "Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead," 2008.

[2] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Booktitle*, vol. 78, 2007.

[3] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: high performance fault tolerance interface for hybrid systems," in *SC '11*.

[4] H. Abbasi et al., "Datastager: scalable data staging services for petascale applications," in *HPDC '09*.

[5] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, and S. Klasky, et al., "PreDatA - Preparatory Data Analytics on Peta-Scale Machines," in *IPDPS, 10*.

[6] S. Kannan, A. Gavrilovska, K. Schwan, D. Milojicic, and V. Talwar, "Using active nvram for i/o staging," in *PDAC '11*.

[7] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. de Supinski, and R. Eigenmann, "Mcrengine - a scalable checkpointing system using data-aware aggregation and compression," in *SC '12*.

[8] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC '10*.

[9] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems," in *SC '09*.

[10] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro, 2005*.

[11] http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf.

[12] J. Bent, G. Gibson, and G. Grider et al., "Plfs: a checkpoint filesystem for parallel applications," in *SC '09*.

[13] http://en.wikipedia.org/wiki/Memristor.

[14] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *ISCA '09*.

[15] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *MICRO '10*.

[16] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating system support for nvm+dram hybrid main memory," in *HotOS'09*.

[17] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: lightweight persistent memory," in *ASPLOS '11*.

[18] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories."

[19] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu, "Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientic applications," in *IPDPS '12*.

[20] G. Zheng, L. Shi, and L. V. Kale, "Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi," in *CLUSTER '04*.

[21] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Hybrid checkpointing using emerging nonvolatile memories for future exascale systems," *ACM Trans. Archit. Code Optim.*

[22] J. Borrill, J. Carter, L. Oliker, and D. Skinner, "Integrated performance monitoring of a cosmology application on leading hec platforms," in *ICPP '05*.

[23] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, and D. Burger, "Better i/o through byte-addressable, persistent memory," in *SOSP '09*.

[24] A. Badam and V. S. Pai, "Ssdalloc: hybrid ssd/ram memory management made easy," in *NSDI'11*.

[25] "LANL Memcpy Benchmark," http://tinyurl.com/92nvgjd.

[26] S. Kannan, A. Gavrilovska, and K. Schwan, "Rich client services using persistent memory," nVMW'12.

[27] "Lammps," http://www.sandia.gov/benchmarks/.

[28] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Trans. Parallel Distrib. Syst.*

[29] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage," in *FAST '09*.