Hybrid Full/Incremental Checkpoint/Restart 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

[2] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN

mueller@cs.ncsu.edu, phone: +1.919.515.7889, fax: +1.919.515.7896

Abstract

As the number of cores in high-performance computing environments 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 high-performance hybrid disk-based full/incremental checkpointing technique for MPI tasks to capture only data changed since the last checkpoint. 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 significantly outweigh the loss on restart operations.

Experiments in a cluster with the NAS Parallel Benchmark suite and mpiBLAST indicate that savings due to replacing full checkpoints with incremental ones average 16.64 seconds while restore overhead amounts to just 1.17 seconds. These savings increase with the frequency of incremental checkpoints. Overall, our novel hybrid full/incremental checkpointing is superior to prior non-hybrid techniques.

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 [17]. 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 [17, 21]. 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 [26].

In such systems, frequently deployed checkpoint/restart (C/R) mechanisms periodically checkpoint the entire process

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@ORNL	23,416	37.5 hrs	Storage, memory

Table 1: Reliability of HPC Clusters

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 [26]. 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: This paper contributes the **first incremental checkpointing mechanism for MPI tasks that is transparently integrated into an MPI environment**. In contrast, prior solutions only operated in *single process* environments [13, 16, 18, 38] or used hash-based blocks and required a *new API* to indicate when to checkpoint and drain in-flight messages [1] instead of our hardware assisted fully transparent scheme. Our incremental checkpoints are complementary to full checkpoints to capture only data changed since the last checkpoint. The implementation, while realized over LAM (Local Area Multicomputer)/MPI's C/R support [31] through Berkeley Labs C/R (BLCR) [11], is in its mechanisms applicable to any process-migration solution, *e.g.*, the OpenMPI FT mechanisms [19, 20]. BLCR is an open source, system-level C/R implementation integrated with LAM/MPI via a callback function. The original LAM/MPI+BLCR combination [30] only provides full C/R mechanisms.

This paper contributes a hybrid full/incremental C/R solution to significantly reduce the size of the checkpoint file and the overhead of the checkpoint operations. Besides the original nodes allocated to an MPI job, it assumes the availability of spare nodes where processes (MPI tasks) may be relocated after a node failure. The paper further reduces the overhead of the restart operation due to roll-back after node failures, *i.e.*, restoration from full/incremental checkpoints on spare nodes, which only moderately increases the restart cost relative to a restore from a single, full checkpoint. After accounting for cost and savings, **savings due to incremental checkpoints significantly outweigh the loss on restart operations for over novel hybrid approach.**

We conducted a set of experiments on an 18-node dual-processor (each dual core) Opteron cluster. We assessed the viability of our approach using the NAS (NASA Advanced Supercomputing) Parallel Benchmark suite and mpiBLAST. Experimental results show that the overhead of our hybrid full/incremental C/R mechanism is significantly lower than that of the original C/R mechanism relying on full checkpoints. More specifically, experimental results indicate that the cost saved by replacing three full checkpoints with three incremental checkpoints is 16.64 seconds while the restore overhead amounts

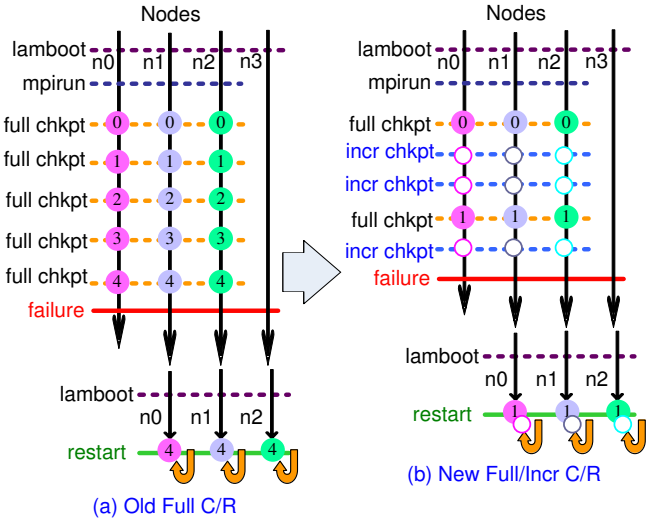


Fig. 1: Hybrid Full/Incremental C/R Mechanism vs. Full C/R

to just 1.17 for an overall savings of 15.47 seconds on average for the NAS Parallel Benchmark suite and mpiBLAST. The potential of savings due to our hybrid incremental/full C/R technique should even be higher in practice as (1) much higher ratios than just 1/3 for full/incremental checkpoints may be employed and (2) the amount of lost work would be further reduced if more frequent, lighter weight checkpoints were employed. Moreover, our approach can be easily integrated with other techniques, such as job-pause and migration mechanisms [34, 35] to avoid requeuing overhead by letting the scheduled job tolerate faults so that it can continue executing with spare nodes.

