Fast Checkpointing by Write Aggregation with Dynamic Buffer and Interleaving on Multicore Architecture

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

***Abstract*—Large scale compute clusters continue to grow to ever-increasing proportions. However, as clusters and applications continue to grow, the Mean Time Between Failures (MTBF) has reduced from days to hours. As a result, fault tolerance within the cluster has become imperative.**

MPI, the de-facto standard for parallel programming, is widely used on such large clusters. Many MPI implementations use Checkpoint/Restart schemes using the Berkeley Lab Checkpoint Restart (BLCR) Library to achieve some level of fault tolerance. However, the performance of the Checkpoint/Restart mechanism does not scale well with increasing job size. As a result, the deployment of Checkpoint/Restart mechanisms for large scale parallel applications is compromised.

In our previous work, we proposed a technique to aggregate certain categories of checkpoint writes to reduce the checkpointing overhead. However, an application still experiences slow checkpoint writing because it is blocked waiting for its checkpoint file writes to complete. In this paper, we propose the Write Aggregation with Dynamic Buffer and Interleaving scheme to reduce the overhead related to checkpoint creation. By aggregating all checkpoint writes into a dynamic buffer pool and overlapping the application progress with the file writes, our algorithm is able to significantly reduce checkpoint creation overhead. In the experiments using 64 processor cores, our design demonstrates a speedup of 2.62 times in terms of checkpoint creation time when compared to the original BLCR design. Our scheme also reduces the impact of checkpointing on the application execution time from 20% to 6% when 3 checkpoints are taken during an application run.

I. Introduction

The trend in the High Performance Computing community over the past couple of years has been to use a large number of distributed processing elements, connected together using a high performance network interconnect. With this exponential increase in the number of components in the cluster, the Mean Time Between Failures (MTBF) has reduced from days to a couple of hours[12], [10]. As a result, it has become vital for such clusters to be equipped with fault tolerance capabilities.

This research is supported in part by DOE grants DE-FC02-06ER25755 and DE-FC02-06ER25749, NSF Grants CNS-0403342, CCF-0833169 and CCF-0702675; grants from Intel, Sun MicroSystems, Cisco Systems, and Linux Networks; and equipment donations from Intel, AMD, Advanced Clustering.

MPI is the de facto standard for parallel programming. Many scientific applications written in MPI take days to complete their computation. Given that the MTBF of modern clusters is smaller than the average running time of the application, failures are expected during the lifetime of a large scale application. Many MPI libraries have builtin checkpointing capabilities that allow applications to be checkpointed at regular intervals. Checkpointing saves the complete state of the MPI process to disk so that in the event of a failure, the process can be restarted from the saved image. Berkeley Lab Checkpoint/Restart software package (BLCR)[8] is a popular Checkpoint/Restart solution that is used by many MPI implementations, including MVAPICH2[1], [18], OpenMPI[11] and LAM/MPI[19].

Although BLCR has the capability to save and restore the MPI process's execution context, most modern interconnects store a substantial amount of information pertaining to the communication endpoint on the interconnect hardware itself. BLCR cannot access this information and so cannot save/restore the communication endpoint. Additionally, all the processes that are part of the MPI job must be in a consistent state before they are checkpointed. As a result, the process of checkpointing an MPI application usually involves the following phases.

Phase 1: Suspend communication between all processes in the parallel application and tear down the communication end points.

Phase 2: Use a checkpoint library to dump the individual process's memory image to a checkpoint file.

Phase 3: Re-establish connections among the MPI processes and continue execution.

At phase 2, a process's context and memory contents

are stored to a file on a reliable storage medium, usually a local disk or a parallel file system. Hence, the time spent in phase 2 dominates the time to create a checkpoint. To understand the process of checkpointing, we profiled the checkpoint writing for several applications from the NAS Parallel Benchmark suite[23] using BLCR and MVAPICH2[1]. The profiling provides insights into the characteristics of checkpoint writing.

Our previous work [24] explored the option of aggregating small and medium writes to a process's local buffer to reduce the checkpoint overhead. However, an application still experiences slow checkpoint writing because it is blocked waiting for its large checkpoint writes to complete. On multicore systems, this constraint becomes more severe due to many processes concurrently writing to the VFS layer.

In order to accelerate checkpointing, one has to decouple checkpoint writing from the slow file IO. However the file IO overhead is fixed for given amount of data and file system capabilities. The choice we make is to hide the checkpoint IO overhead from the application. Multiple application processes interleave their operations to copy their process image to a shared buffer pool, while a set of dedicated IO threads take care of writing the buffered data to disk files. In order to exploit the potentials of interleaving checkpoint data into the buffer pool and overlapping file IO with application progress, several questions must be addressed.

- How to construct the buffer pool to achieve both efficient data copy and low memory footprint?
- How should application processes access the buffer pool in order to improve data copy efficiency?
- What's the strategy for the IO threads to perform file write to improve memory utilization efficiency?

In this paper, we propose a Write Aggregation with Dynamic Buffer and Interleaving strategy to reduce the overhead related to checkpoint creation. By aggregating all checkpoint writes into a dynamic buffer pool and overlapping the application progress with the file writes, our design is able to accelerate checkpoint creation by 2.62 times compared to the original BLCR. In terms of application execution time, our new design brings down the overhead from 20% to 6% when 3 checkpoints are taken during the lifetime of the application.

