The Design and Implementation of a Multi-level Content-Addressable Checkpoint File System

Abhishek Kulkarni*†, Adam Manzanares†, Latchesar Ionkov†, Michael Lang†, Andrew Lumsdaine*
*Indiana University
{adkulkar, lums}@cs.indiana.edu
†Los Alamos National Laboratory
{adamm, lionkov, mlang}@lanl.gov

***Abstract*—Long-running HPC applications guard against node failures by writing checkpoints to parallel file systems. Writing these checkpoints with petascale class machines has proven difficult and the increased concurrency demands of exascale computing will exacerbate this problem. To meet checkpointing demands and sustain application-perceived throughput at exascale, multi-tiered hierarchical storage architectures involving solid-state burst buffers are being considered. In this paper, we describe the design and implementation of cento, a multi-level, content-addressable checkpoint file system for large-scale HPC systems. cento achieves in-flight checkpoint data reduction across all compute nodes through compression and elimination of duplicate blocks over a series of checkpoints. Through a detailed analysis of checkpoint dumps, we assess the benefits of data reduction for scientific applications that are representative of production workloads. We observe upto 40% data reduction within a limited sample of representative workloads. Finally, experiments on existing systems show a decrease in checkpoint commit latencies by 5 to 20% reducing the load on the parallel file system.**

I. Introduction

As supercomputing systems transition from the petascale to the exascale era, individual nodes of these systems are increasing in number and complexity. The System Mean Time Between Failure (SMTBF) of exascale systems have been estimated by some projections to be as low as a few hours [1], [2]. Assuming that there is no significant improvement in component reliability, the SMTBF of the system would largely depend on the MTBF of each of the individual components in the system. The next-generation exascale HPC systems would predictably witness an increase in the order of the number of processors, storage disks, memory chips and network switches. In the face of these frequent failures, it is imperative to compositely tolerate hardware and software faults across the system. Periodic, bulk-synchronous checkpointing of the application state to stable storage has been the most widely used technique to ensure valuable compute-time is not lost. The large-scale of concurrency generated by this workload is challenging for stable storage and exascale class machines will only increase this level of concurrency. Exascale class machines will demand more frequent checkpoints, larger checkpoints, and management of the larger scales of I/O concurrency.

Frequent checkpointing to stable storage at higher node counts is prohibitively expensive due to increasing demands on the I/O infrastructure. To keep up with these demands and achieve a sustained application perceived throughput, leadership-class storage systems are moving towards a hierarchical architecture with intermediary storage involving solid-state disks. Several optimizations to keep checkpoint/restart plausible for exascale systems have been explored recently [3]. Owing to the good compressibility of checkpoint data for certain scientific applications, inline compression of checkpoint data has been considered as a viable option at exascale [4], if the compression latencies are kept lower – possibly through hardware compression or the use of accelerators. The benefits of incremental checkpointing, where only the state modified since the last checkpoint is saved, have been studied extensively [5]–[7]. Two of the most common techniques for incremental checkpointing are: *virtual memory-based* page protection and *hash-based* incremental checkpointing. The VM-based incremental checkpointing technique relies on the operating system or an interposition library to track *dirtied* pages for the application. Since it operates on the granularity of pages, applications modifying a small portion of memory in a page rarely benefit from VM-based incremental checkpointing. Further, VM-based incremental checkpointing cannot differentiate between modifications and equivalent updates. Without operating system support, VM-based incremental checkpointing is expensive since memory accesses of the application have to be monitored for changes [8]. Hash-based incremental checkpointing divides the data into fixed or variable-sized blocks such that unmodified blocks coalesce and point to the same block through deduplication. Hybrid solutions employing both the above techniques have been proposed in the literature [9].

Due to the fact that most scientific applications have distinct *compute* and *I/O* phases, utilizing the abundant computational capabilities to achieve I/O reduction is a profitable proposition. Future exascale machines may rely on many-core processors and accelerators with functional partitioning to gain performance and maintain distinct independent failure domains [10]. Traditional data deduplication eliminates redundant data, and thus decreases the amount of data that needs to be written to stable storage. The reduced load on the storage infrastructure comes at the cost of additional computation to determine redundancies in the data.

Exascale I/O forums have identified the need for hierarchical storage with multiple-levels of memory systems to avoid bottlenecks during bursty I/O activities, such as checkpointing [11]. The multi-level checkpointing model not only permits multiple levels of resiliency associated with the checkpoint data, but also allows for more opportunities for in-transit data reduction without a significant penalty. For the purposes of this paper we define in-transit data as data that has been written by the application, but has not been written to stable storage. Multi-level checkpoint models allow the data to move through various levels of storage eventually being written to stable storage. While application I/O patterns are necessarily varying, most scientific applications follow a much stricter checkpoint I/O pattern. Under these assumptions, a checkpoint file system can be optimized for such usage while extending a familiar file system interface. It can transform the I/O patterns to help reduce, if not eliminate, the I/O scaling bottlenecks.

In this paper, we describe the design and implementation of a multi-level checkpoint file system. In particular, we make the following contributions:

- the design and implementation of `cento`, a distributed, multi-level, content-addressable checkpoint/restart file system.
- an empirical evaluation of the benefits of deduplication for scientific applications that are representative of production workloads.
- assessing the viability of incremental checkpointing through data deduplication for rollback-recovery at extreme-scale.

Our results show that both, system-level and application checkpoints, for certain applications are amenable to data reduction through deduplication. In general, `cento` turns the *N-N* checkpoint pattern to a *N-M* pattern with log-structured writes in order to reduce the load on the PFS (parallel file system) infrastructure.

The remainder of the paper is organized as follows. Section II discusses the background and presents an overview of the related work. The design and implementation of `cento`, a multi-level, content-addressable checkpoint file system is described in Section III. Section IV presents an analysis of the checkpoints of representative scientific applications (*miniapps*) for redundant data and corroborates the need for incremental checkpointing of HPC applications. Section V reports the preliminary performance of our file system on existing systems. Finally, the conclusion and future work is discussed in section VI.