The paper is structured as follows. Section 2 presents the design of our hybrid full/incremental 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.

2 Design

This section presents an overview of the design of the hybrid full/incremental C/R mechanism with LAM/MPI and BLCR. 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 Figure 1(b). Such process-based incremental checkpointing reduces checkpoint bandwidth and storage space requirements, and it leads to a lower rate of full checkpoints.

In the following, we first discuss the schedule of the full/incremental C/R. We then discuss system support for incremental 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 detailed. Second, dirty pages and related meta-data image information are saved at the process/MPI task level, as depicted in Figure 3. We employ filtering of “dirty”

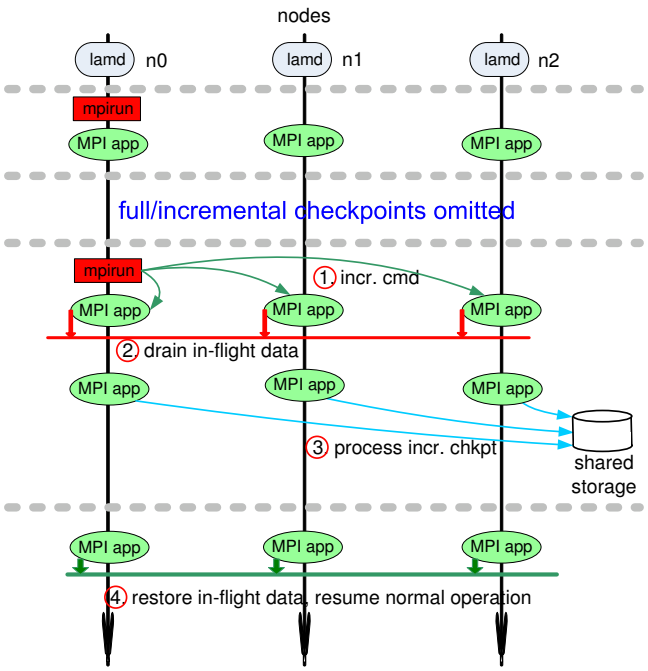


Fig. 2: Incremental Checkpoint at LAM/MPI

pages at the memory management level, *i.e.*, memory pages modified (written to) since the last checkpoint, which are utilized at node failures to restart from the composition of full and incremental checkpoints. Each component of our hybrid C/R mechanism is detailed next.

2.1 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.

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 [34]. 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 2.5.

2.2 Incremental Checkpointing at the Job Level

Incremental Checkpointing at the Job Level is performed in a sequence of steps depicted in Figure 2 and described in the following.

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 Incremental Checkpoint: Once all the MPI tasks (processes) reach a globally consistent state, all the MPI tasks perform the process-level incremental checkpoint operations independently, as discussed in Section 2.3.

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.

2.3 Incremental Checkpointing at the Process Level

Incremental checkpointing of MPI tasks (step 3 in Figure 2) is performed at the process level, which is shown in detail in Figure 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 Figure 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 3.2.

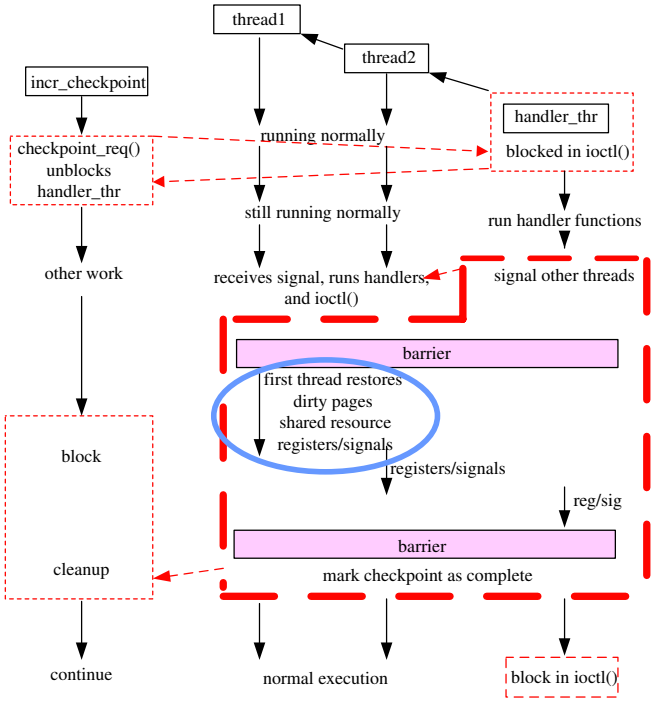


Fig. 3: BLCR with Incremental Checkpoint in Bold Frame

A set of three files serve as storage abstraction for a checkpoint snapshot, as depicted in Figure 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.

