Hybrid Checkpointing for MPI Jobs in HPC Environments

Chao Wang[1], Frank Mueller[1], Christian Engelmann[2], Stephen L. Scott[2]
[1] Department of Computer Science, North Carolina State University, Raleigh, NC (mueller@cs.ncsu.edu)
[2] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN

Abstract

As the core count in high-performance computing systems keeps increasing, faults are becoming common place. Checkpointing addresses such faults but captures full process images even though only a subset of the process image changes between checkpoints.

We have designed a hybrid *checkpointing technique for MPI tasks of high-performance applications. This technique alternates between full and incremental checkpoints: At incremental checkpoints, only data changed since the last checkpoint is captured. Our implementation integrates new BLCR and LAM/MPI features that complement traditional full checkpoints. This results in significantly reduced checkpoint sizes and overheads with only moderate increases in restart overhead. After accounting for cost and savings, benefits due to incremental checkpoints are an order of magnitude larger than overheads on restarts. We further derive qualitative results indicating an optimal balance between full/incremental checkpoints of our novel approach at a ratio of 1:9, which outperforms both always-full and always-incremental checkpointing.*

1. Introduction

Recent progress in high-performance computing (HPC) has resulted in remarkable Terascale systems with 10,000s or even 100,000s of processing cores. At such large counts of cores, faults are becoming common place. Reliability data of contemporary systems illustrates that the mean time between failures (MTBF) / interrupts (MTBI) is in the range of 6.5-40 hours depending on the maturity / age of the installation [1]. The most common causes of failure are processor, memory and storage errors / failures. Table 1 presents an excerpt from a Department of Energy (DOE) study that summarizes the reliability of several state-of-the-art supercomputers and distributed computing systems [1], [2]. When extrapolating for current systems in such a context, the MTBF for peta-scale systems is predicted to be as short as 1.25 hours [3].

System	# Cores	MTBF/I	Outage source
ASCI Q	8,192	6.5 hrs	Storage, CPU
ASCI White	8,192	40 hrs	Storage, CPU
PSC Lemieux	3,016	6.5 hrs	
Google	15,000	20 reboots/day	Storage, memory
Jaguar 4Core	150,152	37.5 hrs	Storage, memory

TABLE 1. Reliability of HPC Clusters

In such systems, frequently deployed checkpoint/restart (C/R) mechanisms periodically checkpoint the entire process image of all MPI tasks. The wall-clock time of a 100-hour job could well increase to 251 hours due to the C/R overhead of contemporary fault tolerant techniques implying that 60% of cycles are spent on C/R alone [3]. Recent investigations [4] revealed that checkpoint/restart efficiency, *i.e.*, the ratio of useful vs. scheduled machine time, can be as high as 85% and as low as 55% on current-generation HPC systems. However, only a subset of the process image changes between checkpoints. In particular, large matrices that are only read but never written, which are common in HPC codes, do not have to be checkpointed repeatedly. Also, coordinated checkpointing for MPI jobs, which is commonly deployed, requires all the MPI tasks to save their checkpoint files at the same time, which leads to extremely high I/O bandwidth demand.

Contributions: (1) This paper contributes a novel approach for *hybrid* checkpointing of MPI tasks that is transparently integrated into an MPI environment. In contrast, prior approaches only considered full or incremental checkpoints in isolation [5], [6]. Our hybrid mechanism complements full checkpoints with incremental ones, which is superior to either checkpointing scheme in isolation.

(2) We contribute a full-fledged implementation of our mechanisms over LAM (Local Area Multicomputer)/MPI's C/R support [7] through Berkeley Labs C/R (BLCR) [8]. Traditional LAM/MPI+BLCR [9] only supports full checkpointing. While implemented within LAM/MPI and BLCR, our mechanisms are applicable to any process-migration solution, *e.g.*, the Open MPI FT mechanisms [10].

(3) We keep the overhead of a restart operation low via a traversal mechanism that ensures linear overhead in the number of *disjoint* pages over a set of full/incremental checkpoints (rather than in the total number of pages saved). While the latest memory mapping information is maintained and used by the restart operation, the pages saved by the preceding checkpoints but unmapped later are skipped.

(4) We conduct experiments on a cluster that quantitatively show significant reductions in the size of checkpoint files and the overhead of checkpoint operations for our hybrid C/R. Hybrid checkpoints save 16 seconds wallclock time on average for all the cases by replacing three full checkpoints with incremental ones while overheads of restarts (if required) are an order of magnitude smaller for our experiments.

(5) From these experiments and an abstract cost model, we derive an optimal balance of full/incremental checkpoints at a ratio of 1:9, which appears to be the first time such rates have been reported based on concrete, hybrid experiments.

The paper is structured as follows. Section 2 presents the design of our hybrid C/R mechanism. Section 3 identifies and describes the implementation details. Subsequently, the experimental framework is detailed and measurements for our experiments are presented in Section 4 and 5, respectively. Our contributions are contrasted with prior work in Section 6. The work is then summarized in Section 7.

This work was supported in part by NSF grants CCR-0237570 (CAREER), CNS-0410203, CCF-0429653, CCF-1058779 and DOE DE-FG02-08ER25837. The research at ORNL was supported by Office of Advanced Scientific Computing Research and DOE DE-AC05-00OR22725 with UT-Battelle, LLC.

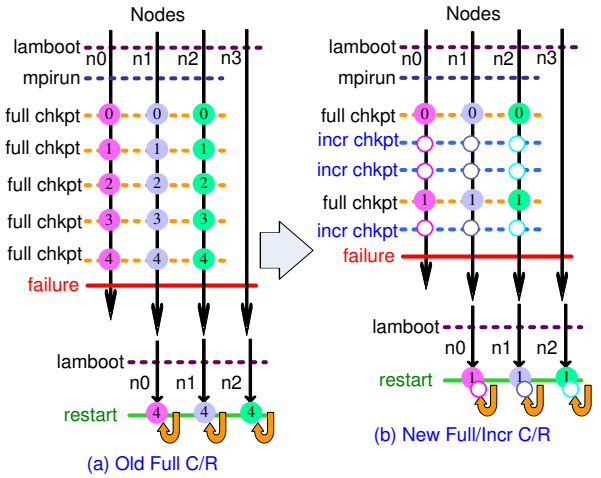


Fig. 1. Full vs. Hybrid C/R Mechanisms

2. Design

This section presents an overview of the design of our hybrid C/R mechanism with LAM/MPI and BLCR. In contrast to prior work, we view incremental checkpoints as complementary to full checkpoints in the following sense. Every *n-th* checkpoint will be a full checkpoint to capture an application without prior checkpoint data while any checkpoints in between are incremental, as illustrated in Fig. 1(b). Such process-based incremental checkpointing reduces checkpoint bandwidth and storage space requirements, and it leads to a lower rate of full checkpoints. Yet, it keeps storage requirements and restore overhead at bay as any full or incremental checkpoints prior to a full one are no longer required.

In the following, we first discuss scheduler integration of hybrid C/R. We then detail system-level aspects of hybrid checkpoints at two levels. First, the synchronization and coordination operations (such as the in-flight message drainage among all the MPI tasks to reach a consistent global state) at the job level are provided. Second, dirty pages and related meta-data image information saved at the process/MPI task level, as depicted in Fig. 3, are discussed. We employ filtering of "dirty" pages at the memory management level, *i.e.*, memory pages modified (written to) since the last checkpoint are restored after node failures to restart from the composition of full and incremental checkpoints.

