2013 IEEE 27th International Symposium on Parallel & Distributed Processing

Adaptive Incremental Checkpointing via Delta Compression for Networked Multicore Systems

Itthichok Jangjaimon and Nian-Feng Tzeng
Center for Advanced Computer Studies
University of Louisiana at Lafayette
Lafayette, LA 70504
{ixj0704, tzeng}@cacs.louisiana.edu

***Abstract*—Checkpointing has been widely adopted in support of fault-tolerance and job migration, with checkpoint files preferably kept also at remote storage to withstand unavailability/failures of local nodes in networked systems. Lately, I/O bandwidth to remote storage becomes the bottleneck for checkpointing on a large-scale system. This paper proposes an adaptive incremental checkpointing (AIC), aiming to reduce the checkpointing file size considerably so that its involved overhead is lowered and thus the expected job turnaround time drops. Given production multicore systems are observed to have unused cores often available, we design AIC to make use of separate cores for carrying out multi-level checkpointing with delta compression at desirable points of time adaptively. We develop a new Markov model for predicting the performance of such multi-level concurrent checkpointing, with AIC performance evaluated using six SPEC benchmarks under various system sizes. AIC is observed to lower the normalized expected turnaround time substantially (by up to 47%) when compared to its static counterpart and a recent multi-level checkpointing scheme with fixed checkpoint intervals.**

Keywords—Adaptive checkpointing; delta compression; fault tolerance; incremental checkpointing; Markov model; multicore systems; two-level checkpointing.

I. INTRODUCTION

Checkpointing saves the states of a running process to a persistent storage, allowing it to be restarted from that stored state (called a checkpoint). It has been applied successfully in support of fault tolerance and job migration essential for virtualization and cloud computing [7], in addition to facilitating code debugging. Given a system node usually contains multiple cores nowadays, checkpointing and execution recovery after failures in the system can be handled more diversely and efficiently than can be possible in its single-core counterpart. This article focuses on checkpointing in networked multicore systems, aiming to reduce its overhead with an aid of multiple cores existing in each system node.

Recent work has shown that checkpointing to remote storage in a large-scale networked system is relatively expensive but necessary to have acceptable reliability of long-running jobs [11], calling for multi-level checkpointing able to tolerate various failure/unavailability types. In general, checkpointing overhead is dictated by the checkpoint file size due to bandwidth constraints on local/remote storages [11, 17, 19]. Given continuing growth in the application program footprint, the checkpoint size is expected to rise going forward. Much research has been carried out to lower or even hide checkpointing overhead [6, 16, 18]. One common strategy follows incremental checkpointing [6, 16], which saves only modified memory pages into the checkpoint. In addition, delta compression (or differencing compression) between successive checkpoints is employed to further reduce the checkpoint size.

Previously, a checkpointing scheme employs simple delta compression (like an XOR method) because job execution is suspended during the delta compression time. In contrast, our interest here lies in process execution on multi-core systems, with a separate core for handling delta compression and writing compressed outcomes to remote storage concurrently when the process is executed on the other core(s). An idle core is frequently available at each node of real-world systems (as demonstrated by computing system logs from the Los Alamos National Lab, LANL, in Section II), and such an available core is exploited in this work for carrying out delta compression and outcome writes to remote storage concurrently without suspending job execution progress in other active computation cores. As compression performance relies on the degree of similarity between two consecutive checkpoints, the desirable points of time to take checkpoints is paramount in any effective checkpointing mechanism (detailed in Section II), usually calling for adaptive checkpointing. This is unlike traditional checkpointing which takes checkpoints periodically in an equidistant interval computed by averaging checkpoint overhead amounts and failure rates to minimize the process's expected runtime [4, 11, 18, 21, 24].

Recently, dynamic checkpoint intervals or skipping some (fixed) checkpoints have been considered [14, 23], because checkpointing overheads and system parameters vary. This article presents design and implementation of adaptive incremental checkpointing (AIC) with delta compression realized by dedicated cores for networked multicore systems. AIC determines the desirable points of time to take checkpoints adaptively based on predicted checkpoint overhead during execution progress and system parameters. It relies on fast prediction governed by Stepwise regression [13] and a Gradient Descent algorithm [1] to estimate at fine granularity in real time, the overhead of incremental checkpointing with delta compression. Unlike its static counterparts treated earlier [4, 11, 21, 24], AIC leverages on the fact that the in-memory process contents of a running task and those of its previous checkpoints have varying degrees of similarity during its execution, dictated by how

1530-2075/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPS.2013.33

IEEE computer society

much the working set is in common to those working sets when previous checkpoints were taken. Hence, it calls for estimating the similarity degree during task execution to dynamically choose a desirable point of time that yields the smallest checkpoint file after delta compression. AIC differs from earlier adaptive checkpointing mechanisms, which are unaware of delta compression dynamics, and are realized by either skipping certain *fixed* checkpoints dynamically [14] or treating single-level checkpoints without separate cores for remote concurrent checkpointing during job execution [23].

