2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Checkpointing Orchestration: Toward a Scalable HPC Fault-Tolerant Environment

Hui Jin, Tao Ke
Illinois Institute of Technology
Chicago, IL 60616
{hjin6, tke1}@iit.edu

Yong Chen
Texas Tech University
Lubbock, TX 79409
yong.chen@ttu.edu

Xian-He Sun
Illinois Institute of Technology
Chicago, IL 60616
sun@cs.iit.edu

***Abstract*- Checkpointing is widely used in technical computing. However, the overhead of checkpointing is a subject of increasing in concern in recent years, especially for large-scale parallel computer systems. In these systems, checkpointing generates a huge number of concurrent I/O writes. The burst of writes plus the worsening I/O-wall problem often leads to network and I/O congestion, and makes the overall system performance painfully slow. Recognizing contention as a dominant performance factor, in this paper we propose a systematic approach named *checkpointing orchestration* to reduce write contention, which combines the marshaling of concurrent checkpoint requests and the adopting of vertical data access in coordination. A prototype of the proposed checkpointing orchestration approach has been implemented at the system-level under Open MPI over the PVFS2 file system. Extensive experiments based on NPB benchmarks have been conducted to verify the design and implementation. Experimental results show that checkpointing orchestration reduced the checkpointing cost at a degree of more than 30%. Checkpointing cost was halved for 4 out of 5 the C class NPB benchmarks.**

Keywords*- *Fault Tolerance, Checkpointing, Parallel File System

I. INTRODUCTION

High-Performance Computing (HPC) has crossed the petaflop (10^{15} FLOPS) mark and is moving forward to reach the exaflop (10^{18} FLOPS) range [1]. Such ultra-scale computing power comes with increased system complexity and has significantly increased the likelihood of failures. Mean-Time-Between-Failures (MTBF) of a large-scale cluster is predicted to drop to hours [2] [3]. Effective fault tolerance support is a necessity for tomorrow's high-end computing systems.

Checkpointing/Restart (C/R) is a widely used mechanism for fault-tolerant computing, where checkpoints are taken periodically to store a snapshot of the application to a stable storage for resuming the application in case of failures [4]. The C/R mechanism mitigates the damage of failures. However, in the meantime, it introduces considerable overhead, and could degrade the overall performance significantly as well. Oldfield, et al. have shown that checkpointing in a 1-petaflop system can potentially harm the system performance by up to 50% [5]. Whether checkpointing is feasible for the future exascale computing environment is a subject under intensive scrutiny in recent years [6] [7].

Parallel file systems (PFS) such as Lustre [8], GPFS [9] and PVFS2 [10] are widely deployed on modern large-scale systems and serve as the storage of checkpoint images. The parallel file systems are usually deployed on dedicated I/O servers that are separated from compute nodes. Conventional PFS are motivated by mitigating the I/O-wall problem. They are designed to mask the performance gap between memory and disks. However, the flood of emerging data-intensive applications has pushed PFS into new scenarios than what they were originally designed for. As one of the representative data intensive applications, checkpointing in a large-scale system usually issues hundreds of thousands of concurrent I/O requests to the PFS in a burst, which introduces considerable data access contentions. I/O contention is a main factor that influences the checkpointing performance, partially due to the fact that the number of compute nodes is often one to two orders of magnitude greater than the number of I/O servers [11] [12]. The gap is further enlarged with the wide adoption of multi-core/many-core processors that supports multiple computation processes on one compute node. The contention limits the scalability of checkpointing, due to the fact that more processes usually lead to higher contention and more waste in bandwidth. A recent study has revealed that checkpointing scalability is the key factor that limits the scalability of applications for the emerging exascale computing era [13].

A. Impact of I/O Contention

To evaluate the impact of I/O contention on checkpointing performance, we have set up experiments on a cluster of 32 compute nodes and 4 I/O server nodes. We set up PVFS2 on these 4 dedicated I/O servers as checkpointing storage. Open MPI was used as the MPI and checkpointing environment. Detailed configuration of the experimental environment can be found in sub-section IV-A.

We ran one synthetic parallel application with the native checkpointing method that is supported in Open MPI v1.4. We minimized the communication in the synthetic application to remove the influence of coordination on checkpointing and focus on the I/O performance. We kept the overall image size at 16GB, varied the number of processes from 16 to 256, and measured their corresponding checkpointing overheads.

Fig. 1 demonstrates the aggregated bandwidth with different number of processes. Though writing the same amount of checkpoint image to the storage, the I/O bandwidth has a huge variance with different number of processes. We observed that the average bandwidth was halved when the number of processes was increased from 16 to 256, which confirms the scalability limitation of traditional checkpointing.

