Accelerating Checkpoint Operation by Node-Level Write Aggregation on Multicore Systems

Xiangyong Ouyang, Karthik Gopalakrishnan and Dhabaleswar K. Panda
Department of Computer Science and Engineering
The Ohio State University
{ouyangx, gopalakk, panda}@cse.ohio-state.edu

***Abstract*—Clusters and applications continue to grow in size while their mean time between failure (MTBF) is getting smaller. Checkpoint/Restart is becoming increasingly important for large scale parallel jobs. However, the performance of the Checkpoint/Restart mechanism does not scale well with increasing job size due to constraints within the file system. Furthermore, with the advent of multi-core architecture, the situation is aggravated due to larger number of processes running on the same node, trying to checkpoint simultaneously. This results in increased number of file writes at the time of checkpointing which leads to performance degradation. As a result, deployment of Checkpoint/Restart mechanisms for large scale parallel applications is limited.**

In this work, we explore the Checkpoint/Restart mechanism in MVAPICH2, which uses BLCR as the checkpointing library. Our profiling of the checkpoints for the NAS parallel benchmarks revealed a large number of small file writes interspersed with large writes. Based on these observation we propose to optimize checkpoint creation by classifying checkpoint file writes into small writes, medium writes and large writes based on their size of data to write, and use write aggregation to optimize the small and medium writes. At the aggregation threshold of 512KB, the implementation of our design in BLCR shows improvements from 27% to 32% over the original BLCR in terms of time cost to checkpoint an MPI application.

I. Introduction

Compute Clusters are continuously growing in terms of their speed and size. With the advent of multi-core processor architectures, the number of MPI processes that can be executed within a single node increases substantially. However, with increasing number of components in the compute cluster, the Mean Time Between Failures (MTBF) is also reduced. In [8], the authors studied existing clusters and predict that the MTBF of peta-scale systems maybe as short as 1.25 hours.

This research is supported in part by DOE grants #DE-FC02-06ER25755 and #DE-FC02-06ER25749, NSF Grants #CNS-0403342, #CCF-0702675 and #CCF-0833169; grant from Wright Center for Innovation #WCI04-010-OSU-0; grants from Mellanox, Intel, Cisco, QLogic and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Advanced Clustering, Appro, QLogic and Sun Microsystems.

To avoid wasting computational cycles caused by failures in the cluster, applications are checkpointed at regular intervals. Checkpointing causes the complete state of the process to be saved from memory to disk so that in the event of a failure before the process completes its execution, the process can be restarted from the image that was saved to disk. In this way, only the computation performed by the process after the most recent checkpoint is lost. Berkeley Labs' Checkpoint/Restart software package (BLCR) [6] is a popular Checkpoint/Restart solution that is used by many MPI implementations, including MVAPICH2 [1] [13], OpenMPI [7] and LAM/MPI [14].

The process of checkpointing a parallel MPI application involves three main steps:

1) Suspend communication between all processes in the parallel application and bring them to a consistent state.
2) Use the checkpoint library to checkpoint the processes individually.
3) Re-establish connections between the processes and continue execution.

Step (2) involves writing the process' context and memory contents to a file on a reliable storage medium, usually a local disk or a parallel file system. Hence, the time to perform Step (2) dominates the time to create a checkpoint.

Modern file systems provide large storage capacity and throughput. However, their performance on large clusters can be vastly improved by optimizing the way in which the checkpoint data is written to the file system. For example, from a file system's perspective, writing one large chunk of data to disk is more efficient than writing a smaller amount of data through multiple times, even though the total size of data written is the same. Hence, coalescing writes to the file system would yield a much better performance.

To understand the process of checkpointing better, we ran several applications from the NAS parallel benchmark suite [18] using MVAPICH2 [1] and checkpointed the applications using a version of BLCR that we modified to provide profiling information. Our traces revealed that BLCR performs a substantial number of writes to file with

data less than 64 bytes. Furthermore, since checkpointing a parallel application involves each process running on every core to take a checkpoint concurrently, the number of writes with small data is increased.