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 profiling information collected for the NAS Parallel Benchmark to characterize checkpoint writing. In section 4, we present our detailed designs and discuss our design choices. In section 5, we conduct experiments evaluating our designs and present results that indicate improvement. In section 6, we discuss the related work. Finally we provide our conclusion and state the direction of the research we intend to conduct in future.

II. Background

Checkpointing is the process of saving the state of a program, usually to stable storage, at a given point of time during its execution, so that the program may be reconstructed at a later point in time. The process of reconstructing the program from a checkpoint is referred to as Restart.

A. Applications of Checkpoint/Restart

Checkpoint/Restart has many applications in the context of High Performance Computing[9].

Multiuser Scheduling: HPC Clusters usually employ a job scheduler which enables multiple users to share the cluster's resources. Based on the scheduling policy used by the scheduler, there may be a necessity to preempt a long running job to run a shorter job that arrived much later. Checkpoint/Restart can be used to achieve this preemption.

Application Migration: Well designed Checkpoint/Restart schemes allow processes to be checkpointed on one node and be restarted on another. This feature can be exploited to achieve process migration on computing clusters.

Application Backup: Checkpointing provides the backbone for fault tolerance through rollback recovery. An application maybe checkpointed periodically so that only the computation performed after the most recent checkpoint is lost in the event of a failure. The rest of the discussion focuses on this application of Checkpoint/Restart.

B. Checkpoint/Restart in MVAPICH2

MVAPICH2 is a MPI library with native support for InfiniBand and 10GigE/iWARP [1]. It supports application initiated and system initiated checkpointing [18], [17] using the BLCR Library for Checkpoint/Restart [15]. Checkpointing in MVAPICH2 involves the following three steps.

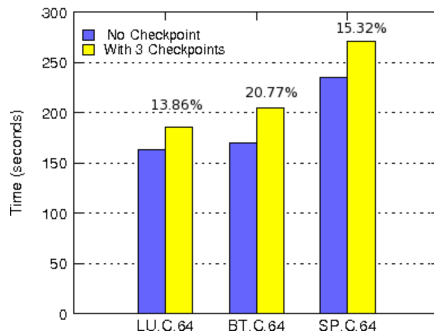
- Draining the communication channels of all pending messages and tearing down the communication endpoints on each process.
- Using the BLCR Library to independently request the checkpoint of every process that is part of the MPI job. The checkpoint is taken by BLCR in a blocking manner with the data being written to one file per process.
- Re-establishing the communication endpoints on every process.

The application continues its execution after the checkpoint is taken.

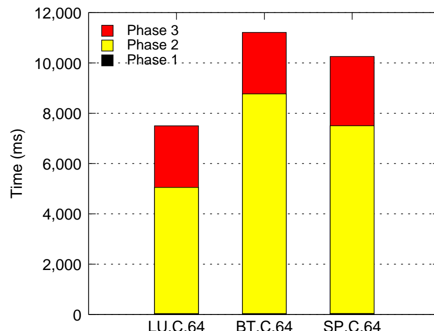
III. Profiling Checkpoint Creation

To understand the characteristics of checkpoint file IO, we ran NAS parallel benchmarks LU, BT and SP using the MVAPICH2 C/R framework with BLCR modified to provide profiling information. We chose Class C with 64 processes. The applications were run on a 64 node cluster. Each node has 8 processor cores on 2 Intel Xeon 2.33 GHz Quad-core CPUs. Each application was run on 8 nodes with one process per core. An application process wrote its checkpoint data to a separate checkpoint file on a local ext3 file system.

First, we measured the execution time of the application without checkpoints and with three checkpoints evenly distributed during the lifetime of the application. The results are displayed in Figure 1(a). The numbers above the bars indicate the overhead in execution time caused by the checkpoints. For example, BT.C.64 takes 169.9s to complete without any checkpoints. With the checkpoints, it takes 205.19s. This translates to an overhead of 20.77%. For larger scale application with thousands of processes, we expect the checkpoint overhead to be more adverse. Our observation is consistent with the results reported in [12].



(a) Application Execution Time



(b) Decomposition of Checkpoint Time

Fig. 1. NAS Parallel Benchmark Checkpoint Profiling

Figure 1(b) decomposes the time required for one checkpoint into phases 1, 2 and 3. In phase 1, communication is suspended and the end points are torn down. The time taken for phase 1 to complete is negligible compared to the other two phases. In phase 2, the process image is written to a checkpoint file, which is the major portion of the checkpoint time. In phase 3, communication channels are reestablished among all processes. The cost of phase 3 is relatively constant for a given number of processes. Table I indicates some basic information pertaining to the number of VFS writes and the size of the checkpoint data.

The information above indicates that phase 2 dominates the checkpoint overhead. In this phase, BLCR performs a lot of VFS writes to dump the process image to disk. We further profiled phase 2 of the checkpoint process and categorized all VFS writes into different classes based on the size of data written in each write. Figure 2 depicts this information. The first bar indicates the percentage of the total number of VFS writes that fall in the indicated range. The second bar indicates the percentage of the total data amount in the indicated range. The third bar indicates the percentage of the total time spent in doing the VFS write in the indicated range. We can see some interesting trend in this figure.