Scheduler

We designed a decentralized scheduler, which can be deployed as a stand-alone component or as an integral process of an MPI daemon, such as the LAM daemon (lamd). The scheduler will issue the full or incremental checkpoint commands based on user-configured intervals or the system environment, such as the execution time of the MPI job, storage constraints for checkpoint files and the overhead of preceding checkpoints. The algorithms to periodically trigger full/incremental checkpoints are not quatitatively analyzed this paper.

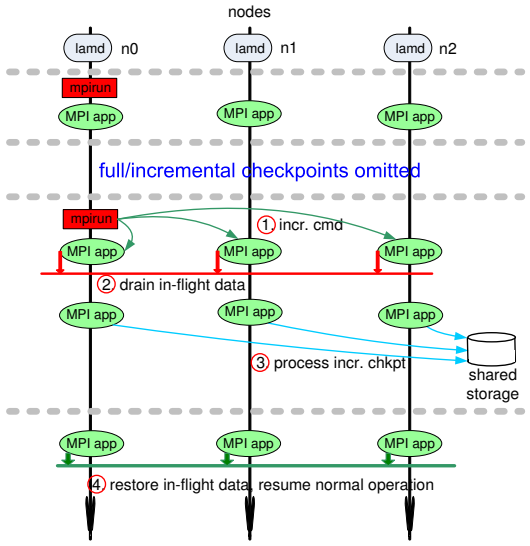


Fig. 2. Incremental Checkpoint at LAM/MPI

Upon a node failure, the scheduler initiates a "job pause" mechanism in a coordinated manner that effectively freezes all MPI tasks on functional nodes and migrates processes of failed nodes [5]. All nodes, functional (paused ones) and migration targets (replaced failed ones), are restarted from the last full plus n incremental checkpoints, as explained in Section 2.

Hybrid Checkpointing at the Job Level

Hybrid checkpointing at the job level is performed in a sequence of steps depicted in Fig. 2 and described below.

Step 1: Incremental Checkpoint Trigger: When the scheduler decides to engage in an incremental checkpoint, it issues a corresponding command to the *mpirun* process, the initial LAM/MPI process at job invocation. This process, in turn, broadcasts the command to all MPI tasks.

Step 2: In-flight Message Drainage: Before we stop any process and save the remaining dirty pages and the corresponding process state in checkpoint files, all MPI tasks coordinate a consistent global state equivalent to an internal barrier. Based on our LAM/MPI+BLCR design, message passing is handled at the MPI level while the process-level BLCR mechanism is not aware of messaging at all. Hence, we employ LAM/MPI's job-centric interaction mechanism for the respective MPI tasks to clear in-flight data in the MPI communication channels.

Step 3: Process-level Incremental Checkpoints: Once all the MPI tasks (processes) reach a globally consistent state, all of them engage in process-level incremental checkpoints independently (see below).

Step 4: Messages Restoration and Job Continuation: Once the process-level incremental checkpoint has been committed, drained in-flight messages are restored, and all processes resume execution from their point of suspension.

Hybrid Checkpointing at the Process Level

Hybrid checkpointing of MPI tasks (step 3 in Fig. 2) is performed at the process level, which is shown in detail

in Fig. 3. Compared to a full checkpoint, the incremental variant lowers the checkpoint overhead by saving only those memory pages modified since the last (full or incremental) checkpoint. This is accomplished via our BLCR enhancements by activating a handler thread (on right-hand side of Fig. 3) that signals compute threads to engage in the incremental checkpoint. One of these threads subsequently saves modified pages before participating in a barrier with the other threads, as further detailed in Section 3.

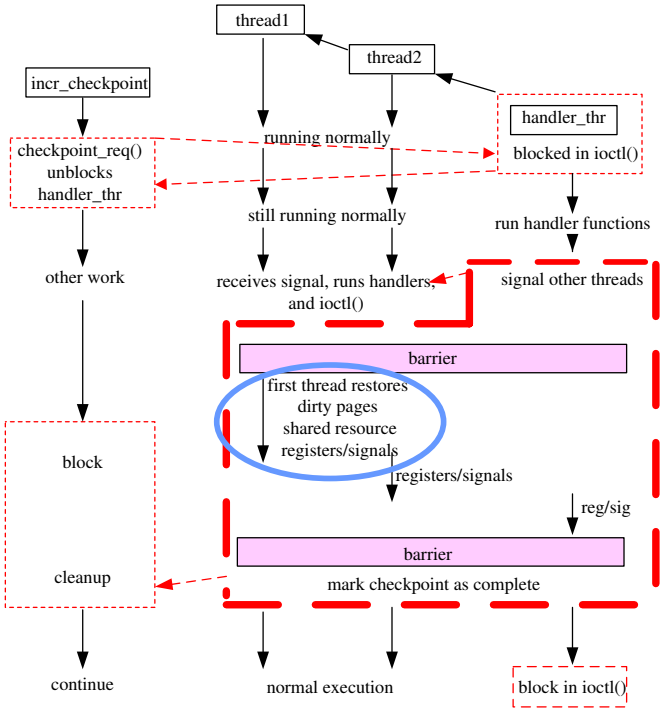


Fig. 3. BLCR with Incremental Checkpoint in Bold Frame

A set of three files serve as storage abstraction for a checkpoint snapshot, as depicted in Fig. 4:
(1) *Checkpoint file a* contains the memory *page content*, *i.e.*, the data of only those memory pages modified since the last checkpoint.
(2) *Checkpoint file b* stores memory *page addresses*, *i.e.*, address and offset of the saved memory pages for each entry in *file a*.
(3) *Checkpoint file c* covers other *meta information*, *e.g.*, linkage of threads, register snapshots, and signal information pertinent to each thread within a checkpointed process / MPI task.

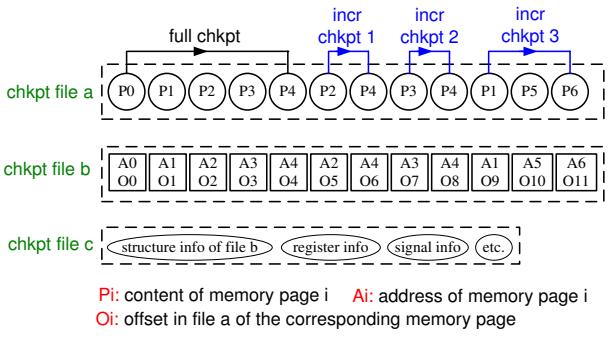


Fig. 4. Structure of Checkpoint Files

File a and *file b* maintain their data in a log-based append mode for successive incremental checkpoints. The last full and subsequent incremental checkpoints will only be discharged (marked for potential removal) once the next full checkpoint has been committed. Their availability is required for the potential restart up until a superseding checkpoint is written to stable storage. In contrast, only the latest version of *file c* is maintained since all the latest information is saved as one meta-data record, which is sufficient for the next restart.