A multi-process job may involve either a lot of communications among its processes during job execution, as exemplified by heroic MPI applications, or limited communications (typically only at the beginning and the end of its execution), such as MapReduce-like jobs and many of Recognition, Mining, and Synthesis (RMS) workloads [2]. For brevity, we refer to these two distinct job types as MPI and RMS tasks, respectively. Given that multi-level checkpointing has just been introduced recently [11], with prior concurrent checkpointing work focusing only on net measured overhead [16] or single-level checkpointing [22], we develop, for the first time, a new Markov model for predicting expected job turnaround time (for single-process, MPI, and RMS jobs) under *multi-level concurrent* checkpointing. Under the actual application and system profiles from Lawrence Livermore National Laboratory (LLNL), our multi-level concurrent checkpointing always reduces the turnaround time noticeably when compared to its Moody's counterpart, the best known multi-level checkpointing model [11].

This work focuses solely on developing AIC for RMS tasks, where each processes can freely checkpoint at different times. (AIC for MPI tasks requires tracking similarity degrees of all MPI processes for coordinated checkpointing, which is beyond the scope of this work and will be treated in a separate article.) Our AIC has been implemented in BLCR (Berkeley Lab Checkpoint/Restart) [8] to evaluate its real performance using six SPEC CPU2006 benchmarks, as representatives of RMS task processes. The results demonstrate that in the absence of failures, AIC lengthens actual execution times only negligibly (upper-bounded by 2.6%) in comparison to those without checkpointing. For real-world networked systems with potential failures, AIC reduces the expected job turnaround time by up to 47% when compared with its non-adaptive counterpart.

II. CHECKPOINTING BASICS AND MOTIVATION

A. Checkpointing Basics

A networked system may write its checkpoint data to various places, involving different levels of overhead and resilience. *Multi-level checkpointing* [11, 21] is the most noticeable example, able to handle different kinds of failures. In addition to local disks, for example, the system may also write its checkpoints to remote nodes [11, 21], to distributed file storage [11], or to the main memory of a group of nodes that form RAID-5 redundancy [11, 18]. Naturally, checkpointing overhead is dictated mainly by the checkpoint size, prompting incremental checkpointing and delta compression for size reduction. An example is given below.

Fig. 1. Process in-memory contents at checkpoints. Each grey block represents a page whose contents have been modified or allocated since the last checkpoint. An underlined block denotes a page kept in the previous checkpoint, available for further size reduction by delta compression.

Scenario 1: Consider a simple process with seven initial memory pages, called *A* to *G*, involving three checkpointing instances. Assume that the process allocates two more pages, *H* and *I*, and modifies pages *A*, *B*, *D*, *E*, *H*, *I*, before the second checkpoint, and that the process frees page *C* and modifies pages *D*, *E*, *F*, *G*, after the second checkpoint but before the third checkpoint.

Fig. 1 illustrates memory contents involved in the three checkpoints. While incremental checkpointing keeps simply modified and new pages (e.g., pages *A*, *B*, *D*, *E*, *H*, and *I*, in the second checkpoint), delta compression further reduces the checkpointed size by writing only difference (called *delta*) between each modified page (called *target* data) and its corresponding old version (called *source* data) written in the previous checkpoint, if available (e.g., pages *A*, *B*, *D*, and *E* in the second checkpoint). Given the *source* data and associated *delta*, decompression can produce the *target*. The very first checkpointing instance is always *full*. To restart a process, incremental checkpointing requires the last full checkpoint and *all incremental checkpoints* generated after that full one. The system may generate a full checkpoint periodically to limit this cumulative overhead.

We classify checkpoint type into *local* and *remote*. Local checkpoint does (possibly incremental) checkpoint at the disk (or memory) of a local node. Remote checkpoint performs a mirror checkpoint over the network (possibly to the remote storage, or to RAID-5 group). To lower its overhead, AIC remote checkpoint is enhanced innovatively by **(1)** monitoring the process page similarity so that the system does checkpoint when the similarity is high, and **(2)** concurrently executing delta compression and delta transmission on separate processor cores, allowing non-interrupted job execution. As a result, AIC remote checkpointing is dynamic (instead of static, carried out periodically), facilitating more aggressive delta compression [10]. It enjoys marked 47% reduction, when compared with its static counterpart.