Based on these experiments, we propose enhancements to the existing checkpointing mechanism by caching small and medium writes within a node, drastically reducing the number of writes to the file system. Our approach reduces the checkpoint creation time on a 8-core system by 30.79%, 32.46%, 27.46% for LU, BT and CG respectively.

The rest of paper is organized as follows. In section 2, we describe the background of checkpoint and restart. In section 3, we analyze the profiles of NAS parallel benchmark to characterize checkpoint writing. In section 4, we present our design in detail and discuss some design alternatives. In section 5, we present the experiments and show the results of improvement of our design. In section 6, we provide our conclusion and state future work we are going to conduct. Finally we discuss the related work.

II. Background

A checkpoint saves the state of the process at a given point of time during its execution. It includes enough information to restart a process from that point. An application maybe checkpointed periodically so that in an event of a failure, it can be restored from the most recent checkpoint to continue its execution, rather than start again from the beginning.

There are two potential approaches to initiating a checkpoint; application initiated and system initiated.

In application initiated checkpointing, the application decides when to start a checkpoint and requests the system to take the checkpoint on the application's behalf. Transparent application-level checkpointing may be achieved through compiler techniques [15]. Additionally, a hybrid approach is possible where the application participates in the creation of a checkpoint but is assisted by a user-level library [11].

In system initiated checkpointing, the system directly initiates the application checkpoint, without interaction with the application. The application maybe unaware of the fact that it is being checkpointed. This is usually achieved through a kernel component, as with Berkeley Lab Checkpoint Restart (BLCR) [6]. BLCR has been combined with several Message Passing Libraries (MPI) such as LAM/MPI [16], OpenMPI [7] and MVAPICH2 [13] to checkpoint parallel jobs running on multiple nodes.

A. MVAPICH2 checkpoint/restart facility

MVAPICH2 is a message passing library for parallel applications with native support for InfiniBand [1]. It supports checkpoint/restart (C/R) for the InfiniBand communication channel [13], [12]. Support for C/R is achieved through interaction between the MVAPICH2 library and the BLCR kernel module [10].

Taking a checkpoint for a parallel job involves bringing the processes to a consistent state using a coordination algorithm. First, all the processes coordinate with one another, and flush and lock the communication channel. Then, all InfiniBand connections are torn-down. Next, the MVAPICH2 library on each node requests the BLCR kernel module to take a blocking checkpoint on the process. The checkpoint data is independently written to a file on disk; one file per process. Finally, all processes again coordinate with each other to rebuild the InfiniBand connections and the application continues its execution. MVAPICH2 supports both system initiated and application initiated checkpointing.

III. Characterizing Checkpoint Writing

In order to understand the characteristics of checkpoint file IO, we run several applications LU/BT/SP/CG from NAS parallel benchmark version 3.2.1 [18] using MVAPICH2 C/R framework [1]. The BLCR is modified to provide profiling information pertaining to checkpoint file writing. The test is conducted on a 64 node cluster. Each node has 8 processor cores on 2 Intel Xeon 2.33 GHz Quad-core CPUs. We choose Class C with 64 processes. Each process runs on a separate processor core, so 8 nodes are used in the test. Each process writes its checkpoint data to a checkpoint file on a local ext3 file system.

Table I
Basic Checkpoint Information(Class C, 64 processes)

	LU	BT	SP	CG
Time for one checkpoint(seconds)	7.6	11.3	10.3	7.1
Total data size(MB) per node	184.0	320.0	316.0	163.2
Number of VFS write per process	975	1057	1367	820
Total number of VFS writes per node	7800	8456	10936	6560

We profiled the checkpoint creation for these applications to understand the checkpoint file write patterns. Table II gives an example of the application LU with class C and 64 processes. It decomposes the checkpoint writing into different categories according to the size of writes. The first column is the size of write belonging to that category. The second column is percentage of writes within that range. The third column is percentage of data amount written by that type of write. The fourth column is percentage of time spent by VFS writes belonging to that category. We can observe some characteristics of checkpoint file write from this profiling.

(1) Most of the file writes only write small amount of data (smaller than 4KB per write). These small writes make up over 60% of all file writes, but they only write about 1.5% of total amount of data being dumped. It costs less than 0.2% of total time to perform these small writes. We investigated the BLCR, and found that these small writes are primarily storing CPU registers, signal handler table, timers, open-file tables, process/group/session ids, and various other of BLCR data structures necessary to restore a process.