II. Background And Related Work

Multi-level checkpointing: To reduce the overall checkpoint overhead, Moody et al. propose the Scalable Checkpoint/Restart (SCR) library, a multi-level checkpointing system that utilizes the storage on compute nodes in addition to using the parallel file system [12]. Multi-level checkpointing using non-volatile memories [13] and cooperative storage [14] have shown reductions in I/O bottlenecks, which results in improved I/O efficiency [15], [16]. While supporting multiple levels similar to the above work, `cento` additionally performs in-transit checkpoint data reduction and transforms the resulting write patterns to log-structured writes.

Checkpoint overhead reduction: Checkpoint images of scientific applications may exhibit high levels of redundancy. This fact has validated the viability of leveraging compression techniques to reduce checkpoint dump times [4], [17], [18]. An overall decrease in checkpoint overhead is obtained only when the gains due to checkpoint data reduction outweigh the cost of compressing the checkpoint data. The use of hardware compression, dedicated cores [10] or accelerators [3], [6] can significantly reduce the cost of deduplicating the redundant data. Adaptive hash-based incremental checkpointing using variable-sized blocks has shown up to a 70% reduction in checkpoint data and a 50% reduction in checkpoint overhead for the NAS parallel benchmarks [5]. Sancho et al. confirmed the feasibility of application-level incremental checkpointing by studying the memory behavior of popular scientific applications [19]. Checkpoint/Restart using incremental checkpointing has, in general, shown improvements in I/O performance and energy consumption under various scenarios [20]. We confirm the above results through a detailed analysis of the checkpoint images of real-world applications and small, self-contained representative *miniapps* that embody the essential performance characteristics of key scientific applications.

Data deduplication: Considerable prior work has been done on finding differences between binary data [9], [21]. Naive hash-based incremental checkpointing with fixed block sizes often suffers from inefficiencies due to insertions or deletions in the input data. A shift-resistant method of variable chunking using Rabin fingerprinting is discussed in [22]. `cento` supports fixed and variable sized chunking, and we show the reduction in checkpoint sizes with each in our analysis in Section IV. There has been a considerable amount of prior work on Content Addressable Storage (CAS) systems. `cento` is based on the design of Venti [23], a Plan 9 archival storage system. Foundation [24] is a system, similar to Venti, for personal archival that uses modified algorithms optimized for single-disk usage to achieve higher throughput. HydraFS [25] is a content-addressable file system built on top of the HYDRAstor CAS system and explores some of the issues in mapping a file system onto a CAS system. While similar to the above systems, `cento` makes simplifying assumptions in its design based on the checkpoint I/O patterns. The hierarchical, multi-layered design of `cento` is better suited for HPC architectures.

Checkpoint Storage Systems Recently, there have been several proposals for storage systems specialized for checkpoint/restart. `stdchk` [14] is the most similar to our system `cento`, but it is focused towards desktop grid computing and uses scavenged disk space from participating desktops. CRFS [26] does transparent write aggregation and asynchronous I/O to the PFS to achieve better write performance. PLFS [27] transforms the commonly occurring *N-1* write pattern to a *N-N* pattern to achieve higher througput. Zest [28] uses log-structured writes and object-based storage to optimize the write latencies. CAPFS [29] demonstrates the benefits of CAS on real-world datasets generated by data intensive applications. None of the above systems have all of the following properties of `cento`: multi-level design, inline deduplication and log-structured *N-M* writes.

III. Cento

Modern large-scale HPC cluster architectures typically have a scalable storage system, often commercial, with parallel file system's being the prevalent storage system. Large-scale concurrent access to files or directories on the parallel file system can severely degrade I/O performance. `cento` is a multi-level, content-addressable checkpoint storage system specifically designed for HPC clusters. It does *in-transit* reduction of checkpoint data through data deduplication and online compression, so that significant space-savings are achieved with an overall reduction of load on the parallel file system. The *N-N* write access pattern is transformed into a *N-M* pattern with log-structured writes reducing the write latencies and mitigates issues like *write amplification* in solid-state disks.

The checkpoint I/O behavior of parallel HPC applications is largely predictable. Based on these observable patterns, the design of `cento` relies on the following simplifying assumptions:

- A *checkpoint snapshot* is either read or written atomically from the start to end without random access. An application does not benefit from a partial checkpoint snapshot.
- Applications usually have distinct *compute* and *I/O* phases, and all checkpoint I/O happens in large bursts.
- Writing out a *checkpoint snapshot* is a more frequent activity than reading back a *checkpoint snapshot*.
- With an increase in intra-node parallelism (including higher number of cores and the presence of accelerators), compute resources are *cheaper* than network resources.

A. Design

A high-level architecture of `cento` is depicted in Figure 1. `cento` is designed to match the hierarchical architecture of bleeding-edge HPC systems. The individual components in the architecture are explained in sections III-C and III-D. The terminology used by `cento` is borrowed from Venti [23]. As shown in Figure 1, the Content Deduplication Server (CDS), instances run on the compute nodes. They are tasked with computing the block identifiers (hashes) for each block and leverage the fact that compute nodes are largely idle during application I/O phases. The MDSS instances are typically run on the I/O nodes of a cluster and are capable of aggregating I/O requests before they are written out in a log-structured fashion to the parallel file system.

B. `cento` File System Client

The `cento` file system client is based on the `v9fs` virtual file system driver in the Linux kernel. It is similar to FUSE but uses the 9P distributed file system protocol [30] to communicate with the `cento` CDS. `v9fs` provides a file system mount point that is capable of intercepting application I/O calls. This

