A Case Study of Incremental and Background Hybrid In-Memory Checkpointing

Xiangyu Dong†‡, Naveen Muralimanohar‡, Norman P. Jouppi‡, Yuan Xie†
Computer Science and Engineering, Pennsylvania State University†
Exascale Computing Lab, Hewlett-Packard Labs‡
{xydong,yuanxie}@cse.psu.edu†
{xiangyu.dong,naveen.muralimanohar,norm.jouppi}@hp.com‡

***Abstract*—Future exascale computing systems will have high failure rates due to the sheer number of components present in the system. A classic fault-tolerance technique used in today's supercomputers is a checkpoint-restart mechanism. However, traditional hard disk-based checkpointing techniques will soon hit the scalability wall.**

Recently, many emerging non-volatile memory technologies, such as Phase-Change RAM (PCRAM), are becoming available and can replace disks with the superior latency and power characteristics. Previous research has demonstrated that taking checkpoints at multiple levels referred to as *hybrid checkpointing* and employing PCRAM for taking local checkpoints can dramatically reduce checkpoint overhead and has the potential to scale beyond the exascale. In this work, we develop two prototypes to evaluate hybrid checkpointing. We find that, although global checkpointing is slow, by carefully scheduling checkpoint operations, we can hide its overhead using an extra checkpoint copy maintained in the local PCRAM of each node. In addition, as local checkpointing gets faster, taking more frequent checkpoints can help reduce the size of incremental checkpoints. However, in order to benefit from incremental checkpointing, the checkpoint interval has to be less than 10 seconds.

I. Introduction

Checkpoint-restart is a classic fault-tolerance technique that helps large-scale computing systems recover from unexpected failures or scheduled maintenance. However, the current state-of-the-art approach, which takes a snapshot of the entire memory image and stores it into a globally accessible storage disk at regular intervals, is no longer feasible.

There are two major roadblocks that severely limit checkpoint scalability. Firstly, future exascale systems will potentially have more than 100,000 processing units and their mean time to failure (a.k.a. MTTF) will be much shorter than existing petascale systems. One extreme example is the "ASCI Q" supercomputer, which has an MTTF of less than 6.5 hours [1]. Even with an optimistic socket MTTF of more than 5 years, we will soon arrive at a situation where MTTF of an exascale system with thousands of nodes will be as low as half an hour. This trend implies that more frequent checkpoint activities are required with an interval as low as a few minutes. Secondly, with hard disk drive being a mechanical device, it is extremely difficult to scale its bandwidth as the rotation speed and the seek latency are limited by physical constraints. Therefore, it is not feasible to use HDD and meet the checkpoint interval requirement that might be as low as several seconds. Table I lists the reported checkpoint time of a few modern supercomputers. It is clear that a checkpoint overhead of 20 to 30 minutes is typical in most systems. However, as the application size grows along with the system scale, the poor scaling of existing techniques can increase the overhead to several hours. Oldfield *et al.* [2] showed that a 1-petaFLOPS system can potentially take more than a 50% performance hit unless we significantly increase the I/O bandwidth of storage nodes. As this trend continues, very soon the failure period will be as small as the checkpoint overhead. In such a situation, a system either has to limit the number of nodes allocated to an application or risk ending up with an indefinite execution time.

TABLE I
TIME TO TAKE A CHECKPOINT ON SOME MACHINES OF THE TOP500. (SOURCE: LLNL)

Systems	Max performance	Checkpoint time (minutes)
LANL RoadRunner	1 petaFLOPS	~ 20
LLNL BlueGene/L	500 teraFLOPS	20
Argonne BlueGene/P	500 teraFLOPS	30
LLNL Zeus	11 teraFLOPS	26

Although the industry is actively looking at ways to reduce failure rates of computing systems, it is impractical to manufacture fail-safe computing components such as processor cores, memories, etc. A feasible solution is to make checkpointing techniques more efficient. As shown in Fig. 1, a major obstacle to the scalability of checkpointing is the limited I/O bandwidth of the centralized storage device. All checkpoint data have to go through these I/O nodes to get stored. Typically during a checkpoint process, all the process nodes (whose scale is much larger than the scale of I/O nodes) must take checkpoints at the same time and this causes high bandwidth stress on the I/O nodes and the centralized storage devices. To maintain the consistency of the system and maximize I/O bandwidth available for checkpointing, the workload is stalled until the checkpoint operation completes and this leads to a significant increase in workload execution time.

A scalable solution to this problem is to take checkpoints in a local storage medium. Fig. 2 shows a new organization in which the global checkpoints are still stored in the globally accessible storage devices through I/O nodes but each process node can take its own private local checkpoint as well. Since

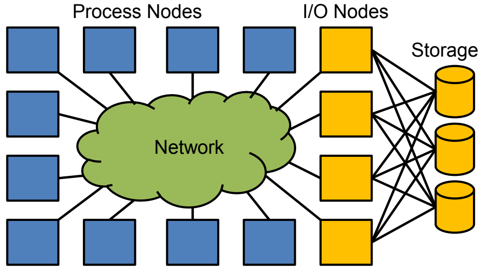

Fig. 1. The typical organization of the contemporary supercomputer. All the permanent storage devices are taken control by I/O nodes.