(2) There are a few large writes (greater than 512 KB per write). These writes constitute only about 0.8% of all writes, but they contribute about 79% of all data dumped. These writes consume 35% of total write time.

(3) In between small and large writes are medium writes, which make up about 38% of all writes. They contribute about 20% of all data, but consume about 65% of all time. The medium and large writes actually store the virtual memory area (VMA) of a process. BLCR scans all VMAs of a process, and saves non-zero contiguous data pages to the checkpoint file. An application process usually has many VMAs. Many VMAs contain a handful contiguous pages that need to be dumped to file, which become a medium write. A few VMAs contain large block of contiguous pages to dump. They are the source of large writes.

Table II
CHECKPOINT FILE WRITE PROFILE OF LU.C.64

	% of Writes	% of Data	% of Time
0-64	50.86	0.04	0.17
64-256	0.61	0.00	0.00
256-1K	0.25	0.01	0.00
1K-4K	9.46	1.53	0.01
4K-16K	36.49	11.36	44.66
16K-64K	0.74	0.77	6.55
64K-256K	0.49	3.79	11.80
256K-512K	0.25	3.58	1.75
512K-1M	0.61	17.72	14.72
$>$ 1M	0.25	61.21	20.35

The profiling above reveals patterns of a typical checkpoint writing. Given these characteristics, we propose to classify checkpoint file writes into 3 categories, and use different approaches to handle them in order to improve checkpoint creation performance.

(1) Although small writes can be buffered in the local page cache if buffered IO is used, they can cause frequent calls to write() system call. This can result in heavy overhead. Therefore, we propose to use a local buffer to coalesce these small writes, and flush this buffer to file when all writes complete. The cost to copy small amount of data to local buffer is very small compared to the VFS write. Our experiment in later sections justify this idea.

(2) For large writes we have to treat them differently for two reasons. First is the cost of memory copy compared to file write. At large data size, the cost of memory copy quickly becomes close to the cost of direct file write. Second is memory consumption. If we treat large write in the same way as small write, then local buffer will consume significant amount of memory to store the large data. This can severely hurt the memory scalability in a large parallel application. Therefore we decide to flush large write directly into checkpoint file.

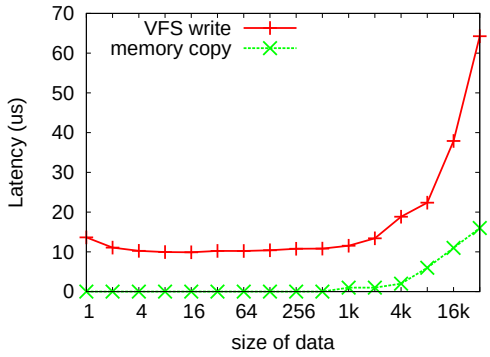
(3) Medium writes are an important source of overhead in checkpoint writing. One approach is to treat them like small-writes to aggregate them in local buffer and flush them to file at later time. There are two sources of overhead in this approach. One is the memory copy cost, the other is cost to flush the aggregated large data chunk to checkpoint file. The alternative is to treat them as large-write to flush it directly to file. Which method to take depends on the relative cost of memory copy compared to file write. It's critical to choose a threshold upon which to make a decision. We conduct detailed experiments to search the optimal division. This is discussed in later sections.

IV. DETAILED DESIGN

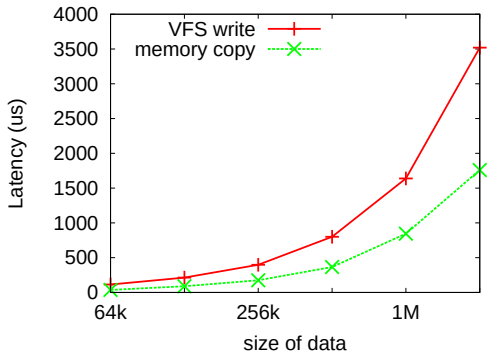
In this section we present our write aggregation optimization design to improve checkpoint creation for parallel jobs on multi-core systems.