978-0-7695-4691-9/12 $26.00 © 2012 IEEE
DOI 10.1109/CCGrid.2012.61

IEEE computer society

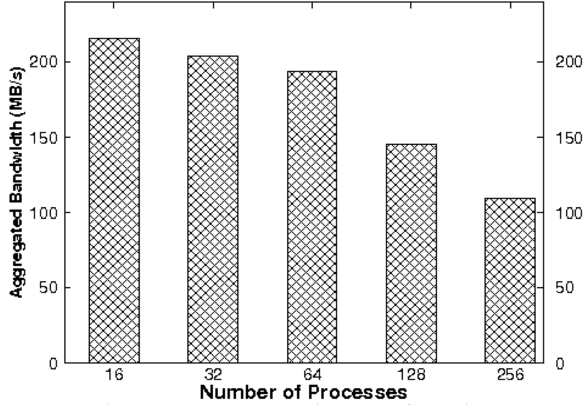

Figure 1. Aggregated Bandwidth under Contentions

B. *Our Solution: Checkpointing Orchestration*

Recognizing I/O contention as a dominant performance factor of checkpointing, we propose a systematic approach to reducing I/O contention of parallel checkpointing in this study. Checkpointing has its own characteristic and is important enough to have its own optimization systems. In this paper, we explore a seamless integration between PFS and checkpointing, and investigate the methodology to adapt both sides for better overall performance.

The main idea of our solution is to *orchestrate the concurrent checkpoints in an optimized and controllable way to minimize the I/O contention*. The contribution of this research is multi-fold:

- We propose a new vertical checkpointing manner to rearrange data layout of the checkpoint files on PFS and reduce the contention.
- We propose a staged checkpointing marshaling technique to serialize the concurrent checkpoints on each compute node and to further reduce the I/O contention
- A prototype of the system-level checkpointing orchestration is implemented under the PVFS2 and Open MPI environments. We have released the software implementation of checkpointing orchestration and made it publicly accessible to the community [14].
- Extensive experiments are conducted with NPB benchmarks to evaluate the proposed checkpointing orchestration approach.

The rest of this paper is organized as follows. We introduce the design of checkpointing orchestration in Section II, followed by the implementation introduced in Section III. Experimental evaluation is presented in Section IV. Section V reviews related work and compares them with our work. Section VI concludes this study and discusses future work.

II. DESIGN OF CHECKPOINTING ORCHESTRATION

A. *Background*

A parallel application can be composed of hundreds of thousands of processes. To complete a checkpointing, the processes first coordinate with one another to get a *consistent state*. Each process will issue a *checkpoint request* after the coordination. We use the term *checkpoint* and checkpoint request interchangeably throughout the paper when context is clear. Checkpoint requests are issued in a burst, which strikingly increases contention and harms the performance. The objective of checkpointing orchestration is to reduce the contention caused by the burst of checkpoint requests. Each checkpoint request generates a sequence of *I/O requests*. The I/O requests are different from each other in size. For example, the I/O requests that store the CPU registers signal handler and pid are usually small, while the I/O requests that store the virtual memory area (VMA) are relatively big. The I/O requests are represented by *VFS requests* at the file system layer.

Checkpointing can be supported at either application-level or system-level. Application-level checkpointing relies on the application to provide its fault tolerant capabilities. This research of checkpointing orchestration aims at enhancing system-level checkpointing, which does not require modification to the source code of applications. However, the idea of checkpointing orchestration is general and can be extended to support application-level checkpointing as well.

From the perspective of file systems, there are two types of checkpointing patterns, N–N and N–1 [15]. An N–N checkpointing pattern is one in which each of the N processes writes to a unique file, with a result of total of N checkpoint files. An N–1 checkpointing pattern differs in that all the N processes write to a single shared file. The implementation of N–1 based checkpointing is limited to application-level checkpointing. From system-level, it is difficult, if not impossible, to combine/retrieve multiple independent checkpoint images into/from one file. Thus, in this research, we focus on the performance optimization of the N-N checkpointing pattern.

PFS generally stripes data over multiple I/O servers for high performance I/O. Fig. 2 illustrates how checkpoint requests are handled with a traditional checkpointing method. Suppose we have two compute nodes. Each node is equipped with dual cores and executes two parallel processes. Assume that PFS is deployed on two dedicated I/O servers. A PFS client daemon resides on each compute node to capture the I/O requests to/from I/O servers. The checkpoint snapshot of each process is divided into 4 stripes, which are evenly distributed onto the two I/O servers. One data stripe is not guaranteed to be serviced exclusively by the I/O server: it could be preempted by the I/O requests of other stripes for the sake of fairness. In the ideal case, the processing time is halved by distributing the checkpoint snapshot onto two I/O servers. The striping design of PFS facilitates fast processing time, which is

defined as the time from the start of file writing to its completion. As a specific data intensive application to PFS, however, checkpointing has several distinctive features that provide room for performance improvement, as discussed below.