TABLE I
Basic Checkpoint Information

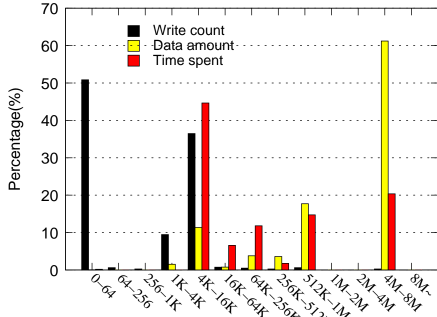
	LU.C.64	BT.C.64	SP.C.64
Checkpoint file size(MB) per process	23	40.0	39.5
Total data size(MB) per node	184	320.0	316.0
Number of VFS write per process	975	1057	1156
Total VFS write per node	7800	8456	9248

Firstly, a large portion of the VFS writes are associated with very small amount of data. For example in LU.C.64, more than 50% of VFS writes are of less than 64 bytes. Such writes account for less than 0.5% out of the total checkpoint data. These small writes are initiated by BLCR to save the process's open file table, CPU register set, timers, process/group/session id, signal handler table, metadata of Virtual Memory regions and other data structures that are small in size. Since these writes are buffered by the VFS layer, the time spent in this category is less than 0.2% of total time cost.

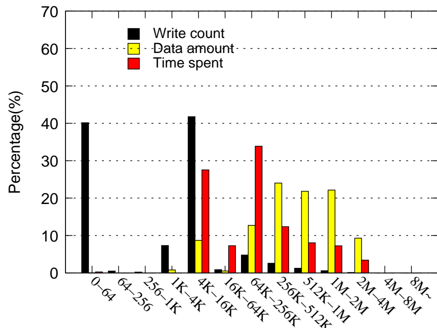
Secondly, there are a lot of writes in the range of 4KB to 64KB. These medium writes correspond to consecutive pages in a process's virtual memory area. For LU, about 37% of the VFS writes fall within the range of 4KB to 64KB. These account for about 12% of total checkpoint data. However, these writes account for about 50% of total write time. BT and SP also show a similar trend.

Thirdly, we find a few large writes that correspond to large blocks of consecutive pages in the process's Virtual

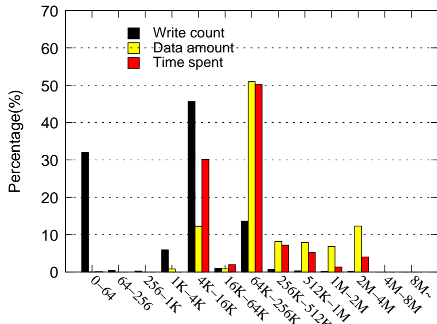
Memory (VM) area. In LU, for example, only 1% of the VFS writes are larger than 512 KB. However, 86% of checkpoint data is dumped by these large writes. They contribute to about 35% of the total checkpoint time.



(a) LU.C.64



(b) BT.C.64



(c) SP.C.64

Fig. 2. Profiling of Time Spent in Phase 2

In our previous work [24], we made initial attempts to aggregate small and medium VFS writes to a node's local buffer to reduce checkpoint overhead. By coalescing these writes and performing large VFS writes directly to the checkpoint file, we demonstrated that the checkpoint time could be significantly reduced. However, the previous work has a few drawbacks. Firstly, large VFS writes are directly performed to disk files. Since these account for the majority of data, a large part of the checkpoint time is wasted in waiting for the VFS layer to complete the writes. In a multicore system, this problem becomes more severe due to many processes concurrently writing to the VFS layer. Secondly, medium writes are aggregated to a buffer shared by all application processes. Although this marginally reduces the memory footprint, each write incurs additional overhead to synchronize access to the shared buffer.

In the following section, we propose new designs that can achieve better performances by overcoming these limitations.

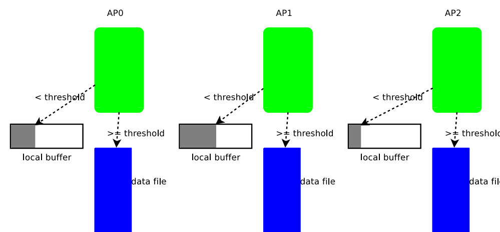
IV. Reduce Checkpoint Time by Write-Aggregation and Interleaving

In this section we present two improved design strategies to accelerate checkpointing parallel applications on multicore systems.

A. *Write Aggregation (WAG)*

Through the profiling data collected in section III, we find that medium VFS writes constitute a significant portion of checkpoint write time. We also find that a large portion of VFS writes are of very small size. Therefore we propose a basic Write Aggregation(WAG) design to coalesce all small and medium writes to a process-specific buffer, which is illustrated by figure 3(a).

This strategy differs from our previous work [24] in two important places. (1) In [24] all processes in a node send their medium writes to a shared buffer. This causes additional synchronization overhead for each write. In WAG, each process copies its medium writes to its own buffer. Therefore no inter-process synchronization is required. (2) In WAG, small writes are also absorbed into the same process-specific buffer to achieve a simplified buffer design. As a comparison, [24] allocates a separate local buffer for each process to aggregate small writes, which complicates the buffer management.