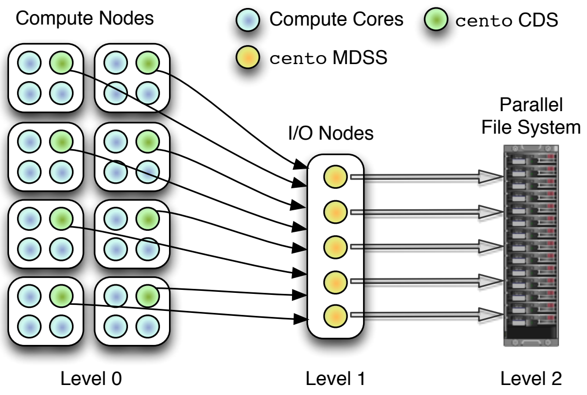


Fig. 1: `cento` High-Level Architecture

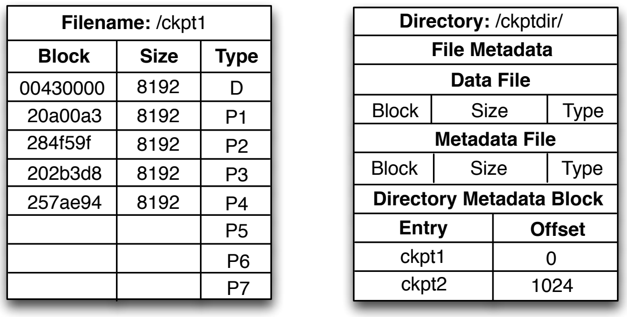
enables applications the ability to use `cento` without any source-code modifications. Based on the assumptions listed in Section III, the file system interface extended to the application does not allow overwriting of files. Writing to files at random offsets is also prohibited. The files can only be written to in append-only mode. Seeking is, however, permitted when reading a file. The file system client can communicate with the CDS over several possible transports including TCP, RDMA and Unix domain sockets.

C. `cento` Content Deduplication Server (CDS)

The CDS acts as a virtual 9P file system server and translates file I/O calls to `cento`-specific calls. I/O requests are forwarded in the form of a request consisting of all relevant details, such as *(file id, [data], size, offset)*. Since the CDS server communicates to the file system client with the 9P distributed network file system protocol, it is possible to run it locally on the same node as the client or on a remote node such as an I/O node. The CDS server bootstraps by connecting to a table of MDSS server instances. The design and operation of the MDSS is described in Section III-D. For all write requests, the CDS computes the hash of the block's contents, hereafter referred to as a *score*. The size of a *score* in our system is limited to 160 bits. Requests are routed to the appropriate MDSS based on the block's *score*. This ensures that the checkpoint write load from *N* CDS clients is distributed evenly to *M* intermediary MDSS servers.

The CDS file server prevents overwriting of files. Existing files can only be appended to. This limitation is generally imposed by CAS systems because a block's *score* uniquely identifies its location. To prevent the size of a write request to the file system from defining the size of a block, CDS aggregates incoming write requests. Incidentally, write aggregation helps improve write performance in the case of small writes. The default block threshold size is 8KB. The file server tracks writes to a file and partially-filled aggregation buffers are flushed upon closing the file.

A much simplified logical layout of CDS file data structures is shown in Figure 2. Each data file has a data block (type *D*)



Filename: /ckpt1		
Block	**Size**	**Type**
00430000	8192	D
20a00a3	8192	P1
284f59f	8192	P2
202b3d8	8192	P3
257ae94	8192	P4
		P5
		P6
		P7

(a) `cento` Data File Layout

Directory: /ckptdir/		
File Metadata		
Data File		
Block	Size	Type
Metadata File		
Block	Size	Type
Directory Metadata Block		
Entry	**Offset**	
ckpt1	0	
ckpt2	1024	

(b) `cento` Directory Layout

Fig. 2: `cento` File System Data Structures

for write buffer aggregation and a few metadata blocks (type P_i) containing a list of scores that constitute a file [31]. The multiple level of metadata blocks form a Merkle tree that can identify a data file by a single unique *score*. Since the size of each block is 8KB, a single metadata block can hold 409 *score* entries. Three levels of metadata blocks can roughly address a file up to 521 GB in size. Directories hold additional metadata blocks that describe the list of entries in the directory.

Algorithm 1 is a description of how a `cento` file is written. All blocks in a file are aggregated in a data block buf unless they reach a threshold (8KB in this case). Procedure writeBuffer first checks if there is space for a *score* entry in the active metadata block s_i. If there is space, the *score* is computed and the block is routed to an appropriate MDSS server. The *score* is appended to the active metadata block s_i. If the active metadata block is filled up, it is written to the data block buf and the resulting *score* is appended to the next active metadata block s_{i+1}. The count of entries in the previous metadata block is reset to zero and the block is written normally. The last active metadata block at depth d holds the *score* of a file. On an open request, the metadata blocks for a file and the first data block is fetched from the MDSS. All subsequent data blocks are fetched as the file is seeked to.

Since the files are evenly distributed across all MDSS servers, the *score* describing the root block of the file system is saved in persistent storage by the clients. For multi-level checkpointing, the resilience factor and access latencies for each level differ. To take advantage of this, the CDS file server can instruct the MDSS to push data to the next level (e.g., from SSD to PFS) or pull it back to a previous level (e.g., from PFS to SSD).

D. `cento` Metadata and Staging Server (MDSS)

The MDSS store blocks in an append-only, log-structured store, which we have named as *arenas*. Each MDSS instance is responsible for a range of *scores*, and maintains a hash table mapping the *scores* to the location of the block in the *arena*. The multiple server instances of MDSS help reduce the amount of memory required on the intermediary nodes to hold this auxiliary data. The *arenas* are memory-mapped by the MDSS server so that writes are fast. When the MDSS server starts to run out of memory, the OS memory manager flushes the dirty blocks in a LRU (Least Recently Used) fashion to the file-backed *arena* on the SSD or the parallel file system. Alternately, the client can send a `sync` request to flush the dirty buffers. To support multi-level checkpointing, the MDSS server can maintain an *arena* for each level. The `push` and `pull` request copy data blocks between these *arenas* at different levels. The protocol to communicate between the CDS and MDSS is shown in Table I. To further eliminate the redundancies within a data block, we use compression when reading and writing data to the *arenas*. The basic operation of a MDSS server is conceptually simple. If a *score* is not present in the hash table, it is appended to the *arena*. Otherwise, the data block corresponding *score* is returned.