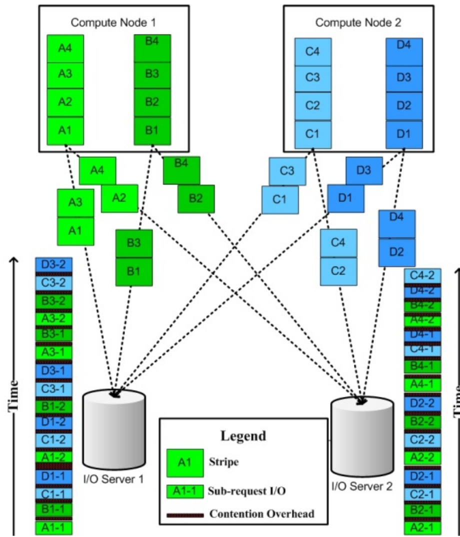

Figure 2. Traditional Checkpointing

- Checkpoints of all parallel processes act as a whole and their performance is evaluated by the time span between the issuance of the first checkpoint request and the completion of last one. The processing time of one single checkpoint has a limited, if not meaningless, representation of the overall performance. As a consequence, we need to reevaluate the role of striping under the scenario of checkpointing.
- Checkpointing usually issues a burst of write requests, which are serviced in a round-robin manner at file system side for the sake of fairness, and introduces additional overhead due to the context switch and the contention that causes extra physical disk head movement. Fig. 2 reflects this overhead by further dividing each stripe into two sub-requests.

In this paper, we propose to orchestrate both the checkpointing system and the parallel file system for better performance. The orchestration is two-fold. First, we propose a *vertical checkpointing* mechanism to reduce the number of checkpoints serviced by each I/O server. Second, we introduce a *staged checkpointing marshaling* to reduce I/O contention.

B. *Vertical Checkpointing*

The striping mechanism of PFS helps to speed up the processing time of single checkpoint. However, it may not be helpful for a burst of concurrent checkpoint requests. On the contrary, striping checkpoint requests across I/O servers introduces contention and can harm the overall performance.

One of the main functionalities of checkpointing orchestration is being able to disable striping and select one dedicated I/O server for each checkpoint to reduce the contention. As illustrated in Fig. 3, checkpointing orchestration first generates a mapping file. PFS later refers the mapping file to choose I/O servers for checkpointing storage. The term of vertical checkpointing comes from the fact that each checkpoint file is stored directly onto one given I/O server. In contrast, as illustrated in Fig. 2, the conventional striping methodology distributes the checkpoint file horizontally among all the I/O servers.

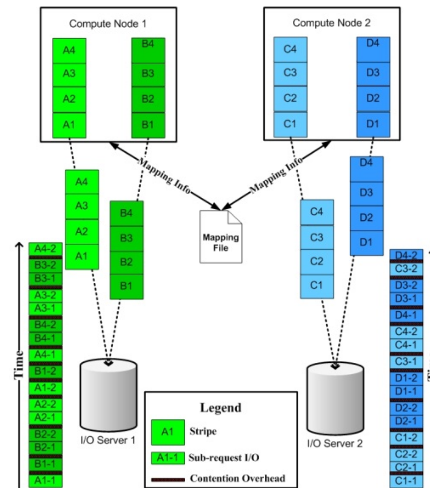

Figure 3. Vertical Checkpointing

With vertical checkpointing, each I/O server only needs to service two checkpoint files as shown in Fig. 3, which considerably reduces the contention.

The mapping file is a hashing function from PFS clients to PFS servers. Checkpoints from one compute node share one identical I/O server to reduce network contention. The mapping file is generated at the beginning of an application run and maintained until the completion of the application or an interruption because of failures. It is important to balance the workload of each I/O server. By default, the mapping file assigns each I/O server with the same number of compute nodes and the associated checkpoints. This strategy works well since the checkpoint image sizes of each process are similar, especially for well optimized MPI applications. For applications with irregular workload patterns, the mapping file can be adjusted accordingly to balance the data distribution on each I/O server. Note that there is no consistency and synchronization issue for sharing the mapping file among multiple clients, because all the requests are read only.

C. *Staged Checkpointing Marshaling*

Vertical checkpointing helps to reduce the number of checkpoint requests served by each I/O server and the cost

of coordination among I/O servers. However, even with the vertical checkpointing method, each I/O server is still burdened by I/O interleaving of checkpoint requests as shown in Fig. 3. The I/O interleaving issue is exaggerated with the fact that each I/O server services multiple compute nodes, and each compute node can spawn multiple checkpoint processes to the I/O server. The proposed checkpointing orchestration approach serializes the checkpoints on each compute node to boost the performance. We employ a *staged marshaling* design for this purpose. Each checkpoint process first stages its image in the local memory, and then flushes the image from memory to the PFS server.