(a) Write Aggregation (WAG) Design

Process Rank	Data size	Original Offset	Data

(b) Format of a Chunk

Fig. 3. Node-level Write Aggregation Design

1) Design Strategy: Figure 3(a) illustrates our Write Aggregation (WAG) design. A parallel job has many application processes (APs) running on one node. When a checkpoint of a AP is requested, a buffer is allocated by BLCR and the VM area of the AP is copied to the allocated buffer, based on the following policy.

(1) If the write size is smaller than some threshold, it is copied to the local buffer till the buffer is filled. When the buffer is filled, it is written to a separate file on disk. The procedure is repeated as long as there is data available. Before the copy can happen, a header is prepended to the data. The structure of the header is shown in Figure 3(b). The header records the rank of the process, the size of data and the offset within the checkpoint file. The "original offset" is used to reconstruct the checkpoint file from the individual file fragments.

(2) If the data is greater than or equal to the threshold, it will not be aggregated. It will instead be directly written to a file. At the same time, a header will be prepended to this data to record the location of this chunk within the checkpoint file. This is again necessary to construct the checkpoint file.

2) Design Choices: The choice of the threshold plays an important role in the Write Aggregation design. It determines which VFS writes are copied to the local buffer, and which ones are directly written to disk. It also decides the amount of data that will be coalesced. In section V, we evaluate WAG design performance for different thresholds.

3) Restart with WAG: The restart process of our design follows the BLCR framework. Since we have altered the file organization, we have to reconstruct the checkpoint file to a format that can be interpreted by BLCR. We have designed an offline tool to construct checkpoint files by parsing the headers in the data files created by WAG. Once the checkpoint file is rebuilt for each process, the parallel job can be restarted from the checkpoint files.

B. Write Aggregation with Dynamic Buffer and Interleaving (WAG-DBI)

Although Write Aggregation effectively reduces the checkpointing overhead for small and medium writes, our previous design in [24] still shows a large delay in writing the checkpoint. Further investigation reveals that an AP spends a lot of time waiting for large VFS writes to complete before it can proceed to handle next chunk of data. Moreover, the wait time is worse for multicore systems where a lot of the APs on a same node, compete for IO. For a given amount of data, traditional checkpoint writing semantics that serialize VFS writes cannot do well.

To speed up checkpoint creation, we have to decouple checkpoint progress and the slow file IO. We propose to aggressively use the residual local memory to buffer all checkpoint writes. While an AP proceeds to buffer all checkpoint data in local memory, a set of IO threads write the buffered data to disk. All APs interleave their data copy into the shared buffer pool. By overlapping the file writing with checkpoint data copy, we expect a significant reduction in checkpoint write time at the cost of additional memory usage. On the other hand, a recent study [2] suggests that even large scale parallel jobs seldom use all available local memory. Therefore we feel it is reasonable to allocate part of the available local memory to the buffer pool to accommodate checkpoint data writing.

1) Design Strategy: Figure 4 illustrates our Write Aggregation with Dynamic Buffer and Interleaving (WAG-DBI) design. The most prominent difference of this design from the previous one is the use of a buffer pool that is shared by all processes on the node. When a checkpoint is requested, certain amount of memory is reserved for the buffer pool. When a process needs to write its checkpoint data, it first grabs a free chunk of buffer from the pool. All its data is copied to this chunk of buffer. If the chunk is filled, the process returns it to buffer pool, and requests for a new free chunk from the pool. This process is repeated by all APs till they finish writing all their checkpoint data.

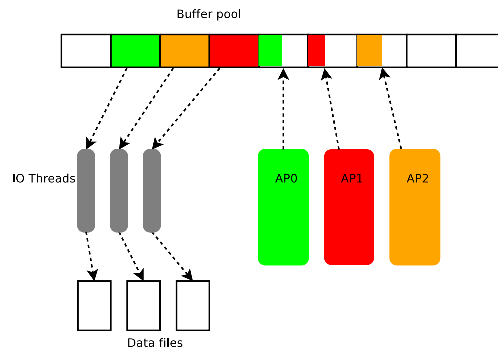


Fig. 4. Write Aggregation with Dynamic Buffer and Interleaving (WAG-DBI)

A set of IO threads constantly monitor the usage of the buffer pool. Once a free chunk is filled with data and returned to the buffer pool, an IO thread is activated to write this chunk to a separate file. The rank information and the offset of the data is encoded within the file. This information is used to rebuild the original checkpoint file. fsync() is called to flush all buffered data to disk before the file is closed. After the file is closed, the IO thread returns this chunk to the buffer pool as a free chunk.

2) Overlapping Between Application Processes and IO Threads: The benefit of WAG-DBI comes from its ability to hide the checkpoint write delay from the application. This is illustrated in Figure 5. At time *t1*, a checkpoint

is requested. All the APs enter phase 1 to suspend communications. They then enter phase 2 to store their process images using WAG-DBI. In this phase an AP repeatedly grabs free chunks from the buffer pool, copies data to the buffer chunk, and returns full chunks to the buffer pool. As full chunks are returned to pool, IO threads are woken up to write these chunks to disk. When an AP finishes copying its checkpoint data, it enters phase 3 to reestablish the communication channels with other processes. At some time *t2*, all APs return from phase 3. At this point of time, the parallel job resumes its computation. From an application's point of view, the "perceived checkpoint time" is $t2 - t1$. However, the checkpoint data is not completely written to disk till time *t3*. Hence, the "actual checkpoint time" is $t3 - t1$. The checkpoint delay experienced by an application is only $t2 - t1$.