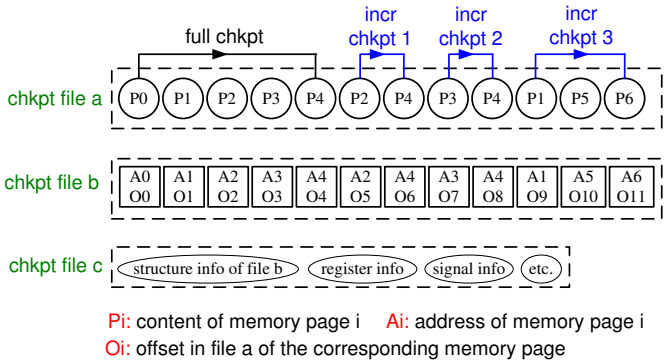


Fig. 4: Structure of Checkpoint Files

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 Figure 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 [28], 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*.

2.4 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 [35]. 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, which incurs negligible overhead (see [35] for details).

2.5 MPI Job Restart from Full+Incremental 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 sequence up to the pages from the last full checkpoint image, as depicted in Figure 5. The scan over all incremental checkpoints and the last full checkpoint allows the recovery of the last stored version of a page, *i.e.*, the content of any page only needs to be written once for 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.

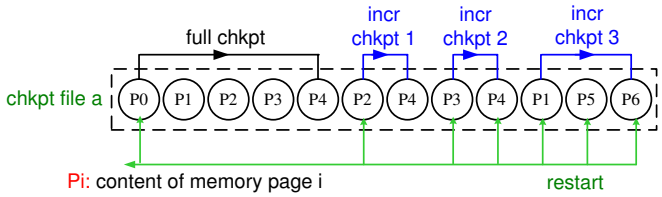


Fig. 5: Fast Restart from Full/Incremental Checkpoints

3 Implementation Issues

Our hybrid full/incremental checkpoint/restart mechanism is currently implemented with LAM/MPI and BLCR. The overall design and implementation allows adaptation of this solution to arbitrary MPI implementations, such as MPICH and OpenMPI. 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.

3.1 Full/Incremental Checkpointing at the Job Level

We developed new commands *lam_full_checkpoint* and *lam_incr_checkpoint* to issue full and incremental checkpoint commands, respectively. The decentralized scheduler relays these commands 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 the in-flight data and reach to a consistent internal state before processes launch the actual checkpoint operation (see step 2 in

Figure 2). We then restore the in-flight data and resume normal operation after the checkpoint operation of the process has completed (see step 4 in Figure 2).

3.2 Full/Incremental Checkpointing at the Process Level

We integrated several new BLCR features to extend its process-level checkpointing facilities, including the new commands *cr_full_checkpoint* and *cr_incr_checkpoint* to trigger full and incremental checkpoints at the process level within BLCR. Both of these commands write their respective portion of the process snapshot to one of the three files (see Section 2 and Figure 4).

Figure 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. When *mpirun* invokes the newly developed *cr_incr_checkpoint* command extensions to BLCR, it provides the process id as an argument. In response, the *cr_incr_checkpoint* mechanism issues 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.

The command *cr_full_checkpoint* performs similar work, except that once the kernel is entered, the first thread saves all the non-empty memory pages rather than only the dirty ones.

3.3 Restart from Full+Incremental Checkpoints at Job and Process Levels

A novel command, *lam_fullplusincr_restart*, has been developed to perform the restart work at the job level with LAM/MPI. Yet another command, *cr_fullplusincr_restart*, has been devised to support the restart work 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 [37] (version 3.3) as well as mpiBLAST [1]. 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 associated with the full and incremental checkpoints, (b) full and incremental checkpoint file size and memory checkpointed (which is the main source of the checkpointing overhead), (c) restart overheads associated with the full and incremental checkpoints, 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 Figure 7(a), or they have excessive memory requirement, such as the benchmarks with class D data inputs on 4 nodes.

Since the version of mpiBLAST we used 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.

5.1 Checkpointing Overhead

The first set of experiments assesses the overhead incurred due to one full or incremental checkpoint. Figures 7(a), 8(a), 9(a) and 10(a) depict the base execution time of a job (benchmark) without checkpointing while Figures 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 [34] 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. Figure 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.