B. Motivating Example for Adaptive Checkpointing

The basic idea of AIC relies on the fact that the time duration for completing delta compression (dubbed *delta latency*) and its resulting size (dubbed *delta size*) are dynamic with respect to the checkpointing moment (dubbed *checkpoint time*). AIC aims to predict the delta latency and

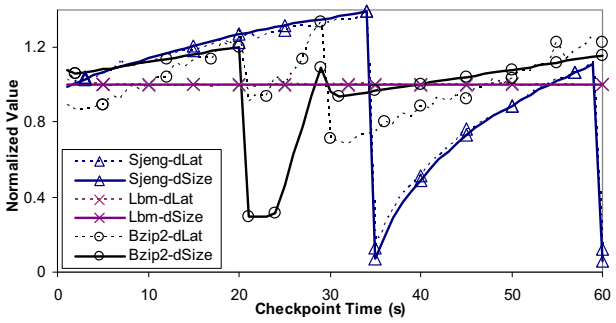


Fig. 2. Normalized delta latency and delta size of three SPEC benchmarks (Sjeng, Lbm, and Bzip2) obtained using our testbed (detailed in Section V) when taking the next (incremental) checkpoint at different points of time over a 60-second interval. The outcomes are normalized over respective benchmark's latency/size means in the interval.

the delta size for determining the most beneficial moments to take checkpoints adaptively. A motivating case observed in our real experiment is given next to illustrate the fact that dynamic two-level checkpointing is preferred.

We took page-aligned delta compression (detailed in Section IV.C) between the first full checkpoint and the second incremental checkpoint at different moments, with the normalized delta latency and delta size outcomes (with respect to their corresponding benchmark's means, i.e., (delta latency (or size)) / (mean latency (or size) over the 60-second interval)) versus the checkpointing time depicted in Fig. 2. Each delta latency result in Fig. 2 includes the time to read two checkpoints, to conduct delta compression, and to write delta back to the local disk.

The results of Fig. 2 reveal clearly that wide swings in the delta latency/size over time may exist for a benchmark, making the *selection of proper times to take checkpoints* especially crucial for overhead reduction. For example, Sjeng exhibits a decrease of 95% in its delta latency and delta size if its checkpoint is taken at the 35th second, instead of the 32nd second (shown by line segments with triangles in Fig. 2). In fact, five (out of those six) SPEC benchmarks examined (as listed in Table 3; see Section V.B) have wide swings in their delta latency/size curves. Our adaptive checkpointing is based on predicting the delta compression latency/size at a given checkpoint time effectively to choose the most desirable point of time for checkpointing.

C. Opportunities for Concurrent Checkpointing

Since AIC aims to exploit an idle core per node for its concurrent checkpointing, one may ask whether such an opportunity exists. To answer this question, we analyze usage logs of 5 computing systems at Los Alamos National Laboratory (LANL) available in the public domain [15]. The 5-year logs consist of over 3 million job records, each with the submit time, dispatch time (from queue), end time, and running node IDs for every execution process. We define a *candidate job* as the job where each of its processes always has one idle core throughout its execution. In other words, a candidate job can exploit those idle cores for concurrent checkpointing without purging or suspending other job execution processes. Table 1 lists properties of 5 systems and the numbers of candidate jobs. It shows that more than 40% of jobs running in 4 systems (i.e., Systems 15, 23, 8, and 16) always have one idle core for each of their processes. On the other hand, System 20 has only 17% candidate jobs, chiefly because the scheduler assigned processes to small subsets of nodes. It is possible to rectify the scheduler slightly to leave one core dedicated for checkpointing, if available, so that the numbers of candidate jobs can be boosted. The last column of Table 1 lists the percentages of candidate jobs under the rectified scheduler, which leads to more candidate jobs for the systems with multiple nodes (i.e., all but System 15). One possible technique for such rectified scheduling is to let the local scheduler in each node reserve the dedicate core for concurrent checkpointing. It can be realized easily by means of the CPU affinity library or `taskset` Linux command (which is adopted by our AIC implementation).

TABLE 1. LANL SYSTEM CHARACTERISTICS

System ID	System Type	# of nodes in logs	# of cores per node	% of candidate jobs	% of candidate jobs after rescheduling
15	NUMA	1	256	50%	50%
20	Cluster	256	4	17%	32%
23	Cluster	5	128	77%	78%
8	Cluster	164	2	47%	75%
16	Cluster	16	128	41%	42%

III. MULTI-LEVEL CONCURRENT CHECKPOINTING

This section treats multi-level concurrent checkpointing, with Section III.A-B laying the groundwork of the Markov model established in Section III.C. Section III.D presents numerical results, comparing our static concurrent model with its earlier Moody's counterpart [11]. Based on obtained results, we can reach the appropriate adaptive checkpointing decision, as outlined in Section III.E.

A. Assumptions and Definitions

Checkpointing is done transparently, without explicitly requested by applications [6, 21, 23]. This article assumes failure inter-arrival times to follow an exponential distribution with the rate of λ over time, as commonly found in earlier work [4, 21, 23, 24]). Additionally, failures are assumed to be independent [11, 21, 23] and to possibly happen at any time (even during process recovery). Once it occurs, a failure is detected by a diagnostic mechanism (not treated in this article). Failure types are detailed next.

