Aggregate Memory as an Intermediate Checkpoint Storage Device

Samer Al-Kiswany [*], Matei Ripeanu [*], Sudharshan S. Vazhkudai [†]
[*] The University of British Columbia, {samera, matei}@ece.ubc.ca
[†] Oak Ridge National Laboratory, vazhkudaiss@ornl.gov

Abstract. Applications that generate bursty I/O load, like checkpointing, require additional support to perform efficiently on next generation petascale supercomputers. Tens of thousands of processors, generating terabytes of snapshot data at once at each timestep, can easily overwhelm a storage system. Further, even at the current peak I/O bandwidth rates, offered by parallel file system deployments at leadership class facilities, an application is likely to spend a significant portion of its runtime checkpointing. To address these issues, we propose a checkpoint storage device, built from memory resources, that acts as an intermediary to the central parallel file system. Our system comprises of a dedicated manager that aggregates memory resources from processors (benefactors) and makes it available as a collective space for checkpointing clients, using a standard POSIX file system interface. We argue that such a system has the potential to alleviate the I/O bandwidth bottleneck for bursty I/O operations like checkpointing by aggregating memory and interprocessor bandwidth.

I. INTRODUCTION

The Challenge: The advent of PetaFlop (PF) supercomputers will pose fundamental challenges to scalable, fault-tolerant HPC. At current sub-petaflop levels, applications use tens of thousands of compute cores (Table 1) for hours or days on end. In such settings, checkpointing is an indispensable failure impact mitigation technique.

Typically, a parallel job checkpoints its state periodically during the course of its run in an attempt to strike a balance between the cost of recovering the application state in case of failure and checkpointing overheads. A high checkpoint frequency results in a low amount of computation that needs to be repeated in case of a failure; it generates, however, more data, stressing the I/O system. Additionally checkpointing is used for other scenarios such as debugging and auditing where the checkpoint frequency is predetermined.

Consider a 10,000 core job on the Oak Ridge National Laboratory (ORNL) Jaguar system which has 2 GB of memory per core and no local disk. Assume further that the job runs for 12 hours and checkpoints every half hour. For this job, in the worst case, when all of the memory per core is saved as state information, 500TB of checkpoint data is produced during a run. Such data volumes can overwhelm any storage system. As we scale to PF systems, this problem is likely to get acute with the increase in the number of computing cores and the amount of data to be saved at each timestep.

Modern parallel file systems (e.g., Lustre [1], PVFS [2] and GPFS [3]) attempt to cope with such intense I/O demands by building atop thousands of I/O servers and tens of thousands of disks. For example, the Jaguar file system (based on Lustre) currently offers an aggregate peak I/O bandwidth of around 55GB/s and plans to scale to around 240 GB/s as the system reaches one PF [4]. On this system, to checkpoint 20TB of data each timestep (10,000 cores * 2 GB per core) at 55 GB/s would consume 6 minutes. Checkpointing every half hour would require spending 20% of the run time just to prepare against failure [5].

A survey of DOE applications [6] suggests that most applications require a sustained 1GB/s I/O throughput for every TeraFlop of peak computing performance. Thus, a PF computer will require 1TB/sec of I/O bandwidth, which is still two orders of magnitude lower than the theoretical estimate of 100TB/s that a balanced PF machine should provide according to Jim Gray [7].

Compounding the problem is the fact that, historically, I/O bandwidth has not scaled with processor frequencies and that when the I/O channel is shared across multiple applications, the effective throughput achieved by any given application significantly deteriorates.

This situation calls for novel techniques to address checkpointing in PF systems, in a way that achieves the desired fault-tolerance level and yet does not compromise the overall system throughput.

Table 1: Number of cores and memory size for recent PF systems.