In addition, memory pages saved in *file a* by an older checkpoint can be discharged once they are captured in a subsequent checkpoint due to page modifications (writes) since the last checkpoint. For example, in Fig. 4, memory page 4 saved by the full checkpoint can be discharged when the first incremental checkpoint saves the same page. Later, the same page saved by the first incremental checkpoint can be discharged when it is saved by the second incremental checkpoint. In our on-going work, we are developing a garbage collection thread for this purpose. Similar to segment cleaning in log-structured file systems [11], the file is divided into segments (each of equal size as they represent memory pages) that are written sequentially. A separate garbage collection thread tracks these segments within the file, removes old segments (marked appropriately) from the end and puts new checkpointed memory data into the next segment. As a result, the file morphs into a large circular buffer as the writer thread adds new segments to the front and the cleaner thread removes old segments from the end toward the front (and then wraps around). Meanwhile, checkpoint *file b* is updated with the new offset information relative to *file a*.

Modified Memory Page Management

We utilize a Linux kernel-level memory management module that has been extended by a page-table dirty bit scheme to track modified pages between checkpoints [12]. This is accomplished by duplicating the dirty bit of the page-table entry (PTE) and extending kernel-level functions that access the PTE dirty bit so that the duplicate bit is set with negligible overhead (see [12] for details).

MPI Job Restart from Hybrid Checkpoints

Upon a node failure, the scheduler coordinates the restart operation on both the functional nodes and the spare nodes. First, the process of *mpirun* is restarted, which, in turn, issues the restart command to all the nodes for the MPI tasks. Thereafter, recovery commences on each node by restoring the last incremental checkpoint image, followed by the memory pages from the preceding incremental checkpoints in reverse order up to the pages from the last full checkpoint image, as depicted in Fig. 5. The reverse-order scan over all incremental checkpoints and the last full checkpoint allows the recovery of the last stored version of a page in linear complexity relative to the number of pages to be restores. While the meta-information of repeatedly stored pages is traversed for

pages that had been modified between checkpoints, the content of any page only needs to be written once to facilitate a fast restart. After process-level restart has been completed, drained in-flight messages are restored, and all the processes resume execution from their point of suspension. Furthermore, some pages saved in preceding checkpoints may be invalid (unmapped) in subsequent ones and need not be restored. The latest memory mapping information saved in *checkpoint file c* is also used for this purpose.

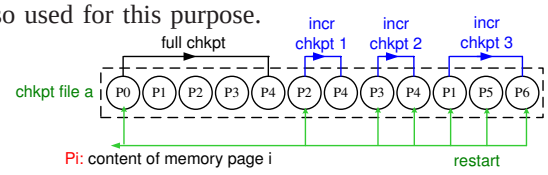


Fig. 5. Fast Restart from a Hybrid Checkpoint

3. Implementation Issues

Our hybrid full/incremental checkpoint/restart mechanism is implemented with LAM/MPI and BLCR. The overall design and implementation allows adaptation of this solution to arbitrary MPI implementations, such as MPICH and Open MPI. Next, we present the implementation details of the full/incremental C/R mechanism, including the MPI-level communication/coordination realized within LAM/MPI and the process-level fundamental capabilities of BLCR.

Hybrid Checkpointing at the Job Level

We developed new commands to issue full and incremental checkpoint commands, which are relayed by a decentralized scheduler to the *mpirun* process of the MPI job. Subsequently, *mpirun* broadcasts full/incremental checkpoint commands to each MPI tasks. At the LAM/MPI level, we also drain in-flight messages to reach a consistent internal state before checkpointing (see step 2 in Fig. 2). We restore in-flight message data and resume normal operation after the checkpointing has completed (see step 4 in Fig. 2).

Hybrid Checkpointing at the Process Level

We integrated several new BLCR features to extend its process-level checkpointing facilities to trigger full and incremental checkpoints at the process level within BLCR with corresponding writes of the respective portion of a process snapshot to one of the three files (see Section 2 and Fig. 4).

Fig. 3 depicts the steps involved in issuing an incremental checkpoint in reference to BLCR. Our focus is on the enhancements to BLCR (large dashed box). In the figure, time flows from top to bottom, and the processes and threads involved in the checkpoint are placed from right to left. Activities performed in the kernel are surrounded by dotted lines. A callback thread (right side) is spawned as the application registers a threaded callback and blocks in the kernel until a checkpoint has been committed. *mpirun* invokes a new command extension to BLCR, parametrized with the process id. This triggers an *ioctl* call, thereby resuming the callback thread that was previously blocked in the kernel. After the callback thread invokes the individual callback for each of the other threads, it reenters the kernel and sends a signal to each thread. These threads, in response, engage in executing the callback signal handler and then enter the kernel through another *ioctl* call.

Once in the kernel, the first thread saves the dirty memory pages modified since the last checkpoint. Then, threads take turns saving their register and signal information to the checkpoint files. After a final barrier, the process exits the kernel and enters user space, at which point the checkpoint mechanism has completed.

BLCR's full checkpoint command performs similar work, except that once the kernel is entered, the first thread saves all non-empty memory pages rather than only the dirty ones.

Restart from Hybrid Checkpoints

Newly developed commands perform the restart work (1) at the job level with LAM/MPI and (2) at the process level within BLCR. In concert, the two commands implement the restart from the three checkpoint files and resume the normal execution of the MPI job as discussed in Section 2.

4. Experimental Framework

Experiments were conducted on a dedicated Linux cluster comprised of 18 compute nodes, each equipped with two AMD Opteron-265 processors (each dual core) and 2 GB of memory. The nodes are interconnected by two networks, both with 1 Gbps Ethernet. The OS used is Fedora Core 5 Linux x86_64 with our dirty bit patch as described in Section 2. We extended LAM/MPI and BLCR with our hybrid full/incremental C/R mechanism of this platform.

For all following experiments we use the MPI version of the NPB suite [13] (version 3.3) as well as mpiBLAST [14]. NPB is a suite of programs widely used to evaluate the performance of parallel system, while the latter is a parallel implementation of NCBI BLAST, which splits a database into fragments and distributes the query tasks to workers by query segmentation before the BLAST search is performed in parallel.

5. Experimental Results

Experiments were conducted to assess (a) overheads, (b) file sizes, (c) restart overheads, and (d) the relationship between checkpoint interval and checkpoint overhead.

Out of the NPB suite, the BT, CG, FT, LU and SP benchmarks were exposed to class C data inputs running on 4, 8 or 9 and 16 nodes, and to class D data inputs on 8 or 9 and 16 nodes. Some NAS benchmarks have 2D, others have 3D layouts for 2^3 or 3^2 nodes, respectively. The NAS benchmark EP is exposed to class C, D and E data inputs running on 4, 8 and 16 nodes. All the other NAS benchmarks were not suitable for our experiments since they execute for too short a

period to be periodically checkpointed, such as IS, as depicted in Fig. 7(a), or they have excessive memory requirement, such as the benchmarks with class D data inputs on 4 nodes.

Since this version of mpiBLAST assigns one process as the master and another to perform file output, the number of actual worker processes performing parallel input is the total process number minus two. Each worker process reads several database fragments. With our experiments, we set the mpiBLAST-specific argument *-use-virtual-frags*, which enables caching of database fragments in memory (rather than local storage) for quicker searches.