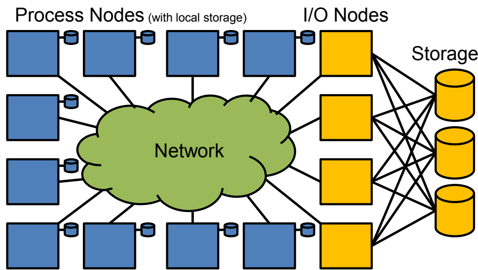

Fig. 2. The proposed new organization that supports local/global hybrid checkpoint. The primary permanent storage devices are still connected through I/O nodes, but each process node also has a permanent storage.

local checkpointing does not involve network transfers, it can be relatively fast. However, unlike global checkpoints that are accessible by any node in a system, local checkpoints are private to each node. Therefore, it cannot be reached in the event of a node loss or other permanent hardware failures. To provide complete protection, it is necessary to take both global and local checkpoints referred to as *hybrid checkpointing*. While this approach looks expensive, Dong *et al.* [3] showed that a significant checkpoint overhead reduction can be achieved by maintaining multiple checkpoints.

In an hybrid scheme, depending upon the percentage of the failures that can be solely recovered by local checkpoints, the local and global checkpoint ratio can be appropriately tuned. Hence, a system that is mostly plagued with soft errors (similar to "ASCI Q") need not have to take frequent global checkpoints. In addition, since a consistent state of the machine is already captured in the local checkpoint, global checkpointing can be done in parallel with program execution and this can dramatically reduce the execution time. Since the traditional HDD device is not fast enough to take frequent local checkpoints as required by the hybrid checkpointing scheme, emerging non-volatile technologies such as PCRAM [4] are ideally suited to hold local checkpoints.

Dong *et al.* [3] quantified the benefits of hybrid scheme through analytical modeling and simulations. However, the hidden overhead of *background checkpointing* and its interplay with other orthogonal optimizations such as *incremental checkpointing* [5]–[7] cannot be captured through simulations. For example, the I/O bandwidth in parallel systems are designed to either handle workload traffic or checkpoint data. While overlapping global checkpointing with execution is beneficial, it is not possible to capture its deleterious effects in the small simulation window. Similarly, incremental checkpointing is a promising technique to reduce checkpoint overhead. However, its effectiveness can vary depending upon the checkpoint interval and the characteristics of the workload. A thorough analysis of incremental checkpointing will require many days worth of simulations.

In this work, we build prototype platforms of a scaled-down parallel system with hybrid checkpointing and study the benefits of background and incremental checkpointing optimizations. We use Berkeley Lab Checkpoint/Restart (BLCR) library [8] to create coordinated checkpoints.

II. Related Work

There has been abundant work to reduce checkpoint overhead in distributed systems [9]. The most widely-used checkpoint protocol is *coordinated checkpointing*, which takes a consistent global checkpoint snapshot by flushing the in-transit messages and capturing the local state of each process node simultaneously [10]. *Uncoordinated checkpointing* reduces network congestion by letting each node take checkpoints at a different time. To rollback to a consistent global state, each node maintains a log of all incoming messages and takes multiple snapshots at different time. During recovery, the dependency information between various checkpoints are communicated to find a consistent state [11].

An alternate approach to reduce checkpoint overhead is to reduce the checkpoint size. *Memory exclusion* [12] is a software approach, in which programmers segregate the data into critical and non-critical, and reduces the checkpoint size by removing non-critical data such as buffers and temporary matrices. *Incremental checkpoint* [5]–[7] is another way to reduce the checkpoint size. It consists of saving only the differences between two consecutive checkpoints. Implemented at the OS level, this approach saves a checkpoint that only contains the virtual memory pages marked as dirty (modified). This approach requires several checkpoint images to be saved for a single process: at least the previous "reconstructed" checkpoint and the last incremental checkpoint. However, the effectiveness of incremental checkpointing reduces with increase in checkpoint interval. With the current generation systems having a checkpoint interval of several hours, the incremental checkpoint size is almost the same as the full checkpoint size.