System	#Cores	Aggregate Memory (TB)	Top500 Rank
RoadRunner (LANL)	122,400	98	1
BlueGene/L (LLNL)	212,992	69	2
BlueGene/P (ANL)	163,840	80	3
Ranger (TACC)	62,976	123	4
Jaguar Cray XT4 (ORNL)	31,328	62	5

An Opportunity: As a supercomputer's cross-section bandwidth is generally higher than its bandwidth to the storage system, harnessing memory resources from a dedicated processor pool to build a memory-based checkpoint storage system (or simply to provide a large buffer to temporarily receive the bursty checkpoint workload) can significantly improve the checkpointing performance as perceived by an application.

For example, each processing element (PE) in the Jaguar Cray XT4 machine is a quadcore connected to its memory using a 6.4 GB/s HyperTransport and is in turn connected to six other PEs using a 3D-torus router (switching speed of 45.6 GB/s). Consequently, Jaguar can offer tens—even hundreds—of TB/s of aggregate memory bandwidth. Table 2 depicts example machine

configurations from Cray [8].

Previous checkpoint solutions that target the supercomputing space [9, 10] have not harnessed the possibility of aggregating a dedicated set of memory resources to accelerate the checkpointing operation and have simply resorted to using the memory available locally.

Table 2: Sample Cray XT3 machine configurations [8].

Cabinets	#cores	Memory (TB)	Aggregate Memory BW(TB/s)	Memory BW/PE (GB/s)	Bisection BW(TB/s)
6	548	4.3	2.5	4.5	0.7
24	2,260	17.7	14.5	6.4	2.2
96	9,108	71.2	58.3	6.4	5.8
320	30,508	239	196	6.4	11.7

Contributions: This article proposes a framework to aggregate memory from dedicated PEs within extreme-scale supercomputers. Such an aggregate memory-based device can be used for in-memory checkpointing or, we argue, as an intermediate staging ground to smooth out bursty workloads such as checkpointing. This technique will accelerate application's data handover and, we speculate, will enable apparent application checkpoint rates of hundreds of GB/s or even a few TB/s. We have further designed a scheme to transparently *"lazily push"* or *drain* from the aggregate memory device to a shared parallel file system as a means both to make room for new incoming data and to reliably store it on stable storage.

II. Feasibility Discussion

The following reasons make introducing an aggregate, memory-based intermediary device to support checkpointing feasible.

It is common in HPC job submission systems for jobs to oversubscribe for processors to prepare for failure. For example, depending on the failure rate of the machine, a particular job might ask for 12,000 cores instead of the 10,000 cores that it actually needs. The remaining cores are used for failing over processes. One can imagine, an aggregated memory device built out of such pools. This approach has the advantage that it uses the application's own over subscribed processor allocation. Depending on the charging scheme and the constraints of each specific system, applications can factor such pools into their requests. For example, often applications are charged proportionally with the *number of processors x walltime* used. As turnaround time is directly dependent on the checkpointing performance, more processors do mean higher charges but, potentially, for shorter time. Additionally, the application may decide what to optimize: the turnaround time or to its costs.

Alternatively, if the HPC center observes that I/O bandwidth bottleneck in checkpointing significantly hampers its serviceability, a dedicated checkpoint device based on harnessing memory resources from dedicated processor pools can be installed as a system-wide option. One can even extend this further and consider better provisioning of the supercomputer by way of providing processor-local flash memory, which can then be aggregated to provide a dedicated checkpointing device.

Aggregation of memory resources is also made feasible as modern supercomputers are equipped with tens—or even hundreds—of TBs of memory and powerful interconnects (Table 2). There has been little effort to use these resources collectively and in concert with the storage system. While the HPC center's shared filesystem is crowded and struggling to meet user I/O bandwidth demands, vast amounts of residual bandwidth across these resources remain untapped.

Finally, a recent survey of Tier 1 applications for the Jaguar system [6] that included application codes from Fusion (GTC), Combustion (S3D), Climate (POP) and Astrophysics (Chimera) suggests that most applications seldom use all the available memory per core and there is a significant amount of unused residual memory. In such cases, local PE's memory buffers can be used in addition to the dedicated aggregate memory-based checkpointing device to temporarily store the PE's snapshot data.

