Leveraging Near Data Processing for High-Performance Checkpoint/Restart

Abhinav Agrawal
North Carolina State University
aragraw2@ncsu.edu

Gabriel H. Loh
Advanced Micro Devices, Inc.
Gabriel.Loh@amd.com

James Tuck
North Carolina State University
jtuck@ncsu.edu

ABSTRACT

With the increasing size of HPC systems, the system mean time to interrupt will decrease. This requires checkpoints to be stored in a smaller time when using checkpoint/restart (C/R) for mitigation. Multilevel checkpointing improves C/R efficiency by saving most checkpoints to fast compute-node local storage. But it incurs a high cost for writing a few checkpoints to slow global-I/O. We show that leveraging NDP to offload writing of checkpoints to global-I/O improves C/R efficiency. We explore additional opportunities using NDP to further reduce C/R overhead and evaluate checkpoint compression using NDP as a starting point.

We evaluate the performance of our novel application of NDP for C/R and compare it to existing C/R optimizations. Our evaluation for a projected exascale system using multilevel checkpointing shows that with NDP, the host processor is able to increase its efficiency on an average from 51% to 78% (i.e., a >50% speedup in performance).

ACM Reference Format:
Abhinav Agrawal, Gabriel H. Loh, and James Tuck. 2017. Leveraging Near Data Processing for High-Performance Checkpoint/Restart. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 12 pages.
https://doi.org/10.1145/3126908.3126918

1 INTRODUCTION

Increasing size and complexity of high-performance computing (HPC) systems to achieve exascale performance is projected to cause a decrease in the system mean time to interrupt (MTTI). Checkpoint/restart (C/R) is a widely used mechanism to deal with failures in HPC systems. It involves saving the state of the application required to resume the application to stable storage at certain intervals. In case of a failure or interrupt, the application's execution resumes from the most recent checkpoint (saved state). In the absence of C/R, a failure would lead to the application having to restart from the beginning, losing all completed work. However, C/R mechanisms add performance overhead due to the time spent saving the checkpoint state, restoring from the saved state, and re-running lost work (work performed since the most recent checkpoint). The efficiency or availability of exascale systems with C/R is projected to be around 50% [1].[1] C/R efficiency or progress rate is the ratio of the time it takes to run an application in the absence of failures and C/R overhead to the time it takes to perform the task in the presence of such overheads. Under some simplifying assumptions, progress rate can be approximated as a function of ratio of MTTI(M) and time to save checkpoint(δ) [2, 3].[2] Figure 1 illustrates this function.

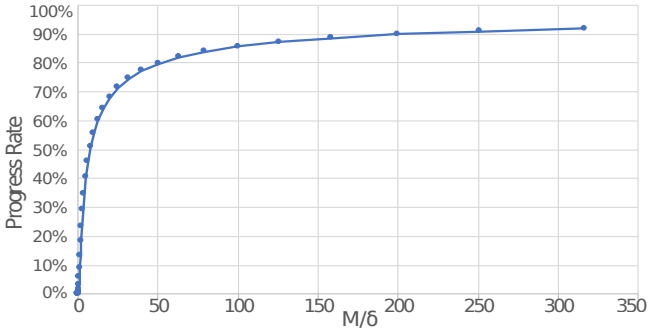


Figure 1: Progress rate of a system with C/R as a function of M/δ. Increasing value of M/δ leads to higher progress rate.

For exascale systems with checkpoint/restart, this ratio of M/δ decreases due to two factors. On one hand, the increasing number of compute nodes needed to reach exaflops performance would lead to a decrease in the system MTTI because the MTTI of a single compute node is not improving [4]. On the other hand, exascale systems are expected to have larger physical memory capacities and are expected to be able to run applications with larger problem sizes. This will lead to larger application state that needs checkpointing. Without a proportional increase in checkpoint commit write to storage, the time to save checkpoint(δ) will increase. The combination of these two factors leads to a reduction in the ratio of M/δ and thus the progress rate.

I/O or storage hierarchy improves the access time to storage by means of having fast intermediate levels between compute and disk based global I/O. These intermediate levels referred to as burst buffers can be in the form of flash-based solid state drives (SSDs). They provide a lower overhead storage site to stage data by the compute before it is drained to slower disk-based storage in I/O node [5]. While this alleviates the performance overhead by reducing the access time to storage (including for C/R), moving large data between storage in different levels of hierarchy is not energy efficient. Near data processing (NDP) is effective in reducing the amount of data movement in many applications by performing computations closer to data. Offloading some application's computations from the host processors to NDP has shown to improve performance and energy efficiency, especially for data intensive applications [6–11]. NDP (or active storage) can potentially address key challenges in

[1] *Progress rate* and *efficiency* are used interchangeably in this work.

[2] While Daly's work [2] provides an equation to calculate the optimal checkpoint interval given MTTI(M) and checkpoint commit time (δ), *Quantifying Checkpoint Efficiency*[3] provides an equation that calculate checkpoint efficiency given M and δ. Checkpoint restore time is assumed to be same as commit time.

SC17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5114-0/17/11…$15.00
https://doi.org/10.1145/3126908.3126918

the areas such as scalability, performance, and reliability for I/O systems in exascale computing [12]. In this work we show that NDP can be leveraged to improve C/R performance.[3]

A number of optimizations and mechanisms [13–27] have been proposed to reduce C/R overhead. While a mechanism like partial redundancy [26] decreases effective MTTI to reduce C/R overhead, many mechanisms aim to either increase the effective checkpoint commit bandwidth [13–19] or reduce the checkpoint size [20–24]. Increasing the checkpoint commit bandwidth decreases the value of δ, thus improving the progress rate. Many of the techniques that reduce the effective checkpoint commit time take advantage of the storage hierarchy. Multilevel checkpointing is a clear example of C/R mechanism that exploits storage hierarchy to improve C/R cost.

Multilevel checkpointing [13, 14] involves writing frequent checkpoints to compute node local storage, while writing occasional checkpoints to global I/O based storage. However, these occasional checkpoints to global I/O typically have a high overhead. Writing a checkpoint out to global I/O in a conventional multilevel checkpointing system requires the host processor to read the checkpoint data from main memory, and then send the data over the network to the remote storage, which requires the host to execute all of the code associated with running the full network stack (e.g., TCP/IP). This can be a particularly slow process as the checkpointing process is typically bottlenecked on the slower I/O (disk) bandwidth at the shared remote I/O nodes. While this is happening, the host processor is generally not available to perform the "useful" computations of the main application. While bandwidth of the local storage in compute node scales, that of global I/O based storage does not scale with the increasing size of application. Moody *et al*. in [13, 14] show that with increasing failure rate and increasing time to save checkpoint to global I/O, the progress rate of a system with multilevel checkpointing decreases, although slower than the decrease for single level checkpointing.

In this work we target this overhead associated with checkpointing to global I/O. To improve C/R performance, multilevel checkpointing utilizes one feature of hierarchical storage (i.e., storage with different speeds and availability at different levels in hierarchy). We propose leveraging additional feature of hierarchical storage - the likely presence of NDP or active storage in (future) HPC systems. Using NDP (i.e., compute capabilities coupled to compute-node local storage) allows the host processor to quickly write checkpoints to the node-local storage and resume execution; NDP can then handle the slower process of sending the checkpoint(s) to global I/O off of the main application's critical path.

NDP can be leveraged for additional optimizations that improve C/R performance. As a starting point, we explore the benefits of adding compression capabilities to our NDP-based checkpointing scheme, as this can reduce network bandwidth requirement for sending checkpoints out to I/O (thereby reducing network contention for the main application's communication needs), and it can also help improve performance by speeding up checkpoint restoration (which is primarily limited by how fast checkpoints can be retrieved from the I/O nodes' disks). While checkpoint compression is not new, the exploitation of an NDP architecture to offload it from the host processor is a new twist: past approaches tolerated higher host-side processing costs because the compression reduced the I/O cost sufficiently to make it a net win, whereas our approach can get the benefits of compression without the host-side overheads.

We start by projecting an exascale system by scaling a petascale system based on technology trends described in literature. Next we review two of the existing C/R optimizations - multilevel checkpointing and checkpoint compression, in the context of our projected exascale system and discuss how they can be combined. Next we present the key ideas of our proposed approach using NDP and describe its operational details. To help determine NDP configuration for the proposed C/R operations including compression, we perform a compression study. This is followed by an evaluation of our proposed NDP approach for C/R.

We make the following contributions:

- We evaluate multilevel checkpoint in the context of our projected exascale system, highlighting the overhead associated with checkpointing to global I/O. We show that adding checkpoint compression to multilevel checkpointing can reduce this overhead.
- We describe the operational details of the checkpoint/restart mechanism using NDP as well as the compute-node's hardware organization to implement such a mechanism. We evaluate checkpoint compression using NDP as a starting point for exploring additional optimizations that can be performed by NDP.
- We perform a detailed evaluation of multilevel checkpointing with NDP support. With our proposed NDP approach for offloading I/O management and compression, the host processor is able to increase its progress rate from 51% to 78% (i.e., a more than 50% speedup in the application performance).