Prior work on multilevel checkpointing considers either DRAM or local HDD for checkpoint storage [13]–[15]. The limited bandwidth of HDD coupled with its poor access time make it not suitable for fast checkpointing. While DRAM has superior latency properties, its volatile nature will significantly increase the complexity of multilevel checkpointing. For example, when using DRAM, to provide a complete fault coverage including power failures happening during checkpointing, we either have to maintain multiple copies of global checkpoints or employ a log based scheme for global

TABLE II
THE STATISTICS OF THE FAILURE ROOT CAUSE COLLECTED BY LANL DURING 1996-2005

Cause	Occurrence	Percentage
Hardware (hard error)	5163	21.7%
Hardware (soft error)	9178	38.7%
Software	5361	22.6%
Network	421	1.8%
Human	149	0.6%
Facilities	362	1.5%
Undetermined	3105	13.1%
Total	23739	100%

checkpoints. The use of non-volatile memory for local checkpointing significantly increases the number of faults covered by local checkpoints and reduces the probability of a global failure in the middle of a global checkpoint to less than 1% [3]. Vaidya [16] proposed a two level recovery scheme to reduce the checkpointing overhead. He employed a log based local checkpointing method for the first level and coordinated global checkpointing for the second level. The two level recovery scheme is aimed at using local checkpoints to recover from all single node failures and using global checkpoints only for multi-node failures. Since the first level checkpoints covers both transient and hard errors, local checkpoints are made in neighboring nodes which is an order of magnitude slower than taking checkpoints in the local storage. In addition, the use of log based scheme can result in domino effect [17].

Several checkpointing library implementations are also available for the HPC community, such as BLCR [8], Condor [18], CRAK [19], Libckpt [20], and C3 [21]. However, none of them implements the hybrid checkpointing scheme.

III. Local/Global Hybrid Checkpoint

State-of-the-art checkpointing relies on global centralized storage as shown in Fig. 1. Its main motivation is to recover from all failures including a complete node loss. However, global checkpoint availability is obtained at the cost of slow access speed and hence large performance overhead. After studying the failure events logged in the Los Alamos National Laboratory (LANL) from 1996 to 2005, which covers 22 high-performance computing systems, including a total of 4,750 machines and 24,101 processors [22], Dong *et al.* [3] observed that a majority of failures are transient in nature (Refer Table II. Transient errors such as soft errors (38.7% of total errors) together with software errors (22.6% of total errors) can be recovered by a simple reboot command. As a result, a significant number of failures can be recovered by taking local checkpoints private to each node.

Dong *et al.* [3] also projected that more than 83% of failures in a 1-petaFLOPS system can be recovered by local checkpoints while the remaining 17% of failures that include hard errors or node loss require globally accessible checkpoints. Based on this observation, more than 90% of the checkpointing operations can be made locally at a high speed without compromising the failure coverage, since the remaining globally accessible checkpoints can be used as the backup if necessary. Fig. 3 shows the conceptual view of the local/global hybrid checkpointing scheme, in which τ indicates the computation time slot. Every two computation slots, τ, are divided by either a global checkpoint indicated by δ_G or a local checkpoint indicated by δ_L. When a failure happens, a global recovery time R_G or a local recovery time R_L is added depending on whether the local checkpoint or the global checkpoint is used during the recovery

IV. Prototype Platforms

The primary motivation of this work is to study the overhead of background checkpointing and effectiveness of incremental checkpointing. We develop two prototypes: the first prototype uses existing libraries to model hybrid checkpointing and can execute MPI applications. The second prototype is built from scratch to specifically study incremental checkpointing.

As PCRAM is not yet available to the commercial market, we use half of the DRAM main memory space to be the local checkpoint storage. This device emulation is reasonable since the future PCRAM can be also be mounted on a Dual-Inline Memory Module (DIMM). While the write speed of PCRAM is slower than DRAM, data on PCRAM DIMM can be interleaved across PCRAM chips so that write operations can be performed at the same rate as DRAM without any stalls [3]. We also expect to see the write endurance of PCRAM improve to 10^{12} as projected by ITRS [23], which will eliminate the write limitations associated with PCRAM. Therefore, we can use DRAM to model PCRAM and assume the DRAM-based and the PCRAM-based checkpointing have the same behavior in terms of performance, regardless of the fact that PCRAM will be more energy-efficient due to its non-volatile nature.

A. Prototype 1

The first prototype is built using existing *Berkeley Labs Checkpoint/Restart* (BLCR) [8] and OpenMPI [24] solutions. The BLCR kernel is modified to add a “dump to memory” feature. We modify the **uwrite** kernel function that is responsible for BLCR to enable memory-based checkpointing. As the BLCR library is an independent module which merely controls the program execution, it can directly execute existing MPI application binaries without any changes to the source code. We further extend the kernel function to track and log the overhead of checkpointing overhead.

The overhead of each checkpoint-to-memory operation is measured by: 1. *kmalloc* that allocates memory; 2. *memcpy* that copies data to the newly-allocated memory space; 3. *free* the allocated memory. However, in Linux 2.6 kernel, *kmalloc* has a size limit of 128K, thus each actual memory-based checkpoint operation is divided into many small ones. This constraint slightly impacts on the memory write efficiency.