III. Requirements for a Memory-Aggregation System that Supports Bursty IO

In brief, we plan to aggregate memory form donor nodes and use it as a large buffer between an application that produces bursty write traffic (e.g., checkpointing) and a storage system that cannot accommodate these bursts without slowing down the application. This architecture offers a number of tuning knobs:

- Relatively straightforward benchmarking to evaluate the size and the frequency of checkpoint operations can be used to approximate the size of the burst generated by applications. This, in turn, offers a first approximation for size of the aggregated memory required to completely decouple the application and the slower I/O channel. However, since data can be simultaneously drained to stable storage a smaller aggregate memory will suffice. For a more precise estimate that takes the above factor into account we additionally need to evaluate application's write throughput to the aggregate memory and the I/O throughput to stable storage.
- The system can be configured to persist data form the aggregated buffer to stable storage at every N checkpoint operations (or never) rather than at each checkpoint operation. Such a configuration would still make checkpoint images temporarily available (e.g., for application debugging and monitoring), while preserving controlled reliability of the application.

While checkpointing through an aggregate memory device has the potential to improve the throughput of the I/O system perceived by applications, it also poses several constraints and challenges.

- *Data transfer to stable storage:* Snapshot data stored in memory needs to be drained to stable storage (generally a shared parallel file system). This should be performed in an asynchronous fashion so that the application perceived throughput for writing a checkpoint does not suffer a significant impact.
- *Automated management of transient checkpoint data:* Checkpoint data is transient in nature and is usually not used beyond the lifetime of the run except for debugging purposes. An application should be able to specify the longevity of the snapshots and inform the storage so that this can automate data management and optimize space usage accordingly.
- *Scalability:* The aggregation system should be able to amass memory resources from a large number of processors and deal with parallel writes from numerous client PEs.
- *Transparency*: The proposed system should not require application support: applications should be oblivious of the use of the aggregated memory buffer.

IV. ARCHITECTURE

Starting from MosaStore storage system codebase [11] we aim to build our aggregate memory-based device. MosaStore is a storage system that aggregates disk space contributions from connected machines. Our basic architecture comprises *benefactor processes*, running on each PE that contributes memory to the system and a *manager* that aggregates these memory contributions into a collective space.

Benefactor PEs can come and go depending on volatility of the processor. To accommodate this transient behavior, benefactors register with the manager using a *soft-state registration protocol*: initially they declare their intention to participate in memory aggregation, then, every 30s they update the manager about their status. This way the manager knows which benefactor processes are alive and can approximate the free space at each benefactor.

The manager keeps track of the memory contributions from the benefactors and helps presenting a unified storage space to client PEs. The manager maintains metadata regarding individual benefactor contributions, each benefactor's status and potentially some history about the benefactor. When a client contacts the manager, the file to write is divided by the system in equally sized chunks. The manager computes a striping plan, determining a set of benefactors to send chunks to, and a benefactor mapping. One striping policy we have implemented is to sort the benefactors on available memory space and then perform a round-robin striping across a top subset (stripe width) of them. Once clients obtain a striping 'map', they interact with the benefactors directly to send the chunks to benefactors. Since the size of the checkpoint data, at any timestep, is not known a priori, the client will need to adapt to situations such as an overrun of the initial width of benefactors. In such cases, the client contacts the manager again to readjust the width. Once the entire checkpoint operation is completed, the client commits the map to the manager, indicating a successful operation.

Incremental Checkpointing: In our previous work [12], we have built techniques to detect similarity between successive checkpoint images in order to minimize the amount of data written during each timestep. A simple, yet elegant, strategy is to compute a hash of the chunks and to store them as metadata at the manager at each timestep, t. At $(t+1)^{th}$ timestep, the hashes for chunks of the checkpoint image are compared against the stored metadata to detect similarity. If the chunk hashes are similar, then the new chunk in question need not be written but only a reference to the old chunk needs to be retrieved.