Checkpointing Overhead

The first set of experiments assesses the overhead incurred due to one full or incremental checkpoint. Figs. 7(a), 8(a), 9(a) and 10(a) depict the base execution time of a job (benchmark) without checkpointing while Figs. 7(b), 8(b), 9(b) and 10(b) depict the checkpoint overhead. As these results show, the checkpoint overhead is uniformly small relative to the overall execution time, even for a larger number of nodes. Prior work [5] already compared the overhead of full checkpointing with the base execution, and the ratio is below 10% for most NPB benchmarks with Class C data inputs. Fig. 6 depicts the measured overhead for single full checkpointing relative to the base execution time of NPB with Class D data inputs and mpiBLAST (without checkpointing). The ratio is below 1%, except for MG, as discussed in the following.

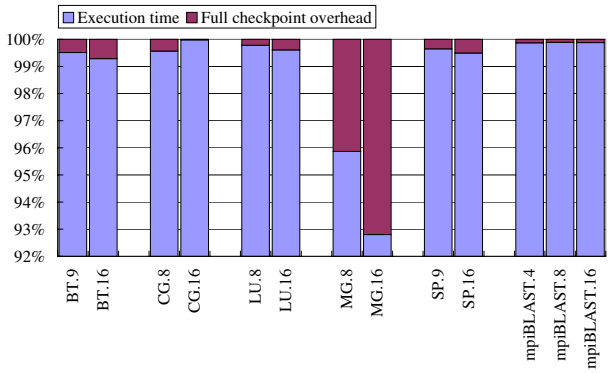


Fig. 6. Full Checkpt. Overhead: NPB Class D, mpiBLAST

MG has a larger checkpoint overhead (large checkpoint file), but the ratio is skewed due to a short overall execution time (see Fig. 8(a)). In practice, with more realistic and longer checkpoint intervals, a checkpoint would not be necessitated within the application's execution. Instead, the application would have been restarted from scratch. For longer runs with larger inputs of MG, the fraction of checkpoint/migration overhead would have been much smaller.

Figs. 7(b), 8(b), 9(b) and 10(b) show that the overhead of incremental checkpointing is smaller than that of full checkpointing, so the overhead of incremental checkpointing is less significant. Hence, a hybrid full/incremental checkpointing mechanism reduces runtime overhead compared to full checkpointing throughout, i.e., under varying number of nodes and input sizes.

Checkpoint Times and File Size

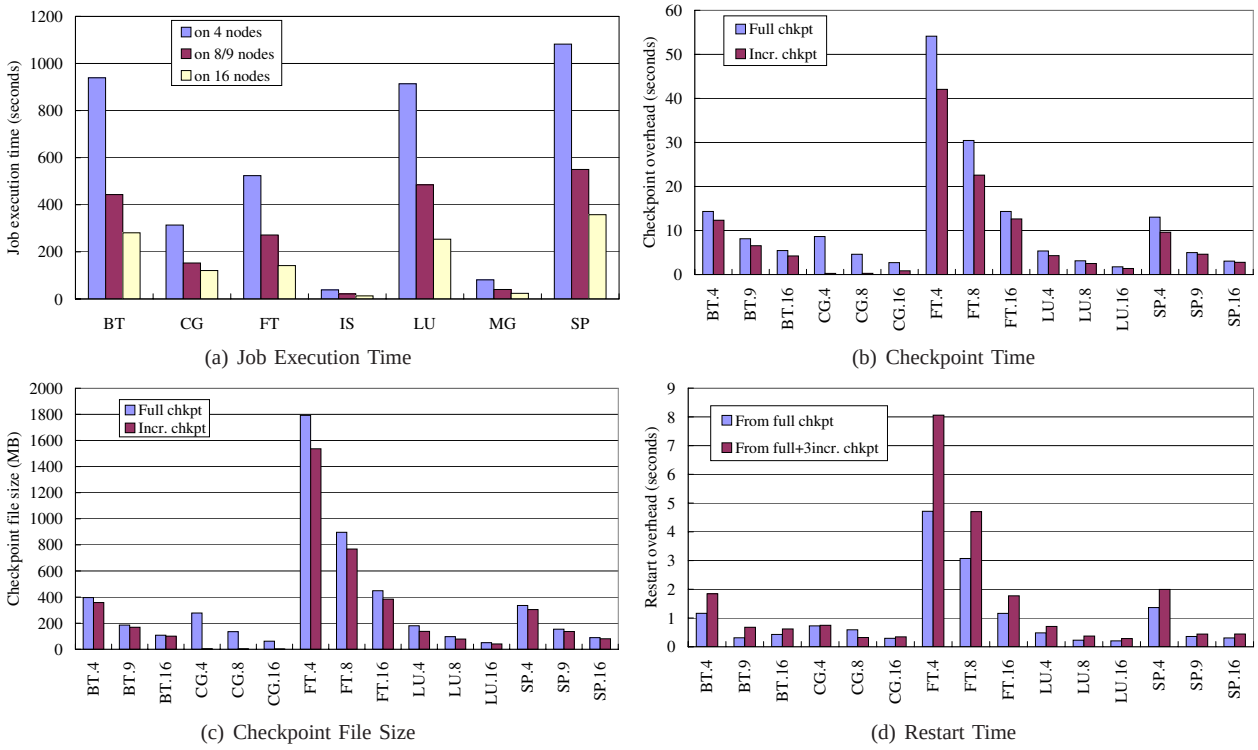
We also assessed the actual footprint of the checkpointing file. Figs. 7(c), 8(c), 9(c) and 10(c) depict the size of the checkpoint files for one process of each MPI application. Writing many files of such size to shared storage synchronously may be feasible for high-bandwidth parallel file systems. In the absence of sufficient bandwidth for simultaneous writes, we provide a multi-stage solution where we first checkpoint to local storage. After local checkpointing, files will be asynchronously copied to shared storage, an activity governed by the scheduler. This copy operation can be staggered (again governed by the scheduler) between nodes. Upon failure, a spare node restores data from the shared file system while the remaining nodes roll back using the checkpoint file on local storage, which results in less network traffic.

Overall, the experiments show that:
(1) the overhead of full/incremental checkpointing of the MPI job is largely proportional to the size of the checkpoint file;
(2) the overhead of full checkpointing is nearly the same at any time of the execution of the job;
(3) the overhead of incremental checkpointing is nearly the same at any interval; and
(4) the overhead of incremental checkpointing is lower than that of full checkpointing (except some cases of EP, which are lower than 0.45 seconds, which is excessively short. If required at this sort rate, one can employ full checkpointing only).

The first observation indicates that the ratio of communication overhead to computation overhead for C/R of the MPI job is relatively low. Since checkpoint files are, on average, large, the time spent on storing/restoring checkpoints to/from disk accounts for most of the measured overhead. This overhead is further reduced by the potential savings through incremental checkpointing.

For full/incremental checkpointing of EP (Fig. 9(c)), incremental checkpointing of CG with Class C data inputs (Fig. 7(c)) and incremental checkpointing of mpiBLAST (Fig. 10(c)), the footprint of the checkpoint file is small (smaller than 13MB), which results in a relatively small overhead. Thus, the checkpoint overhead mainly reflects the variance of the communication overhead inherent to the benchmark, which increases with the node count. However, the overall checkpoint overhead for these cases is smaller than 1 second. Hence, communication overhead of the applications did not significantly contribute to the overhead or interfere with checkpointing. This indicates a high potential of our hybrid full/incremental checkpointing solution to scale to larger clusters, and we have analyzed our data structures and algorithms to assure suitability for scalability. Due to a lack of large-scale experimentation platforms flexible enough to deploy our kernel modifications, new BLCR features and LAM/MPI enhancements, such larger scale experiments cannot currently be realized, neither at National Labs nor at larger-scale clusters within universities where we have access to resources.