The *transient* failures (e.g., intermittent failures [3] and faults due to external interferences like alpha particles and neutron) can be recovered by re-running the application on the same core. On the other hand, *permanent* faults can result in partial node failures or total node failures. A partial node failure in a multicore node damages some cores but leaves one or multiple operational cores for application recovery on the node, whereas a total node failure brings down all its cores and also causes its local disk to become unavailable. Any application run on a totally failed node can

TABLE 2. RELEVANT SYMBOLS

Symbol	Definition
t	Base process execution time
T	Total expected runtime
T_{int}	Expected runtime of an interval
NET^2	Normalized expected turnaround time
c_k	Checkpoint latency of level-k checkpoint
r_k	Recovery time of level-k checkpoint
f_k	Level-k failure
λ_k	Level-k failure rate
λ	System failure rate
n_k	Moody's parameter for level-k checkpoint
w	Work time span
w_L^*	Local optimum work time span
B_k	Estimated bandwidth of level-k checkpoint
dl	Delta latency
ds	Delta size

Note: The additional subscript (i) denotes the parameter value during checkpoint Interval i.

be recovered only when the checkpoints of the application are kept remotely (either at remote storage or remote group of nodes for use to resume its execution). Similar to prior work [11], we assume an infinite pool of spare cores. This assumption is likely to hold since real jobs often allocate extra cores and the repair rate is higher than the failure rate.

The system has multiple checkpoint levels via different checkpointing means. We denote level-k checkpoint as L_k. Let c_k be the checkpoint latency of L_k and r_k be the recovery time of L_k. Level-k failure is denoted by f_k, with an arrival rate of λ_k. Note that summation of all λ_k's equals the system failure rate λ. A higher level checkpoint can recover all lower-level failures. Being the most basic checkpoint operation, L_1 is embedded in all higher-level L_k, for k>1. Incremental checkpointing or delta compression can be applied to reduce c_k. In Section III.B-D, the static model is assumed, where all c_k and r_k are constant. This assumption is relaxed in Section III.E for adaptive checkpointing.

An example multi-level checkpointing system has been considered recently [11], with three levels involved, as follows. The local checkpoint L_1 has a latency of c_1, equal to the time for writing a checkpoint to the local disk or memory. In the case of MPI programs, c_1 also includes the time for *coordinated checkpointing*, where all in-flight messages and synchronization are properly handled. Let L_2 be the remote checkpoint to a RAID-5 group of nodes, while L_3 be the remote checkpoint to remote storage. Since L_2 and L_3 must inherently execute L_1 at the beginning, their latencies equal c_1 plus the time to send checkpoint over the network (to a RAID-5 group or remote storage). In our model, each constituent node of a networked system at hand includes multiple cores, with at least one core pre-allocated for supporting remote checkpoints. A local checkpoint is always followed immediately by one or multiple remote checkpoints. Relevant notations are listed in Table 2.

B. Process Execution

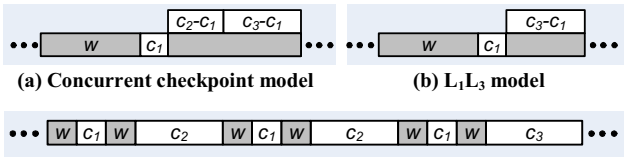


Fig. 3. An interval of process execution under concurrent and sequential 3-level checkpoint models. The shaded block represents time spent for actual work. Note that work time spans w in (a), (b), and (c) are not necessarily identical.

A *base* process execution time, t, refers to the time for its execution in the absence of checkpointing or failures. However, actual process execution usually involves checkpoints, say n of them, during the course of its execution. Within the checkpoint interval i in our concurrent model, for $1 \le i \le n$, **(1)** the process does its actual work (for w seconds, dubbed *work time span*), **(2)** the system performs local L_1 *checkpoint* **sequentially**, requiring the process to halt its execution until L_1 has been completed, and **(3)** the system initiates remote checkpoint (L_3 or both L_2 and L_3) **concurrently** at the checkpointing core when the process is being executed on separate cores. Fig. 3(a) illustrates an interval of process execution for the 3-level concurrent model (composed of L_1, L_2, and L_3). Since at the end of Block c_1, the checkpoint file is already generated, L_2 and L_3 can initiate checkpoint transfer immediately, yielding the latencies of c_2-c_1 and c_3-c_1. Note that Blocks c_2-c_1 and c_3-c_1 refer to execution instances on the pre-allocated checkpointing core. As only one available checkpointing core is assumed, our model does not initiate any L_1 until the last L_3 has finished. Should a failure occur, the system resumes job execution at the latest recovery point, with its restart time determined by the checkpoint type chosen (e.g., r_1, r_2, or r_3).

In contrast, the Moody model performs multi-level checkpointing sequentially (as depicted in Fig. 3(c)). It is governed both by work time span w and by parameter n_k, which indicates how many level-k checkpoints are taken in between level-k+1 checkpoints. The Moody model restarts the checkpoint from the latest checkpoint able to recover from the arising failure.