MG has a larger checkpoint overhead (large checkpoint file), but the ratio is skewed due to a short overall execution time (see Figure 8(a)). In practice, with more realistic and longer checkpoint intervals, a checkpoint would not be necessitated

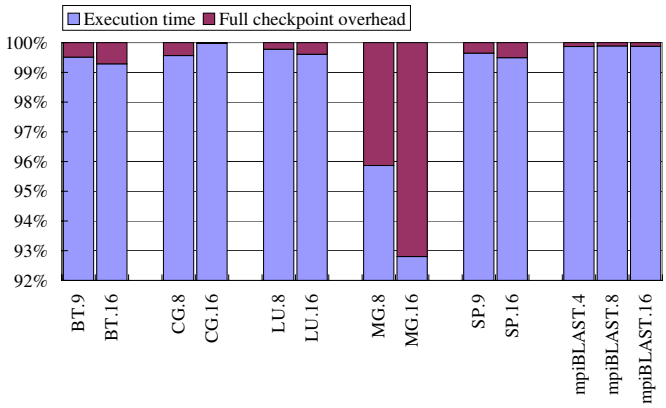
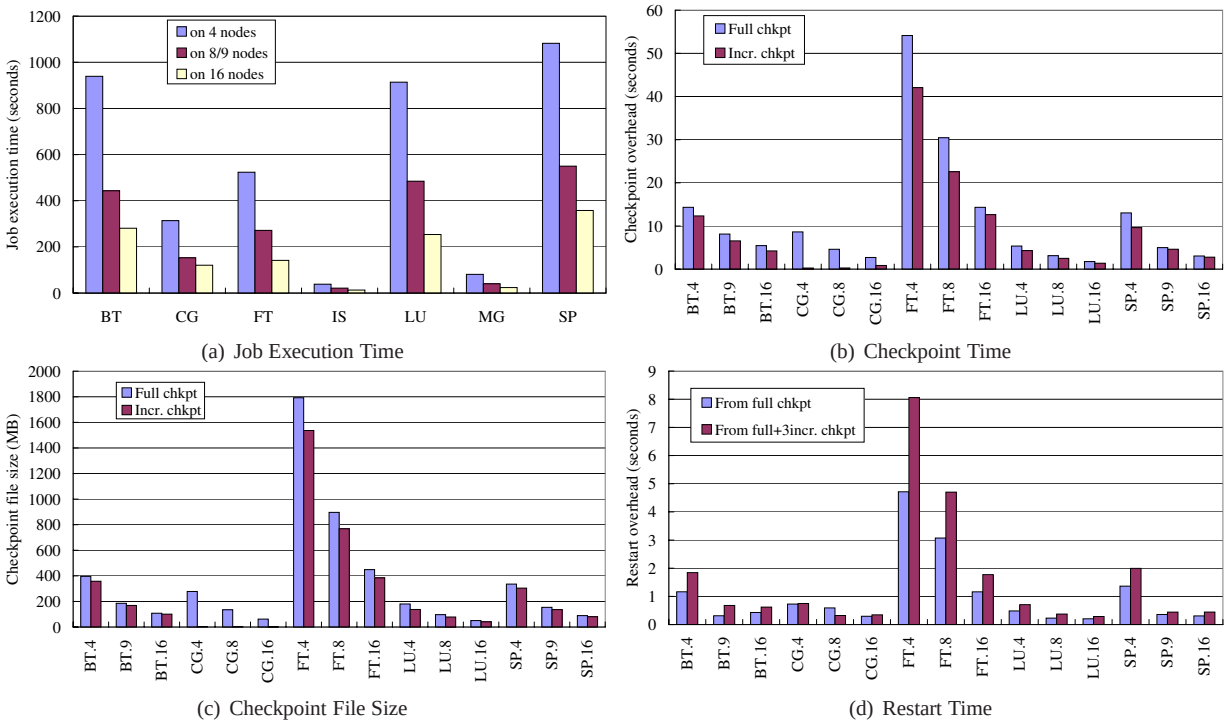


Fig. 6: Full Checkpoint Overhead of NPB Class D and mpiBLAST

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.