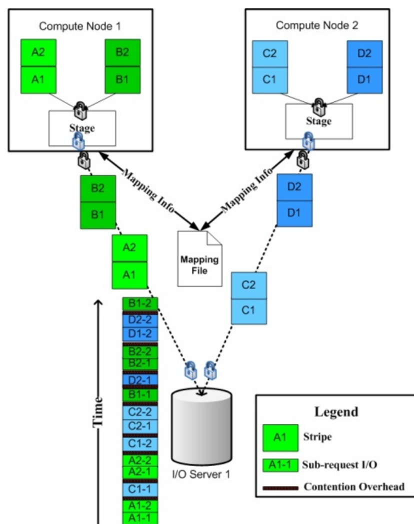

Figure 4. Staged Checkpointing Marshaling

In Fig. 4 we have two compute nodes whose checkpoints are mapped onto one shared I/O server. The staged checkpointing marshaling is adopted on each compute node. Each checkpoint file is first staged onto the local memory, and then flushed into the PFS I/O server. The motivation of the staging phase is to mitigate the impact of small VFS writes [16]. The flushing phase has a mutex to govern the multiple checkpoint requests in a stream and reduce the contention. Multiple checkpoint requests from one compute node are served sequentially by the I/O server. From Fig. 4 we can observe that at the I/O server side, the VFS requests from one compute node are not interleaved. As a result, we have more contiguous requests in Fig. 4.

The pseudo code of the staged checkpointing marshaling is shown in Algorithm 1. We have two interleaved mutexes. The first mutex is used to limit only one checkpoint process carrying out the staging at one time. The second mutex marshals the concurrent checkpoints in a serialized manner. The two mutexes are interleaved, which means a checkpoint will not release the staging lock until it gets the PFS flushing lock. The staging phase operates in memory and is faster than the PFS operation, and by interleaving the two mutexes we avoid the excessive memory usage. Staged marshaling requires the memory to accommodate up to two checkpoint files, which is affordable. The staged marshaling technique facilitates checkpointing by both mitigating the burden of small VFS requests and limiting the memory usage. Checkpoint requests are serviced in a pipeline manner with staged checkpointing marshaling: The flush operation of the previous request runs in parallel with the stage operation of the next request.

```
Wait(StageMutex)
    Stage Checkpoint onto local memory
Wait(PFSMutex)
Signal(StageMutex)
    Flush checkpoint to PFS server
Signal(PFSMutex)
```

Algorithm 1. Pseudo Code for Staged Marshaling

III. IMPLEMENTATION OF CHECKPOINTING ORCHESTRATION

A prototype of checkpointing orchestration system is implemented under PVFS2 and Open MPI. In particular, we implement vertical checkpointing under PVFS2 and optimize the performance from the PFS side. The staged checkpointing marshaling is implemented under Open MPI and optimizes the performance from the perspective of checkpointing middleware.

A. Implementation under PVFS2

As a parallel file system that provides high performance I/O for parallel applications, PVFS2 [10] not only supports MPI-IO interface, but also provides POSIX interface to benefit general application such as system-level checkpointing. We choose to implement the checkpointing orchestration under PVFS2. We made the following modifications to PVFS2 to facilitate vertical checkpointing.

First, the striping mechanism is disabled to enforce that one checkpoint is served by a single I/O server. As checkpoint files are stored within a PVFS2 directory that the user specifies, we reset the attributes of the PVFS2 directory to enforce single I/O server access.

Second, we need to balance the workload of each I/O server to achieve the overall optimal performance. In the original implementation of PVFS2, I/O server is chosen randomly for each I/O request, which provides approximately balanced distribution in general. However, if each task has a similar workload, as in the checkpointing application, a controlled distribution achieves a better load balance. We use a mapping file to hash checkpoints to I/O servers for balanced distribution. Each checkpoint process first reads the mapping file, and piggybacks hashed I/O server information in the *hint* field of PFS clients, which is used later by the PFS metadata server to map requests to the corresponding I/O servers.

PFS usually service mixed workloads and our implementation should not affect I/O requests from other ordinary applications. We take two approaches to handling

this problem. First, each checkpoint goes to the specified directory that disables the striping mechanism. This approach isolates the effect of checkpointing orchestration from regular applications. Furthermore, since the hint filed of each checkpoint request is flagged, PFS can easily identify the checkpoint I/O requests from regular I/O requests and route the request to the right I/O server. Regular I/O requests from other applications will not be affected by checkpointing orchestration.

B. *Implementation under Open MPI*

Open MPI is an open source MPI-2 implementation that provides high-performance, robust, parallel execution environment for a wide variety of MPI applications [17] [18]. Open MPI supports a transparent, coordinated checkpoint/restart implementation primarily based on BLCR library [19]. The staged checkpointing marshaling component of checkpointing orchestration is implemented under Open MPI.