To evaluate the performance overhead of the background global checkpointing, the original memory-based checkpoint implementation cannot be used because it does not generate any real checkpoint region in the main memory. To have a

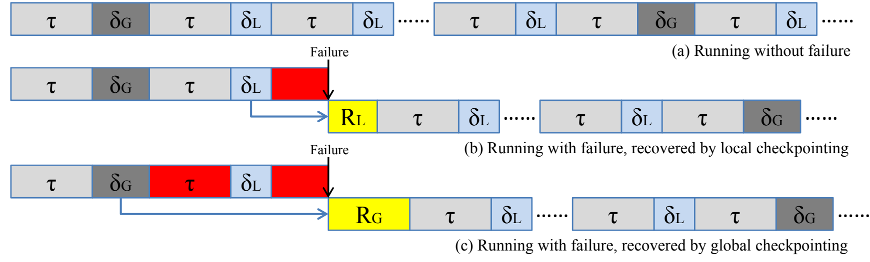

Fig. 3. A conceptual view of execution time broken by the checkpoint interval: (a) an application running without failure; (b) an application running with a failure, where the system rewinds back to the most recent checkpoint, and it is recovered by the local checkpoint; (c) an application running with a failure that cannot be protected by the local checkpoint. Hence, the system rewinds back to the most recent global checkpoint. The red block shows the computation time wasted during the system recovery.

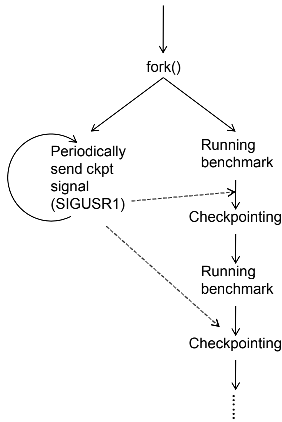

Fig. 4. The basic programming structure of checkpointing library for single-thread applications.

workable solution, the RAM file system, *ramfs*, is used to generate the actual checkpoint files.

B. Prototype 2

Although the BLCR kernel is a widely-used checkpoint/restart library, it does not have support for incremental checkpointing. Hence, we developed a second prototype fully from scratch which can be used to evaluate the timing overhead of both in-HDD and in-memory checkpoints. It supports both C and Fortran single-thread programs, and it enables the incremental checkpoint feature through bookkeeping [5].

The prototype consists of two parts: a primary thread that launches the target application and manages checkpoint intervals; a checkpoint library to be called by application. As shown in Fig. 4, a running shell spawns a new process to run the application that requires checkpointing. After that the shell periodically sends **SIGUSR1** signal to the application. The **SIGUSR1** signal handler is registered as a function to store checkpoints to hard disk or main memory. This approach requires modification to the source code, although the changes are limited to a couple of lines to invoke the handler.

The incremental checkpoint feature is implemented using the bookkeeping technique. After taking a checkpoint, all the writable pages are marked as read-only using an *mprotect* system call. When a page is overwritten, a page fault exception occurs, which sends the **SIGSEGV** signal, and the page fault exception handler saves the address of the page in an external data structure. The page fault signal handler also marks the accessed page as writable by using an *unprotect* system call. At the end of the checkpoint interval it is only necessary to scan the data structure that tracks the dirty pages. In this prototype, the register file and data in main memory are essential components of a whole checkpoint. Other components, such as pending signal and file descriptor, are not stored during the checkpointing operation because their attendant overhead can be ignored [8].

V. Experiments

A. Hardware Configuration

The aforementioned prototypes were developed using the C language on a hardware configuration with 2 Dual-Core AMD Opteron 2220 Processors and 16GB of ECC-protected registered DDR2-667 memory. Since the MPI synchronization overhead is on the scale of tens of microseconds [25] and is negligible compared to the checkpointing latency, we can use a scaled-down system employing two identical machines with the same configuration to evaluate MPI applications running across multiple nodes. Each machine is equipped with a Western Digital 740 hard disk drive that operates at 10,000 RPM with a peak bandwidth of 150MB/s reported in its datasheet. The experiment is run on a 64-bit Ubuntu Linux 2.6.28-15 and the checkpoint libraries are all compiled by gcc 4.3.3.

B. In-HDD and In-Memory Checkpoint Speed

We first investigated the actual speed difference between in-HDD and in-memory checkpointing. As a block device, the HDD had a large variation in effective bandwidth depending upon the access pattern. In our system, although the data sheet reports a peak bandwidth of 150MB/s, the actual working bandwidth is much smaller. We measure the actual HDD bandwidth by randomly copying files with different sizes and use the system clock to track the time spent. The result is plotted in Fig. 5, which shows all the points fall into two regions: one is near the y-axis and the other is at the 50MB/s line. When the write size is relatively small, the effective write

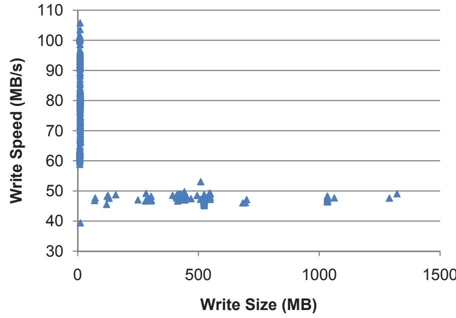

Fig. 5. The hard disk drive bandwidth with different write size.