2 BACKGROUND AND RELATED WORK

Projections show that the system MTTI of exascale machines could be in the range of minutes to tens of minutes [1, 12, 28]. The total system memory is projected to be in the range of tens to hundreds of petabytes [1, 12, 28]. Due to this dual factor of decreasing system MTTI and increasing δ, the progress rate of systems with C/R is decreasing exponentially as shown in Figure 1. If one were to achieve a progress rate of 90% with checkpoint/restart solely by improving checkpoint commit time, based on Daly's analytical solution [2], the checkpoint commit time (and restore time) would need to be in the order of seconds. Assuming that a large part of the system physical memory may need to be checkpointed, the required checkpoint read/write bandwidth would increase exponentially for increasing size of HPC systems. Hierarchical I/O consisting of burst buffers [5] as well as file systems optimized for C/R [18, 19] help with the high I/O bandwidth requirement. Many prior works show the benefits of adding NDP to different levels of storage hierarchies such as burst buffer for HPC systems or fast SSD storage [6–11]. While in this work we use NDP in context of processing coupled to local NVM based storage, NDP is also used in the context of adding compute capability or specialized logic in the main memory [29, 30]. A review of prior work on NDP in both contexts can be found in [31].

[3] NDP could be in the context of main memory (like DRAM) or in the context of storage (like SSDs). In this work NDP refers to compute capabilities coupled to compute-node local storage (which would likely be flash SSDs or other NVM based storage due to high speed requirement).

To overcome the limitations of traditional I/O based C/R, many proposed solutions [13, 15–17, 25] store some or all of the checkpoint data closer to compute nodes. These schemes are some variation of multilevel checkpoint scheme [13] where checkpoints are frequently saved to faster compute node local storage and less frequent checkpoints are stored to slower global I/O. The compute node local memory could be DRAM [13, 17] or non-volatile memory NVM [15, 16]. While the bandwidth to local storage is expected to scale with checkpoint size, the bandwidth to global I/O is not expected to scale [32]. Moody *et al*. [13] shows that while multilevel checkpointing performs better than single level checkpointing for increasing failure rate and increasing time to save checkpoints to I/O, its performance still reduces. Therefore the global I/O component of multilevel checkpointing will increase with increasing cost of checkpointing to I/O.

On the other hand, solutions like *checkpoint compression* [20, 21], *incremental checkpointing* [22], and *checkpoint deduplication* [23, 24], reduce the checkpoint overhead by reducing the amount of data that needs to be saved during checkpointing. These solutions have been shown [20] to be cost effective ways of reducing C/R overhead.

3 SCALING STUDY

3.1 Exascale System Projection

Similar to prior projections in [15, 28], we project an exascale system that we use to study existing C/R mechanisms and help guide the design of our proposed solution. The projection is made by scaling an existing petascale system to exaflops performance. The assumptions made when scaling are based on cited technology trends. Furthermore, in our projections, we generally err on the side of more optimistic or lower C/R overheads. The intent is to show that even with these optimistic assumptions, the overhead of existing C/R mechanisms on exascale systems would be high, resulting in a lower progress rate. One implication of this preference is that we project a conservative increase in physical memory size and, consequently, the checkpoint size. Similarly compared to other projections a conservative increase in failure rate is projected.

In this study, we scale the Titan Cray XK7 system [33], a petascale system, to exaflops performance. Titan has 18,688 compute nodes each consisting of a 16-core AMD Opteron processor coupled with additional GPU acceleration. Each node has a 38 GB of memory (2GB per CPU core plus the GPU's 6GB). Each node has a theoretical peak performance of 1.44 teraflops with a theoretical system peak performance of 27 petaflops. A ~37x factor increase is required to reach exaflops performance. This can be accomplished by a combination of an increase in performance per compute node and an increase in the number of compute nodes. We assume that the performance of a single compute node can scale to 10 teraflops [34], a ~7x increase compared to Titan's per-node performance. We assume a uniform 7x increase in both CPU and GPU performance. For the CPU, the performance increase is assumed to be achieved by a combination of a 75% increase in performance per core and an increase in core count from 16 to 64. If the ratio of 2 GB/core is maintained, the memory for the CPU would increase to 128 GB. We conservatively assume that the memory of the GPU is doubled to 12GB (and not increased 7x, proportional to performance). The total memory for the node would be 140 GB. This is a conservative estimate compared to projections made in past work [15]. With a 7x increase in the compute node's performance, the remaining increase in performance comes from a 5.3x increase in node count (37x/3x). This leads to 100,000 compute nodes, which at 10 teraflops each, provide a system peak performance of 1 exaflops. With 100K compute nodes, the total memory of the system would be 14 PB, again a conservative projection compared to other projections [12, 15, 28]. The aggregated data bandwidth of Titan to its file system is 1000 GB/s. We project this to increase to 10 TB/s, a conservative projection than one provided in [35].

Parameter	Titan Cray XK7	Exascale Projection	Factor change
Node Count	18,688	100,000	5.35x
System Peak	27 petaflops	1 exaflops	37x
Node Peak	1.44 teraflops	10 teraflops	7x
System Memory	710 TB	14 PB	19.72x
Node Memory	38 GB	140 GB	3.68x
Interconnect BW	20 GB/s	50 GB/s [28]	2.5x
I/O Bandwidth	1000 GB/s	10 TB/s	10x
System MTTI	160 minutes[4]	30 minutes	(1/5.33)x

Table 1: Exascale system projection scaled from the Titan Cray XK7

3.2 MTTI Projection

We project the *system MTTI* based on previously observed or projected *node MTTI* and scale it for the compute node count of our projected exascale system. A study by Schroeder and Gibson [4] on petascale systems introduced in 2002-2004 showed ~0.2 failures per socket per year. This is equivalent to a 5 year mean time to failure (MTTF) per socket. Similar to previous work [20, 36], we assume a node/socket MTTF of 5 years. This results in a system MTTF of ~26.28 minutes with 100K nodes. We assume each failure leads to an interrupt requiring recovery using checkpointed application state, and thus system MTTI would also be ~26.28 minutes. For the sake of simplicity we make an optimistic assumption of system MTTI being 30 minutes, which falls in the range projected in previous work [28]. Key parameters of our projected exascale system are listed in Table 1.

3.3 Checkpoint Commit Time and Frequency

For basic C/R without optimization, to achieve a progress rate of 90%, we calculate that the checkpoint commit time and restore time should be ~1/200 the system MTTI (also see Figure 1) and the checkpoint period should be ~1/10 the system MTTI.[5] Therefore checkpoint commit time should be 9 seconds and the checkpoint period should be 3 minutes for a system MTTI of 30 minutes. In this work we assume 80% of the physical memory needs to be checkpointed (i.e., 112 GB per compute node or 11.2 PB for the overall system). This would require a checkpoint commit rate of (112/9) ~12.44 GB/s per compute node or 1.244 PB/s for the system. The 1.244 PB/s far outpaces the projected 10 TB/s of global I/O bandwidth, thus requiring additional C/R optimizations.

3.4 Multilevel Checkpointing

Many optimizations to C/R are based on multilevel checkpointing (surveyed in section 2). As mentioned previously, if all checkpoints

[4]Prior work [25] reports 9 failures per day for Titan, which converts to failure every 160 minutes.

[5]We use a 90% target progress rate throughout the paper when calculating or projecting parameters using Daly's equation.

are saved and restored from just one level (e.g., compute-node local storage), a progress rate of 90% would require 12.44 GB/s bandwidth to that storage. While this bandwidth can be achieved when storing checkpoints in DRAM, DRAM does not provide resiliency against a variety of failure modes (temporary loss of power to the DRAM) that non-volatile storage can handle. Future HPC systems are expected to provide some local storage in the form of NVM [16]. For instance, multiple banks of NVM express (NVMe) solid state drives (SSDs) can be added to each compute node to achieve the required bandwidth. For example, a recently introduced SSD, the Seagate Nytro XP7200 drive, has a sequential write speed of 3600 MB/s [37]. Four such drives in parallel would provide 14.4 GB/s of write bandwidth.

In multilevel checkpointing, in addition to storing the checkpoint data of a compute node to just the same node's local storage (referred to as *local*-level), some checkpoints are redundantly stored to multiple compute node's storage (referred to as *partner*-level) or stored to global I/O (referred to as *I/O*-level). This allows failures to recover from *partner* or *I/O* when they cannot recover from *local*. Even with the assumption that one can achieve a checkpoint read/write time of 9 seconds for *local*/*partner* level, the overall efficiency is still hurt due to the overhead to save (a few) checkpoints to global I/O. Based on our projection of I/O bandwidth, the effective per node bandwidth to global I/O is 100 MB/s. It will take ~18.67 minutes to save checkpoints to I/O. Even if checkpoints are only rarely stored remotely to amortize the checkpointing overhead, this still results in higher recovery costs because the checkpoint is more likely to have been taken further in the past, causing the application to have to redo more work (i.e., the work lost since the last checkpoint and the point of failure). We show these competing effects in the section 6.2 in Figure 4.