E. Implementation

The `cento` checkpoint file system is implemented in C. It is about 13000 lines of code which includes a library to

Algorithm 1: Basic algorithm for CDS file write.

```
Input: File $\mathcal{F}$
Output: Score $score$
$threshold \leftarrow 8192$;
$buf \leftarrow$ empty buffer of type D;
$buflen \leftarrow 0$;
$s_i \leftarrow$ empty list of type $P_i$;
$numscore_i \leftarrow 0$;

procedure writeBuffer(Block $b$):
if $numscore_i < 20$ then
    $score \leftarrow$ hash($b$);
    send $b$ to $mdss(score)$;
    append $score$ to $s_i$;
    $numscore_i \leftarrow numscore_i + 1$;
    return $score$;
else
    $pscore \leftarrow$ writeBuffer(block($s_i$));
    append $pscore$ to $s_{i+1}$;
    $numscore_i \leftarrow 0$;
    $numscore_{i+1} \leftarrow numscore_{i+1} + 1$;
    return writeBuffer($b$);

procedure main():
foreach block $b$ in $\mathcal{F}$ do
    $buflen \leftarrow buflen +$ length($b$);
    if $buflen < threshold$ then
        append $b$ to $buf$;
    else
        writeBuffer($buf$);
$score \leftarrow$ writeBuffer(block($s_{i+1}$));
```

Message	Description
Ping	Check the reachability of the server
Hello	Session negotiation message
Read	Read request
Write	Write request
Sync	Synchronize memory to stable store
Push	Write a block to the $(k+1)$ level
Pull	Read a block from $(k+1)$ to the k level
Goodbye	Session termination message

TABLE I: MDSS Protocol

write userspace 9P file servers, a library that implements the extended Venti protocol, the CDS server and the MDSS server.

IV. Analysis of Checkpoint Dumps

In this section, we present the results of our deduplication analysis on checkpoint dumps of *miniapps* from the Mantevo project [32] and three production workloads: POP, VPIC and xNOBEL. We analyzed a series of checkpoint dumps from each of these workloads for duplicate content using fixed and variable-sized blocking techniques. In addition we have also produced a set of visualizations representing the variability of data within a checkpoint dump across a set of checkpoint dumps.

A. *Miniapps*

Mini-applications (*miniapps*) from the Mantevo project are small, self-contained proxies of real HPC applications being used in production at several national laboratories. They are divided into different classes based on the performance characteristics of the real applications that they embody. We used the following *miniapps* in our evaluation: HPCCG 0.5, miniFE 1.1, miniMD 0.1, phdMesh 0.1 and pHPCCG 0.4. HPCCG is a simple conjugate gradient benchmark for a 3D chimney domain that generates a 27-point finite difference matrix with a user-prescribed sub-block size on an arbitrary number of processors. miniFE is a similar finite-element benchmark which uses recursive coordinate bisection (RCB) partitioning to provide a nearly perfect load balance between the nodes. pHPCCG is a parameterized version of HPCCG with features for arbitrary scalar and integer data types, as well as different sparse matrix data structures. We ran the above three apps (HPCCG, miniFE and pHPCCG) with 4 processes and a problem size of nx=100, ny=100 and nz=100.

miniMD is a miniature version of the popular LAMMPS molecular dynamics (MD) application. miniMD uses spatial decomposition MD to simulate Lennard-Jones pair interaction. The miniMD tests were run on 4 processors with a problem size of 40x40x40 for 100 timesteps with 10 neighbor bins in each direction. phdMesh is a proxy application for the performance-critical parallel geometric proximity search algorithm. It uses an in-memory unstructured mesh and field data structure with an oct-tree geometric search algorithm. It includes dynamic load balancing to divide the load between the processors. This app was run on a single processor with a problem set size of 4x5x3.

B. *Production Workloads*

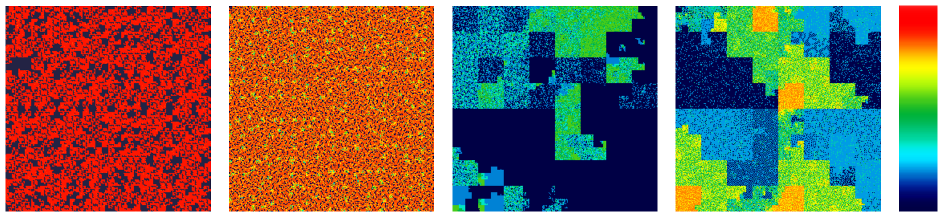
POP [33] is an ocean circulation model in which depth is used as the vertical coordinate. The model solves the three-dimensional primitive equations for fluid motions on the sphere under hydrostatic and Boussinesq approximations. VPIC is an explicit three-dimensional, relativistic, charge-conserving, electromagnetic, particle-in-cell (PIC) code [34] developed at Los Alamos National Laboratory. To simulate plasma behavior, VPIC tracks particle motion in three dimensions of simulated particles as simulated electric and magnetic force fields push the particles, accounting for increases in particle masses as their speeds approach that of light. xNOBEL simulates the shock hydrodynamics using Eulerian methods and adaptive mesh refinement (AMR). xNOBEL is a one, two, three dimensional, multi-material Eulerian hydrodynamics code developed for solving a variety of high deformation flow of materials problems, with the ability to model high explosives [35]. POP and VPIC were run with 256 processors where as the xNOBEL tests were run on 512 processors.

Checkpointer	App	# Ckpts	Avg. Interval(s)	Avg. Size(MB)
BLCR	HPCCG	9	7.22	780
	miniFE	5	7	1195
	miniMD	11	7.36	105
	phdMesh	9	6.77	129
	pHPCCG	6	6.66	741
Application	POP	10	900	4128
	VPIC	10	600	3863
	xNOBEL	5	300	3819

TABLE II: Checkpoint snapshot information

C. *Experimental Setup*

For collecting the *miniapps* checkpoints, we used a 128 node, AMD 2GHz Dual-Core Opteron machine with 4 GB of memory per compute node. Compute nodes were connected with Gigabit Ethernet and InfiniBand. The operating system was RHEL 5.8 with the 2.6.18-274.18.1.el5 kernel. We used Open MPI 1.5.4 and Berkeley Lab Checkpoint/Restart (BLCR) 0.8.4 to take random periodic checkpoints. System-level whole checkpointing results in large checkpoint sizes. To make the analysis viable, we limited these runs to a smaller scale. Since most *miniapps* have weak-scaling workloads, we expect that the patterns revealed by our analysis will hold true at larger scales. To collect the other application checkpoints, we used two different clusters consisting of 256 nodes and 128 nodes respectively, with a setup similar to the one described above. All *N-N* checkpoints were combined into a series of *N-1* checkpoints to ease the process of analyzing and visualizing them. The checkpoint dump details, including the average checkpoint interval and average checkpoint size, for each of the applications are summarized in Table II.