(a) Job Execution Time

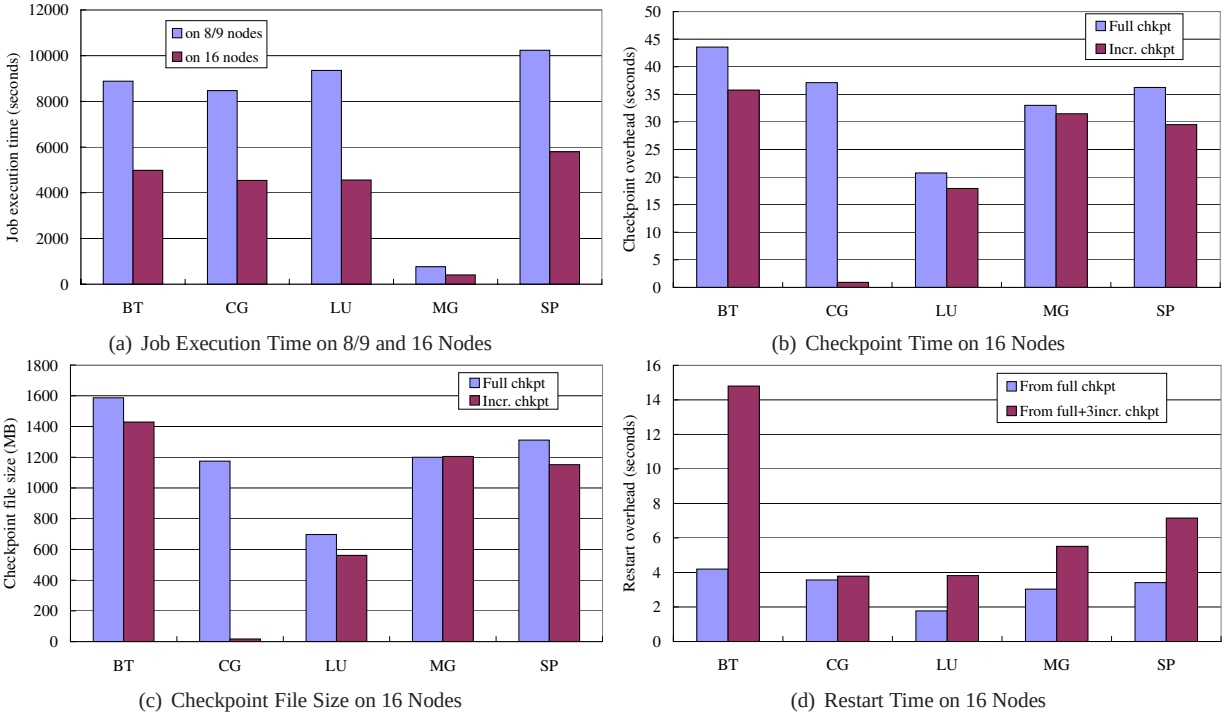
(b) Checkpoint Time

(c) Checkpoint File Size

(d) Restart Time

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

Figures 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.



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

(b) Checkpoint Time on 16 Nodes

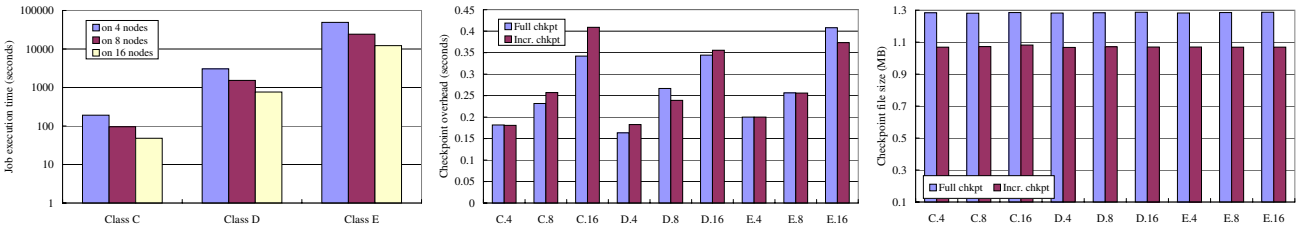
(c) Checkpoint File Size on 16 Nodes

(d) Restart Time on 16 Nodes

Fig. 8: Evaluation with NPB Class D

5.2 Checkpointing File Size

Besides overhead due to checkpointing, we assessed the actual footprint of the checkpointing file. Figures 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.