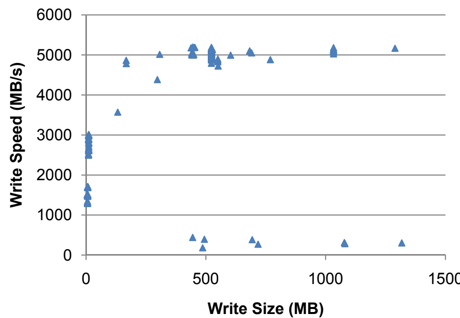

Fig. 6. The main memory bandwidth with different write size.

bandwidth of the HDD can be as high as 100MB/s and as low as 60MB/s. This amount of variation is caused by the HDD internal buffer. If the buffer is empty, the incoming data can be directly written to buffer which is significantly faster than the disk media; once the buffer overflows, the HDD needs to rotate and seek the proper sector to store the data. Hence, it can be observed that when the write size is in megabyte scale, the effective write bandwidth of HDD drops dramatically and the actual value is 50MB/s, which is only one third of its peak bandwidth of 150MB/s.

We performed a similar experiment to find the effective checkpointing speed in main memory by copying data of different sizes to newly-allocated memory addresses. The result is shown in Fig. 6. Similar to the hard disks, all the collected data fall into two regions. Note, for a very few cases, we noticed significant drop in write speed to DRAM. However, the frequency of their occurrence is low and we attribute this to noise in the simulation. However, unlike hard disks, the attainable bandwidth is higher when the write size is large due to the benefit achieved from spatial locality. This is desirable for checkpointing since checkpoint sizes are usually large. In addition, the achievable bandwidth is very close to 5333MB/s, which is the theoretical peak bandwidth of the DDR2-667 memory used in this experiment. Compared to the in-HDD checkpoint speed, the attainable in-memory speed can be two orders of magnitude faster.

We also recorded the actual time spent on in-HDD and in-memory checkpointing for a real application with 1.6GB memory footprint. The analysis is performed on a 4-threaded application, and we used BLCR platform that supports both HDD-based and memory-based checkpointing.

Table III and Table IV show the checkpoint size and the checkpoint completion time of HDD-based and memory-based techniques. Not surprisingly, taking checkpoints in memory is 50 times faster than taking checkpoints in hard disks. It is also clear that when all threads take checkpoint simultaneously, the effective bandwidth available per thread goes down. In addition, checkpointing process is further delayed compared to an application with single thread due to loss of spatial locality as illustrated in Fig. 5 and Fig. 6. More quantitatively, the HDD bandwidth reduces from 50MB/s to 33.7MB/s, and the memory bandwidth falls sharply from 5000MB/s to 1690MB/s due to increase in row buffer conflicts.

Hence, contention existing in multi-threaded checkpointing will cause a significant drop in performance especially when main memory is used to make checkpoints. This issue can be solved by:

- Serializing the checkpointing operations of multiple threads, so that only one thread can access the checkpoint storage device at a time.
- Only one thread is assigned to each process node, so that the entire memory bandwidth is available for the in-memory checkpointing.

C. Background Global Checkpointing

As mentioned earlier, the existence of local checkpoints in the hybrid scheme makes it possible to overlap global checkpointing with program execution. In order to find whether background checkpointing can effectively hide latency, we studied the following three scenarios:

1) *Without checkpointing:* The program is executed without triggering any checkpointing activities. This is the actual execution time of the program.
2) *With foreground checkpointing:* The program is executed with checkpoint enabled. Every checkpointing operation stalls the program, and takes snapshots into HDD directly.
3) *With background checkpointing:* The program is executed with checkpoint enabled. Every checkpointing operation stalls the program, takes snapshots into memory, and then copies them to HDD in the background.

While background checkpointing stalls the program to make local checkpoints, the overhead is significantly smaller due to the low DDR latency compared to HDD or network latencies. Background checkpointing makes it feasible to overlap the slow in-HDD global checkpoint process with program execution. In this experiment, the in-memory local checkpoint is implemented by *ramfs*, which mounts a portion of main memory as a file system. To study the impact of the number of involved cores on background checkpointing, 1-thread, 2-thread, and 4-thread applications were run in a quad-core processor, respectively[1]. The results are listed in Table V

[1]A 3-thread application is not included in the experiment setting because some benchmarks only allow radix-2 task partitioning.

TABLE III
HARD DISK-BASED CHECKPOINT PERFORMANCE OF A 4-THREAD APPLICATION

	Thread 1	Thread 2	Thread 3	Thread 4	Total
Checkpoint size (Byte)	417828448	417861216	417558112	417803872	1593.64M
Checkpoint latency (Second)	42.656668	44.998514	46.308757	47.220858	47.273500
Achieved bandwidth (MB/s)	9.3	8.9	8.6	8.4	33.7

TABLE IV
MEMORY-BASED CHECKPOINT PERFORMANCE OF A 4-THREAD APPLICATION

	Thread 1	Thread 2	Thread 3	Thread 4	Total
Checkpoint size (Byte)	417836216	417565880	417860792	417860872	1593.70M
Checkpoint latency (Second)	0.202096	0.501741	0.507173	0.687554	0.943159
Achieved bandwidth (MB/s)	1972	794	786	580	1690