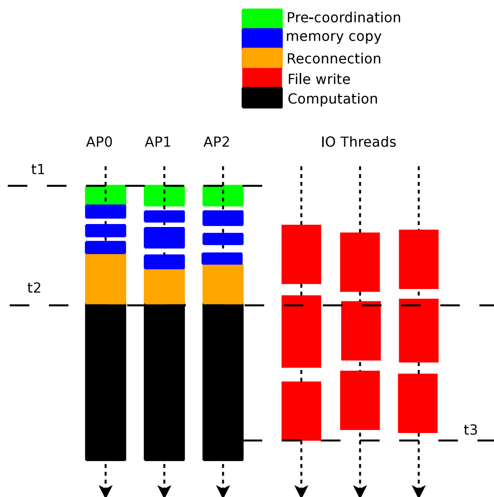


Fig. 5. How the APs and IO Threads Overlap

WAG-DBI effectively relaxes the file IO semantics. Instead of waiting for the data to be written to stable storage, a process returns from checkpoint writing once it has copied all its data to the buffer pool. In doing so we risk the possibility to lose a checkpoint if a failure happens during the period *t2* to *t3*. In situations where high data reliability is desired, we also provide an interface for an application process to poll the IO threads for the completion of all file writes. In the next section we measure the overlapping time $t3 - t2$ for different applications.

3) Design Choices: Several parameters play important roles in WAG-DBI design.

1) Buffer pool size. This parameter determines the degree of overlap between APs and IO threads. Large buffer pools provide higher opportunities to overlap, since more data can be held in the buffer pools. We measure the impact of this parameter in checkpoint creation in next section.
2) Chunk size. For a given size of the buffer pool, the chunk size determines the number of chunks in the buffer pool, and therefore impacts the waiting time of an AP to grab a free chunk. We intend to study the effect of this parameter in our future work.

V. Experimental Results

We have implemented WAG and WAG- designs into BLCR-0.8.0. We have also integrated the modified BLCR into MVAPICH2 1.2 [1] checkpoint/restart framework. In this section, we conduct various experiments to evaluate the performance of our design. A 64 nodes RedHat Enterprise Linux 5 cluster is used in the evaluation. Each node has 8 processor cores on 2 Intel Xeon 2.33 GHz Quad-core CPUs. All our experiments are based on MVAPICH2 1.2 as the MPI library with modified BLCR 0.8.0. In all the experiments conducted, the checkpoint files are written to the local ext3 file system. However our design is generic and can be applied to any file system.

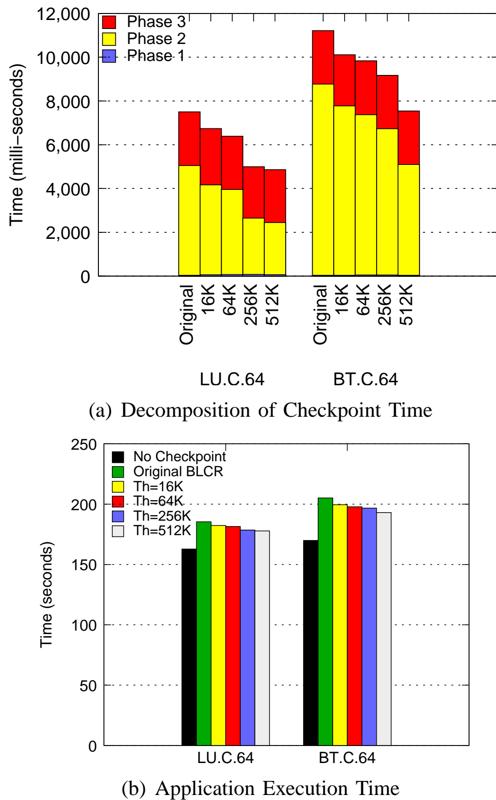
A. Performance of WAG

In this section we measure the checkpointing performance of the WAG scheme using the LU and BT applications from the NAS Parallel Benchmark suite 3.2.1 [23]. Each application is evaluated with class C and 64 processes. Each application process runs on a separate core, so 8 nodes(8 cores per node) are used in this experiment.

Figure 6(a) compares the time to make one checkpoint at different aggregation thresholds. The checkpoint time has been categorized into 3 phases as described in section I. "Phase 1" and "Phase 3" are the time spent by the MPI library to tear down and reestablish the communication end points. "Phase 2" is the time spent by MPI processes within the BLCR library to take a local checkpoint. "Original" refers to the current BLCR design without any optimization. The rest of the bars in the figure refer to the WAG design suggested in this work. The numbers below these bars indicate the value of the aggregation threshold.

From Figure 6(a), it can be seen that WAG can consistently reduce the checkpoint time. Furthermore, it is observed that the checkpoint time decreases with increasing values of aggregation threshold. In the example of LU.C.64, a threshold value of 16 KB reduces the checkpoint time by 10.18%. Larger threshold values 64 KB, 256 KB and 512 KB reduces checkpoint time by 14.84%, 33.41% and 35.20%, respectively. For BT.C.64, the corresponding numbers are 9.8%,12.30%, 18.21% and 32.75%. This is due to the fact that more writes are coalesced for larger thresholds.