The system call *fcntl* is chosen for the implementation of mutex locks. Each process requires an exclusive write request to the lock file in order to carry out the staging or flushing operation. We use ram-based file as the mutex lock file for better efficiency. Since the lock file is only shared by a limited number of processes inside one compute node, it provides sufficient and required concurrency control.

Note that checkpointing orchestration is currently implemented as an augment to PVFS2 and Open MPI. Users can flexibly decide whether to take advantage of checkpointing orchestration by setting different flag parameters. PVFS2 and Open MPI work normally if checkpointing orchestration is disabled.

IV. PERFORMANCE EVALUATION

A. *Experimental Settings*

Our experiments were conducted on a cluster of 32 Sun Fire Linux-based compute nodes. Each node is equipped with dual 2.7 GHz Opteron quad-core processors, 8 GB memory and 250GB SATA hard drive. All the nodes are connected with 1 Gigabit NICs in a fat tree topology. We use Open MPI v1.4 as the MPI and checkpointing environment, and use PVFS2 as the parallel file system for checkpointing storage. PVFS2 is built on 4 extra nodes as I/O servers. The PVFS2 stripe size for traditional checkpointing is set as 64KB. The I/O servers have the same hardware configuration as the compute nodes. Each I/O server also works as a metadata server.

We use NAS Parallel Benchmark (NPB) as parallel applications for performance evaluation [20]. NPB is a suite of programs widely used to evaluate the performance of parallel applications. We select 5 representative benchmarks with the consideration of image sizes, number of processes available for the performance evaluation. Table I summarizes the benchmarks selected, their problem sizes (class C/D), process number and the corresponding checkpointing image sizes. Among the five benchmarks, the number of processes for BT and SP can be only set as the squared integers, while the other three require the process number as the power of 2*. The applications with 36 and 196 processes are assigned 12 and 28 nodes such that the images can be evenly distributed over the 4 I/O servers.

Three environments have been set up to evaluate the proposed checkpointing orchestration. The traditional environment is the default configuration without orchestration. The vertical environment employs only vertical checkpointing and disables staged checkpointing marshaling. The orchestration environment reflects the performance with the deployments of both the two techniques. Measuring the checkpointing overhead is naturally supported by Open MPI.

Problem Size	Class=C	Class=D			
Benchmarks/# of Procs	256	32/36	64	128/196	256
LU	2.5GB	12GB	12GB	14GB	16GB
CG	2.1GB	20GB	20GB	21GB	22GB
BT	4.2GB	26GB	28GB	31GB	32GB
SP	3.7GB	22GB	24GB	27GB	28GB
FT	9.3GB	N/A	81GB	81GB	82GB

Table I: Benchmarks and the Overall Image Size (GB)

B. *Performance with Different Benchmarks*

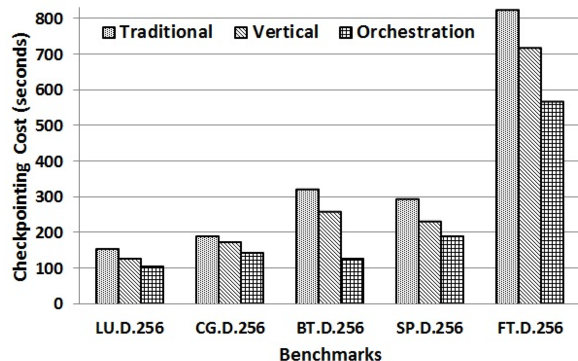

Figure 5. Checkpointing Performance Comparison

We first measured the checkpointing performance for all five benchmarks with class D as reported in Fig. 5. The number of processes is set as 256. The first observation is that checkpointing orchestration presents better performance than that of the traditional approach. As an example, checkpointing orchestration saved 254 seconds compared to the traditional approach for benchmark FT. The checkpointing overhead of benchmark LU was reduced from 157.41 seconds of the traditional approach to 105.99 seconds, with a speedup close to 30%. Vertical checkpointing outperforms the traditional approach but still leaves a considerable performance improvement space for staged checkpointing marshaling. In the traditional approach, one I/O server services 256 concurrent

* Class C of benchmark FT was omitted in the experiment since this configuration couldn't complete the compilation phase.

checkpointing processes, which is reduced to 64 by vertical checkpointing, and further optimized to 8 when equipped with staged checkpointing marshaling.

C. *Task Scaling Performance*

In Fig. 6 we scaled the number of processes from 32/36 to 256 to observe the performance changes. It shows that the growth of processes constantly increases traditional checkpointing costs. Checkpointing orchestration presents considerable performance improvement over both the traditional approach and vertical checkpointing.