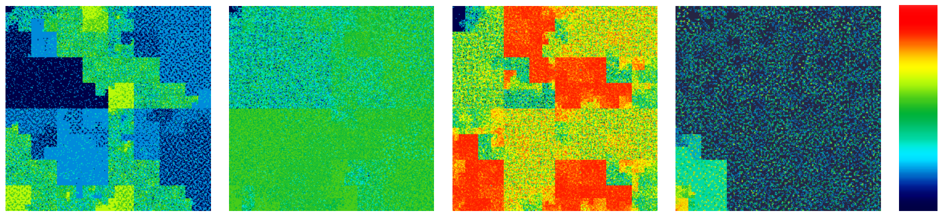


Fig. 3: Checkpoint dump heatmaps of applications

D. Methodology

To understand the benefits of deduplication on checkpoint dumps, it is essential to understand how the checkpoint snapshots change over time. We selected consecutive checkpoint images and sampled the blocks for similarity. The differences were laid out as a Hilbert curve in a delta image and all the deltas were combined to get a resulting heatmap for the application. The darker regions (blue) indicate little to no changes whereas the lighter region (red) indicates maximum changes for a given block across a series of checkpoint snapshots. The color intensity was not normalized and hence depends on the number of checkpoints analysed for a given application. Rather than associating an absolute measure with the intensity, the heatmaps are only indicative of the pattern in which the checkpoint data changes over time. The block sizes were chosen depending on the size of the checkpoint dumps and the resolution of the resulting heatmap visualization. Since the blocks in the checkpoint dumps are compared pairwise, this method returns only a fair estimate of the similarity. Using methods that are more applicable to CAS systems, the deduplication ratios were computed using fixed (512, 4K and 8K bytes) and adaptive chunking. Adaptive chunking using Rabin fingerprinting is shift-resistant and any insertions or deletions in the middle of a file do not affect the deduplication ratio. Both aggregate and cumulative analyses of checkpoint dumps were performed.

A majority of the system-level checkpoint dump is comprised of the virtual memory area (VMA) regions including the initialized and uninitialized data section, anonymous mappings, including the stack and heap, and the open files. BLCR, by default, excludes the executable file (code section), private mapped files (shared libraries) and shared mapped (System V IPC) files from the checkpoint image. The application checkpoint dump consists of data structures specific to the application. In effect, the important distinction is that since system-level checkpointers dump the entire memory region of an application, we are more likely to see high deduplication ratios, whereas the deduplication ratio of application checkpoints depends only on the data structures that the application writes to stable storage.

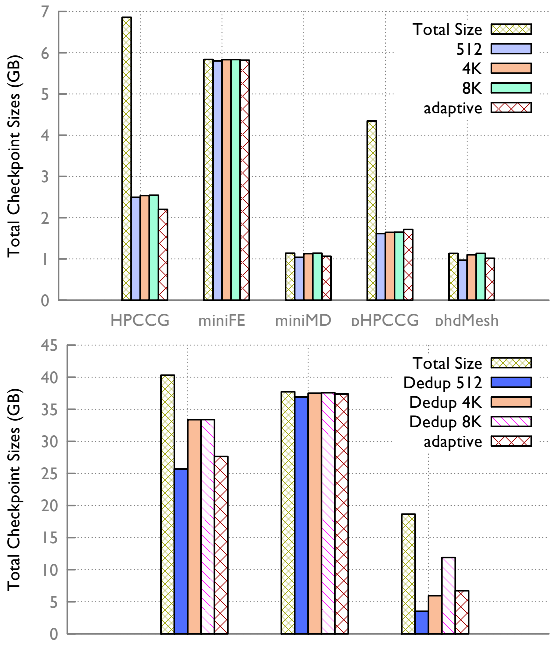


Fig. 4: Deduplication benefits for fixed and adaptive chunking

E. Results

As seen in Figure 3, the applications involving particle simulation or unstructured meshes (like VPIC, miniMD, phdMesh) show either i) a lot of modifications (red regions) or ii) equally-spaced uniform but infrequent modifications (no dark blue regions). The miniMD, miniFE, VPIC and

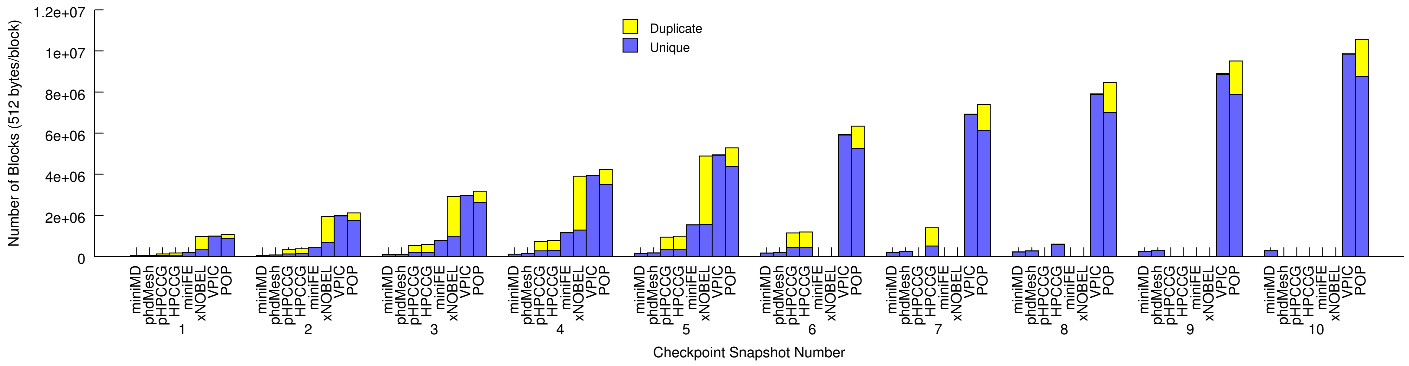


Fig. 5: Cumulative block distribution.

phdMesh application dumps change significantly between each checkpoint snapshot and are, hence, a bad candidate for hash-based incremental checkpointing. This can be corroborated by the very low deduplication ratio of these applications as shown in Figure 4. In other words, the total sizes of checkpoints and their sizes after eliminating duplicate data using varying block sizes remains nearly constant. Figure 4 compares the sizes across a series of checkpoint dumps resulting from a completed run. Parts of HPCCG, pHPCCG and xNOBEL show little to no modification at all. POP writes heavily to restricted regions in its dump while not touching the other regions at all. Figure 5 shows the cumulative ratio of unique blocks against duplicate blocks for all of the applications.