We experimented with this technique in the context of checkpointing in a desktop grid environment. For an aggregate memory based checkpointing device, a similar technique can be adopted. The hash comparison is a metadata operation that can be performed between the client and the manager and does not involve the chunks stored at the benefactors. Consequently, the snapshot data from timestep, t, need not be maintained in the aggregate memory during timestep, t+1.

Draining data to a central parallel file system (PFS). The aggregate memory buffer needs to create room for incoming snapshot data. Thus the data in aggregate memory needs to be saved in stable storage. Pushing the chunks in benefactor memory to a central PFS (e.g., Lustre, PVFS, GPFS) is coordinated independently by each benefactor (using a FIFO order) and decoupled from accepting application data.

The drained snapshot data needs to be saved in a way that checkpoints can be easily recovered in case a restart is needed. To this end, benefactor nodes write data chunks as independent files but encode, in the file name, all the chunk related information needed to restore the original checkpoint file by coalescing the files corresponding to individual chunks. We preserve the space savings enabled by similarity detection by using hard links.

Traditional file system API. To provide complete client-side transparency we use FUSE [13], a Linux kernel module. It provides the callback mechanism enabling us to offer client access to this aggregate memory system as a user-level file system. Once a

FUSE volume is mounted, all file system calls to the mount point are forwarded to the FUSE kernel module, which preprocesses and forwards them to our user-level file system callbacks. FUSE is officially merged into the Linux kernel starting with 2.6.14 version, further simplifying adoption of our user space file system.

V. Preliminary Evaluation

We report here preliminary performance evaluation results obtained on a small cluster. While these results cannot support our claim that the proposed solution will scale to petascale machines they do confirm that, on smaller deployments, our prototype performs as expected and achieves the desired performance level. We hope to present a more thorough evaluation in case this paper is accepted,.

Additionally, we have installed our aggregate memory prototype on the BlueGene/P supercomputer at Argonne National Laboratory as part of a larger software suite. Therein, our system is used to aggregate memory from multiple nodes for a mostly-read data flow, in the opposite of the write-only data flow checkpointing generates. Zhang et al. [14] report results for this setup where our system was deployed at the BlueGene/P pSet level (that is aggregating memory from 64 nodes) on over 2,000 pSets.

A. The Platform

We evaluated our prototype using a range of limited scale benchmarks. We used a 22-node cluster. Each node has an Intel Quad Core Xeon 2.33GHz Processor, 4GB of RAM, and 1Gbps Ethernet card. For all configurations, we report averages and standard deviations based on 20 runs.

We first evaluated the performance and the overhead of each individual component of our test platform. Unlike in a supercomputer our cluster nodes have local disks. The sustained write throughput on a local disk with write caches enabled was 93.2MB/s. The peak node-to-node network bandwidth, measured using iPerf [15] is 117.38 MB/s.

B. Write Throughput to Aggregate Memory

Our implementation uses write buffers at the client to decouple the application from the actual data transfer to benefactor nodes. Therefore, we define two performance metrics. First, *the observed application bandwidth* (OAB) is the write bandwidth observed by the application: the file size divided by the time interval between the application-level *open()* and *close()* system calls. Second, *the achieved storage bandwidth* (ASB) uses the time interval between file open() and until the data is entirely transferred off the processing element (i.e., all remote I/O operations have completed). Three main factors affect the performance of the write operation: the client-side buffer size, the memory size offered at each benefactor, and the number of benefactors used in the write operation.

In this preliminary evaluation we performed a number of experiments to estimate the effect of each of these factors and we evaluate the maximum throughput at which the aggregated cache can receive data.