The second observation about full checkpoint overheads



(a) Job Execution Time (b) Checkpoint Time

(c) Checkpoint File Size (d) Restart Time

Fig. 7. Evaluation with NPB Class C on 4, 8/9, and 16 Nodes

above indicated that the size of the full checkpoint file remains stable during job execution. The benchmarks do not allocate or free heap memory dynamically within timesteps of execution; instead, all allocation is performed during initialization, which is typical for many parallel codes.

The third observation is obtained by measuring the checkpoint file size with different checkpoint intervals for incremental checkpointing, i.e., with intervals of 30, 60, 90, 120, 150 and 180 seconds for NPB Class C and intervals of 2, 4, 6, 8, 10 and 12 minutes for NPB Class D and mpiBLAST.

Thus, we can assume the time spent on checkpointing is constant. This assumption is critical to determine the optimal full/incremental checkpoint frequency. The fourth observation verifies the superiority and justifies the deployment of our hybrid full/incremental checkpointing mechanism.

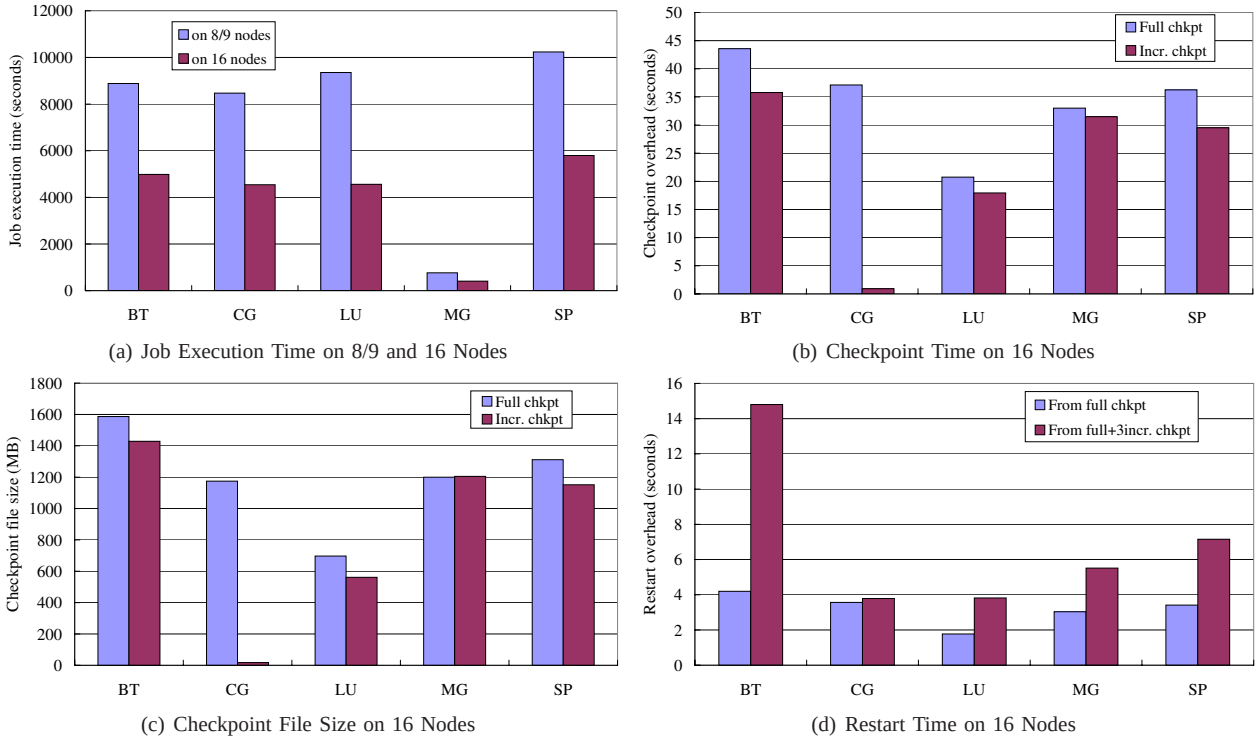
Restart Overhead

Figs. 7(d), 8(d) and 10(d) compare the restart overhead of our hybrid full/incremental solution from one full checkpoint plus three incremental checkpoints with that of the original solution restarting from one full checkpoint. The results indicate that the wall clock time for restart from full plus three incremental checkpoints exceeds that of restart from one full checkpoint by 0-253% depending on the application, and it is 68% larger (1.17seconds) on average for all cases. The largest additional cost of 253% (10.6 seconds) was observed for BT under class D inputs for 16 nodes due to its comparatively large memory footprint of the incremental checkpointing. Yet, this overhead is not on the critical path as failures occur significantly less frequently than periodic checkpoints, *i.e.*, our hybrid approach reduces the cost along the critical path of checkpointing. For mpiBLAST and CG, the footprint of incremental checkpointing is comparatively so small that the overhead of restarting from full plus three incremental checkpoints is almost the same as that of restarting from one full checkpoint. Yet, the time saved by three incremental checkpoints over three full checkpoints is 16.64 seconds on average for all cases. Even for BT under class D inputs for 16 nodes (which has the largest restart cost loss ratio), the saving is 23.38 seconds while the loss is 10.6 seconds. We can further extend the benefit by increasing the incremental checkpointing count between two full checkpoints.

We can also assess the accumulated checkpoint file size of one full checkpoint plus three incremental checkpoints, which is 185% larger than that of one full checkpoint. However, as just discussed, the overhead of restarting from one full plus three incremental checkpoint is only 68% larger. This is due to the following facts:
(1) a page saved by different checkpoints is only restored once;
(2) file reading for restarting is much faster than file writing for checkpointing; and
(3) some pages saved in preceding checkpoints may be invalid and need not be restored at a later checkpoint.

Benefits of the Hybrid C/R Mechanism

Fig. 11 depicts sensitivity results of the overall savings (the cost saved by replacing full checkpoints with incremental ones minus the loss on the restore overhead) for different number of incremental checkpoints between any adjacent full ones. Savings increase proportional to the number of incremental



(a) Job Execution Time on 8/9 and 16 Nodes

(b) Checkpoint Time on 16 Nodes

(c) Checkpoint File Size on 16 Nodes

(d) Restart Time on 16 Nodes

Fig. 8. Evaluation with NPB Class D

checkpoints (as the y axis in the figure is on a logarithmic base), but the amount of incremental checkpoints is still limited by stable storage capacity (without segment-style cleanup). The results are calculated by using the following formulae:

$S_n = n \times (O_f - O_i) - (R_{f+i} - R_f)$

where S_n is the saving with n incremental checkpoints between two full checkpoints, O_f is the full checkpoint overhead, O_i is the incremental checkpoint overhead, R_{f+i} is the overhead of restarting from full+i incremental checkpoints and R_f is the overhead of restarting from one full checkpoint. For mpiBLAST and CG, we may even perform only incremental checkpointing after the first full checkpoint is captured initially since the footprint of incremental checkpoints is so small that we will not run out of drive space at all (or, at least, for a very long time). Not only should a node failure be the exception over the set of all nodes, but the lower overhead of a single incremental checkpoint provides opportunities to increase checkpoint frequencies compared to an application running with full checkpoints only. Such shorter incremental checkpoint frequencies reduce the amount of work lost when a restart is necessitated by a node failure. Hence, the hybrid full/incremental checkpointing mechanism effectively reduces the overall overhead relative to C/R.