One important idea of our design is to coalesce many small and medium writes into a relatively large write. Without write aggregation, the overhead of a file write comes from two sources: (a.1) Overhead at the VFS layer and (a.2) Overhead to actually move the data from memory to back-end storage device. If a file write is aggregated, then the overhead consists of: (b.1) Cost to copy the data to a buffer (b.2) Overhead at the VFS layer when the aggregated buffer is written to a file later (b.3) The overhead to move the data from memory to back-end storage device. Given that the VFS layer usually buffers data in the page cache and later on moves data to storage device as large chunks, we assume that the cost (b.3) is close to cost (a.2) of the non-aggregation design. Most file systems do buffered IO automatically, so we believe our assumption here is valid. So, the relative cost of (a.1) comparing to the sum of (b.1) and (b.2) becomes critical. To come up with an optimal design, we need to know the relative cost of memory copy vs. VFS write. We measure the VFS write latency and memory copy latency at different data size.

Figure 1 compares their latency at different data size. This measurement plays an important role as we propose our write-aggregation design.



(a) size $<$ 64KB



(b) size $>=$ 64KB

Figure 1. Latency of VFS Write and Memory Copy

A. Distinguish different write sizes

From Figure 1 we can see that the VFS overhead remains relative constant at about 10 us when write size is small ($<$ 1K bytes). Given large number of small writes, the total overhead of small writes will become significant. On the other hand, memory copy cost is very low at small data size. For data size ranging from 1 byte to 1k bytes, the latency is close to 0. Therefore, we propose to use a local buffer to accommodate all small writes.

If we aggregate these large writes and flush them to a file later on, then the cost of memory copy plus cost of VFS write for the aggregated data can exceed the original cost to write the data without any aggregation. High memory usage of large writes is another concern if aggregation is used. Therefore, we choose to let each application process perform a file write to send large data to checkpoint files.

Medium-sized writes happen quite frequently and their size is big enough to make the cost of copying them to a buffer noticeable. Therefore we decide to set a threshold size in this category. All writes lower than this size are copied to buffer, while writes larger than this size is directly written to checkpoint file. The choice of this threshold size is very critical.

A very large threshold size can cause too much data being copied to local buffer. This can lead to extra usage of memory, and high overhead to do memory copy. On the contrary a too small threshold will make all medium writes perform a VFS write, thus losing the opportunity to coalesce some writes into a large one. In the next section we conduct extensive experiments to measure the performance with different thresholds.

B. Write aggregation design

We incorporate all the aforementioned considerations into our design for multi-core systems.

Figure 2 illustrates our write-aggregation design. Each node has an IO process (IOP) that is responsible for performing aggregated writing. A parallel job usually has many application processes (APs) running on one node. The IOP creates a shared memory region that can be accessed by all application processes (APs). At initialization time, an AP allocates a piece of buffer local to the AP to accommodate its small writes. Since total data amount of small writes is not huge (usually less than 50 KB per process), the memory usage dedicated for small writes for all processes in a same node is not going to produce much pressure on available system memory.

1) Application processes: When taking a checkpoint, an AP will perform a series of data writes as described below.

(1) If the data size is small, it's put in a chunk with a header prepended. Then this chunk is stored in a local buffer. The structure of a header is illustrated in Fig 3. A header records the rank of this process, the size of data, as well as the original offset of this data if it was written to an exclusive checkpoint file. This "original offset" field is used to reconstruct an exclusive checkpoint file used by the process at restart phase. Since we make use of BLCR framework to restart a process, and BLCR parses the checkpoint file according to the way it writes a file, this "original offset" is necessary to rebuild a BLCR checkpoint file from the aggregated data.

(2) If the data is regarded as "large", then the AP won't perform any aggregation on it. Instead it performs a VFS write to write it out to a checkpoint file. At the same time, a header recording this large chunk is stored into the same local buffer as small writes. The header contains similar information as for small writes.