While a smaller block size results in a better deduplication ratio, it takes longer to compute the hash for the block and further reduces the write throughput considerably. From Figure 4, we see that there was no observable difference between block sizes 4K and 8K, except in the xNOBEL case. Although adaptive chunking is more expensive in terms of computation, it strikes a balance between a good deduplication ratio and a higher distribution of bigger block sizes as depicted in Figure 6. Applications with minimal redundancies show a block size distribution with many smaller blocks and few or no large blocks. It can be concluded that these applications deal with smaller, temporal structures that change frequently.

In summary, the benefits of deduplication for an application does not only depend on its checkpoint I/O behavior, but also largely depends on the nature of its problem too. Contrary to the popular belief that scientific applications do not benefit from deduplication at all, we see a benefit of 0 to 40% within a small sample of representative applications. Furthermore, we see that adaptive chunking can result in good deduplication ratio with bigger block sizes resulting in an overall increase in checkpoint throughput.

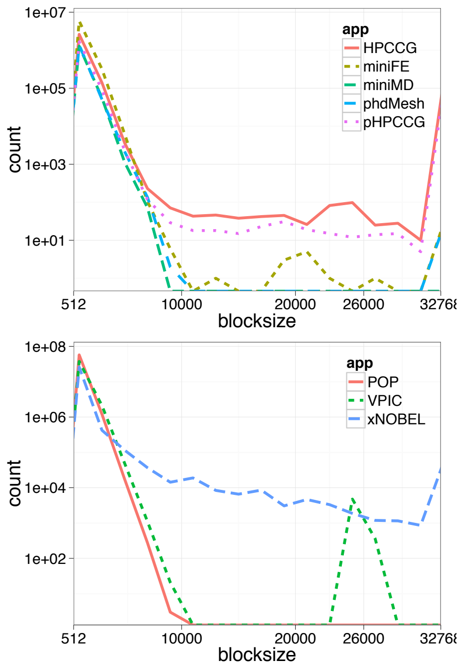


Fig. 6: Distribution of block sizes for adaptive chunking

V. Performance Evaluation

The multi-level design of `cento` matches closely with the design of upcoming hierarchical storage architectures – disk-based parallel storage systems augmented with a tier of solid-state burst buffers. The natural aggregation points provided by the I/O nodes and the increasing performance of manycore processors open up possibilities for *in-transit* data reduction and compression. The performance gains for the `cento` checkpoint file system derive primarily from the following factors: i) Write-aggregation of checkpoint data ii) Data reduction using deduplication and iii) Log-structured writes to the intermediary storage and the parallel file system.

We evaluate the performance of checkpointing to both a parallel file system and solid-state burst buffers separately, but do not include the hybrid model due to unavailability of such a setup. The performance evaluation was performed for a fixed workload with and without deduplication, varying the number of writers. The resulting checkpoint patterns are denoted by the following terminology: An $N-N$ checkpoint pattern is the one in which every process writes to its own unique file whereas an $N-M$ checkpoint pattern involves write-aggregation of the data resulting in M files being written out to the storage. Our

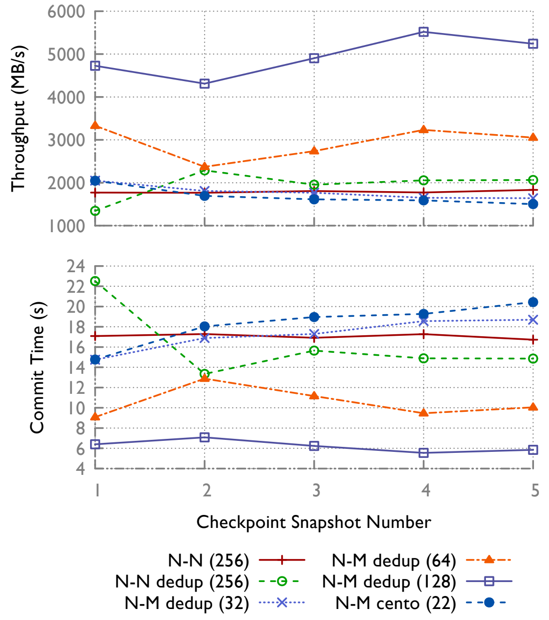


Fig. 7: Performance of checkpointing to the PFS

preliminary evaluation shows a comparable performance gain at modest scales.

A. Checkpointing to the Parallel File System

1) Testbed Configuration: The testbed cluster used for our performance evaluation of checkpointing to a parallel file system is a 154 node, dual-socket based 8-core Intel Sandy Bridge system (Xeon E5-2670 2.60GHz) with 32 GB RAM per node. The cluster uses a QLogic QDR Infiniband interconnect in a full bisectional bandwidth fat-tree. The parallel file system is accessed through the I/O gateway nodes that route the I/O traffic from the Infiniband network to the 10GigE fabric where the Panasas parallel filesystem is connected. The Panasas file system was used through a single volume, engaging a single director blade (for metadata service and control) and 340 OSDs to provide the storage capacity.

2) Experimental Setup: For these tests, we checkpointed the xNOBEL application running with 256 processors, *N-N (256)*. The commit time and throughput in writing 5 checkpoint snapshots, each roughly 32GB in size for a total of 160GB, was measured. To determine the overhead of `cento`, we ran microbenchmarks that capture the I/O behavior of our file system. The *N-N dedup (256)* test, assuming a deduplication ratio of 0.67 from Figure 4, wrote 67% of the total checkpoint data to the parallel file system. The *N-M dedup (M)* tests involved writing the same amount of deduplicated checkpoint data with *M* writers (where *N* >> *M*). Finally, the *N-M* `cento` test was run with 256 processors and 22 MDSS server instances.