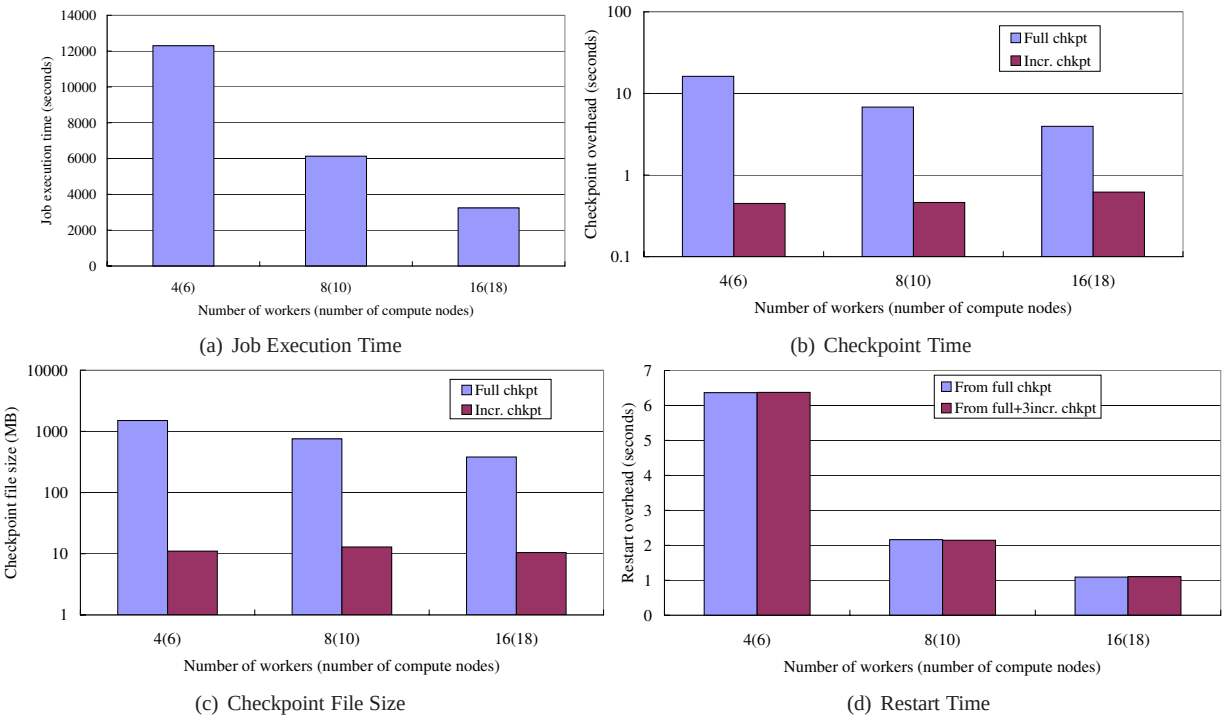
(a) Job Execution Time

(b) Checkpoint Time

(c) Checkpoint File Size

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

Overall, the experiments show that:



(a) Job Execution Time

(b) Checkpoint Time

(c) Checkpoint File Size

(d) Restart Time

Fig. 10: Evaluation with mpiBLAST

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 (Figure 9(b)), incremental checkpointing of CG with Class C data inputs (Figure 7(b)) and incremental checkpointing of mpiBLAST (Figure 10(b)), 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 above indicated that the size of the full checkpoint file remains stable during job execution. The benchmarks codes do not allocate or free heap memory dynamically within timesteps of execution; instead, all allocation is performed during initialization, which is typical for most parallel codes (except for adaptive codes [36]).

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.

5.3 Restart Overhead

Figures 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.

5.4 Benefit of Hybrid Full/Incremental C/R Mechanism

Figure 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 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:

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

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+n_i} is the overhead of restarting from full+n 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_i} - R_f) is low, compared to the 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 [24]. They further develop a method to determine the optimal number

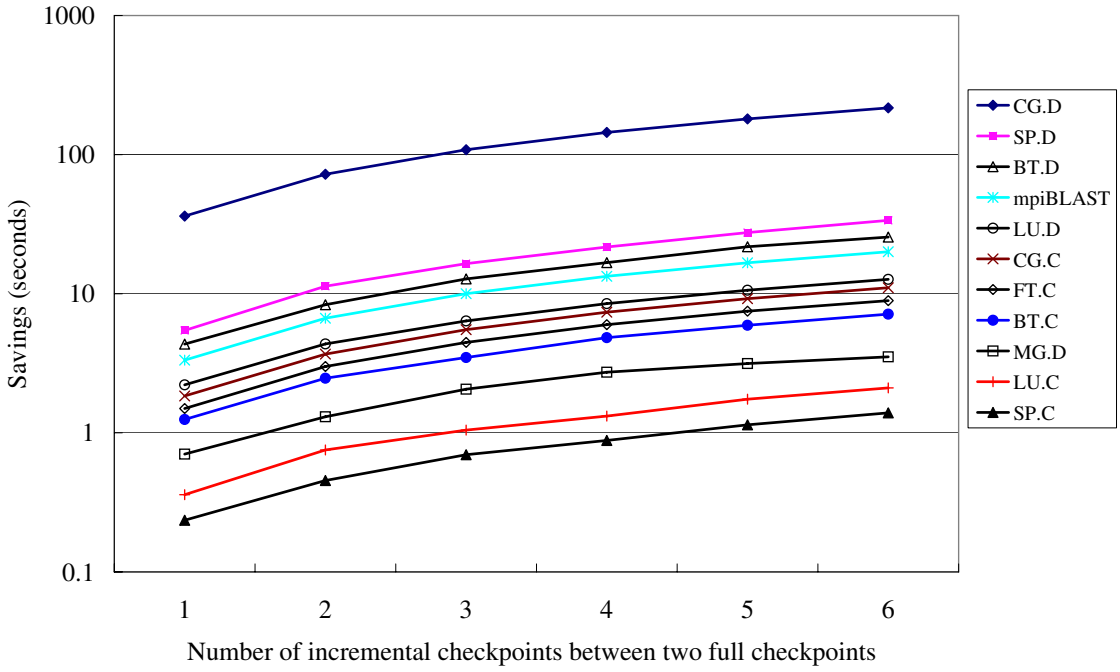


Fig. 11: Savings of Hybrid Full/Incremental C/R Mechanism for NPB and mpiBlast on 16 Nodes