We observe that checkpointing orchestration presents much better scalability through these tests. The overhead increase was less than 15% for benchmarks LU and CG when the number of processes was doubled. The performance variance of checkpointing orchestration was also trivial for benchmark FT, BT and SP when the number of processes was doubled from 128 to 256. The gap between the traditional and orchestration approaches is enlarged as the number of processes increases. These observations confirm the potential of checkpointing orchestration in fostering the scalability for large-scale computing environment.

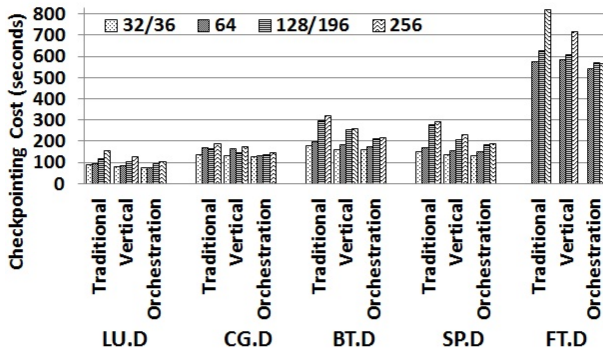

Figure 6. Task Scaling Performance (class=D)

Fig. 7 demonstrates the aggregated bandwidth with different number of processes for a better understanding of the scalability. Both traditional approach and vertical checkpointing exhibit considerable bandwidth degradation while the number of processes is increased. Checkpointing orchestration presents relatively stable bandwidth for CG and FT. For other benchmarks, the performance degradation of checkpointing orchestration is less than 25% when increasing the number of processes from 32/36 to 256, compared to over 50% bandwidth reduction of the traditional approach.

D. *Problem Size Scaling Performance*

Fig. 8 demonstrates the performance by varying the problems size from class C to D. Due to the increase in the image size, the checkpointing cost rises for all the three approaches. A quantitative study shows that the advantage of checkpointing orchestration actually drops as the problem size increases from class C to D. Further investigation of the data reveals that checkpointing orchestration boosts the performance more for class C than class D. For example, checkpointing orchestration reduces the cost of traditional checkpointing at a degree of 50% or more for the first four benchmarks of class C. However, the average gain of checkpointing orchestration is reduced to about 30% for class D. The increase of the problem size enlarges the portion of I/O in the overall checkpointing cost. The contention overhead, however, does not increase at the same pace since the number of processes is fixed at 256. These facts explain a relatively low performance improvement room for a larger problem size.

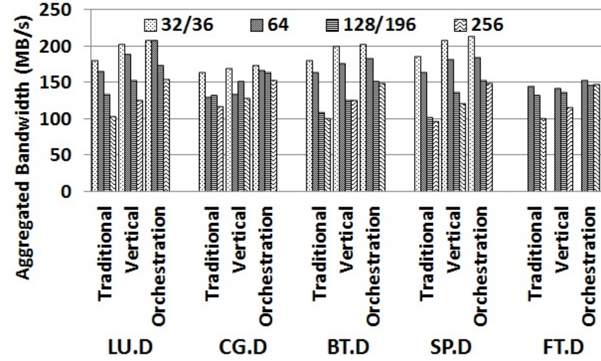

Figure 7. Task Scaling Bandwidth (class=D)

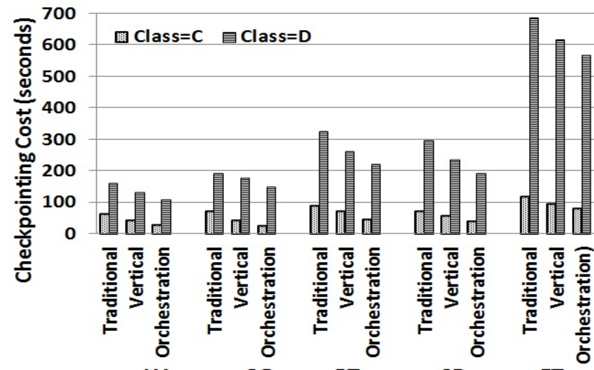

Figure 8. Problem Size Scaling Performance

V. RELATED WORK

It is well recognized by the community that checkpointing overhead is a critical issue that hinders the scalability and performance of large-scale systems and the upcoming exascale computing environment [6][25][11][20]. The existing efforts that tackle this issue can be roughly classified into three categories.

A. *File System Optimizations for Checkpointing*

Several efforts have been made to develop file systems that are tailored to checkpointing. Lightweight File System (LWFS) [11] allows secure, direct access to storage, bypassing certain features of traditional file systems that potentially cause performance bottlenecks for checkpointing. In [15], Bent et al. proposed a parallel log-structured file system(PLFS) that sits between the applications and the underlying parallel file system to achieve higher checkpointing bandwidth. PLFS transparently rearranges N-1 checkpointing pattern into an

N-N pattern to leverage the high I/O bandwidth achieved via an N-N pattern while maintaining an N-1 pattern to users such that the failed applications can be restarted with a different number of processes. Neither of these two considers the I/O contention issue of surging checkpoints. Our work complements these studies by mitigating the impact of I/O contentions and can be used in combination with LWFS and PLFS to further boost the checkpointing performance.