Moody *et al*. [13] observed that 85% of the failures could be recovered from *local*-level + *partner*-level and only 15% needed recovery from *I/O*-level. They further state that this can be further improved so that only ~4% of the failures need recovery from *I/O*-level. For multilevel checkpointing, we observe (see section 6.4 and figure 7 for more details) that just 4% recoveries from *I/O*-level account for 19% of application run-time. Almost all of this time is spent re-running lost work. On the other hand, 17% of application run-time is consumed for the 96% recoveries from *local*+*partner*-level. **Therefore, just 4% recoveries from *I/O*-level checkpoints account for a big component of the overhead for multilevel checkpointing.**

3.5 Multilevel Checkpointing with Compression

Checkpoint compression [20] reduces the size of checkpoints, thereby reducing the checkpoint commit and restore time. Previous work has shown that checkpoints could be compressed to as little as 10% [38] of their original size using utilities such as pbzip2. But a reduction in checkpoint size alone does not result in a decrease in C/R overhead. Checkpoint compression only helps when the time it takes to compress and write a compressed checkpoint is smaller than the time it takes to write an uncompressed checkpoint. The time to write a checkpoint varies for the different levels in multilevel checkpointing. We derived a value of 18.67 minutes to write checkpoints (of size 112GB) to *I/O*-level. Assuming that writing of checkpoint data can be overlapped with compression of checkpoint data, any compression rate exceeding 100 MB/s (112-GB/18.67-minutes) per compute node will help reduce the time to save a checkpoint to I/O. For example, we can get a rate of 640 MB/s using 64 compression threads (on 64 CPU cores of our projected compute-node) with a per thread rate of 10 MB/s which is easily achievable based on the work in [20] and our analysis in section 5.2.

On the other hand for checkpoints saved to compute-node local storage (*local*/*partner*-level), in section 3.4, we consider a checkpoint commit time of 9 seconds. To get the benefits of compression for these levels, one would need an aggregated compression rate of 12.44 GB/s (112-GB/9-seconds). This is not achievable based on the compression rates observed in [20] even with 64 threads. There are multiple means to achieve higher compression rate (listed in [39]), for example by using hardware accelerators like FPGAs/ASICs. But, in this work we do not compress data when writing to compute-node local storage, and only compress data being stored to *I/O*-level. Multilevel checkpointing with compression, reduces the overhead component associated with *I/O*-level. See section 6.4 for details. We observe that 4% recoveries from *I/O*-level now account for just 10% of the application run-time against the 19% observed in case of multilevel checkpointing without compression. However the overhead associated with I/O is still a significant fraction of total execution time and our proposed NDP approach aims to reduce this further.

4 NDP FOR CHECKPOINTING

To eliminate the overhead of checkpoints to global I/O, we propose utilizing data processing capabilities coupled to the compute node's local storage or NDP. NDP in each compute node offloads the bulk of the checkpoint data management operations from the primary host CPU cores, thereby allowing the host to spend more time on useful application execution and less on checkpoint overheads.

4.1 Compute Node with NDP

Figure 2 shows the hardware organization of a compute node that can support our proposed NDP approach for C/R. In such an organization, compute capabilities are coupled to the fast node local NVM based storage.[6] The processor could be leveraged from the existing controllers for the NVM (e.g., FTL controllers in flash-based systems [40]), off-the-shelf standard processors (e.g., x86, ARM), or custom hardware. We assume the use of point-to-point interconnects (e.g., QPI, HyperTransport[41], CCIX[42], Gen-Z[43], OpenCAPI[44]); these are shown as thick black lines in the figure which can be used for the different system components to communicate. The NDP's processor communicates with the node's network interface controller (NIC) through these links.[7] The operational details of C/R with NDP are described below.

4.2 Operation of Multilevel Checkpointing with NDP

The timeline of the operational steps of two-level checkpointing is shown in Figure 3. For reference, Figure 3a shows a baseline scenario for multilevel checkpointing without the benefit of NDP. The host CPU must periodically stop to write a checkpoint to its local NVM (*local*-level), which occurs relatively quickly due to the

[6] All references to NVM in this work refer to NVM technology based storage like flash SSDs

[7] While the figure shows a direct connection from the NDP to the NIC, the NDP could also route messages through the host processor's external interconnect/fabric controller. So long as communication between the NDP and the NIC can occur without interrupting the host CPU cores, the NDP can offload writing checkpoints to *I/O*-level.

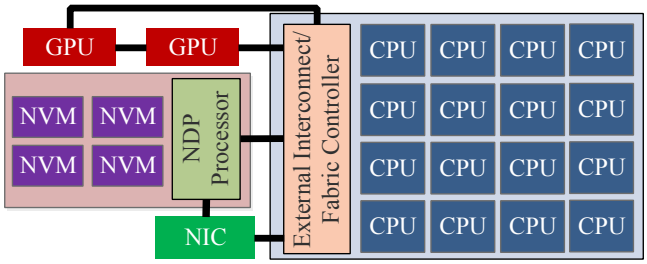
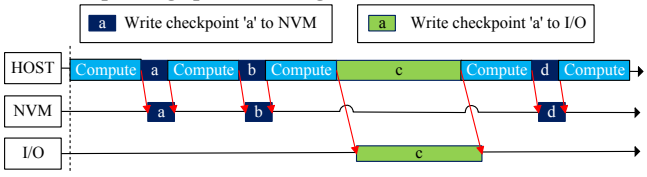
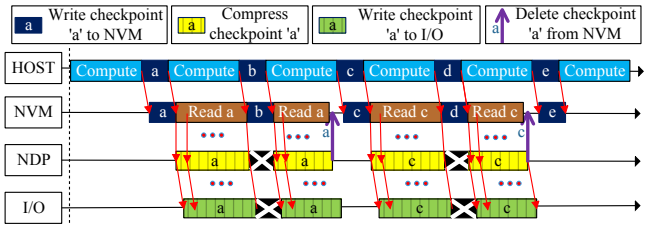


Figure 2: Hardware organization of a compute node to support multi-level checkpointing optimized using NDP.



(a) Time-line of two level checkpointing without NDP.



(b) Time-line of two level checkpointing with NDP.

Figure 3: Time-line of multilevel checkpointing with and without NDP. 'HOST': Primary processing of compute node + DRAM; 'NVM': Compute node local storage; 'I/O': I/O nodes based storage or global I/O.

high write NVM bandwidth (compared to global I/O). Note that the application must completely pause because continued execution could result in modifications to memory resulting in an inconsistent checkpoint. To protect against node-level failures, the checkpoints still must occasionally be written out to global I/O (e.g., - checkpoint 'c' in the figure), but in a conventional system, the network and remote file system operations are handled by the main CPU cores. The main application stalls until this global I/O-limited process completes. Due to the long latency to write to global I/O (*I/O*-level), the system sends only a subset of checkpoints to I/O, which increases re-execution costs. This is because the system will on average have to roll-back further and re-execute more lost work when recovering from *I/O*-level. Checkpointing operations using NDP are described next.

4.2.1 Writing Checkpoints to Local NVM. In multilevel checkpointing with NDP support, all checkpoints are initially written to the local NVM. Figure 3b shows five checkpoints (a - e) written to local NVM. This operation is in the critical path of application execution as execution is paused on the host processors (coordinated checkpoint) during this operation. To ensure that this operation is completed in the least amount of time, all the bandwidth of the NVM is made available for this write operation. Any operation being done by the NDP that uses NVM bandwidth is paused during this time. This is shown in Figure 3 where processing of checkpoint 'a' by NDP is paused while the host writes 'b' to NVM. The NVM capacity is organized as a circular buffer where each checkpoint is written in a FIFO manner.

As an example, consider the transparent coordinated C/R functionality of OpenMPI using integrated Berkeley Lab Checkpoint/Restart (BLCR) [45] library support. This tool creates system-level checkpoints. It creates a process context file for each MPI process (rank) of the application. All of the context files are stored in a folder at a logical address specified in the OpenMPI configuration. For a system with local NVM storage, the implementation of such a tool can be modified to save the checkpoint files to the local NVM instead of a shared location such as remote I/O nodes. BLCR also adds metadata that contains the process ID of the parent application process, the MPI process ID, and a unique checkpoint ID for each checkpoint of the application. This metadata will be used to keep track of the latest checkpoint and its location for each application.