(3) If the data is regarded as "medium size", then we will store this data into the share memory created by IOP. The shared memory comes with two pointers P1 and P2 in a ring manner. P1 points to the beginning of available memory, P2 points to the beginning of "dirty data" that have been copied by APs and ready to be flushed to checkpoint file by the IOP. Another pointer P3 points to the end of "dirty data"

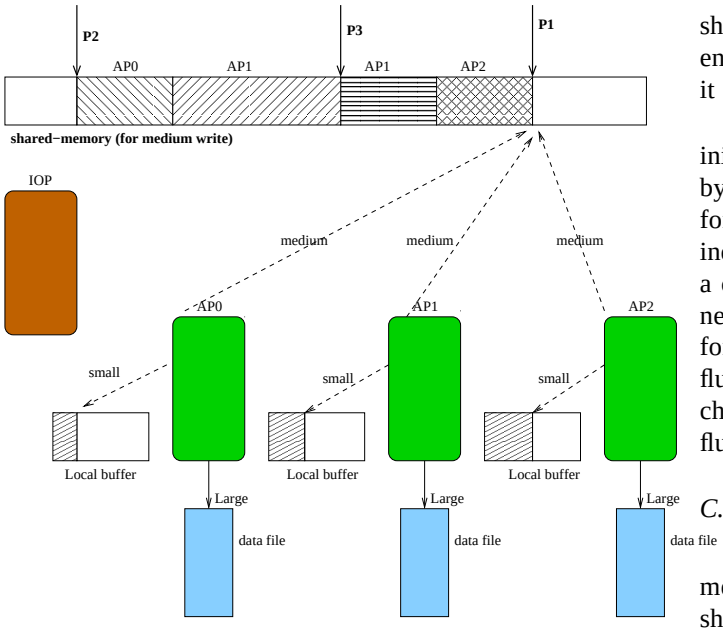


Figure 2. Node-level write-aggregation design

Process Rank	Data size	Original Offset	Data

Figure 3. Format of a chunk

that is ready for IOP to flush out in an aggregated manner. P3 and P1 are different when some area in between is being copied to by an AP.

The data between P2 and P3 can be flushed to checkpoint file by IOP. All operations to manipulate pointers P1/P2/P3 are protected by locks for mutual exclusion.

In case of a medium write, an AP grabs a piece of buffer from shared memory, and increments P1 by an amount which is the sum of data to be copied plus size of the header. The AP also enters itself into a list. This list is designed to keep track of the updates to pointer P3. Then the AP fills in a header and copies the data into buffer. After the copy finishes, the AP checks to see if it's in the closest position to P3. Then it examines whether its predecessor and successor in the list have finished copying. P3 is moved to the end of the newly copied data after careful cooperation by all APs in that list.

At any given time, the shared memory may contain dirty data from multiple APs. Figure 2 gives a possible snapshot of the shared memory. At that moment the shared memory contains data (shadowed areas) from AP0, AP1 and AP2 interwoven together. Each chunk of data is tagged with a header so that the mixed data can be parsed and returned to their owning APs at restart phase. After an AP finishes copying data to the shared memory, it checks if the share memory has accumulated enough dirty data. If it finds enough data there, then it sends a signal to the IOP to notify it of the pending dirty data.

2) IO process: An IOP's task is relatively simple. At initialization it creates shared memory region to be accessed by all APs. Then it sits idle and becomes blocked waiting for a signal from any one of APs. The arrival of a signal indicates that the dirty data in shared memory has exceeded a certain threshold. Then the IOP is unblocked. It creates a new thread (IO thread), immediately returns back and waits for the next signal. The IO thread performs a file write to flush all dirty data between pointers P2 and P3 to a separate checkpoint file. This file is meant for the aggregated data flushed by IO threads.

C. Insights into the design

One alternative is to store small write data to the shared memory instead of a local buffer. However each access to shared memory region involves acquiring / release locks. If the large number of small writes go to shared memory then the synchronization cost becomes significant. Therefore we choose to let each AP store its small writes to local buffer.

If all APs share the same data file, then we need a shared variable (with protecting lock) to keep current position in the file for write. Although the APs acquire this position in ascending order, they issue write requests concurrently. The order in which these requests come to the back-end storage server is also unpredictable. Therefore the file server may have to seek its disk head to service these requests going to different positions in the same file. This can cause severe overhead. Our experiment with shared checkpoint file shows a significant performance degradation due to this reason. Therefore we choose to let each AP write to a separate checkpoint file.