First, we evaluate the performance metrics form the viewpoint of a single client, working in isolation. Figure 1 and Figure 2, show the effect of client-side buffer size on the observed application bandwidth (OAB) and achieved storage bandwidth (ASB), respectively, while varying the stripe width from one to eight benefactors.

In these experiments the client successively writes 1GB files to the aggregated memory device and the data is lazily saved to local disks (akin to draining to a central file system). Each benefactor offers 1GB of memory for aggregation.

The experiments show that the client-side interface's buffer size directly affects the observed application bandwidth (OAB) and does not have a noticeable effect on achieved storage bandwidth (ASB). Two contributing nodes with 1Gbps NICs can saturate the client 1Gbps link, achieving around 110MB/s of write throughput, even when only 128MB client-side buffer is used at the client.

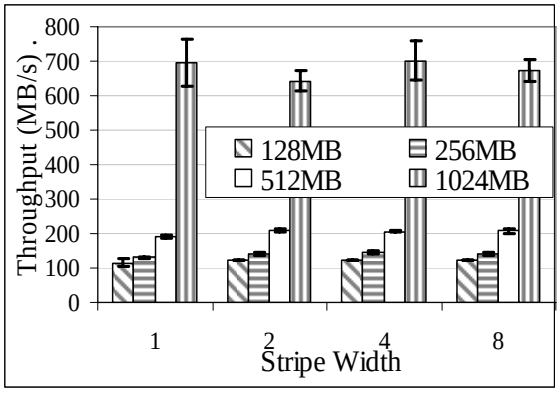

Figure 1. The average observed application bandwidth while varying the client-side buffer size.

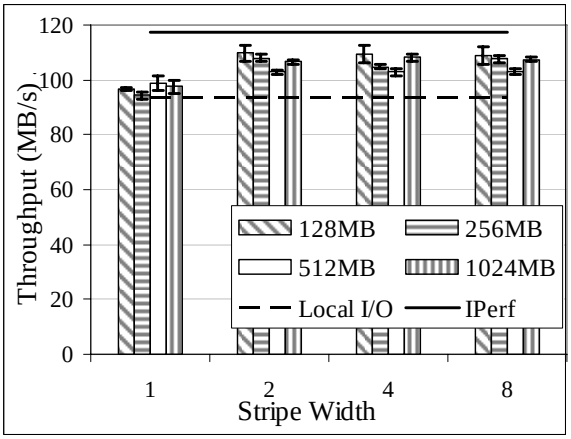

Figure 2. The average ASB while varying the file system buffer size.

Second, we asses the aggregate throughput multiple clients can achieve when writing concurrently to the aggregated memory device. In this experiment, seven clients generate load as follows: each client writes, back-to-back 20 files of 1GB each, amounting to around 140 GB of data. To ramp-up the load, clients start at 10s intervals. Figure 3, presents the aggregate client throughput. We observe a sustained peak throughput of about 560MB/s. This indicates that the aggregate memory device may be able to receive an intense workload as we expect.

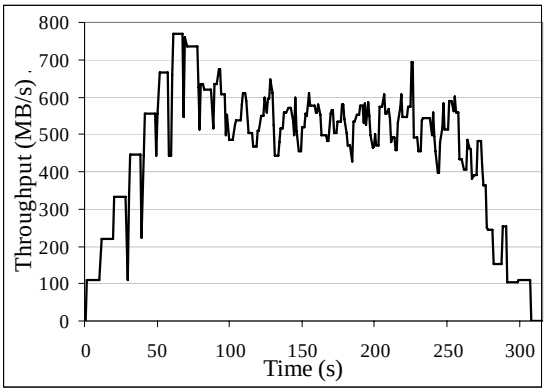

Figure 3. Throughput at larger scale: seven clients generate a synthetic workload to stress an aggregate memory device pool supported by 14 benefactor nodes.