3) Results: The *N-N* write pattern is slow at higher scales due entirely to the metadata overhead involved in parallel open, create and close requests. At a smaller scale, the regular *N-N* write pattern has a comparably good performance. From Figure 7, we see that `cento` performs as good as, and at times better than, the baseline *N-N* case. The *N-M dedup* test achieves much higher throughput and lower latencies due to data reduction and lower metadata overhead. An increase in throughput and decrease in commit times was observed when the number of writers (*M*) were increased with the best throughput and commit time at $M = 128$. The comparable performance of *N-M dedup (32)* with *N-M* `cento` illustrates the low overheads incurred by `cento`. In addition to the observed increase in performance, considerable savings in storage were seen. Out of the average 7447MB of checkpoint data to be written by each writer, only 4729MB of deduplicated data was written per writer which further compressed to an average of mere 13.46MB of checkpoint data per writer.

B. Checkpointing to Solid-State Disks

1) Testbed Configuration: The tests for evaluating the performance of checkpointing to solid-state disks were run on a testbed cluster with 2 gateway nodes, 10 HP XW8200 nodes and 2 burst-buffer (SSD) nodes. The compute nodes were 2-core Intel Xeon 3.4GHz processors with 8GB RAM. The burst-buffer nodes were equipped with 8-socket 10-core Intel Xeon (E7-8800 2.40GHz) processor, a total of 128GB RAM per node and an enterprise-grade XceedIOPS SAS SSD with sustained sequential read/write performance of up to 250/230 MB/s and up to 26000/20000 random read/write IOPS.

2) Experimental Setup: The HPCCG application with system-level BLCR checkpoint dumps was used for these tests. Each checkpoint snapshot was roughly 1GB in size for a total of 9 checkpoints equaling 9GB of total write size. We ran it with 36 and 10 processors, *N-N (36)* and *N-N (10)* respectively. The terminology of the tests' name is similar to the one described previously. The *N-M* `cento` test was run with 36 processors and 2 MDSS server instances.

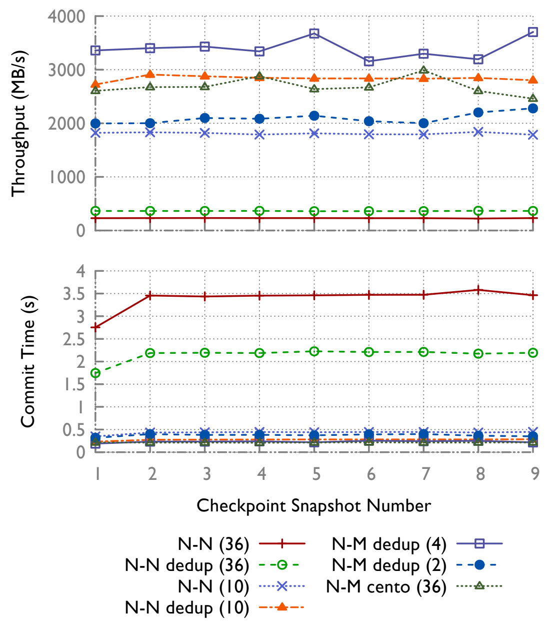


Fig. 8: Performance of checkpointing to the SSDs

3) Results: Solid-state disks perform much better with sequential read/write accesses than random read/write accesses. By the virtue of their design, they are inherently suited for log-structured access patterns. In Figure 8, we see that higher throughput and lower commit latencies are obtained with fewer writers. The best performance was achieved with 4 writers. `cento` performs a notch better than the *N-M dedup (2)* test due to write aggregation and the log-structured contiguous write pattern. The memory-mapped *arenas* in `cento` are flushed on a `sync` request from the clients on closing the file.

VI. Future Work

Future work includes evaluating the performance of `cento` at much larger scales. We also plan to test the performance on a hierarchical storage architecture with an external parallel storage system and a tier of solid-state disks. Faster hash computating using GPUs, FPGAs or the newer manycore processors like the Intel MIC architecture would increase the performance significantly. We also intend to extend the file system to support *N-1* workloads by maintaining the relevant metadata (like offsets and sizes). The performance of our implementation could be vastly improved through caching and better memory management. Use of Bloom filters to maintain a block cache would result in lower block commit latencies. We further plan to investigate the role of resilience factor of different storage levels through fault simulations. A better model based on networks of queues can better predict the non-linearity due to contention of I/O requests at higher scales. Finally, we would also like to investigate how our file system could fit under a I/O forwarding layer and offer the proposed benefits to applications.

VII. Conclusion

Hierarchical storage architectures with node-level *burst buffers* present more opportunities of doing in-situ or post-process deduplication thereby resulting in data and metadata overhead reduction. For exascale systems, this would result in much lower checkpoint commit latencies and increased machine efficiency. In fact, the checkpoint demands and application-perceived throughput at these scales would have to be met by decreasing the checkpoint overheads. Data deduplication through a content-addressable parallel file system can achieve a size reduction comparable to compression and also offer additional performance benefits. It can help transform the I/O pattern to lessen some of the performance bottlenecks exacerbated at scale. We observed upto 40% space reductions in checkpoint data and 5 to 20% increase in write performance at relatively modest scales, validating the benefits of deduplication for checkpoint/restart at extreme scale.

Acknowledgment

This work was supported, in part, by a grant from the Lilly Endowment; National Science Foundation NSF ANI-0330620 and EIA-0202048; and U.S. Department of Energy DE-FC02-06ER25750. This publication has been assigned the LANL identifier LA-UR-12-23731.

References

[1] F. Cappello, A. Geist, B. Gropp, L. V. Kal, B. Kramer, and M. Snir, "Toward Exascale Resilience." *IJHPCA*, vol. 23, no. 4, pp. 374–388, 2009.

[2] V. Sarkar *et al.*, "ExaScale Software Study: Software Challenges in Extreme Scale Systems," *DARPA Information Processing Techniques Office, Washington DC.*, vol. 14, p. 159, 2009. [Online]. Available: http://users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/ECSS%20report

[3] K. B. Ferreira, *Keeping Checkpoint/Restart Viable for Exascale Systems*. Thesis/Dissertation, Universoty of New Mexico, Jul. 2011.