We have also measured the overall application execution time, which is depicted in Figure 6(b). “No Checkpoint” represents the application execution time in the absence of checkpoints as the baseline for comparison. The rest of the bars indicate the application execution time with three checkpoints taken at equal intervals. Original BLCR produces an overhead of 13.86% and 20.77% for LU.C.64 and BT.C.64, respectively. WAG can reduce the corresponding overheads to 9.21% and 13.64% at threshold value 512 KB.



(a) Decomposition of Checkpoint Time

(b) Application Execution Time

Fig. 6. Evaluation of WAG

As the aggregation threshold is increased, the memory required to perform the aggregation also increases as shown in Table II. In this experiment with WAG, we let each process allocate a sufficiently large buffer at the beginning of a checkpoint, and report the actual amount of usage at different aggregation threshold. This simplifies the test, but doesn’t affect the characteristics of our study. This table can be related to Figure 2. Figure 2 shows the profiling of an process’s virtual memory size distribution, which is a snapshot of a process’s memory usage pattern at the moment when a checkpoint is taken. A process’s virtual memory usage pattern evolves over time, and Table II captures the total memory usage at certain threshold values after 3 checkpoints have completed. We can find that different applications require varied amount of memory for a given aggregation threshold, while increasing the threshold value always enlarges memory usage. On a multicore system where a lot of processes run on a same node, a large threshold can quickly exhaust local available memory. This makes using very large thresholds impractical for large parallel applications on multicore systems.

TABLE II
MEMORY USAGE PER NODE(IN MB)

	16 KB	64 KB	256 KB	512 KB
LU.C.64	42.6	50.0	78.2	80
BT.C.64	33.6	44.8	81.2	160.5

B. *Performance of WAG-DBI*

In this section we measure the checkpointing performance of the WAG-DBI scheme using 64 processes class C LU and BT applications from the NAS Parallel Benchmark suite. Since each process runs on a separate processor core, 8 nodes (8 cores per node) are used.

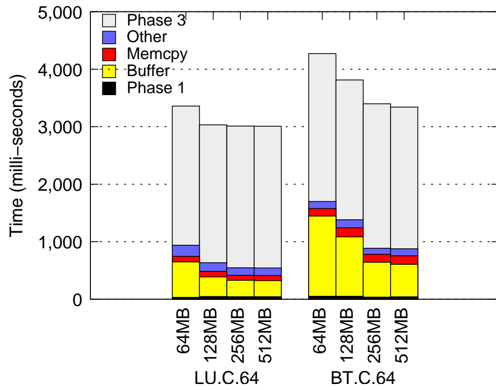
Figure 7(a) shows the breakdown of the time to do one checkpoint. The overhead is categorized into 3 phases as described in section I. Phase 2 is further divided into 3 parts. “Buffer” denotes the time spent by a process to acquire/return buffers. “Memcpy” denotes the time for a process to copy its memory image to the acquired buffers. “Other” denotes the time spent on the rest of the operations in phase 2, such as freezing threads, etc.

The values below each bar indicate the buffer pool sizes. A chunk size of 4 MB is used in each case. From this figure, it can be seen that WAG-DBI significantly reduces the time cost to make a checkpoint. WAG-DBI does a very good job in reducing the time spent in phase 2 to write checkpoint data. Although time spent in phase 1 and phase 3 remains constant, the total checkpoint time drops significantly as phase 2 dominates the overhead of a checkpoint. In the test with BT.C.64, original BLCR incurs an overhead of 11.2 seconds to make a checkpoint (indicated in Figure 1(b)), while WAG-DBI reduces this overhead to only 4.27 seconds when a 64 MB buffer pool is used. This leads to a speedup of 2.62 times. Faster checkpoint creation can be achieved when the buffer pool is enlarged, as can be seen in Figure 7(a).

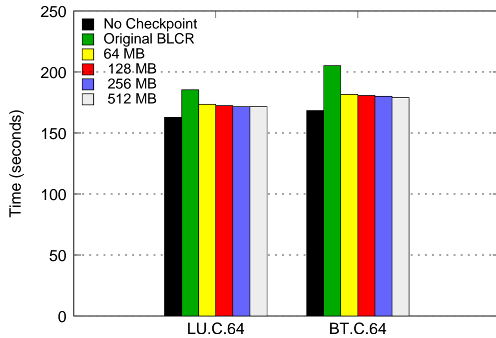
Figure 7(b) reports the overall application execution time at different buffer pool sizes. Take BT.C.64 for example. Without any checkpoints the application completes in 169.9 seconds. When 3 checkpoints are taken at equal intervals using original BLCR, the execution time is prolonged to 204.9 seconds, which implies the checkpoint overhead to be 20.77%. When WAG-DBI with 64 MB

buffer pool is used to take 3 checkpoints, the application completes in 181.5 seconds. The overhead is driven down to only 6.86%.

Using large buffer pool can further reduce checkpoint time, but the improvement flattens beyond certain amount of buffer. This is because the IO threads and application processes are totally overlapped at certain buffer pool size. Increasing buffer pool beyond this level isn't able to yield additional benefits. This "critical level" depends on an application's virtual memory usage pattern. We plan to investigate along this direction in future work.