C. Markov Model for Concurrent Multi-level Checkpointing

We evaluate the expected runtime of the checkpoint interval of multi-level checkpointing using the Markov model, which is a directed-edge graph representing states and state transitions. The state is annotated with the time spent in that states if no failure occurs. Each edge is associated with **(1)** the probability of state transition on that edge, and **(2)** the expected time spent in the old state before transition into the new one. In a general form, each state has up to k+1 edges, one corresponding to the success case, and the rest corresponding to k recovery states when a level-k failure, f_k, occurs. Since the time between failures follows an exponential distribution, the edge-associated values (i.e., the transition probability and the expected time spent) can be calculated. Once the model is constructed, the formula for

calculating expected runtime of the checkpoint interval can be obtained by solving a set of linear equations [21]. We also apply simplification technique [12] to merge edges and states in our model.

While the system under consideration is equipped with three checkpointing levels, in practice, it may enable only one or two checkpointing levels. To be specific, we construct the Markov model for a networked system with **(1)** L_1 and L_3, **(2)** L_2 and L_3, and **(3)** L_1, L_2, and L_3, denoted by L_1L_3, L_2L_3 and $L_1L_2L_3$, respectively, as illustrated in Fig. 4. Note that L_3 is always enabled to avoid restarting from the process beginning. Process execution of an interval in L_1L_3 model is presented in Fig. 3(b). The process running in w and c_1 intervals in Fig. 3(b) is denoted by State 1, labeled by a shaded square, in Fig. 4(a). If no failure occurs during the whole $w+c_1$ interval (corresponding to the black arrow), State 2 (c_3-c_1) is entered, where L_3 starts a remote checkpoint to remote storage concurrently. If L_3 succeeds, the process finishes its execution for the current interval. Note that at the end of Block c_1, L_1 successfully generates a checkpoint file, which includes the application state at the end of Block w. After that, L_3 sends this generated file to remote storage concurrently while the main process continues its execution for c_3-c_1 seconds. If any failure occurs before the next L_1 checkpoint, the process must rerun this lost execution. Next, let us consider two failure edges leaving State 1. Since L_1L_3 enables only L_1 and L_3 checkpointing, f_2 must be recovered by L_3 checkpoints (i.e., State 4). Once the recovery state succeeds (from State 3 or 4), the application must rerun the unsaved work (State 5) before returning to State 1. If a failure occurs during these states, it moves to State 3 or 4 accordingly. Let us consider the failure arising in State 2. In this case, L_1 checkpoint is already saved. If f_1 occurs, the application moves to State 6 and then starts over at State 2. However, if f_2 or f_3 occurs during State 2 or 6 (where L_3 has not finished), the application must restart from the old L_3 checkpoint files, forcing it to enter State 4 and then rerun State 5 (c_3-c_1 of the previous interval) afterwards.

Note that transition from State 1 to State 2 acts as if the application has two independent tasks running on different cores, one for its execution and the other for L_3 checkpointing, where the failure might occur to any of those two tasks. Since the tasks are independent, the overall expected runtime is the maximum of the two. In addition, recovery from a failure at the checkpointing core does not require entering State 5. As a result, the expected runtime of the application over segment c_3-c_1 in Fig. 3(b) is always greater than that of the L_3 process, enabling us to simplify the model as presented in Fig. 4(a).

The models for L_2L_3 and $L_1L_2L_3$ are derived similarly. Figs. 4(b) and 4(c) present the constructed models where all state transitions to the same states are merged. Given w, λ_1, λ_2, λ_3, c_1, c_2, c_3, r_1, r_2, and r_3, the expected runtime of the interval, T_{int}, can be calculated. Finally, the total expected runtime of the application, T, is the summation of all interval expected runtimes. Our performance metric of interest is the normalized expected turnaround time, NET^2, defined as

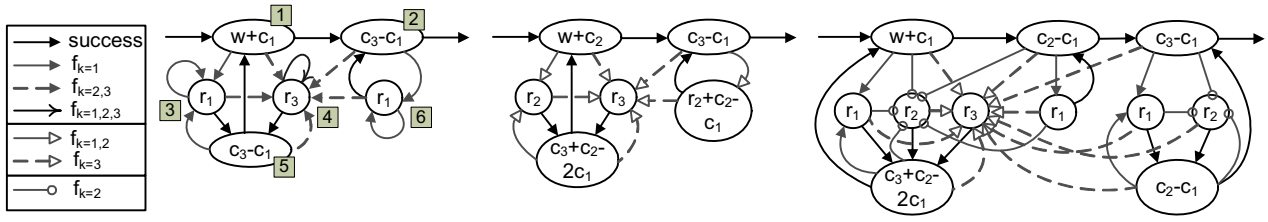
$$NET^2 = T/t,$$

where T is the application total expected runtime and t is its base process execution time. NET^2 gives the estimation of how much longer the application is expected to run in the system versus its base runtime. Our goal is to find the lowest NET^2 value by varying work time span w. This can be done numerically, like in earlier work [11, 21].