4.2.2 Writing Checkpoints to Global I/O. Once a checkpoint has been written to local NVM, the host processor notifies the NDP. The notification can be implemented with any standard mechanism such as signals, memory-mapped doorbell registers, or a shared-memory location monitored by the NDP. If the NDP determines that the checkpoint should be saved to remote storage, then it locks the checkpoint (to prevent it being over-written by a future checkpoint writing operation) and then sends the checkpoint contents to the remote storage. The NDP must be able to operate the relevant system code for running the network stack (e.g., TCP/IP) and other code necessary for interfacing with the remote file-system. Upon completion of writing the checkpoint to remote I/O, the corresponding NVM capacity is unlocked (represented by a vertical arrow to delete checkpoint 'a' and 'c' in Figure 3b) which allows it to be reused for saving new checkpoints.

The operation of checkpoint compression and sending it to remote storage over interconnect network can be serialized or overlapped. To do it serially would involve first compressing all the checkpoint data and writing it to NVM. Next initiating a remote DMA (direct memory access) transaction to copy the checkpoint data from node local NVM to remote storage over the network though the network interface card (NIC). This would mean that the time to complete saving checkpoints to remote storage would be a sum of time to compress checkpoint data and time to write the compressed checkpoint data. To mitigate this, we propose overlapping the checkpoint compression and its transfer to remote storage. This overlapping compression and write operations are shown in Figure 3. Instead of completing the compression of all checkpoint data before initiating a DMA transaction, we propose multiple such transactions on small blocks of compressed checkpoint data. The compressed stream of checkpoint data (of some block size) is written to the node's network interface card (NIC) buffer. If the NIC buffer gets full due to conflicting network traffic, checkpoint compression can either be paused till additional space is available or the data could be spilled to NVM.

4.2.3 Recovering from a Failure. When there is a failure, the system must determine a rollback point for the overall parallel application. Each compute node first attempts to recover the application's checkpointed state from the local NVM, as this would be the fastest. If the local NVM does not contain the designated checkpoint, then recovery must be performed from global I/O. In either case, the checkpointed state is written directly to the host processor's main memory. In the case that recovery occurs from global I/O, it is possible that the NDP is also in the middle of writing some other checkpoint to I/O. In such a scenario, the NDP should pause the

checkpoint-to-storage operation to minimize network contention which should help avoid slowing down the host processor's recovery operation. Recovery scenarios are discussed in more details at the end of the next section 4.3.

4.3 NDP for Checkpoint Data Compression

While the primary task of NDP is to offload the task of checkpointing to *I/O*-level from host CPU processors, it can be used for additional optimization to further improve C/R performance. As a starting point we explore checkpoint compression using NDP.

Prior proposals to compress checkpoints faced the challenge that any savings in the latency to send a smaller checkpoint to remote I/O had to more than offset the performance cost of compressing the checkpoint. However, once checkpoint management has been offloaded to the NDP, we now have the opportunity to offload compression activities as well. After an uncompressed checkpoint has been saved to NVM, the NDP starts compressing the checkpoint. We partition the NVM into two separate regions. The first is a circular buffer that holds uncompressed checkpoints, and the second is a circular buffer that holds compressed checkpoints. The NDP simply reads a checkpoint from the head of the first buffer, compresses the data, and writes it to the tail of the second buffer. When saving a checkpoint to remote I/O, the NDP reads from the second buffer containing the compressed checkpoints.

Compression provides two key benefits. The first benefit is that the (compressed) checkpoint size is smaller. Because the bandwidth of global I/O is limited, a smaller checkpoint size enables the remote-write operation to complete faster. This can either allow the system to write checkpoint to *I/O*-level more frequently, or reduce the cost of the I/O system by decreasing the peak bandwidth supported. The second benefit is that on recovery, the smaller compressed checkpoint can be retrieved in less time.

Based on the compression operations described above, there are two possible scenarios for recovery. The first is that the designated checkpoint to recover from is still available in the first circular buffer. In this case, the checkpoint is uncompressed and can simply be copied back to the host processor's main memory.

In the second case, the checkpoint is located in the I/O nodes. The naive solution would be to first stream the checkpoint to the local NVM, and then decompress the checkpoint from there back to the host's main memory. This unfortunately serializes both the checkpoint retrieval latency and the decompression latency. Instead, we stream the compressed checkpoint from remote I/O directly back to the host processor, and the host processor handles the decompression. The process is pipelined out so that each page (or other sized block) of compressed data can be sent to a different core and the decompression of different pages can be processed in a concurrent manner. This overlaps the read from the remote I/O with the decompression, which reduces the overall recovery to only about however long it takes to retrieve the compressed checkpoint from the remote I/O. Due to the likely higher processing capability of host processors compared to NDP, performing decompression on the host will lead to a higher decompression speed compared to the speed achieved using NDP. However, if the remote I/O bandwidth is low enough that performing decompression on the NDP processor does not become a bottleneck, then the host processor cores can be kept in a low power mode throughout the checkpoint restoration process, while using NDP for decompression.

4.4 NDP Performance Requirements

The primary task of NDP is to write checkpoints to global I/O without interrupting the host processors of the compute node. This operation can be performed by leveraging the FTL controllers in flash SSDs. But we also explore additional optimization by using NDP for checkpoint compression. Compression would only be useful if it reduces the time to save the checkpoint. Therefore, the compression rate should be greater than the write bandwidth to I/O to get some benefits. This is with the assumption that writing to I/O and compression is overlapped. This gives us the lower limit for compression rate that NDP needs to achieve. For our projected exascale system, the compression rate should be greater than 100 MB/s to obtain some advantage from compression using NDP.

Recall that writing checkpoints to I/O using NDP does not stop application execution on the host processor. Thus, writing checkpoints using NDP more frequently does not add performance overhead. On the other hand with increasing checkpointing frequency to I/O, there would be reduction in amount of unsaved computations between checkpoints to I/O. So, in case of a failure that needs recovery from *I/O*-level, the overhead associated with rerunning lost work would reduce with increasing checkpointing frequency to I/O. The rate at which checkpoint is saved to I/O is bounded by both the compression rate of NDP as well the write bandwidth to I/O. Therefore, NDP generating compressed data at a rate faster than the rate at which it can be written to I/O would not help. The compression rate that would saturate the I/O write bandwidth can be calculated as: Compression_rate = (Uncompressed_size/Compressed_size) * IO_bandwidth. Therefore, NDP should be designed to achieve a compression rate no more than the value obtained using the above equation. To select a checkpoint utility for NDP, we perform a compression study on checkpoint data, to obtain the compression factors and compression speeds for multiple compression utilities.

5 COMPRESSION STUDY

We perform the compression study using methodology similar to the one used in [20, 21]. The compression study is performed on a 64-bit Intel Core i7-4770HQ CPU running at 2.2 GHz, with 1600 MHz 16 GB DDR3 memory, running MacOS. This system has a high bandwidth PCIe-based SSD. This would be a more accurate proxy for compression speed expected on NDP compared to a study on a system with disk based storage. This is because NDP is coupled to high bandwidth NVM storage and compression performance will not be bottlenecked by disk storage's bandwidth. It is not clear if the prior checkpoint compression studies (e.g., Ibtesham *et al*. in [20]) were performed on a system with fast SSDs. This is the primary reason we do not directly use compression data from prior work.

5.1 Tools and Methodology

5.1.1 Checkpoint data collection. We use seven mini-apps from the Mantevo project [46]. They are CoMD version 1.1, HPCCG version 1.0, MiniAero version 1.0, miniFE version 1.5, miniMD version 1.2, miniSMAC2D version 2.0 and pHPCCG version 0.4. Please refer to the work by Ibtesham *et al*. [20] and Heroux *et al*. [46] for the description of the mini-apps.

The mini-apps are run on a single compute node. The compute node is a 2-way SMP with AMD Opteron processors with 8 cores per socket (16 cores per node) and 2GB memory per core. We use BLCR's (Berkeley Lab Checkpoint/Restart) [45] checkpoint/restart support integrated in OpenMPI framework [47]. The BLCR library provides a transparent coordinated checkpoint/restart implementation. Each application is run with 16 MPI processes using OpenMPI [48] version 1.3.3. We take three checkpoints from each application at approximately 25%, 50% and 75% of the execution time. The input size for the mini-apps is selected such that the physical memory usage per MPI process fits in the available physical memory per core while giving a good variation of checkpoint sizes across the mini-apps. We observed that the checkpoint size is close to the memory usage of the application. Table 2 shows the total size of all checkpoint data (in gigabytes) generated for each application.

5.1.2 Compression utilities. The compression utilities used for the study are gzip [49], bzip2 [50], xz [51], and lz4 [52]. Each of these utilities has a configurable compression level that ranges from zero or one to nine where a lower value gives faster execution but lower compression factor. For each compression utility except lz4 we use two compression levels: the suggested default level and compression level one. For lz4 we just use the default compression level which is compression level one. The suggested compression level of a utility is supposed to give the best compression speed and compression factor trade-off for that utility. For each utility and compression level, we measure the compression speed (MB/s) and compression factor on the collected checkpoint data. Compression factor is calculated as 1 - (compressed_size)/(uncompressed_size).