[4] D. Ibtesham, D. C. Arnold, K. B. Ferreira, and P. G. Bridges, *On the Viability of Checkpoint Compression for Extreme-Scale Fault Tolerance*. 4th Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids in conjunction with the 17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011), Sep. 2011.

[5] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *Proceedings of the 18th annual international conference on Supercomputing*, ser. ICS '04. New York, NY, USA: ACM, 2004, pp. 277–286.

[6] 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*, ser. EuroMPI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 272–281.

[7] N. Naksinehaboon, Y. Liu, C. B. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott, "Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, ser. CCGRID '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 783–788.

[8] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis, "Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 9–.

[9] J. Plank, J. Xu, and R. Netzer, "Compressed differences: An algorithm for fast incremental checkpointing," University of Tennessee, Tech. Rep. CS-95-302, 1995.

[10] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman, "Functional Partitioning to Optimize End-to-End Performance on Many-core Architectures," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–12.

[11] N. Liu, C. Cope, P. Carns, C. Carothers, R. Ross, and G. Grider, "On the Role of Burst Buffers in Leadership-class Storage Systems," in *Proceedings of 28th IEEE MSST/SNAPI conference (MSST 2012)*, Pacific Grove, CA, 2012.

[12] 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*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[13] 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.*, vol. 8, no. 2, pp. 6:1–6:29, Jun. 2011.

[14] S. Al-Kiswany, M. Ripeanu, S. S. Vazhkudai, and A. Gharaibeh, "stdchk: A Checkpoint Storage System for Desktop Grid Computing," in *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, ser. ICDCS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 613–624.

[15] B. Nicolae and F. Cappello, "BlobCR: efficient checkpoint-restart for HPC applications on IaaS clouds using virtual disk image snapshots," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 34:1–34:12.

[16] Y. Li and Z. Lan, "A fast restart mechanism for checkpoint/recovery protocols in networked environments," in *DSN*, 2008, pp. 217–226.

[17] J. Hursey and A. Lumsdaine, "A Composable Runtime Recovery Policy Framework Supporting Resilient HPC Applications," Indiana University, Bloomington, Indiana, USA, Tech. Rep. TR686, August 2010.

[18] D. Marques, G. Bronevetsky, R. Fernandes, K. Pingali, and P. Stodghil, "Optimizing Checkpoint Sizes in the C3 System," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 10 - Volume 11*, ser. IPDPS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 226.1–.

[19] J. C. Sancho, F. Petrini, G. Johnson, J. Fernández, and E. Frachtenberg, "On the Feasibility of Incremental Checkpointing for Scientific Computing," in *IPDPS*, 2004.

[20] L. B. Costa, S. Al-Kiswany, R. V. Lopes, and M. Ripeanu, "Assessing Data Deduplication trade-offs from an Energy Perspective," in *ERSS: Workshop on Energy Consumption and Reliability of Storage Systems*, Jul. 2011.

[21] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, ser. FAST'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 18:1–18:14.

[22] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 174–187, Oct. 2001. [Online]. Available: http://doi.acm.org/10.1145/502059.502052

[23] S. Quinlan and S. Dorward, "Venti: A New Approach to Archival Storage," in *FAST*, D. D. E. Long, Ed. USENIX, 2002, pp. 89–101.

[24] S. C. Rhea, R. Cox, and A. Pesterev, "Fast, Inexpensive Content-Addressed Storage in Foundation," in *USENIX Annual Technical Conference*, R. Isaacs and Y. Zhou, Eds. USENIX Association, 2008, pp. 143–156.

[25] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra, "HydraFS: a high-throughput file system for the HYDRAstor content-addressable storage system," in *Proceedings of the 8th USENIX conference on File and storage technologies*, ser. FAST'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 17–17. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855511.1855528

[26] X. Ouyang, R. Rajachandrasekar, X. Besseron, H. Wang, J. Huang, and D. K. Panda, "CRFS: A Lightweight User-Level Filesystem for Generic Checkpoint/Restart," in *Proceedings of the 2011 International Conference on Parallel Processing*, ser. ICPP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 375–384. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2011.85

[27] 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*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 21:1–21:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654081

[28] P. Nowoczynski, N. Stone, J. Yanovich, and J. Sommerfield, "Zest Checkpoint storage system for large supercomputers," in *3rd Petascale Data Storage Workshop*, Nov. 2008, pp. 1–5. [Online]. Available: http://dx.doi.org/10.1109/PDSW.2008.4811883

[29] P. Nath, B. Urgaonkar, and A. Sivasubramaniam, "Evaluating the usefulness of content addressable storage for high-performance data intensive applications," in *Proceedings of the 17th international symposium on High performance distributed computing*, ser. HPDC '08. New York, NY, USA: ACM, 2008, pp. 35–44. [Online]. Available: http://doi.acm.org/10.1145/1383422.1383428

[30] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 from Bell Labs," *EUUG Newsletter*, vol. 10, no. 3, pp. 2–11, Autumn 1990. [Online]. Available: http://plan9.bell-labs.com/cm/cs/cstr/158b.ps.gz

[31] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. Bressoud, and A. Perrig, "Opportunistic Use of Content Addressable Storage for Distributed File Systems," in *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, TX, Jun. 2003, pp. 127–140. [Online]. Available: http://www.soe.ucsc.edu/~{}carlosm/Papers/tolia-usenix03.pdf

[32] 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," Sandia National Laboratory, Technical Report SAND2009-5574, 2009.

[33] P. W. Jones, P. H. Worley, Y. Yoshida, J. B. White, III, and J. Levesque, "Practical performance portability in the Parallel Ocean Program (POP): Research Articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 10, pp. 1317–1327, Aug. 2005. [Online]. Available: http://dx.doi.org/10.1002/cpe.v17:10

[34] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation," *Physics of Plasmas*, vol. 15, no. 5, p. 055703, 2008. [Online]. Available: http://link.aip.org/link/PHPAEN/v15/i5/p055703/s1&Agg=doi

[35] Los Alamos National Laboratory, "xNobel benchmark," 2009. [Online]. Available: LA-CC09-61