D. Restart design

The restart process of our design follows the BLCR framework. Since we have altered the file organization, we have to reconstruct the checkpoint files to the structure that can be interpreted by BLCR. First, the IOP reads from an aggregation file to extract medium-sized data and hand it over to their owner APs. Then each AP starts to read its own data file to get small and large-sized data. All these data are stored to a new checkpoint file at offsets specified by the header coming with each piece of data. When the new checkpoint file is ready, an AP makes a call to invoke BLCR to perform a restart. Although an additional overhead is involved in this process, it is fairly reasonable since the the application is restarted from the checkpoint only in the event of a failure, although the checkpoint maybe taken multiple times during the lifetime of the application.

V. Experimental Results

We have implemented the node-level write-aggregation design into BLCR-0.8.0 kernel module. We also have integrated the modified BLCR into MVAPICH2 [1]. In this section we conduct experiments on our design to evaluate its performance. The experiments are conducted on a cluster with 64 nodes. Each node has Intel Xeon 2.33 GHz 8 cores CPU. All of them run RedHat Enterprise Linux 5 with kernel 2.6.18-53.1.13.el5. Checkpoint files are stored to a local ext3 file system.

A. Checkpoint Time of Different Applications

In this section we evaluate performance of our design with NAS parallel benchmarks. We run NAS benchmark applications (LU,BT,CG) of class C and 64 processes. Eight nodes out of the 64 nodes are used to run one process on one processor core. While the application is running, the system initiates a checkpoint request to checkpoint the MPI job. The time to take a complete checkpoint is measured. In this experiment all MPI processes write their checkpoint files to local disks. As a part of our next step, we plan to extend our experiments to store checkpoint files to Luster [5] file system. Lustre file system provides huge aggregated IO throughput, so we expect higher IO performance in future experiments with Lustre file system.

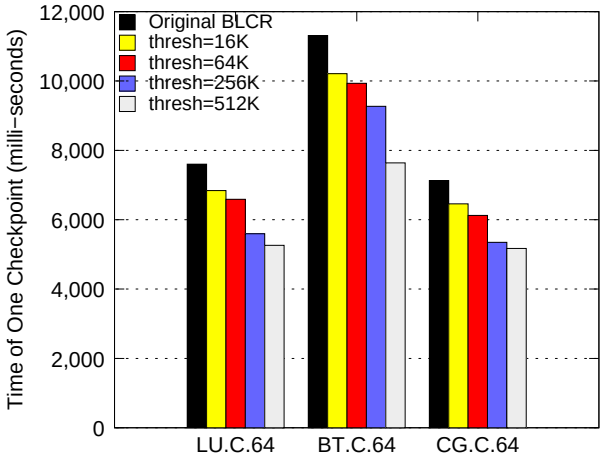


Figure 4. Time to Take One Checkpoint

Figure 4 shows the time to do one checkpoint using MVAPICH2 C/R framework for different applications in the NAS parallel benchmark suite. The results with different aggregation thresholds are compared against the time to take one checkpoint using the original BLCR (marked "Original BLCR"). Figure 4 demonstrates that write-aggregation can reduce the total time to take a checkpoint. In the example of LU.C.64, threshold=16K reduces one checkpoint time from 7601 ms to 6840 ms (a 10.01% improvement). Larger thresholds can further drive down this cost. As with LU.C.64, the improvements in checkpoint time are 13.32%, 26.39% and 30.79% for 64K, 256K and 512K thresholds respectively. Similar improvements can be obtained for BT.C.64 and CG.C.64 as shown in Figure 4. For BT.C.64, threshold values of 16K,64K,256K,512K can improve checkpoint time by 9.73%, 12.19%,18.05% and 32.46%. The corresponding numbers for CG.C.64 are 9.41%, 14.11%, 25.02% and 27.46%.

B. Decomposition of Checkpoint Time

The time cost to take a checkpoint includes 3 phases as described in section I: (1) Phase 1: synchronize all processes and drain all in flight messages in the communication channels. (2) Phase 2: each process perform a local checkpoint to save its image to a checkpoint file. (3) Phase 3: all processes re-establish communication channels among them and resume previous work. In order to understand the source of improvement from write-aggregation, we decompose the total time cost into the aforementioned three phases. This enables us to know exactly the sources of time cost to take a checkpoint, and gives us a better understanding of improvement achieved by write-aggregation. Table III shows the decomposed time cost (in milliseconds) at different phases to checkpoint an application with or without write-aggregation.