5.2 Data: Compression Speed and Factor

Table 2 shows the compression speed (in MB/s) corresponding to each utility and mini-app obtained by compressing using a single thread on that utility. The average compression speed observed varies from 441.9 MB/s for lz4 to 4.8 MB/s for xz(compression level six). Table 2 shows the compression factor. The average compression factor for the utilities varies from 64.75% for lz4(1) to 83.25% for xz(6).[8]

In this work we only collect checkpoint compression data for mini-apps from the Mantevo project. Prior work by Ibtesham *et al.* in [20] collected the compression data for checkpoints of two large scale production apps - LAMMPS and CTH. For LAMMPS, they observed a compression factor of 91.6% and 92.7% and a compression speed of 209.1 MB/s and 154.2 MB/s for zip and pbzip2 (concurrent threads of bzip2) respectively. For CTH, they observed a compression factor of ~83% and ~85% and a compression speed of ~175 MB/s and ~40 MB/s for zip and pbzip2 (concurrent threads of bzip2) respectively. These checkpoint compression factors for production apps is similar to that observed in our checkpoint compression study for mini-apps.

5.3 Configuring NDP for Compression

For the exascale system projected in section 3.1, the compression data helps us determine which compression algorithm to use and the compression performance requirement. In section 4.4, we derived the upper limit of the compression speed required from NDP given the global I/O bandwidth and compression ratio: Compression_rate = (Uncompressed_size/Compressed_size) * IO_bandwidth. A compression rate higher than that obtained using this equation does not help improve checkpointing rate to I/O as it gets bottlenecked by I/O bandwidth.

Let us take the case of gzip(1). It has an average compression factor of 72.77%. Converting this to (uncompressed_size / compressed_size) ratio, gives a value of 3.67. Therefore the compression speed required is 367 MB/s ($3.67 * 100$ MB/s). A single thread of gzip(1) gives an average compression speed of 110.1 MB/s per processing core. Four such cores can reach a speed of 440.4 MB/s meeting the required compression speed. Using the average compression factor for gzip(1) as reference, 112 GB of checkpoint data per node would be compressed to 30.5 GB, which would need 305 seconds (5.08 minutes) to write to I/O.

By performing this analysis for all compression utilities (and compression level combinations) using the average compression factor and the average single thread compression speed, we can get the upper limit of the required number of cores in NDP and the minimum checkpoint interval for checkpoints to global I/O. The smaller the checkpoint interval, the smaller would be the amount of lost work that needs to be rerun after a recovery using I/O stored checkpoint. The results are shown in Table 3.

For core count, we round up to the nearest integer value. Using lz4 would require just a single NDP core, while gzip(1) and gzip(6) would require 4 and 8 NDP cores respectively. For bzip2 and xz, the number of NDP cores required vary from 34 to 125. Even though bzip2 and xz would allow for higher checkpoint to I/O frequencies, the NDP processing requirement of using bzip2 and xz makes them infeasible. While lz4 can be used with lower hardware cost, we choose gzip(1) for our evaluation. With 4 NDP cores for compression, it allows saving a checkpoint to I/O much more frequently than lz4(305s vs 395s) and not much less frequently than gzip(6)(305s vs 283s) which requires 8 NDP cores.

6 EVALUATION

In this section, we evaluate the impact of using our NDP approach with checkpoint compression on the performance of our projected exascale system for different scenarios. We compare the performance (progress rate) of our proposed solution with existing optimizations that reduce C/R cost.

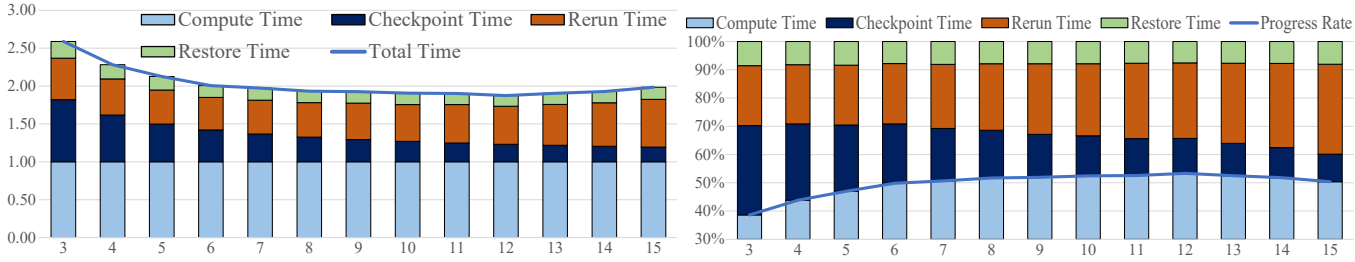
6.1 Methodology

6.1.1 Performance model. To evaluate the performance of different C/R mechanisms, we develop a performance model to obtain the expected execution time of an application for different configurations. This model is based on Daly's analytical model [2] and is similar to the performance model described in [38] with two key differences. We model multilevel checkpointing faithfully and we add support for evaluating our proposed optimization using NDP. To evaluate the performance of multilevel checkpointing, the work in [38] used a single effective checkpoint read/write bandwidth which was higher than the bandwidth of checkpointing to remote storage but slower than that of writing checkpoints to local storage. In our performance model, we model different bandwidths for checkpointing to local storage and to global I/O based storage. We also model different frequencies for writing checkpoints to local storage and global I/O

[8] Value inside () after the compression utility name is the compression level used.

Mini-Apps	Checkpoint Data Size	gzip(1)		gzip(6)		bzip2(1)		bzip2(9)		xz(1)		xz(6)		lz4(1)	
		Factor	Speed	Factor	Speed	Factor	Speed	Factor	Speed	Factor	Speed	Factor	Speed	Factor	Speed
CoMD	25.07 GB	84.2%	153.7	84.4%	92.3	85.1%	32.5	85.0%	30.4	86.0%	23.5	86.2%	8.2	82.8%	658.3
HPCCG	45.92 GB	88.4%	150.7	92.3%	61.6	92.4%	5.9	93.6%	4.6	96.9%	47.5	98.7%	7.4	81.6%	447.8
miniFE	52.31 GB	71.5%	84.5	77.6%	24.1	80.7%	10.7	82.3%	10.1	87.6%	18.3	91.1%	1.6	54.8%	253.9
miniMD	23.94 GB	57.0%	52.2	58.4%	27.7	59.1%	10.0	59.5%	9.2	63.4%	8.0	67.9%	2.5	47.0%	345.3
miniSmac	28.11 GB	35.0%	37.3	35.5%	24.4	31.4%	6.9	32.4%	6.0	47.5%	5.1	48.8%	2.6	24.1%	342.7
miniAero	0.78 GB	84.3%	138.5	85.7%	61.2	86.6%	12.0	87.1%	8.2	88.1%	28.4	92.8%	4.3	80.5%	567.9
pHPCCG	46.18 GB	89.1%	154.0	89.1%	63.2	93.1%	6.8	94.0%	4.8	94.7%	45.9	97.3%	7.0	82.4%	477.7
Average	31.76 GB	72.8%	110.1	74.7%	50.6	75.5%	12.1	76.3%	10.5	80.6%	25.3	83.3%	4.8	64.8%	441.9

Table 2: Checkpoint Data Details. Second column shows the total checkpoint data size collected for each mini-app. Further columns show compression factor and compression speed (in MB/s). Compression speed is for a single thread of each utility.



(a) C/R overhead breakdown normalized to compute time.

(b) C/R overhead % breakdown of total execution time.

Figure 4: C/R overhead breakdown for one Local + I/O-Host configuration on y-axis for increasing ratio of locally-saved to I/O-saved checkpoints on x-axis. Note that for % breakdown (fig. b), y-axis starts at 30% and not at 0% for better resolution.

Utility (level)	Required Compression Speed	Number of Cores	Checkpoint Interval
gzip(1)	367 MB/s	4	305 s
gzip(6)	395 MB/s	8	283 s
bzip2(1)	407 MB/s	34	275 s
bzip2(9)	421 MB/s	41	266 s
xz(1)	515 MB/s	21	217 s
xz(6)	596 MB/s	125	188 s
lz4(1)	283 MB/s	1	395 s

Table 3: The required compression speed, required number of processor cores in NDP and the smallest possible checkpoint interval to I/O based on average compression factor and speed.