Benchmarks	CG.D	SP.D	BT.D	mpiBLAST	LU.D	CG.C	FT.C	BT.C	MG.D	LU.C	SP.C
Savings of (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 of (R_{f+1_i} - 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 caused by 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

of incremental checkpoints between full checkpoints. They obtain

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

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 can determine

$$m = \left\lceil \frac{9.92}{P_i} - 1 \right\rceil$$

. Since $0 < P_i < 1$, a lower bound for m is 8.92, which indicates the potential for even more significant savings than just

those depicted in Figure 11.

Overall, the overhead of the hybrid full/incremental C/R mechanism is significantly lower than the original periodical full C/R mechanism.

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 [30, 11], CoCheck [32], etc.) and uncoordinated (MPICH-V [3, 4]). Coordinated techniques commonly rely on a combination of OS support to checkpoint a process image (*e.g.*, via the BLCR Linux module [11]) or user-level runtime library support. Collective communication among MPI tasks is used for the coordinated checkpoint negotiation [30]. Uncoordinated C/R techniques generally rely on logging messages and possibly their temporal ordering for asynchronous non-coordinated checkpointing, *e.g.*, MPICH-V [3, 4] that uses pessimistic message logging. The framework of OpenMPI [2, 19] 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 [13, 16, 18]. TICK (Transparent Incremental Checkpointer at Kernel Level) [13] is a system-level checkpointer implemented as a kernel thread. It supports incremental and full checkpoints. However, it checkpoints only sequential applications running on a single process that do not use inter-process communication or dynamically loaded shared libraries. *In contrast, our solution transparently supports incremental checkpoints for an* **entire MPI job** *with all its processes. Pickpt* [16] 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.* [38] 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, just as TICK was, while we cover an entire MPI job with all its processes and threads within processes. Agarwal *et al.* [1] 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 [13]. Another solution is to exploit page write protection, such as in *Pickpt* [16], 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 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.

Reactive FT vs. Proactive FT: Besides reactive fault tolerance (FT), including the full/incremental C/R technique discussed so far and reactive migration [23, 27, 10], proactive FT has recently become a hot research area. The feasibility of proactive FT has been demonstrated at the job scheduling level [25], within OS virtualization [33] and in Adaptive MPI [5, 6, 7] using a combination of (a) object virtualization techniques to migrate tasks and (b) causal message logging [12] within the MPI runtime system of Charm++ applications. Wang *et al.* [35] provide a live migration mechanism which is coarser grained than the Charm++ approach as FT is provided at the process level, thereby encapsulating most of the process context, including open file descriptors, which are beyond the MPI runtime layer. Proactive FT relies on failure predictors [29, 14, 15], whose accuracy still has to be further developed. Yet, reactive FT is still a requirement for HPC systems, and our solution improves reactive FT, optionally complemented by proactive support, at reduced overhead due to incremental process-level checkpoints for all MPI tasks.

Checkpoint Interval Model: Aiming at optimality for checkpoint 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. Yong [39] presented a checkpoint model and obtained a fixed optimal checkpoint interval. Based on Youngs work, Daly [8, 9] 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 [22]. Their scheme focuses on the fault tolerance challenge, especially in a large-scale HPC system, by providing optimal checkpoint placement techniques that are derived from the actual system reliability. As we discussed in Section 5, Naksinehaboon *et al.* provide a model to perform a set of incremental checkpoints between two consecutive full checkpoints [24] 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 full/incremental 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 incremental 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 full/incremental C/R mechanism among the MPI tasks. Results indicate that the performance of the hybrid full/incremental C/R mechanism is significantly lower 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. Even more significant saving would be obtained if the rate of incremental checkpoints between two full checkpoints was increased. Our hybrid approach can further be utilized to (1) develop an optimal (or near-optimal) checkpoint placement algorithm, which combines full and incremental checkpoint options in order to reduce the overall runtime and application overhead; (2) create and assess applications with varying memory pressure to measure the tradeoff between full and incremental checkpoints and to provide heuristics accordingly; and (3) combine related job pause/live migration techniques [33, 34, 35] with incremental checkpoints to provide a reliable multiple-level fault tolerant framework that incurs lower overhead than previous schemes. Overall, our hybrid full/incremental checkpointing approach is not only novel but also superior to prior non-hybrid techniques.