D. Numerical Results for Static Concurrent Checkpointing

We compare the performance of our static concurrent multi-level checkpointing with that of the Moody model pursued recently [11]. System and application profiles used for deriving our numerical results are taken directly from the prior work [11], as briefly described below. The application is pF3D, an MPI program for laser-plasma simulation, which requires 1-GB memory per process. The system of interest is the Coastal cluster, which has 1024 nodes, with $\lambda_1=2\times10^{-7}$, $\lambda_2=1.8\times10^{-6}$, and $\lambda_3=4\times10^{-7}$. Its L_1 is a coordinated checkpoint to RAM disk with $c_1 = 0.5$. L_2 writes checkpoints to the main memory of a RAID-5 group of nodes with $c_2 = 4.5$. L_3 performs checkpointing to the Lustre distributed file system, with $c_3 = 1052$. Each recovery time r_k is set to equal c_k. Taken from [12], the Moody model code explores its variables, searching for the optimal one, which yields the highest *efficiency* possible. In fact, this metric of efficiency is the inverse of our metric of interest, NET^2, which is believed to better reflect job execution behavior. The same set of λ and c parameters is used in our concurrent model to search for lowest NET^2 by varying w.

We obtain results for pF3D job execution under different system size scaling, which signifies anticipated future systems with more nodes and cores. The system size affects MPI applications in two ways. First, the I/O bandwidth to remote storage is more congested as the size grows, i.e., c_3 increases proportionally. By contrast, c_1 and c_2 are expected to remain unchanged since their corresponding bandwidth amounts are expected to grow with the system size. Second,



(a) L_1L_3 Model **(b) L_2L_3 Model (merged)** **(c) $L_1L_2L_3$ Model (merged)**

Fig. 4. Proposed Markov model for multi-level concurrent checkpointing, with each edge representing possible state transition. For simplicity, all edges that point to the same state are merged in (b) and (c).

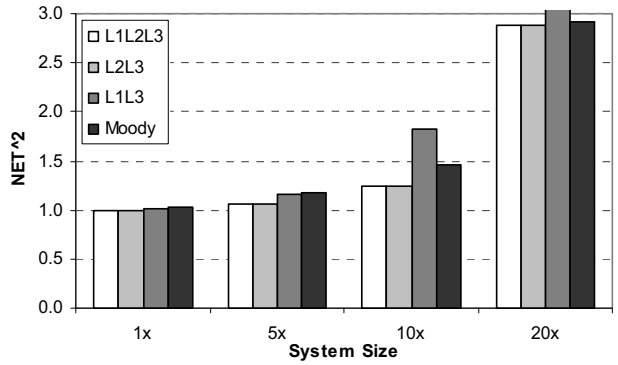


Fig. 5. NET^2 (normalized expected turnaround time) of the MPI program, pF3D, under various system sizes.

the failure rate is to increase proportionally since a failure of any MPI process fails entire job execution. Fig. 5 illustrates NET^2 of pF3D under various concurrent checkpoint models versus the Moody model. It shows that L_2L_3 and $L_1L_2L_3$ are very closed to each other consistently, always yielding the lowest NET^2. This suggests that L_2L_3 is preferred over $L_1L_2L_3$ since L_1 does not add any measurable benefit and can be dropped. Interestingly, Moody's optimal results also accommodate only L_2 and L_3 without incurring L_1. The degree of NET^2 improvement for L_2L_3 (compared with the Moody results) rises as the system size grows, until it reaches 10×, mostly due to increases in the failure rate and c_3. On the other hand, L_1L_3 incurs much more NET^2 at large system sizes. This is because L_1L_3 must recover all f_2 failures (which account for the most frequent failures with λ_2 equal to 1.8×10^{-6}) from level-3 checkpointing whose latency c_3 is higher in a large system, involving much bigger overhead than with level-2 checkpoints. As can be seen in Fig. 5, NET^2 improvement under L_2L_3 almost disappears at the size of 20×, where the system then experiences exceeding overhead due to frequent leave-3 recovery which overwhelms L_3, the Lustre distributed file system [11].

Next, we study the effect of concurrent checkpointing for RMS applications, which require limited inter-process communications. In this case, the system size has little effect on the failure rates under RMS applications (RMS for short) as each process therein can run *almost independently*. We

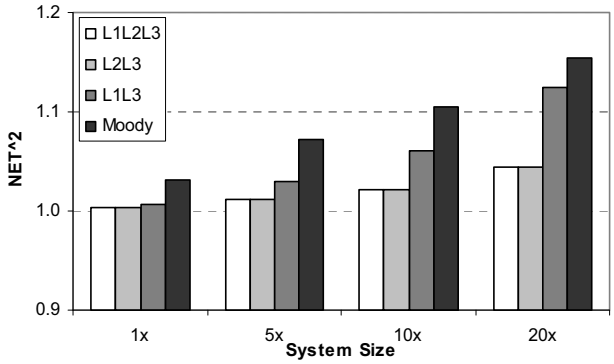


Fig. 6. NET^2 of RMS under various system sizes.