At the parallel I/O middleware level, there has been significant amount of work on the I/O optimization for parallel applications too. The notable optimizations include collective I/O and data sieving [21]. Collective I/O exploits the correlations among I/O accesses from multiple processes of a parallel application and combines them to form large and contiguous accesses. Data sieving is another optimization strategy that makes large and contiguous I/O requests and then filters out demanded data to service many non-contiguous requests. Both collective I/O and data sieving optimization techniques have been well implemented in the most notable MPI-IO implementation, ROMIO [22], and are thus beneficial transparently to parallel applications with MPI-IO interface. Unfortunately, these optimizations are not directly available to checkpointing utilities and can hardly improve the checkpointing performance for parallel applications because many checkpointing utilities adopt POSIX API. The system-level checkpointing tools such as Open MPI, MVAPICH only have the support of POSIX IO, which excludes the possibility to take the advantage of MPI-IO and its optimizations for checkpointing, at least not currently.

B. Checkpointing System Optimizations

In [23], the authors proposed to modify the coordination protocols such that the checkpointings are taken in smaller groups to reduce the contention. Our work provides a simpler and more reliable solution due to the fact that the coordination protocol is kept intact in checkpointing orchestration. There is no concern about the consistency, group formation and connection management in checkpointing orchestration.

Aggregating the write requests at checkpointing system layer is another alternative to accelerate the checkpointing performance. Ouyang et al. categorized the VFS write requests by size, aggregates small and medium writes to relatively large writes for better performance for multi-core systems [16]. Our work differs from [16] in three aspects.

First, the motivation of [16] is to mitigate the impact of massive small VFS writes. However, checkpointing orchestration aims at relieving the contention of concurrent parallel checkpoints. The staged marshaling technique of checkpointing orchestration is designed to marshal the multiple checkpoints from one compute node and reduce the contention. The contention from multiple checkpoint processes is not optimized in the design of [16].

Second, checkpointing orchestration mitigates the impact of small VFS writes by staging each checkpoint in the local memory, and drops the inter-process requests aggregation as proposed in [16]. This design makes checkpointing orchestration easier to rebuild the image for restart than the design of [16].

Third, checkpointing orchestration also proposes to improve the performance from the perspective of PFS, which is not considered in [16].

Open MPI has a native support of staging, which first stages the checkpoint file locally and then transfer it out in the background. Even though such a staging technique can be used to implement our proposed staged marshaling technique*, the contention optimization is not considered in Open MPI.

In a summary, most existing works optimize the checkpointing performance from the perspective of either file system [11] [15] or checkpointing system [16] [23]. Checkpointing orchestration explores the optimization of both sides to foster a better integration of both sides and to provide a systematic approach to boost the checkpointing performance. We believe that the checkpointing overhead problem for the exascale systems will become even more imminent, and we need to explore all the possible ways to make a breakthrough.

VI. CONCLUSIONS, SOFTWARE AVAIALABILITY AND FUTURE WORK

Checkpointing is a widely adopted fault tolerance mechanism in parallel computing, while in the meantime it is highly criticized for its costly I/O access overhead. The overhead challenge could be further amplified with the emerging of exascale computing environment [6].

In this paper we start with analyzing the underlying sources of performance bottlenecks of coordinated checkpointing. In observing data access contention as a dominant contributor for performance degradation, we propose to orchestrate checkpointing to minimize the contention and improve its performance by managing the concurrency at different levels. The proposed checkpointing orchestration approach suggests a controlled management of both PFS and checkpointing system for better integration. From the perspective of PFS, we propose to customize the data distribution to benefit checkpointing. From the perspective of checkpointing system, we adopt a methodology of reorganizing the checkpointing order to avoid potential I/O contentions.

The prototype of checkpointing orchestration has been implemented under PVFS2 and Open MPI. The performance evaluation confirms that the proposed checkpointing orchestration can reduce the checkpointing cost by nearly 50% for 4 out of 5 the class C NPB

* Checkpointing local staging was broken on the version of Open MPI we tested so we dropped the idea of implementing staged marshaling based on the native staging support. See https://svn.open-mpi.org/trac/ompi/ticket/2139

benchmarks. Checkpointing orchestration also helps to improve the scalability of checkpointing. Checkpointing orchestration considers the mixed workloads of the system, keeps the regular I/O requests intact by treating the checkpointing requests with a special PVFS2 hint.

The design of checkpointing orchestration is based on existing hardware and software architecture of HPC environment. Software patching from PFS and checkpointing system is sufficient to deploy the proposed checkpointing orchestration.