In table III, columns 2, 3 and 4 show the time cost in phases 1, 2 and 3, respectively. The last column gives the improvement achieved by write-aggregation in phase 2 at different aggregation thresholds. "LU-orig" indicates the result of LU.C.64 using the original BLCR. "LU th=X" indicates the LU.C.64 result produced by write-aggregation using a threshold value of X. Similar legends are used for BT.C.64 and CG.C.64. From this table we can observe some characteristics. The first and third phases are relatively constant with or without write-aggregation. The improvement in total time cost comes primarily from phase 2 where write-aggregation significantly reduces the cost to save a process image to a file. Take LU.C.64 for example. Write-aggregation reduces time cost in phase 2 by 14.88%, 18.99%, 35.88%, 43.13% for threshold values of 16K, 64K, 256K and 512K, respectively. Similar trends exist for BT.C.64 and CG.C.64.

The improvement in phase 2 comes from several factors. First, write-aggregation significantly reduces time cost for small writes. In original BLCR, each small write causes a VFS write that results in a system call. Although the actual data is implicitly buffered in page cache by VFS write, the total cost of so many small writes can sum up to a considerable amount. On the contrary, write-aggregation

Table III
TIME DECOMPOSITION(MILLISECONDS)

	Phase 1	Phase 2	Phase 3	Improvement in phase 2 (%)
LU-orig	33	5418	2150	
LU th=16K	59	4612	2169	14.88
LU th=64K	67	4389	2132	18.99
LU th=256K	74	3474	2047	35.88
LU th=512K	64	3081	2115	43.13
BT-orig	34	9136	2141	
BT th=16K	34	8142	2034	10.88
BT th=64K	48	7725	2159	15.44
BT th=256K	48	7084	2137	22.46
BT th=512K	34	5463	2142	40.20
CG-orig	40	4987	2103	
CG th=16K	42	4344	2073	12.89
CG th=64K	43	4055	2026	18.69
CG th=256K	44	3178	2124	36.27
CG th=512K	45	2959	2168	40.67

explicitly stores small data into a process's local buffer. This greatly reduces the overhead related to small writes. Medium writes are also aggregated into memory. Even the large writes benefit from write-aggregation, because write-aggregation greatly reduces the total number of write requests that are sent to the VFS layer.

C. Memory Usage at Different Thresholds

We also measured the memory usage on a node for different applications at different threshold values. On each node, enough memory is allocated into the shared memory region at the beginning of a checkpoint, and the actual memory usage is reported in Table IV. This simplifies our design, while doesn't change the question we are studying. We can observe that larger amount of memory is required by a larger threshold value. In large scale applications that produce huge process image files at checkpointing, a large threshold necessitates a huge memory usage that quickly exhausts available memory. This implies that a large threshold isn't practical for very large parallel applications.

Table IV
MEMORY USAGE PER NODE(IN MB)

	16 KB	64 KB	256 KB	512 KB
LU.C.64	42.6	50.0	78.2	81.1
BT.C.64	33.6	44.8	81.2	160.5
CG.C.64	39.2	48.8	64.8	76.0

VI. RELATED WORK

Checkpointing an application and restarting it from the last checkpoint is a widely adopted mechanism to serve fault tolerance. Many works have been done to provide checkpoint/restart facilities for a single application [9] [6] [19] [11] [3]. Checkpoint/restart mechanism has been incorporated into some message-passing libraries such as LAM/MPI [14], MVAPICH2 C/R [13], MPICH-V [4] and OpenMPI [7]. The heavy burden to do file IO at checkpoint/restart is already noticed in [8]. However existing work hasn't explicitly studied the multi-core characteristics in terms of checkpoint operations. Hence they cannot exploit the benefits of multi-core systems that host multiple processes in a same node. Milo et.al. [2] proposes to use a log-based file structure at server side to serialize all writing requests for checkpoint. It's tuned for a checkpoint writing pattern where multiple processes write to a single file. The server needs to be altered to adopt this file structure. This modification at file server side is usually not feasible for many existing applications. Therefore its application is severely constrained.