Table 2 presents detailed measurements on the savings of incremental checkpointing, the overhead of restart from full plus incremental checkpoints, the relationship between the checkpoint file size and restart overhead, and the overall benefit from the hybrid full/incremental C/R mechanism. The benchmarks are sorted by the benefit. The table shows that (1) the cost caused by restart from one full plus one incremental checkpoints (which is R_{f+1} - R_f) is low, compared to the

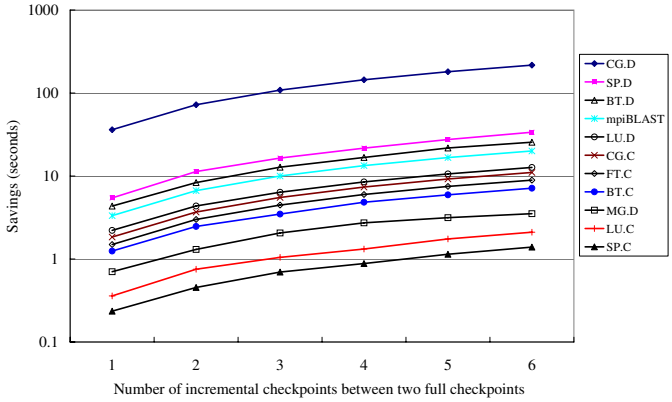
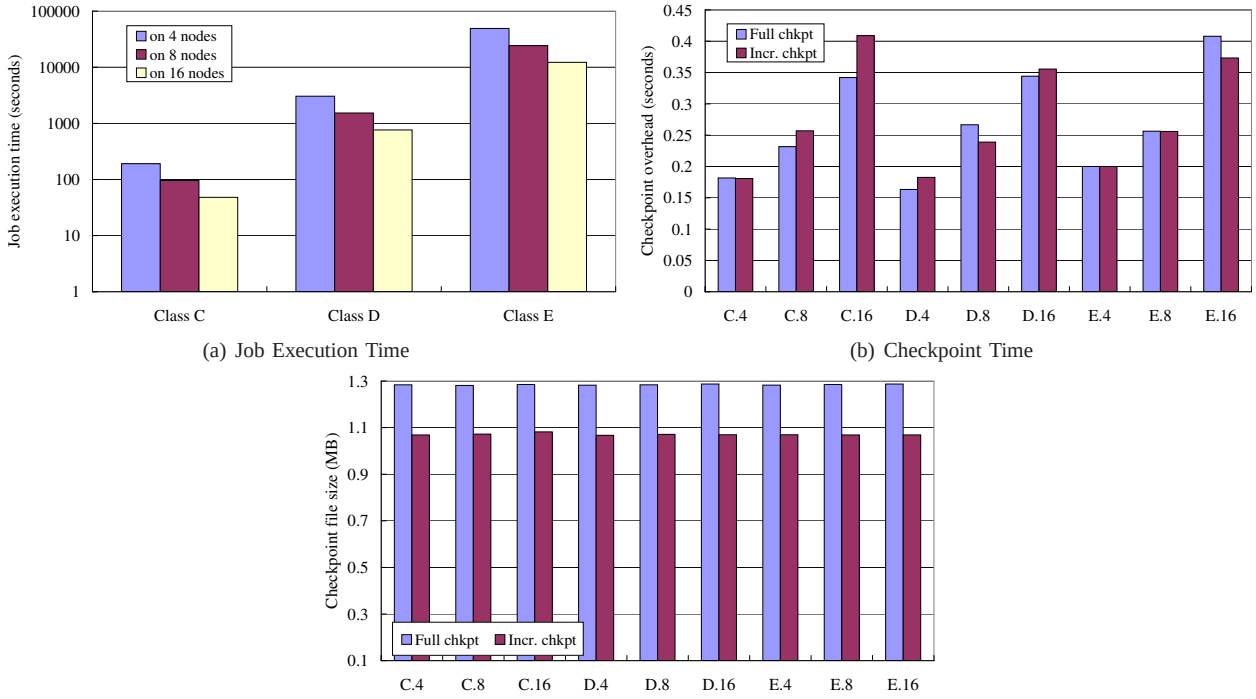


Fig. 11. Savings of Hybrid Full/Incremental C/R Mechanism for NPB and mpiBlast on 16 Nodes

savings by replacing full checkpoints with incremental ones (which is O_f - O_i), and can be ignored for most of the benchmarks; (2) the restart cost is nearly proportional to the file size (except that some pages are checkpointed twice at both full and incremental checkpoints but later only restored once and thus lead to no extra cost); (3) for all the benchmarks, we can benefit from the hybrid full/incremental C/R mechanism, and the performance improvement depends on the memory access characteristics of the application.

Naksinehaboon *et al.* provide a model that aims at reducing full checkpoint overhead by performing a set of incremental checkpoints between two consecutive full checkpoints [15]. They further develop a method to determine the optimal number of incremental checkpoints between full checkpoints. They obtain

$$m = \left\lceil \frac{(1-\mu) \times O_f}{P_i \times \delta} - 1 \right\rceil$$



(a) Job Execution Time

(b) Checkpoint Time

(c) Checkpoint File Size

Fig. 9. Evaluation with NPB EP Class C/D/E on 4, 8 and 16 nodes

Benchmarks	CG.D	SP.D	BT.D	mpiBLAST	LU.D	CG.C	FT.C	BT.C	MG.D	LU.C	SP.C
Savings (O_f - O_i)	36.20	6.73	7.79	3.34	2.81	1.85	1.69	1.22	1.51	0.38	0.28
Restart overhead ($\delta = R_{f+1}$ - R_f)	0.03	1.28	3.45	0.01	0.59	0.01	0.20	-0.02	0.81	0.02	0.04
File increases for 1 incr. chkpt [MB]	17.26	1151.88	1429.14	10.45	561.46	2.10	384.41	100.67	1205.23	41.09	80.98
Benefit of hybrid C/R (S_1)	36.17	5.45	4.34	3.33	2.21	1.84	1.50	1.25	0.70	0.36	0.24

TABLE 2. Savings by Incremental Checkpoint vs. Overhead on Restart

where m is the number of incremental checkpoints between two consecutive full checkpoint, μ is the incremental checkpoint overhead ratio ($\mu = O_i/O_f$), P_i is the probability that a failure will occur after the second full checkpoint and before the next incremental checkpoint, and δ is additional recovery cost per incremental checkpoint. With the data from Table 2, we obtain averages of $O_f - O_i = 5.8$ and $\delta = 0.58$, which, after transformation, gives us

$$m = \left\lceil \frac{O_f - O_i}{P_i \times \delta} - 1 \right\rceil = \left\lceil \frac{5.8}{P_i \times 0.58} - 1 \right\rceil$$