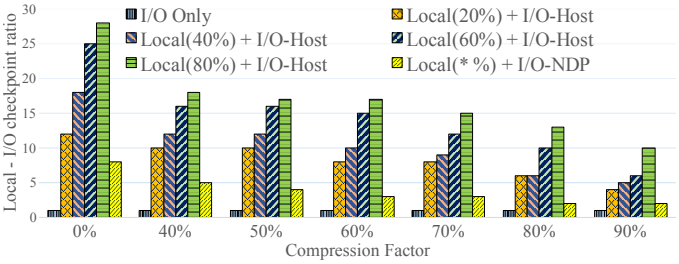


Figure 5: Ratio of the number of locally saved to the number of I/O saved checkpoints for different configurations and compression factors

based storage. In addition, we model failures during recovery from local storage (*local*-level) and make the probability of this failure configurable. To evaluate the performance of our NDP approach, we model that certain tasks of checkpointing (i.e. compressing and writing compressed checkpoints to global I/O) are performed in the background and do not add to execution time. The modelling of our multilevel checkpointing with NDP approach is based on the operational details we described in section 4.2.

The performance model uses the assumption that interrupts occur randomly and are exponentially distributed. The basic (single level) implementation of C/R involves writing all checkpoints to I/O and the performance model requires the following inputs to evaluate its performance: MTTI, checkpoint size and aggregated system I/O bandwidth. For evaluation of multilevel checkpointing, the model needs local NVM bandwidth and the ratio of locally-saved to I/O-saved checkpoints as additional inputs.[9] For analysis of checkpoint compression, checkpoint compression factor and compression/decompression speeds are the inputs to the model.

6.1.2 Checkpoint/Restart Configurations. To compare the performance of our proposed C/R optimization with existing solutions, we evaluate the following configurations:

(1) *I/O Only*: All checkpoints are saved to I/O based storage. The data collected are for scenarios with and without checkpoint compression.
(2) *Local + I/O-Host*: Checkpoints are saved to compute node local storage and I/O based storage. The optimal ratio of locally-saved checkpoints to I/O-saved checkpoints is obtained empirically (see section 6.2). The data are collected for conventional multilevel checkpointing as well as our proposed combination of multilevel with compression. Checkpoint compression is performed for I/O-saved checkpoints only. After a failure an attempt

[9] I/O-saved checkpoint refers to checkpoints saved to global I/O (*I/O*-level) of multilevel checkpointing. Locally-saved checkpoint refers to checkpoints saved to local storage of one ore more compute nodes and thus refers to both the *local* and *partner* level. See section 3.4 for details.

Checkpoint/restart Parameter	Value or Range
System MTTI	30 minutes
Checkpoint Size (80% of memory)	112 GB/node
Compute local NVM BW	15.0 GB/s
Checkpoint interval(to local)	150 seconds
Probability of recovery from Local	20% - 96%
Compression Factor	Mini-app specific
Compression Rate (4 cores NDP)	440.4 MB/s
Decompression Rate (64 cores Host)	16.0 GB/s

Table 4: C/R parameters for evaluation using performance model

is made to first recover from locally-saved checkpoint, but if it fails, it is recovered from *I/O*-level. The probability of successful recovery from local is configurable and we collect data varying in between 20% and 96%. This value is mentioned in configuration name as *Local(x%) + I/O-Host*.

(3) *Local + I/O-NDP*: This is our proposed optimization to multilevel checkpointing using NDP. All checkpoints are first saved to compute node local storage. Some or all of these checkpoints are then saved to I/O based storage by NDP. Data are collected for scenarios with and without checkpoint compression by NDP. Recovery from failure works the same as described in *Local + I/O-Host*.

6.1.3 Checkpoint/Restart Parameters. We derive the C/R related parameters based on our projections of the exascale configuration from section 3.1 and summarized in table 1. Parameters specific to C/R are shown in table 4. Node local NVM read/write bandwidth is set to 15 GB/s which is within the limit of PCIe-3 and slightly higher than the bandwidth projected to achieve a 90% progress rate for single level checkpointing. We chose gzip(1) for compression and calculated a compression speed of 440.4 MB/s in section 5.3. Checkpoint decompression is performed by host processors. We observed an average decompression rate of 350 MB/s when using gzip(1) across the seven mini-apps. With a total of 64 CPU cores, this would give a speed of 22.4 GB/s. We conservatively assume it to be 16 GB/s.

The optimal *checkpoint interval* (i.e. the time an application performs useful computation between two checkpoints), is selected based on Daly's estimate of optimal checkpoint interval [2]. For the multilevel checkpoint mechanism, in addition to *checkpoint interval* for local checkpoints, we derive the ratio of locally-saved to I/O-saved checkpoints, which is another way of determining the checkpoint interval for global I/O. The next subsection discusses the impact of these two parameters on C/R overhead.

6.2 Checkpoint/Restart Overhead Components

The total execution time of an application with C/R can be broken down into four components.

(1) *Compute time*: Total time spent by an application doing useful work.
(2) *Checkpoint time*: Total time spent by an application to save all checkpoint data.
(3) *Restore time*: Total time spent by an application in restoring information from checkpoints.
(4) *Rerun time*: Total time spent by an application in rerunning from a checkpoint to the point of failure (equivalent to progress lost from aborting non-checkpointed work).

C/R total overhead is the sum of components 2-4. Given the cost of committing and restoring a single checkpoint and the system MTTI, Daly's estimate of optimal *checkpoint interval* [2] minimizes the total overhead. Taking checkpoints too frequently results in a higher *checkpoint time* as more checkpoints are saved. On the other hand it results in a lower *rerun time*, because on average less work is lost. Taking checkpoints too infrequently has the opposite impact. At the optimal checkpointing interval, the total overhead is minimum.

For the multilevel checkpointing scheme, the ratio of locally-saved to I/O-saved has a similar impact. Figure 4 shows the impact of this ratio for the *Local + I/O-Host* configuration. Figure 4a shows the breakdown of total execution time into the four components normalized to compute time while Figure 4b shows percentage breakdown of the same. We see in these figures that as the ratio increases (i.e. as checkpoints to I/O are less frequent), the contribution of *checkpoint time* decreases but that of *rerun time* increases. Total C/R overhead initially decreases but eventually reaches a minimum and increases again.

Figure 4b shows the same impact in terms of fraction of time doing useful work (i.e. the percentage of *compute time* in total execution time). This is the efficiency of execution or the progress rate. The progress rate initially increases, reaches a maximum value, and then decreases. For data in this work corresponding to all *Local + I/O-Host* configurations, we use the optimal ratios derived empirically. The optimal ratios for various configurations are shown in Figure 5. Lower compression factor results in a higher cost for writing checkpoints to I/O. This results in a lower optimal frequency of checkpoints to I/O and thus a higher locally-saved to I/O-saved ratio. This ratio decreases with higher compression factor as the cost to write checkpoints to I/O decreases. Similarly the higher the probability of recovery from local (the number in parentheses in configuration name), the higher is the ratio of locally-saved to I/O-saved checkpoints.

On the other hand for *Local + I/O-NDP* configuration, checkpoints to I/O are compressed and written by NDP off the critical execution path. Thus writing checkpoints to I/O more frequently will not increase the *checkpoint time* but will decrease the *rerun time*. Therefore in this case we save checkpoints to I/O as frequently as possible leading to a lower locally-saved to I/O-saved checkpoint ratio compared to *Local + I/O-Host*. Note that the probability of recovery from local does not play a role in determining this ratio for *Local + I/O-NDP* and thus the Figure 5 just has one ratio per compression factor irrespective of the probability of recovery from local.

6.3 Progress Rate Comparison

Figure 6 shows a comparison of progress rate for the three C/R configurations described in section 6.1.2. Each mini-app is sized to have the same checkpoint size while the compression factor is the one obtained using gzip(1) as shown in table 2. The figure shows progress rate for all configurations without compression (0% compression factor). For *Local + I/O-Host* and *Local + I/O-NDP* the probability of recovery from checkpoints saved in local storage is

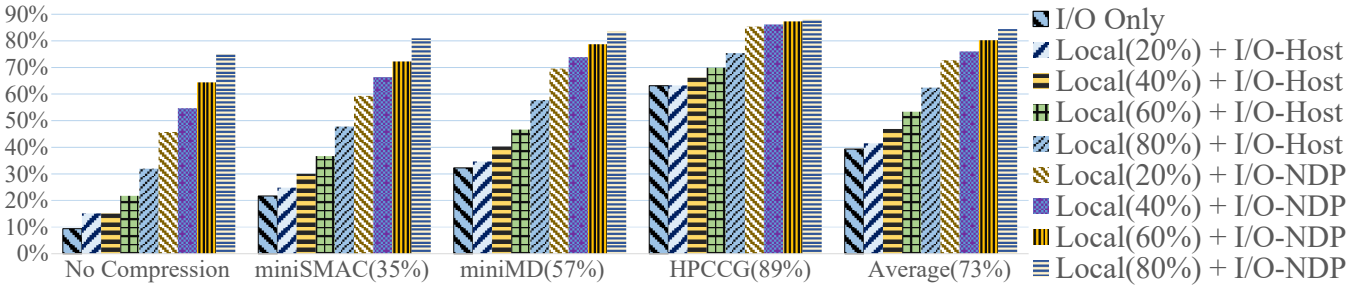


Figure 6: Progress rate comparison between different configurations. Data are shown for 3 of the 7 mini-apps studied and an average progress rate over the 7 mini-apps. The first set of bars is for no compression while for others the compression factor used is for gzip(1) as shown in table 2. Compression factor is specified in parentheses in the label on x-axis.