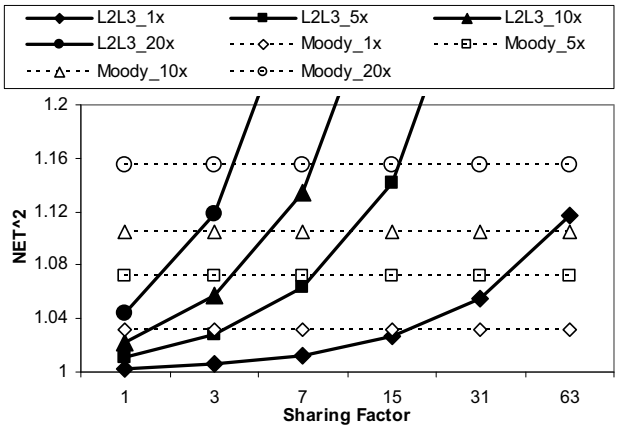


Fig. 7. NET^2 of L_2L_3 under different sharing factor (SF) values and system sizes for RMS applications.

assume an RMS application with a similar profile as that of pF3D (such as the memory footprint size and the number of processors) upon its execution. As revealed in Fig. 6, our concurrent checkpoint models always outperform the Moody counterpart for RMS. Again, L_2L_3, and $L_1L_2L_3$ have very close outcomes, yielding the lowest NET^2. The improvement gaps between L_2L_3 and its Moody counterpart expands as the system size scales up. Hence, concurrent checkpointing is demonstrated to benefit both MPI and RMS application, with L_2L_3 yielding nearly the best NET^2. Given its lower complexity (better suitable for online decision of AIC) than that of $L_1L_2L_3$, *L_2L_3 will be our focus for the remainder of this article, aiming to support RMS program execution*.

While idle cores are likely to present in real systems (as detailed in Section II.C), they may be scarce in many situations, requiring one idle core to cover multiple active computation cores which execute application processes. We define *sharing factor* (SF) as the number of computation cores that share one checkpointing core. We assume the worst case of sharing where all sharing processes ask for the checkpointing core to handle their checkpoints *at exactly the same time*, with checkpointing core resources (such as I/O bandwidth) shared evenly. Fig. 7 illustrates L_2L_3 performance under different SF values and system sizes. Moody's NET^2 results are also added for gauging how many cores can be shared while concurrent checkpointing still outperforms the Moody counterpart. As can be found in Fig. 7, L_2L_3 is still profitable when 3-15 processes share one checkpointing core under 1×-20× system sizes.

E. Adaptive Incremental Checkpionting (AIC) with Delta Compression

The treatment so far assumes that c_1, c_2, c_3, r_1, r_2, and r_3 are constant, with the model searching for the optimal work span w^* to yield the lowest NET^2 (normalized expected turnaround time). However, when incremental checkpointing and delta compression are applied to reduce overhead, the checkpoint latency may vary greatly (as shown in Section II.B), signifying good opportunities for further overhead reduction by taking checkpoint adaptively at desirable points

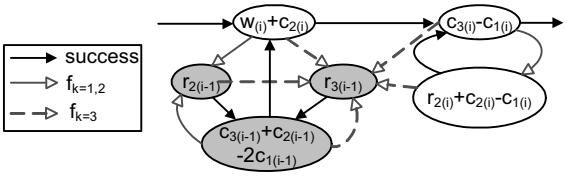


Fig. 8. Non-static multi-level concurrent checkpoint model of Interval *i*.

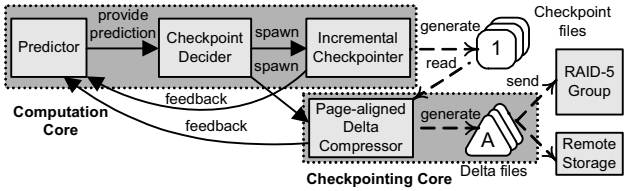


Fig. 9. AIC key components in networked multicore system.

of time. In this case, the selected work time span, checkpoint latencies, and recovery times at each interval are varied.

Our Adaptive Incremental Checkpionting (AIC) requires an enhanced prediction model to capture dynamics. Given the two-level concurrent model of L_2L_3 is sufficient for obtaining near-optimal NET^2 (as demonstrated earlier), we enhance L_2L_3 to arrive at a non-static multi-level concurrent checkpoint model. To this end, the subscript (*i*) is added to variables for indicating their values at Interval *i*. For example, $c_{k(i)}$ is the level-*k* checkpoint latency at Interval *i*. Fig. 8 illustrates the AIC model for Interval *i*, where those states that are different from L_2L_3 are marked in grey. The grey states involve parameters from Interval *i*-1 since Interval *i* uses checkpoints produced therein.