We have developed the software named ORCHECK to implement the idea of checkpointing orchestration. The software has been released and is available at http://www.cs.iit.edu/~scs/invention/orcheck/.

In the future, we plan to evaluate the potential of checkpointing orchestration for large-scale computing environment. We are also interested in studying the impact of checkpointing orchestration for newly emerging storage media such as SSD. Our long term goal along this direction is to build a coordinated framework with the cooperation of both checkpointing and parallel file systems to facilitate scalable fault tolerance for large-scale computing and the upcoming exascale computing environment.

ACKNOWLEDGMENT

The authors are thankful to anonymous reviewers for their valuable suggestions that help improve this study. This research was supported in part by National Science Foundation under NSF grant CCF-0621435, CCF-0937877 and CNS-0751200. The authors would like to acknowledge Joshua Hursey of Open MPI group at Indiana University and Samuel Lang of PVFS2 group at Argonne National Lab for their valuable assistance in the implementation of checkpointing orchestration.

REFERENCES

[1] Top 500 Supercomptuer website. [Online]. http://www.top500.org

[2] I.Philp, "Software Failures and The Road to A Petaflop Machine," in *Proc. of Workshop on High Performance Computing Reliability Issues*, 2005.

[3] B. Schroeder and G. A. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," in *in Proc. of International Conference on Dependable Systems and Networks (DSN)*, 2006.

[4] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Survey*, vol 34, issue 3, 2002.

[5] R. Oldfield, S. Arunagiri, P. Teller, S. Seelam, M. Varela, R. Riesen, and P. Roth, "Modeling the Impact of Checkpoints on Next-Generation Systems," in *Proc. of the 24th IEEE Conference on Mass Storage Systems and Technologies*, 2007.

[6] F. Cappello, A. Geist, B. Gropp, B. Kramer, M. Snir, "Toward Exascale Resilience," *International Journal of High Performance Computing Applications*, vol 23, issue 4, 2009.

[7] N. DeBardeleben, J. Laros J. T. Daly, S. L. Scott, C. Engelmann, B. Harrod, "High-End Computing Resilience: Analysis of Issues Facing the HEC Community and Path-Forward for Research and Development," White Paper 2009.

[8] Lustre File System Website. [Online]. http://www.lustre.org

[9] F. Schmuck and R. Haskin. , "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proc. of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[10] PVFS2 Website. [Online]. http://www.pvfs.org/

[11] R. Oldfield, L. Ward, R. Riesen, A. Maccabe, P. Widener, and T. Kordenbrock. , "Lightweight I/O for Scientific Applications," in *In Proc. of IEEE Cluster Computing (Cluster)*, 2006.

[12] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling and Evaluation of a Scalable Multi-Level Checkpointing System.," in *Proc. of the International Conference for High Performance Computing, Networks, Storage and Analysis (Supercomputing)*, 2010.

[13] H. Jin, Y. Chen, H. Zhu, and X.-H. Sun, "Optimizing HPC Fault-Tolerant Envionment: An Analytical Approach," in *International Conference on Parallel Processing (ICPP)*, 2010.

[14] ORCHECK, An Open-Source Software Implementation of Checkpointing Orchestration. [Online]. http://www.cs.iit.edu/~scs/invention/orcheck/

[15] 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 *in Proc. of the International Conference for High Performance Computing, Networks, Storage and Analysis (Supercomputing)*, 2009.

[16] X. Ouyang, K. Gopalakrishnan, and D. K. Panda, "Accelerating Checkpoint Operation by Node-Level Write Aggregation on Multicore Systems," in *Proc. of International Conference on Parallel Processing (ICPP)*, 2009.

[17] E. Gabriel, G. E. Fagg, G. Bosilca, and etc , "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proc. of European PVM/MPI Users Group Meeting.*, 2004.

[18] J. Hursey, T. I. Mattox, and A. Lumsdaine, "Interconnect Agnostic Checkpoint/Restart in Open MPI," in *in Proc. of International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)* , 2009.

[19] J. Duell, P. Hargrove, and E. Roman, "The Design and Implementation of Berkeley Labs Linux Checkpoint/Restart, Lawrence Berkely National Laboratory," Lawrence Berkeley National Laboratory Technical Rep, LBNL-54941, 2002.

[20] NAS Parallel Benchmarks (NPB). [Online]. http://www.nas.nasa.gov/Resources/Software/npb.html.

[21] Rajeev Thakur, William Gropp, Ewing Lusk, "Data Sieving and Collective I/O in ROMIO," in *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[22] ROMIO Website. [Online]. http://www-unix.mcs.anl.gov/romio/

[23] Q. Gao, W. Huang, , M. J. Koop, and D. K. Panda, "Group-based Coordination Checkpointing for MPI: A Case Study on InfiniBand," in *Proc. of International Conference on Parallel Processing (ICPP)*, 2007.