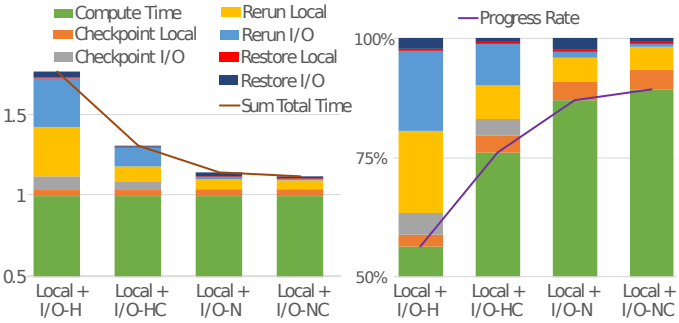


Figure 7: C/R overhead breakdown normalized to compute time (left) and as % of total execution time (right). Y-axis does not start at 0 for both plots. Four configurations from the left: multilevel (*Local + I/O-H*), multilevel+compression (*Local + I/O-HC*), NDP (*Local + I/O-N*) & NDP+compression (*Local + I/O-NC*). The probability that recovery from local fails: 4%. Compression factor: 73%

varied from 20% to 80%. The progress rate is higher for application checkpoints with higher compression and for configurations with higher probability of recovery from local. The progress rate for *Local(x%) + I/O-NDP without compression* is better than *Local(x%) + I/O-Host* for the average compression factor of 73% for all values of x. **Thus, the progress rate with NDP without compression on an average is higher than using multilevel checkpoint with compression.** For the case with probability of recovery from local of 80%, adding compression to multilevel checkpointing improves the progress rate from 32% to 62%. Further with NDP but without the benefits of compression, progress rate improves to 75%. Finally, performing compression using NDP leads to a progress rate of 84%. On average, over the four values of probability of recovery from local, the progress rate of multilevel checkpointing with compression improves from 51% without NDP to 78% when using NDP.

6.4 C/R Overhead - Breakdown (4% I/O Recovery)

Figure 7 shows the breakdown of C/R overhead for compression factor of 73% (average case) when the probability that a failure cannot be recovered from checkpoint stored in compute-node local storage and thus needs recovery from global I/O is set at just 4%. This is based on the work by Moody *et al.* in [13]. They observed that 15% failures needed recovery from parallel file system (or global I/O), but this could be reduced to less than 4% with certain changes. The two plots in the figure show the same data, but use different scaling for the y-axis. The plot on the left shows execution time normalized to compute time while the figure on the right shows the percentage breakdown of the total execution time C/R overhead components. The three components of C/R overhead - *checkpoint*, *restore* and *rerun* time are further divided based on whether the checkpointing or restoring is to/from compute-node local storage or global I/O. This is done to highlight that for multilevel checkpointing, even though a small fraction of C/R operations are associated with global I/O, they contribute to a high fraction of the total overhead.

For multilevel checkpointing without compression (*Local + I/O-H*), the plot on the right in Figure 7 shows that *rerun* overhead when recovering from I/O ("Rerun I/O") accounts for 17% of the execution time. This high overhead is due to the low frequency of checkpointing to I/O, which leads to higher amount of lost work. When compression is added to multilevel checkpointing (*Local + I/O-HC*), this component drops to 9% of the execution time. Compression of checkpoint data being written to global I/O reduces the checkpoint commit time to I/O. This leads to writing checkpoints to I/O at a higher frequency and thus reduces the average amount of computations that need to be rerun after a failure. When using NDP approach, no time is spent in writing checkpoints to I/O in the critical path. This is seen in the figure as absence of "Checkpoint I/O" component in execution time for *Local + I/O-N* and *Local + I/O-NC*. For NDP without compression, checkpoints are written to I/O by the NDP as frequently as the bandwidth to I/O allows, leading to the "Rerun I/O" component being reduced to 1.2% of the execution time. For NDP with compression, checkpoints can be written to I/O even more frequently due to a reduction in their size by 73%. This reduces the "Rerun I/O" component to 0.6% of execution time. The figure also shows that for NDP with compression, the C/R progress rate approaches 90%, as the overhead associated with C/R to I/O is negligible. Recall that, the system was configured to have a 90% progress rate with single level checkpointing to local.

6.5 C/R Overhead - Sensitivity Study

In this section we show the impact of varying checkpoint size and MTTI on the progress rate for various C/R configurations. The C/R configurations evaluated are *L-*GBps + I/O-HC* (multilevel + compression), *L-*GBps + I/O-N* (NDP without compression) and *L-*GBps + I/O-NC* (NDP + compression). The value after *L-* indicates the bandwidth of compute node local storage. Multilevel checkpointing with compression is evaluated for only 15 GB/s compute

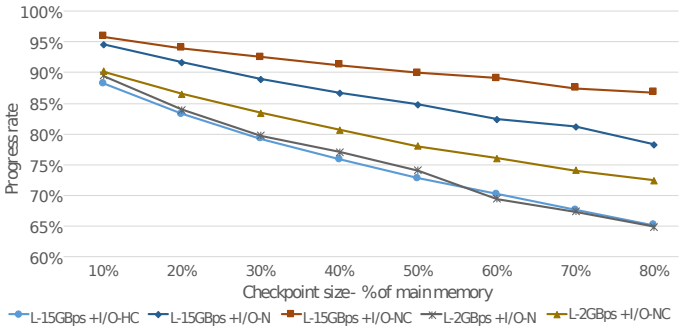


Figure 8: Progress for five C/R configurations for increasing checkpoint size. Y-axis does not start at 0%. MTTI: 30 minutes

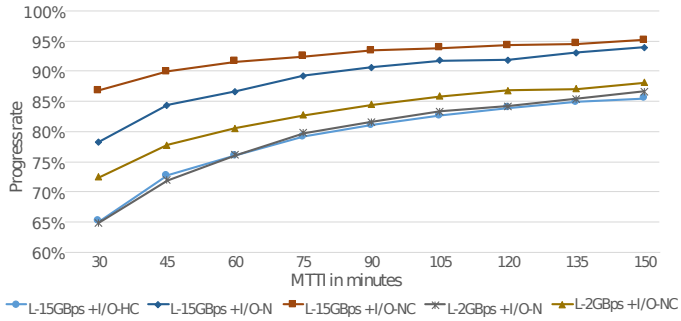


Figure 9: Progress for five C/R configurations for increasing MTTI. Y-axis does not start at 0%. Checkpoint size: 112 GB per compute node

node local storage bandwidth, while both the NDP configurations are evaluated for a fast 15 GB/s and a slow 2 GB/s compute node local storage bandwidth. Probability of successful recovery from *local*-level is set to 85% [13] and checkpoint compression factor of 73% (average for the seven mini-apps) is used.

Figure 8 shows the progress rate for increasing value of checkpoint size. Checkpoint size is expressed in terms of the percentage of main memory and is varied from 14 GB per compute node (10% of main memory from table 1) to 112 GB per compute node (80% of main memory). The figure shows that when the checkpoint size is 10% of the main memory, *L-15GBps + I/O-NC* configuration has a progress rate of 96% versus a progress rate of 88% for *L-15GBps + I/O-HC*. On the other hand, for checkpoint size that is 80% of the main memory, the progress rate improves from 65% for *L-15GBps + I/O-HC* to 87% for *L-15GBps + I/O-NC*. Thus NDP with compression gives a bigger gain in progress rate over multilevel with compression when the checkpoint size is larger. This is because for larger checkpoint sizes the overhead associated with checkpointing to I/O is larger. When this larger overhead gets hidden due to the use of NDP, there is a bigger increase in progress rate. The figure also shows that even with a slower compute node local storage with a bandwidth of 2 GB/s, using NDP with compression (*L-2GB/s + NC*) gives a higher progress rate than multilevel checkpointing with compression with fast compute node local storage (*L-15GB/s + HC*). For *L-2GB/s + N* (NDP without compression and 2 GB/s compute local storage bandwidth), the progress rate is similar to that of *L-15GB/s + HC*. Thus one can substitute a 15 GB/s local storage with a 2 GB/s storage with NDP and get a similar or better progress rate.