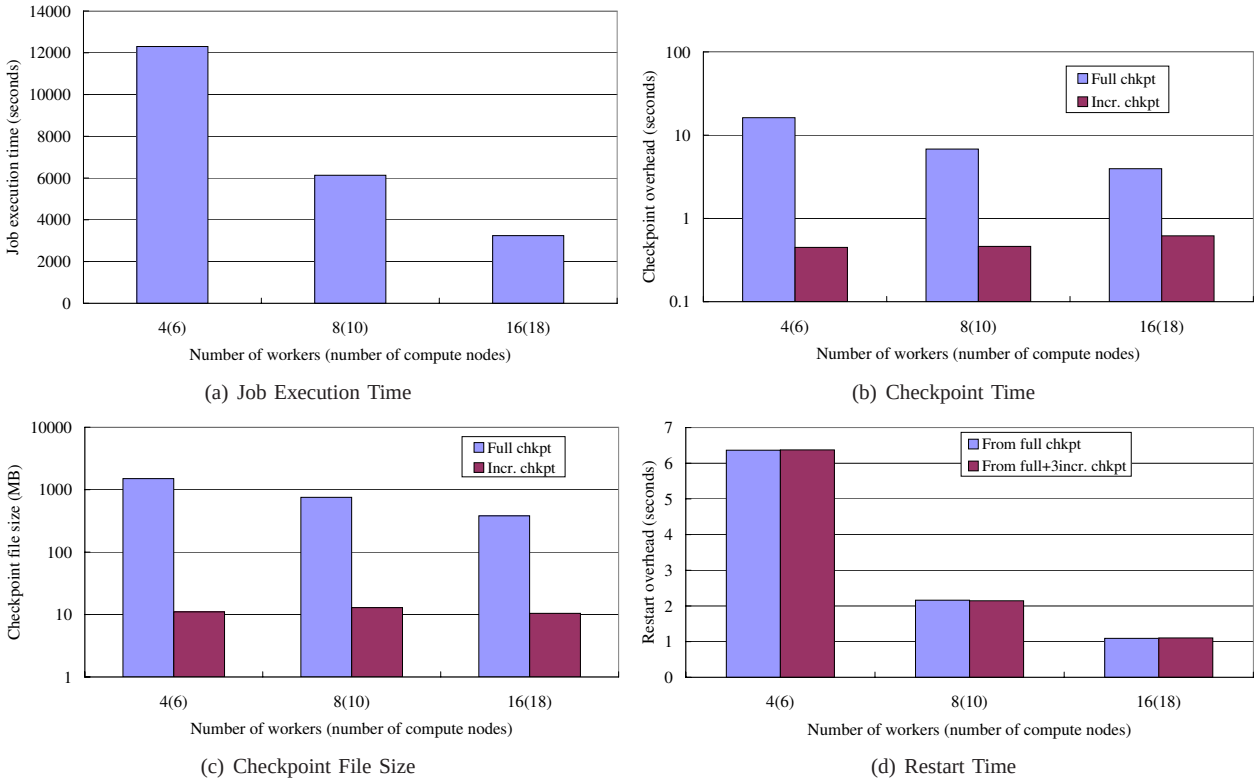
Since $0 < P_i < 1$, a lower bound for m is 9, which indicates the potential for even higher savings at nine incremental checkpoints between any full checkpoints for an optimal balance.

Overall, the overhead of the hybrid C/R mechanism is significantly lower than the original periodical full C/R mechanism, and the resulting checkpointing frequency can be increased to reduce the loss of computation should a node fail.

6. Related Work

Checkpoint/Restart: C/R techniques for MPI jobs frequently deployed in HPC environments can be divided into two categories: coordinated (LAM/MPI+BLCR [9], [8], CoCheck [16], etc.) and uncoordinated (MPICH-V [17], [18]). Coordinated techniques commonly rely on a combination of OS support to checkpoint a process image (*e.g.*, via the BLCR Linux module [8]) or user-level runtime library support. Collective communication among MPI tasks is used for the coordinated checkpoint negotiation [9]. Uncoordinated C/R techniques generally rely on logging messages and possibly their temporal ordering for asynchronous non-coordinated checkpointing, *e.g.*, MPICH-V [17], [18] that uses pessimistic message logging. The framework of Open MPI [19], [10] is designed to allow both coordinated and uncoordinated types of protocols. However, conventional C/R techniques checkpoint the entire process image leading to high checkpoint overhead, heavy I/O bandwidth requirements and considerable hard drive pressure, even though only a subset of the process image of all MPI tasks changes between checkpoints. With our hybrid full/incremental C/R mechanism, we mitigate the situation by checkpointing only the modified pages and at a lower rate than required for full checkpoints.

Incremental Checkpointing: Recent studies focus on incremental checkpointing [20], [21], [22]. TICK (Transparent



(a) Job Execution Time

(b) Checkpoint Time

(c) Checkpoint File Size

(d) Restart Time

Fig. 10. Evaluation with mpiBLAST

Incremental Checkpointer at Kernel Level) [20] is a system-level checkpointer implemented as a kernel thread. It supports incremental and full checkpoints. However, it does not checkpoint dynamically loaded shared libraries. Dejavu [6] integrates TICK within MVAPICH over sockets and Infiniband for incremental checkpoints. In contrast, our solution transparently supports *hybrid* full/incremental checkpoints for an MPI job, which is unprecedented. *Pickpt* [21] is a page-level incremental checkpointing facility. It provides space-efficient techniques for automatically removing useless checkpoints aiming to minimizing the use of disk space that differ from our garbage collection thread technique. Yi *et al.* [23] develop an adaptive page-level incremental checkpointing facility based on the dirty page count as a threshold heuristic to determine whether to checkpoint now or later, a feature complementary to our work that we could adopt within our scheduler component. However, Pickpt and Yi's adaptive scheme are constrained to C/R of a single process while we cover an entire MPI job with all its processes and threads within processes. Agarwal *et al.* [24] provide a different adaptive incremental checkpointing mechanism to reduce the checkpoint file size by using a secure hash function to uniquely identify changed blocks in memory. Their solution not only appears to be specific to IBM's compute node kernel on BG/L, it also requires hashes for each memory page to be computed, which tends to be more costly than OS-level dirty-bit support as caches are thrashed when each memory location of a page has to be read in their approach.

A prerequisite of incremental checkpointing is the availability of a mechanism to track modified pages during each checkpoint. Two fundamentally different approaches may be employed, namely page protection mechanisms or page-table dirty bits. Different implementation variants build on these schemes. One is the bookkeeping and saving scheme that, based on the dirty bit scheme, copies pages into a buffer [20]. Another solution is to exploit page write protection, such as in *Pickpt* [21] and XtreemOS for Grids [25], to save only modified pages as a new checkpoint. The page protection scheme has certain draw-backs. Some address ranges, such as the stack, can only be write protected if an alternate signal stack is employed, which adds calling overhead and increases cache pressure. Furthermore, the overhead of user-level exception handlers is much higher than kernel-level dirty-bit shadowing. Thus, we selected the dirty bit scheme in our design, yet in our own implementation within the Linux kernel. Our approach is unique among this prior work in its ability to capture and restore hybrid checkpoints of an *entire MPI job* with all its tasks, including all relevant process information and OS kernel-specific data. Hence, our scheme is more general than language specific solutions (as in Charm++), yet lighter weight than OS virtualization C/R techniques.