(a) Decomposition of Checkpoint Time



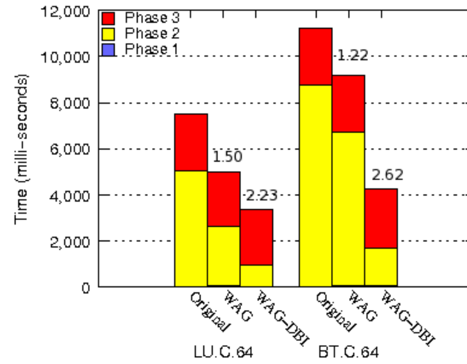
(b) Application Execution Time

Fig. 7. Evaluation of WAG-DBI

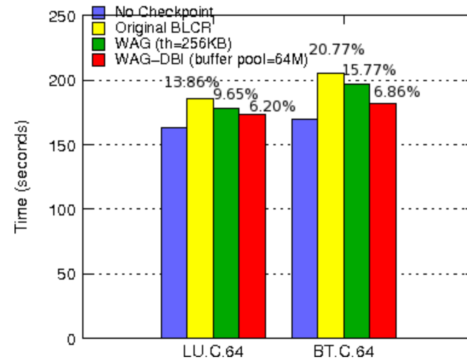
C. Comparing WAG and WAG-DBI

In this section we compare the performance of WAG and WAG-DBI at similar memory usage. We set the aggregation threshold of WAG to be 256 KB. With this setting, 78.2 MB and 81.2 MB memories are used at each node for one checkpoint by LU.C.64 and BT.C.64 respectively(as indicated in Table II). The buffer pool size is set to be 64 MB for WAG-DBI at each node.

Figure 8(a) shows the breakdown of the time to do one checkpoint. "Original" refers to the current BLCR design without any optimization. The numbers above each bar indicate the speedup achieved by different strategies compared to original BLCR. For application LU.C.64, WAG can speed up checkpoint creation by 1.5 times at memory usage of 78.2 MB. As a comparison, WAG-DBI yields a speedup of 2.23 times at 64 MB buffer pool. For BT.C.64, WAG can accelerate checkpointing by 1.22 times at memory usage of 81.2 MB, while WAG-DBI accelerates checkpointing by 2.62 times with 64 MB buffer pool.



(a) Decomposition of Checkpoint Time



(b) Application Execution Time

Fig. 8. Comparing WAG and WAG-DBI

WAG-DBI outperforms WAG in terms of checkpoint creation time with less amount of memory usage. The reason behind is that, WAG forces a process to wait for its large VFS write to complete before handling the next chunk of data, while WAG-DBI allows a process to proceed once it has handed over its checkpoint data to the buffer pool.

Figure 8(b) compares the application execution time with different strategies when 3 checkpoints are taken at equal intervals during the application run. "No Checkpoint" represents the execution time without any checkpoints. The numbers above each bar represent the overhead caused by 3 checkpoints with different checkpoint strategies. We can observe that WAG performs better than original BLCR. We also observe that WAG-DBI consistently outperforms WAG. At BT.C.64, WAG reduces

the overhead in execution time from 20.77% to 15.77%. WAG-DBI can further drive this overhead down to 6.86%.

D. WAG-DBI: Overlapping

WAG-DBI effectively overlaps IO and computation. In this experiment we measure WAG-DBI's overlapping time between IO threads and application processes at different buffer pool sizes. Figure 5 illustrates how IO threads are overlapped with application by WAG-DBI. After copying checkpoint data to the buffer pool, a checkpoint is regarded as completed at time *t2* when communication end points are reestablished between all processes. But IO threads haven't finished writing checkpoint data to disk files until time *t3*. The period between *t2* and *t3* is overlapping between IO and computation. Figure 9 reports this time at varied buffer pool sizes. The legend "N MB" represents the buffer pool size of N MB. Table I indicates that LU.C.64 generates 184 MB of data per node in one checkpoint, while BT.C.64 generates 320 MB data per node in one checkpoint. Therefore IO threads need longer time to write BT.C.64's checkpoint data to disk files, which leads to a longer overlapping time than LU.C.64. We also find this overlapping time tends to be shorter for a smaller buffer pool size. This is because an application process spends longer time in phase 2 to acquire free buffers at a smaller buffer pool(as can be seen in Figure 7(a)). IO threads start writing to files at phase 2. A longer phase 2 hides part of IO time. As a result, the remaining IO time after phase 2 becomes shorter leading to a shorter overlapping time.

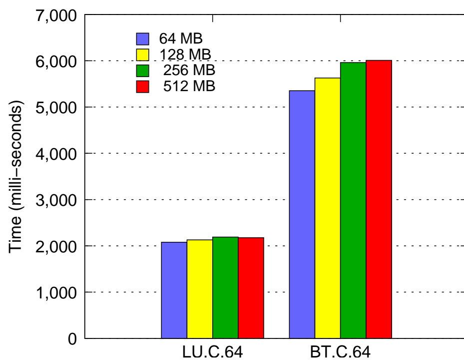


Fig. 9. IO Threads Overlapping with Application

E. WAG-DBI: Restart

In WAG-DBI, an IO thread writes each chunk of buffer to a separate file. The file name encodes the process id which generates this data, and the offset of this chunk in original checkpoint file. Reconstructing a checkpoint file only requires concatenating the individual chunk files in order of their offsets, the cost of which is linearly proportional to the original checkpoint file size. After recovering the checkpoint files, an application can be restarted using BLCR as usual. Table III measures the time cost to rebuild the checkpoint file for one application process. It also reports the time cost to restart an application. Take LU.C.64 for example. It takes 1.59 seconds to rebuild a checkpoint file for one process of LU.C.64. 9.25 seconds are required to restart the application LU.C.64. Cost to rebuild a checkpoint file is 17.19% of the restart cost. However the checkpoint data is read only at restart after a failure. Therefor the cost of rebuilding checkpoint files is largely amortized in the lifespan of an application run.