Another direction for fault tolerance is to proactively migrate the processes on a failing node to a spare node before the failure actually happens. The studies in [17] [16] propose to migrate a process while the parallel application keeps running. This approach can reduce the frequency to write a checkpoint file and hence alleviate the overhead of file IO. However, the effectiveness of migration approach heavily depends on the accuracy to predict a pending failure. If it fails to predict a failure, or if the warning is too late and a failure happens while the migration is going on, then the complete system has to do a restart to rollback to a latest checkpoint. Checkpoint files are still required, and the problem of high cost with checkpoint file I/O still exists.

VII. CONCLUSION AND FUTURE WORK

In this paper we aim to improve checkpoint operations within a multi-core system by using write aggregation. We profile the checkpoint data of the NAS parallel benchmarks with MVAPICH2 Checkpoint / Restart on a multi-core cluster, and characterize the checkpoint file writing patterns. Based on these patterns we have designed and implemented a framework to perform node-level write-aggregation to optimize small and medium writes within BLCR. Using an aggregation threshold of 512KB, the design shows improvements ranging from 27% to 32% in terms of time to take one checkpoint for the NAS parallel benchmarks used in our experiments.

Further optimization is possible to aggregate data from multiple nodes and perform a collective IO. We plan to extend our write-aggregation design to perform inter-node aggregation as the next step.

REFERENCES

[1] MPI over InfiniBand and iWarp. In *http://mvapich.cse.ohio-state.edu/*.

[2] Milo Polte and Jiri Simsa etc. . Fast log-based concurrent writing of checkpoints . In *PDSI 2008 workshop in conjunction with SC08* , Nov. 2008.

[3] Micah Beck, Jack J. Dongarra, and Graham E. Fagg. Harness: a next generation distributed virtual machine. *Future Generation Computer Systems*, 15(5-6):571–582, 1999.

[4] George Bosilca, Aurelien Bouteiller, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Vincent Neri, and Anton Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing*, pages 1–18, 2002.

[5] Cluster File System, Inc. Lustre: A Scalable, High Performance File System. http://www.lustre.org/docs.html.

[6] Duell, J., Hargrove, P., and Roman, E. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, 2002.

[7] J. Hursey, J.M. Squyres, T.I. Mattox, and A. Lumsdaine. The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. In *12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems*, March 2007.

[8] I.R. Philp. Software failures and the road to a petaflop machine. In *First Workshop on High Performance Computing Reliability Issues (HPCRI)*, February 2005.

[9] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. In *Technical Report UW-CS-TR-1346, University of Wisconsin-Madison, Computer Sciences Department*, April 1997.

[10] Paul H. Hargrove and Jason C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In *SciDAC*, 6 2006.

[11] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. Technical report, Knoxville, TN, USA, 1994.

[12] Q. Gao, W. Huang, M. Koop, and D. K. Panda. Group-based Coordinated Checkpointing for MPI: A Case Study on InfiniBand. In *Int'l Conference on Parallel Processing (ICPP)*, XiAn, China, 9 2007.

[13] Q. Gao, W. Yu, W. Huang and D. K. Panda. Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand. In *International Conference on Parallel Processing (ICPP)*, August 2006.

[14] S. Sankaran and J. M. Squyres and B. Barrett etc. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. *LACSI*, October 2003.

[15] V. Strumpen. Compiler Technology for Portable Checkpoints. submitted for publication (http://theory.lcs. mit.edu/ strumpen/porch.ps.gz). citeseer.ist.psu.edu/strumpen98compiler.html, 1998.

[16] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance. In *IPDPS*, pages 1–10, 2007.

[17] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration in HPC environments. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[18] Frederick C. Wong and Richard P. Martin etc. Architectural requirements and scalability of the NAS parallel benchmarks. In *Supercomputing*, page 41, 1999.

[19] Hua Zhong and Jason Nieh. CRAK: Linux Checkpoint/Restart As a Kernel Module. Technical report, Department of Computer Science, Columbia University, November 2001.