TABLE V
EXECUTION TIME OF A 1-THREAD PROGRAM WITHOUT GLOBAL CHECKPOINTING, WITH GLOBAL CHECKPOINTING, AND WITH BACKGROUND GLOBAL CHECKPOINTING (UNIT: SECOND)

	1	2	3	4	5	6	Average
Without checkpointing	6.24	6.29	6.34	6.33	6.33	6.32	6.31±0.0014
With foreground checkpointing	9.18	9.69	7.03	7.03	6.99	7.03	7.83±1.58
With background checkpointing	6.36	6.35	6.36	6.37	6.22	6.39	6.34±0.0037

TABLE VI
EXECUTION TIME OF A 2-THREAD PROGRAM WITHOUT GLOBAL CHECKPOINTING, WITH GLOBAL CHECKPOINTING, AND WITH BACKGROUND GLOBAL CHECKPOINTING (UNIT: SECOND)

	1	2	3	4	5	6	Average
Without checkpointing	18.15	18.08	21.80	18.17	18.88	17.99	18.85±2.20
With foreground checkpointing	25.40	24.85	24.97	23.25	21.05	22.46	23.66±2.92
With background checkpointing	18.41	23.69	21.90	18.44	18.33	18.32	19.84±5.53

TABLE VII
EXECUTION TIME OF A 4-THREAD PROGRAM WITHOUT GLOBAL CHECKPOINTING, WITH GLOBAL CHECKPOINTING, AND WITH BACKGROUND GLOBAL CHECKPOINTING (UNIT: SECOND)

	1	2	3	4	5	6	Average
Without checkpointing	14.15	14.11	14.31	14.10	14.15	13.34	14.03±0.12
With foreground checkpointing	20.03	16.78	17.02	17.56	19.65	18.67	18.29±1.89
With background checkpointing	19.10	22.46	20.47	19.87	18.82	19.58	20.05±1.73

to Table VII, which show the total execution time with a single checkpointing operation performed in the middle of the program. Each configuration is run multiple times and the average value is considered for the evaluation.

We observe from the results that:

- Foreground checkpointing always takes about 25% performance loss due to low HDD bandwidth and this value is consistent with previous analytical evaluation [2].
- When main memory is used for taking checkpoints, the checkpoint overhead for a 1-thread application is around 0.5% (as listed in Table V. This overhead is 50 times smaller than the foreground case, and it is consistent with our previous finding that in-memory checkpointing is 50 times faster than in-HDD checkpointing.).
- The background checkpoint overhead increases from 0.5% to 5% when the application to be checkpointed becomes multi-thread. This is because of conflicts in row buffer due to interleaving of workload accesses with checkpointing. In addition, the MPI synchronization overhead is another source of the extra latency, since our checkpointing scheme is *coordinated*.
- The background checkpointing becomes ineffective when the number of threads equals to the number of available processor cores. Its associated overhead is even larger than in the foreground case. This is because there is no spare processor core to handle the I/O operations generated by the background checkpointing activity.

Therefore, although the background checkpointing technique is an effective tool to hide the impact of slow in-HDD checkpointing, designers need to ensure that a spare processor core is always available on each node when partitioning the computation task. Also, it is necessary to perform global checkpoints in a sequential manner to minimize row buffer conflicts.

D. Incremental Checkpoint Size

Since in-memory checkpointing makes it possible to take checkpoints every few seconds, it reduces the overhead of incremental checkpointing. As the checkpoint interval decreases, the probability of polluting a clean page becomes smaller, hence, the average size of an incremental checkpoint decreases.

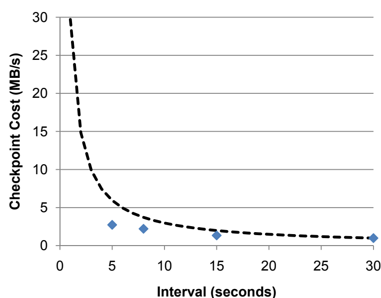

Fig. 7. Incremental checkpoint size (dot) and full checkpoint size (line) of CG.B

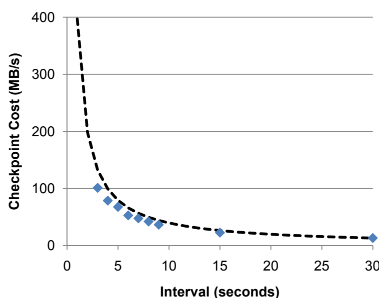

Fig. 8. Incremental checkpoint size (dot) and full checkpoint size (line) of MG.C

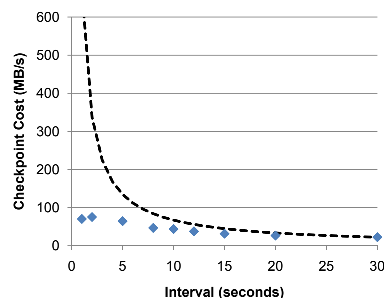

Fig. 9. Incremental checkpoint size (dot) and full checkpoint size (line) of IS.C