TABLE III
OVERHEAD AT RESTART (SECONDS)

	Time to rebuild one checkpoint	Restart time	percentage
LU.C.64	1.59	9.25	17.19
BT.C.64	3.18	9.52	33.40

VI. RELATED WORK

Checkpointing an application and restarting it from the last checkpoint is a widely adopted mechanism for serve fault tolerance. Many works have been done to provide checkpoint/restart facilities for standalone applications [14], [8], [25], [16], [5]. Checkpoint/restart mechanisms have been incorporated into MPI libraries such as LAM/MPI [19], MVAPICH2 C/R [18], MPICH-V [6] and OpenMPI [11].

The overhead of checkpoint/restart on file IO has been studied by [12]. Milo etc. [3] proposes the use of log-based file structures at the server side to serialize all file writing requests for checkpoint. This structure is optimized for a checkpoint writing pattern where multiple processes write to a single file. The server has to be altered to adopt this file structure which makes it infeasible for many existing applications. Stdchk [4] tries to scavenge spare storage resources from all participating nodes to form a dedicated storage space for checkpoint data. Our work differs from it in that we focus on utilizing local residual memory as a buffer pool. [13] proposes a CLL algorithm to reduce checkpoint overhead. It's a user-level optimization, and its buffer management incurs significant overhead to synchronize the copier thread and application thread on every common page access. On the contrary, our work is purely in kernel level, and our algorithms synchronize at chunk level which can largely mitigate buffer management overhead.

Another direction for fault tolerance is to proactively migrate the processes on a failing node to a spare node before the failure actually happens. [22], [21] propose to

migrate a process while the parallel application is active. However, the effectiveness of this scheme heavily relies on the accuracy to predict a pending failure. If it fails to predict a failure, or if the prediction comes too late, the migration itself can fail, in which case, the entire system has to rollback to the previous checkpoint. Hence, a complete checkpoint is still mandatory, in which case, our design is still relevant.

VII. Conclusion and Future Work

In this paper we propose a Write Aggregation with Dynamic Buffer and Interleaving (WAG-DBI) strategy to reduce the overhead related to checkpoint creation. By aggregating all checkpoint writes into a dynamic buffer pool and overlapping the application progress with the file writes, our design is able to accelerate checkpoint writing significantly.

As part of the future work, we plan to study the effect of varying chunk sizes on the WAG-DBI scheme. We also intend conducting experiments on the Lustre File System [7]. Additionally, we plan to explore the use of I/O AT to copy a process's memory contents to the buffer pool. Furthermore, we intend to perform collective IO [20] to aggregate data from multiple nodes.

References

[1] MPI over InfiniBand Project. In *http://nowlab.cse.ohio-state.edu/projects/mpi-iba/*.

[2] Application Requirements and Objectives for Petascale Systems . HPCwire, February 2008.

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

[4] S. Al-Kiswany, M. Ripeanu, S.S. Vazhkudai, and A. Gharaibeh. stdchk: A checkpoint storage system for desktop grid computing. June 2008.

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

[6] 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 *In Supercomputing*, pages 1–18, 2002.

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

[8] J. Duell. The design and implementation of berkeley lab's linux checkpoint/restart. Lawrence Berkeley National Laboratory,Paper LBNL-54941, April 2005.

[9] Duell, J., Hargrove, P., and Roman, E. Requirements for Linux Checkpoint/Restart. Technical Report LBNL-49659, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, 2002.

[10] Chung hsing Hsu and Wu chun Feng. A power-aware run-time system for high-performance computing. *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, Nov. 2005.

[11] 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.

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

[13] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 1994.

[14] 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.

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

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

[17] 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.

[18] 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.

[19] S. Sankaran and J. M. Squyres and B. Barrett etc. The lam/mpi checkpoint/restart framework: System-initiated checkpointing. *LACSI*, Oct. 2003.

[20] Rajeev Thakur and Ewing Lusk. Data sieving and collective i/o in romio. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, 1999.

[21] 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.

[22] 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.

[23] Frederick C. Wong and Richard P. Martin etc. Architectural requirements and scalability of the nas parallel benchmarks. In *Supercomputing '99*, page 41, 1999.

[24] Xiangyong Ouyang, Karthik Gopalakrishnan and Dhabaleswar K. Panda. Accelerating checkpoint operation by node-level write aggregation on multicore systems. *To appear in ICPP 2009*, September 2009.

[25] Hua Zhong and Jason Nieh. Crak: Linux checkpoint/restart as a kernel module. Technical report, Department of Computer Science, Columbia University, November 2001.