Checkpoint Interval Model: Aiming at optimality for checkpoint frequency, overhead and rollback time over a set of MPI jobs, several models have been developed to determine job-specific intervals for full or incremental checkpoints. Young [26] presented a checkpoint model and obtained a fixed optimal checkpoint interval. Based on Young's work, Daly [27] improved the model to an optimal checkpoint placement from a first order to a higher order approximation. Liu *et al.* provide a model for an optimal full C/R strategy toward minimizing rollback and checkpoint overheads [28]. Their scheme focuses on the fault tolerance challenge, es-

pecially in a large-scale HPC system, by providing optimal checkpoint placement techniques that are derived from the actual system reliability. Naksinehaboon *et al.* (see Section 5) provide a model to perform a set of incremental checkpoints between two consecutive full checkpoints [15] and a method to determine the optimal number of incremental checkpoints between full checkpoints. While their work is constrained to simulations based on log data, our work focuses on the design and implementation of process-level incremental C/R for MPI tasks. Their work is complementary in that their model could be utilized to fine-tune our incremental C/R rate. In fact, the majority of their results on analyzing failure data logs show that the full/incremental C/R model outperforms full checkpointing. Furthermore, our reverse scanning restart mechanism is superior to the one used in their model.

7. Conclusion

This work contributes a novel hybrid C/R mechanism with a concrete implementation within LAM/MPI and BLCR with the following features: (1) It provides a dirty bit mechanism to track modified pages between checkpoints; (2) only the subset of *modified* pages is appended to the checkpoint file together with page metadata updates for incremental checkpoints; (3) incremental checkpoints complement full checkpoints by reducing I/O bandwidth and storage space requirements while allowing lower rates for full checkpoints; (4) a restart after a node failure requires a scan over all incremental checkpoints and the last full checkpoint to recover from the last stored version of a page, *i.e.*, the content of any page only needs to be written to memory once for fast restart; (5) a decentralized scheduler coordinates the hybrid C/R mechanism among the MPI tasks. Results indicate that the performance of the hybrid C/R mechanism is significantly better than that of the original full C/R. For the NPB suite and mpiBLAST, the average savings due to replacing three full checkpoints with three incremental checkpoints is 16.64 seconds — at the cost of only 1.17 seconds if a restart is required after a node failure due to restoring one full plus three incremental checkpoints. Hence, the overall saving amounts to 15.47 seconds. Overall, our *hybrid* checkpointing approach is not only novel but also superior to prior non-hybrid techniques as an optimal balance is reached around a ratio of 1:9 between full/incremental checkpoints. These results illustrate that the resulting checkpointing frequency can be increased to reduce the potential loss of computational work should a node fail.

References

[1] C.-H. Hsu and W.-C. Feng, "A power-aware run-time system for high-performance computing," in *Supercomputing*, 2005.

[2] O. R. N. Laboratory, "National center for computational sciences," http://info.nccs.gov/resources/jaguar, Jun. 2007.

[3] I. Philp, "Software failures and the road to a petaflop machine," in *Workshop on High Performance Computing Reliability Issues*, 2005.

[4] J. T. Daly, L. A. Pritchett-Sheats, and S. E. Michalak, "Application MTTFE vs. platform MTTF: A fresh perspective on system reliability and application throughput for computations at scale," in *Workshop on Resiliency in High Performance Computing*, May 2008, pp. 19–22.

[5] C. Wang, F. Mueller, C. Engelmann, and S. Scott, "A job pause service under LAM/MPI+BLCR for transparent fault tolerance," in *International Parallel and Distributed Processing Symposium*, Apr. 2007.

[6] J. Ruscio, M. Heffner, and S. Varadarajan, "Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems," in *International Parallel and Distributed Processing Symposium*, 2007.

[7] J. M. Squyres and A. Lumsdaine, "A Component Architecture for LAM/MPI," in *European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, no. 2840, Sep. 2003, pp. 379–387.

[8] J. Duell, "The design and implementation of berkeley lab's linux checkpoint/restart," Lawrence Berkeley National Laboratory, TR, 2000.

[9] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," in *LACSI Symposium*, Oct. 2003.

[10] 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 *Workshop on Dependable Parallel, Distributed and Network-Centric Systems*, 03 2007.

[11] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," in *ACM Trans. on Computer Systems, Vol. 10, No. 1*, Feb. 1992.

[12] C. Wang, F. Mueller, C. Engelmann, and S. Scott, "Proactive process-level live migration in hpc environments," in *Supercomputing*, 2008.

[13] F. Wong, R. Martin, R. Arpaci-Dusseau, and D. Culler, "Architectural requirements and scalability of the NAS parallel benchmarks," in *Supercomputing*, 1999.

[14] A. Darling, L. Carey, and W. Feng, "The design, implementation, and evaluation of mpiBLAST," in *ClusterWorld Conference and Expo*, 2003.

[15] N. Naksinehaboon, Y. Liu, C. B. Leangsuksun, R. Nassar, M. Paun, and S. Scott, "Reliability-aware approach: An incremental checkpoint/restart model in hpc environments," in *Symposium on Cluster Computing and the Grid*, 2008, pp. 783–788.

[16] G. Stellner, "CoCheck: checkpointing and process migration for MPI," in *Proceedings of IPPS '96. The 10th International Parallel Processing Symposium: Honolulu, HI, USA, 15–19 April 1996*, IEEE, Ed. 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: IEEE Computer Society Press, 1996, pp. 526–531.

[17] G. Bosilca, A. Boutellier, and F. Cappello, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," in *Supercomputing*, Nov. 2002.

[18] B. Bouteiller, F. Cappello, T. Herault, K. Krawezik, P. Lemarinier, and M. Magniette, "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," in *Supercomputing*, 2003.

[19] B. Barrett, J. M. Squyres, A. Lumsdaine, R. L. Graham, and G. Bosilca, "Analysis of the component architecture overhead in Open MPI," in *European PVM/MPI Users' Group Meeting*, September 2005.

[20] R. Gioiosa, J. C. S., S. Jiang, and F. Petrini, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," in *Supercomputing*, 2005.

[21] J. Heo, S. Yi, Y. Cho, J. Hong, and S. Y. Shin, "Space-efficient page-level incremental checkpointing," in *ACM Symposium on Applied computing*, 2005, pp. 1558–1562.

[22] S.-T. Hsu and R.-C. Chang, "Continuous checkpointing: joining the checkpointing with virtual memory paging," *Softw. Pract. Exper.*, vol. 27, no. 9, pp. 1103–1120, 1997.

[23] S. Yi, J. Heo, Y. Cho, and J. Hong, "Adaptive page-level incremental checkpointing based on expected recovery time," in *ACM Symposium on Applied computing*, 2006, pp. 1472–1476.

[24] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *International Conference on Supercomputing*. New York, NY, USA: ACM, 2004, pp. 277–286.

[25] J. Mehnert-Spahn, E. Feller, and M. Schoettner, "Incremental checkpointing for grids," in *Linux Symposium*, Jul. 2009.

[26] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[27] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 303–312, 2006.

[28] Y. Liu, R. Nassar, C. B. Leangsuksun, N. Naksinehaboon, M. Paun, and S. Scott, "An optimal checkpoint/restart model for a large scale high performance computing system," in *International Parallel and Distributed Processing Symposium*, Apr. 2008.