VI. Summary and Future Work

This paper presents an architecture for an intermediate storage device, built by aggregating memory resources, as means to accelerate HPC applications that generate bursty I/O workloads (e.g., checkpointing). Our system preserves the standard POSIX I/O interface and transparently interspaces the buffer obtained by aggregating memory between the application and a shared, parallel file system. As the intermediate memory device not only aggregates memory but also interprocessor bandwidth, the checkpointing clients can store data at higher data rates. To strike a balance between performance and fault tolerance, our system also provides an asynchronous mechanism to drain the snapshot data to stable storage, from where the data can be retrieved if needed. We have conducted a preliminary evaluation of our proof-of-principle prototype. Our results indicate that the aggregate memory device is able to deliver, as expected, a high-write throughput.

We are working towards large-scale deployments on ORNL's Jaguar machine: aggregating at least a few thousand PEs and scaling to an equal number of clients. Next steps include optimized management of the drained data on the central storage system, autonomic sizing of the aggregate buffer, enabling seamless restarts of applications using the snapshot data written through our device, and harnessing residual memory on the checkpointing client's own PE.

VII. Acknowledgments

This research was sponsored in part by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL), managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725. It was also supported by grants from the National Science and Engineering Research Council of Canada (NSERC) and the Canadian Foundation for Innovation (CFI).

VIII. References

1. *Lustre website* [cited 2008; Available from: http://www.lustre.org/.
2. Philip H. Carns, Walter B. Ligon-III, Robert B. Ross, and Rajeev Thakur. *PVFS: A Parallel File System for Linux Clusters*. in *4th Annual Linux Showcase and Conference*. 2000. Atlanta, GA.
3. Frank Schmuck and Roger Haskin. *GPFS: A Shared-Disk File System for Large Computing Clusters*. in *1st USENIX Conference on File and Storage Technologies (FAST'02)*. 2002.
4. Buddy Bland. *Leadership Computing Facility (LCF) Roadmap*. 2007 [cited 2008; Available from: www.csm.ornl.gov/SC2007/pres/Bland_LCFRoadmap_Judy/Bland_LCFRoadmap_SC07.ppt.
5. Garth Gibson, Bianca Schroeder, Joan Digney, Volume on Software Enabling Technologies for Petascale, and Science., *Failure Tolerance in Petascale Computers.* CTWatch Quarterly, Volume on Software Enabling Technologies for Petascale Science, 2007. **3**(4).
6. *Application Requirements & Objectives for Petascale Systems* in *HPCwire*. 2008.
7. Gordon Bell, Jim Gray, and Alex Szalay, *Petascale computational systems.* IEEE Computer, 2006. **39**(1): p. 110-112.
8. *Cray XT3 datasheet.* [cited 2008; Available from: www.cray.com.
9. James S. Plank, Kai Li, and Michael A. Puening., *Diskless Checkpointing.* IEEE Transactions on Parallel and Distributed Systems, 1998. **9**(10): p. 972-986.
10. L.M. Silva and J.G. Silva, *Using two-level stable storage for efficient checkpointing.* IEE Proceedings - Software, 1998. **145**(6): p. 198-202.
11. *MosaStore website.* 2008 [cited 2008; Available from: www.mosastore.com.
12. Samer Al-Kiswany, Matei Ripeanu, Sudharshan Vazhkudai, and Abdullah Gharaibeh. *stdchk: A Checkpoint Storage System for Desktop Grid Computing*. in *International Conference on Distributed Computing Systems (ICDCS '08)*. 2008. Beijing, China.
13. *FUSE, Filesystem in Userspace*. [cited 2008; Available from: http://fuse.sourceforge.net/.
14. Zhao Zhang, et al. *Design and Evaluation of a Collective I/O Model for Loosely-coupled Petascale Programming*. in *Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*. 2008.
15. *Iperf website.* [cited 2008 April]; 2.0.2:[Available from: http://dast.nlanr.net/Projects/Iperf/.