References

[1] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 277–286, New York, NY, USA, 2004. ACM.

[2] 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*, Sorrento, Italy, September 2005.

[3] G. Bosilca, A. Boutellier, and F. Cappello. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing*, November 2002.

[4] Bouteiller Bouteiller, Franck Cappello, Thomas Herault, Krawezik Krawezik, Pierre Lemarinier, and Magniette Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *Supercomputing*, 2003.

[5] S. Chakravorty, C. Mendes, and L. Kale. Proactive fault tolerance in large systems. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*. IEEE Computer Society, 2005.

[6] S. Chakravorty, C. Mendes, and L. Kale. Proactive fault tolerance in MPI applications via task migration. In *International Conference on High Performance Computing*, 2006.

[7] S. Chakravorty, C. Mendes, and L. Kale. A fault tolerance protocol with fast fault recovery. In *International Parallel and Distributed Processing Symposium*, 2007.

[8] J. T. Daly. A model for predicting the optimum checkpoint interval for restart dumps. In *International Conference on Computational Science*, pages 3–12, 2003.

[9] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, 2006.

[10] Fred Douglis and John K. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Softw., Pract. Exper.*, 21(8):757–785, 1991.

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

[12] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Comput.*, 41(5):526–531, 1992.

[13] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Supercomputing*, 2005.

[14] Xiaohui Gu, Spiros Papadimitriou, Philip S. Yu, and Shu-Ping Chang. Toward predictive failure management for distributed stream processing systems. In *IEEE ICDCS*, June 2008.

[15] Prashasta Gujrati, Yawei Li, Zhiling Lan, Rajeev Thakur, and John White. A meta-learning failure predictor for BlueGene/L systems. In *ICPP*, September 2007.

[16] Junyoung Heo, Sangho Yi, Yookun Cho, Jiman Hong, and Sung Y. Shin. Space-efficient page-level incremental checkpointing. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1558–1562, New York, NY, USA, 2005. ACM.

[17] C. Hsu and W. Feng. A power-aware run-time system for high-performance computing. In *SC*, 2005.

[18] Shang-Te Hsu and Ruei-Chuan Chang. Continuous checkpointing: joining the checkpointing with virtual memory paging. *Softw. Pract. Exper.*, 27(9):1103–1120, 1997.

[19] Joshua Hursey, Jeffrey M. Squyres, and Andrew Lumsdaine. A checkpoint and restart service specification for Open MPI. Technical report, Indiana University, Computer Science Department, 2006.

[20] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew 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*, 03 2007.

[21] Oak Ridge National Laboratory. Resources - national center for computational sciences (nccs). http://info.nccs.gov/resources/jaguar, June 2007.

[22] Yudan Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and Stephen Scott. A reliability-aware approach for an optimal checkpoint/restart model in hpc environments. *Cluster Computing, 2007 IEEE International Conference on*, pages 452–457, Sept. 2007.

[23] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys (CSUR)*, 32(3):241–299, 2000.

[24] Nichamon Naksinehaboon, Yudan Liu, Chokchai (Box) Leangsuksun, Raja Nassar, Mihaela Paun, and Stephen L. Scott. Reliability-aware approach: An incremental checkpoint/restart model in hpc environments. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 783–788, Washington, DC, USA, 2008. IEEE Computer Society.

[25] A. Oliner, R. Sahoo, J. Moreira, M. Gupta, and A. Sivasubramaniam. Fault-aware job scheduling for bluegene/l systems. In *International Parallel and Distributed Processing Symposium*, 2004.

[26] Ian Philp. Software failures and the road to a petaflop machine. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*. IEEE Computer Society, 2005.

[27] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Symposium on Operating Systems Principles*, pages 110–119, October 1983.

[28] 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*, February 1992.

[29] R. Sahoo, A. Oliner, I. Rish, M. Gupta, J. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 426–435, 2003.

[30] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, October 2003.

[31] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

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

[33] J. Varma, C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Scalable, fault-tolerant membership for MPI tasks on hpc systems. In *International Conference on Supercomputing*, pages 219–228, June 2006.

[34] 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*, April 2007.

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

[36] Andrew Wissink, Richard Hornung, Scott Kohn, and Steve Smith. Large scale parallel structured amr calculations using the samrai framework. In *Supercomputing*, November 2001.

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

[38] Sangho Yi, Junyoung Heo, Yookun Cho, and Jiman Hong. Adaptive page-level incremental checkpointing based on expected recovery time. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1472–1476, New York, NY, USA, 2006. ACM.

[39] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, 1974.