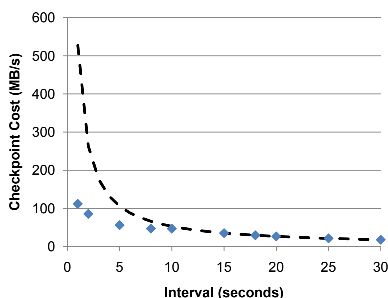

Fig. 10. Incremental checkpoint size (dot) and full checkpoint size (line) of BT.C

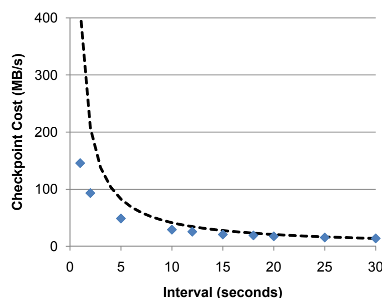

Fig. 11. Incremental checkpoint size (dot) and full checkpoint size (line) of UA.C

To measure the size difference between full checkpoints and incremental checkpoints, *Prototype 2* is used to trigger checkpoint operations with the interval ranging from 1 second to 30 seconds. Five workloads from the NPB benchmark with CLASS B and CLASS C configurations are tested. In order to have a fair comparison, a new metric, *checkpoint size per second*, is used to quantify the timing cost of checkpointing by assuming the checkpointing bandwidth is stable during the process. Fig. 7 to Fig. 11 show the checkpoint size of both schemes for different intervals.

It can be observed that, in all the five workloads, the incremental checkpoint size is almost the same as the full checkpoint size when the checkpoint interval is greater than 20 seconds. This shows that the incremental checkpointing scheme is not effective when the interval is not sufficiently small. Hence, checkpointing processes that involve accessing HDD or network transfers cannot benefit from incremental checkpointing. This could be the reason why the most popular checkpoint library, BLCR [8], does not support incremental checkpointing. As the interval size goes down, all the workloads except MG.C show a large reduction in checkpoint cost with incremental checkpointing. Based on this observation, it is clear that in-memory checkpointing is essential to achieve benefits from incremental checkpointing.

E. Frequent Incremental Checkpointing

Although the previous experiment shows the advantage of incremental checkpointing over full checkpointing, and the size of incremental checkpoints becomes smaller when the checkpoint interval is shortened, the number of checkpoints taken during a unit time is also increased. Therefore, the amount of data to be stored during a unit time might increase, which means the cumulative cost of frequent checkpointing might be bigger compared to using larger intervals.

Fig. 12 shows the result of seven different workloads. All costs are normalized to the cost at the 0.5-second interval. It shows that, for all the workloads, the incremental checkpoint costs begin to go down for checkpoint intervals longer than 0.5 seconds. Based on Fig. 7 to Fig. 12, it is clear that incremental checkpointing is beneficial only if the checkpoint interval is between 0.5 to 10 seconds. Taking more frequent checkpoints can always enhance the system reliability as it reduces the amount of useful work lost in the event of a failure. However, it is clear that when considering realistic checkpoint intervals, most workloads do not benefit from incremental checkpointing. Note, in our current implementation, the dirty pages are tracked via page fault handler at the OS level. In this case, the ultra-frequent checkpointing would cause almost all the memory accesses to result in a page fault and adds extra overhead to the program execution time. However, this issue can be solved by using hardware mechanisms to track dirty pages.

VI. Conclusion

Current in-HDD checkpointing cannot scale to future exascale systems. Emerging PCRAM technologies provide us a fast-access and non-volatile memory and make in-memory checkpointing an interesting alternative, especially for a hybrid

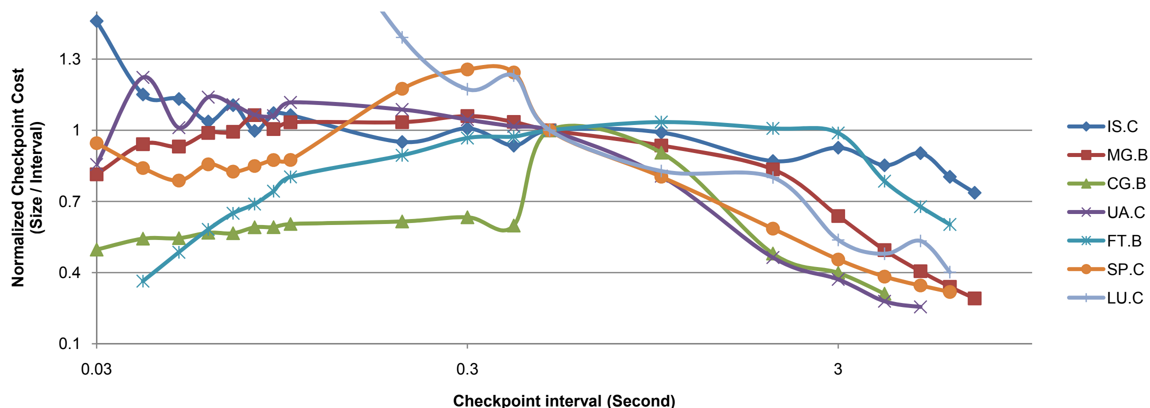

Fig. 12. The normalized incremental checkpoint cost under ultra-frequent checkpointing.