Figure 9 shows the trends for progress rate for increasing values of MTTI for different C/R configurations. MTTI is varied between 30 minutes and 150 minutes. The figure shows that for an increasing value of MTTI (i.e. decreasing failure rate), there is a decreasing gain in performance from using NDP over using multilevel checkpointing with compression . This plot also shows that 15 GB/s local storage (*L-15GB/s + HC*) can be substituted with 2 GB/s local storage with NDP to achieve similar (*L-2GB/s + N*) or better (*L-2GB/s + NC*) progress rate.

7 CONCLUSION

C/R has been a widely used tool to provide resilience in HPC systems. As we approach exascale systems, C/R overheads are projected to likely become intolerable. Our analysis of multilevel checkpointing on projected exascale system showed that while the overhead for local checkpoints is tolerable, the overhead of checkpoints to global I/O is large. We proposed leveraging NDP that couples compute capabilities to the compute-node local NVM to offload the worst checkpoint-related overheads from the host processor and use optimizations such as checkpoint compression. We show that our C/R approach with NDP even without compression achieves a better performance compared to multilevel checkpointing with compression in most cases. Using NDP without compression can be used to improve C/R performance with low cost NDP hardware. The proposed application of NDP for C/R is but one possible application of NDP as NVM becomes more common. Just for C/R, additional optimizations using NDP need to be explored. For instance, NDP is well suited to compare data for consecutive checkpoints and checkpoints of neighboring MPI rank. Such explorations have a potential to achieve a much larger reduction in the amount of checkpoint data.

REFERENCES

[1] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead," 2008.

[2] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Gener. Comput. Syst.*, vol. 22, pp. 303–312, Feb. 2006.

[3] J. T. Daly, "Quantifying checkpoint efficiency," *Nuclear Weapons Highlights*, pp. 188–189, 2007.

[4] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," in *Journal of Physics: Conference Series*, vol. 78, p. 012022, IOP Publishing, 2007.

[5] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia, S. Byna, S. Farrell, D. Gursoy, C. Daley, V. Beckner, B. V. Straalen, D. Trebotich, C. Tull, G. Weber, N. J. Wright, and K. Antypas, "Accelerating science with the nersc burst buffer early user program," in *Proceedings of Cray Users Group [Online]*, 2016.

[6] Y. Kang, Y.-S. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart ssd.," in *MSST*, pp. 1–12, IEEE Computer Society, 2013.

[7] H. Choe, S. Lee, H. Nam, S. Park, S. Kim, E.-Y. Chung, and S. Yoon, "Near-data processing for machine learning," *arXiv preprint arXiv:1610.02273*, 2016.

[8] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A user-programmable ssd," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), pp. 67–80, USENIX Association, 2014.

[9] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: Opportunities and challenges," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), pp. 1221–1230, ACM, 2013.

[10] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. J. Desnoyers, and Y. Solihin, "Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, (Berkeley, CA, USA), pp. 119–132, USENIX Association, 2013.

[11] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger, "Active disk meets flash: A case for intelligent ssds," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, (New York, NY, USA), pp. 91–102, ACM, 2013.

[12] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The international exascale software project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, pp. 3–60, Feb. 2011.

[13] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[14] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello, "Optimization of multi-level checkpoint model for large scale hpc applications," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 1181–1190, May 2014.

[15] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), pp. 57:1–57:12, ACM, 2009.

[16] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using nvm as virtual memory," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 29–40, May 2013.

[17] G. Zheng, L. Shi, and L. V. Kale, "Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi," in *Cluster Computing, 2004 IEEE International Conference on*, pp. 93–103, Sept 2004.

[18] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "Plfs: A checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), pp. 21:1–21:12, ACM, 2009.

[19] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. D. Panda, "A 1 pb/s file system to checkpoint three million mpi tasks," in *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, (New York, NY, USA), pp. 143–154, ACM, 2013.

[20] D. Ibtesham, K. B. Ferreira, and D. Arnold, "A checkpoint compression study for high-performance computing systems," *Int. J. High Perform. Comput. Appl.*, vol. 29, pp. 387–402, Nov. 2015.

[21] D. Ibtesham, D. Arnold, K. B. Ferreira, and P. G. Bridges, "On the viability of checkpoint compression for extreme scale fault tolerance," in *Proceedings of the 2011 International Conference on Parallel Processing - Volume 2*, Euro-Par'11, (Berlin, Heidelberg), pp. 302–311, Springer-Verlag, 2012.

[22] K. B. Ferreira, R. Riesen, R. Brighwell, P. Bridges, and D. Arnold, "Libhashckpt: Hash-based incremental checkpointing using gpu's," in *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, (Berlin, Heidelberg), pp. 272–281, Springer-Verlag, 2011.

[23] J. Kaiser, R. Gad, T. SÃijÃ§, F. Padua, L. Nagel, and A. Brinkmann, "Deduplication potential of hpc applications' checkpoints," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 413–422, Sept 2016.

[24] B. Nicolae, "Towards scalable checkpoint restart: A collective inline memory contents deduplication proposal," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 19–28, May 2013.

[25] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, "Exploring automatic, online failure recovery for scientific applications at extreme scales," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (Piscataway, NJ, USA), pp. 895–906, IEEE Press, 2014.

[26] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining partial redundancy and checkpointing for hpc," in *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pp. 6f15–626, June 2012.

[27] I. Doudalis and M. Prvulovic, "Euripus: A flexible unified hardware memory checkpointing accelerator for bidirectional-debugging and reliability," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, (Washington, DC, USA), pp. 261–272, IEEE Computer Society, 2012.

[28] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 58:1–58:11, IEEE Computer Society Press, 2012.

[29] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, "Design and evaluation of a processing-in-memory architecture for the smart memory cube," in *Proceedings of the 29th International Conference on Architecture of Computing Systems – ARCS 2016 - Volume 9637*, (New York, NY, USA), pp. 19–31, Springer-Verlag New York, Inc., 2016.

[30] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 105–117, ACM, 2015.

[31] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: Insights from a micro-46 workshop," *IEEE Micro*, vol. 34, pp. 36–42, July 2014.

[32] K. Ferreira, "Keeping checkpoint/restart viable for exascale systems," 2012.

[33] J. Rogers, "Power efficiency and performance with ornl's cray xk7 titan," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, (Washington, DC, USA), pp. 1040–1050, IEEE Computer Society, 2012.

[34] D. Vantrease, R. Schreiber, M. Monchiero, M. McLaren, N. P. Jouppi, M. Fiorentino, A. Davis, N. Binkert, R. G. Beausoleil, and J. H. Ahn, "Corona: System implications of emerging nanophotonic technology," in *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pp. 153–164, June 2008.

[35] Y. Chen, "Towards scalable i/o architecture for exascale systems," in *Proceedings of the 2011 ACM International Workshop on Many Task Computing on Grids and Supercomputers*, MTAGS '11, (New York, NY, USA), pp. 43–48, ACM, 2011.

[36] R. Riesen, K. Ferreira, J. Stearley, R. Oldfield, J. H. Laros III, K. Pedretti, R. Brightwell, *et al.*, "Redundant computing for exascale systems," *Sandia National Laboratories*, 2010.

[37] http://www.seagate.com/www-content/product-content/ssd-fam/nvme-ssd/nytro-xp7200/_shared/docs/nytro-xp7200-add-in-card-ds1905-1-1607us.pdf, 2016.

[38] D. Ibtesham, D. Arnold, P. G. Bridges, K. B. Ferreira, and R. Brightwell, "On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance," in *2012 41st International Conference on Parallel Processing*, pp. 148–157, Sept 2012.

[39] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on fpgas using opencl," in *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, (New York, NY, USA), pp. 4:1–4:9, ACM, 2014.

[40] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A framework for near-data processing of big data workloads," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 153–165, IEEE Press, 2016.

[41] B. Holden, D. Anderson, J. Trodden, and M. Daves, *HyperTransport 3.1 Interconnect Technology*. Mindshare Press, first ed., 2008.

[42] http://www.ccixconsortium.com/about-us.html.

[43] http://genzconsortium.org/.

[44] http://opencapi.org/.

[45] J. Duell, "The design and implementation of berkeley lab's linux checkpoint/restart," *Lawrence Berkeley National Laboratory*, 2005.

[46] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Technical Report SAND2009-5574*, 2009.

[47] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for Open MPI," in *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE Computer Society, 03 2007.

[48] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, (Budapest, Hungary), pp. 97–104, September 2004.

[49] P. Deutsch, "Deflate compressed data format specification version 1.3." https://tools.ietf.org/html/rfc1951.

[50] J. Seward, "bzip2 and libbzip2, version 1.0.5, a program and library for data compression." http://www.bzip.org, 2007.

[51] T. Developers, "Xz utils." http://tukaani.org/xz/, 2015.

[52] Y. Collet, "Lz4 explained." http://fastcompression.blogspot.cz/2011/05/lz4-explained.html, 2011.