scheme that takes both local and global checkpoints and provide complete fault coverage. In this work, we extend the previous research on local/global checkpointing, and build two prototypes to study the benefits of background and incremental checkpointing. The background global checkpointing is targeted to hide the long global in-HDD checkpoint latency, and frequent incremental checkpointing is targeted to reduce the overall checkpoint size. In our evaluation we found that with proper scheduling, background checkpointing is effective in hiding global checkpoint latency. On the other hand, incremental checkpointing is only beneficial if checkpoints are made at a high frequency.

References

[1] D. Reed, "High-End Computing: The Challenge of Scale," Director's Colloquium, May 2004.

[2] R. A. Oldfield, S. Arunagiri, P. J. Teller *et al.*, "Modeling the Impact of Checkpoints on Next-Generation Systems," in *MSST '07. Proceedings of the IEEE Conference on Mass Storage Systems*, 2007, pp. 30–46.

[3] X. Dong, N. Muralimanohar, N. P. Jouppi *et al.*, "Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Future Exascale Systems," in *SC '09. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2009.

[4] S. Raoux, G. W. Burr, M. J. Breitwisch *et al.*, "Phase-Change Random Access Memory: A Scalable Technology," *IBM Journal of Research and Development*, vol. 52, no. 4/5, 2008.

[5] R. Gioiosa, J. C. Sancho, S. Jiang *et al.*, "Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers," in *SC '05. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2005.

[6] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive Incremental Checkpointing for Massively Parallel Systems," in *ICS '04. Proceedings of the ACM International Conference on Supercomputing*, 2004, pp. 277–286.

[7] J. Heo, S. Yi, Y. Cho, J. Hong, and S. Y. Shin, "Space-Efficient Page-Level Incremental Checkpointing," in *SAC '05. Proceedings of the ACM Symposium on Applied Computing*, 2005, pp. 1558–1562.

[8] J. Duell, P. Hargrove, and E. Roman., "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-54941, 2002.

[9] E. N. Elnozahy, D. B. Johnson, and W. Zwaenpoel, "The Performance of Consistent Checkpointing," in *SRDS '92. Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 1992, pp. 39–47.

[10] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.

[11] Y.-M. Wang, P.-Y. Chung, I.-J. Lin *et al.*, "Checkpoint Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 5, pp. 546–554, 1995.

[12] J. S. Plank, Y. Chen, K. Li *et al.*, "Memory Exclusion: Optimizing the Performance of Checkpointing Systems," *Software - Practice and Experience*, vol. 29, no. 2, pp. 125–142, 1999.

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

[14] K. Li, J. F. Naughton, and J. S. Plank, "Low-Latency, Concurrent Checkpointing for Parallel Programs," in *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[15] J. S. Plank, "Improving the Performance of Coordinated Checkpointers on Networks of Workstations Using RAID Techniques," in *SRDS '96. Proceedings of the 15th Symposium on Reliable Distributed Systems*, 1996, pp. 76–85.

[16] N. H. Vaidya, "A Case for Two-Level Distributed Recovery Schemes," in *SIGMETRICS '95. Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, 1995, pp. 64–73.

[17] B. Randell, "System Structure for Software Fault Tolerance," in *IEEE Transactions on Software Engineering*, 1975.

[18] M. Litzkow, T. Tannenbaum, J. Basney *et al.*, "Checkpoint and Migration of Unix Processes in the Condor Distributed Processing System," University of Wisconsin, Madison, Tech. Rep. CS-TR-199701346, 1997.

[19] H. Zhong and J. Nieh, "CRAK: Linux Checkpoint/Restart as a Kernel Module," Columbia University, Tech. Rep. CUCS-014-01, 2001.

[20] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," in *USENIX '95. Proceedings of the USENIX Technical Conference*, 1995, pp. 213–223.

[21] G. Bronevetsky, D. Marques, K. Pingali *et al.*, "C3: A System for Automating Application-Level Checkpointing of MPI Programs," in *LCPC '03. Proceedings of the International Workshop on Languages and Compilers for Parallel Computers*, 2003, pp. 357–373.

[22] Los Alamos National Laboratory, Reliability Data Sets, http://institutes.lanl.gov/data/fdata/.

[23] International Technology Roadmap for Semiconductors, "Process Integration, Devices, and Structures 2007 Edition," http://www.itrs.net/.

[24] Open MPI: Open Source High Performance Computing, http://www.open-mpi.org/.

[25] W. Jiang, J. Liu, H.-W. Jin, D. K. Panda *et al.*, "High Performance MPI-2 One-Sided Communication over InfiniBand," in *CCGRID '04. Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*, 2004, pp. 531–538.