The formula for calculating the expected runtime of the checkpoint interval T_{int} can be obtained, although the optimal work span w^* has no closed form. Instead of exploring the whole search space (like an offline algorithm [11, 21], which may take several minutes to finish), we follow the *Extreme Value Theorem* to search for a local optimum work time span, w_L^*, by comparing NET^2 at both search boundaries and one local point with $\partial(NET^2)/\partial w = 0$ (zero derivative) obtained via the *Newton-Raphson* (NR) approximation method. Our NR returns a point for comparison after it either reaches desired precision or iterates 200 times, involving O(1) complexity. In general, AIC requires less than 5 NR iterations in our experiments, incurring low total overhead (< 3%; detailed in Section V).

Our AIC examines *periodically* the estimated checkpoint latency (i.e., $c_{k(i)}$) to decide if a checkpoint should be taken. At a decision time, AIC calculates w_L^* from the current $c_{k(i)}$ and other constant parameters. If w_L^* is smaller than the current interval elapsed time, AIC takes a checkpoint immediately. Subsection IV.D provides details of AIC prediction on the checkpoint latency, $c_{k(i)}$. NET^2 under varying parameters can be expressed by

$$NET^2 = \sum_{i=1}^{n} T_{\text{int}(i)} \Big/ t, \quad (1)$$

where $T_{int(i)}$ is the expected runtime of Interval *i*.

IV. Design and Implementation Details

AIC seeks to meet two design goals: **(1)** low overhead, thus calling for lightweight strategy or offloading tasks to separate cores, and **(2)** usability, suitable for applications with any memory footprint on diverse machines without profiling or code-intruding. Fig. 9 outlines AIC key components detailed as follows.

Periodically, AIC ***Predictor*** makes its prediction from monitored lightweight metrics. The predictor does not need any profiling, able to adjust its prediction model online based on feedbacks from other components. Given predicted data, AIC ***Checkpoint Decider*** may spawn the ***Incremental Checkpointer*** to get a checkpoint, which would be delta compressed by the page-aligned ***Delta Compressor*** launched on the dedicated core. The measurements of predicted value (e.g., delta latency) are sent back to the predictor for its model update. Finally, the delta files are sent over the network to the **RAID-5 Group** (under L_2 checkpointing) and **Remote Storage** (under L_3 checkpointing), where the system may use them to restart the application. While Predictor, Checkpoint Decider, and Incremental Checkpointer are executed on the ***Computation Core*** (together with the application, see Fig. 9), the delta compressor and remote chekcpointer are run concurrently on a dedicated core (dubbed ***Checkpointing Core***) for low time overhead.

A. BLCR and Xdelta3

We have modified Berkeley Lab Checkpoint/Restart (**BLCR,** a checkpointing implementation for high-performance applications in Linux) [8] to arrive at AIC. Additionally, a page-aligned delta compressor, **Xdelta3-PA,** has been developed to enable AIC. BLCR consists of kernel modules, shared libraries, and a set of command-line tools. Application codes must be compiled with the BLCR shared library. We use BLCR version 0.8.2, which does not support incremental checkpointing. Originally based on Rsync algorithm [20], Xdelta3 is a delta compressor that hashes blocks of *source* data and uses the hash table to identify the longest match in *target* data. Our Xdelta3-PA is derived from the Xdelta3 (version 3.0y) library.

B. Incremental Checkpointing and AIC Portability

We have enhanced BLCR's kernel modules and shared libraries to support incremental checkpointing via the Unix `mprotect()` system call for collecting the list of dirty pages during each checkpoint interval. The `mprotect()` system call lets a program set its memory page protection from writing. If the program attempts to modify such a page, the page-fault signal is raised and can be caught by the signal handler. At the beginning of each checkpoint interval, AIC write-protects target pages in process address space. Each first writing attempt to a protected page (**1**) triggers the signal handler to add the page to the dirty page list and (**2**) unprotects the page. AIC kernel module then uses this dirty page list to write modified pages into the checkpoint.

In regard to its portability, AIC inherits BLCR's support for wide-ranged architectures (i.e., `x86`, `x86_64`, `PPC`/`PPC64` and `ARM`) on Linux. L_k can be easily setup and reconfigured via environmental variables. AIC additionally deploys its

own signal handlers for page faults and alarms, invalidating any application that also uses these signals. However, we found that they are rarely used in our target processor-memory intensive applications such as SPEC and RMS.

C. Page-aligned Delta Compression

AIC uses Xdelta3-PA (page-aligned delta compression) to difference *each* dirty page in the current checkpoint against its previous version, if existing in the previous checkpoint, for a smaller checkpoint footprint. Other data, such as CPU states, process linkages, and opened file descriptors, constitute a minor fraction of the checkpoint file and are thus not compressed. While applying delta compression between two *entire checkpoints* might yield smaller footprint sizes in certain cases, our Xdelta3-PA, being page-aligned, is indispensable because it **enables** our checkpoint latency predictor to estimate the delta compression cost on the per-page basis.

Let a **hot page** be the dirty page of the current interval which was also modified during the previous checkpoint interval. Xdelta3-PA performs delta compression only between **hot** pages and their corresponding old pages in the previous checkpoint. In addition, the order of memory contents resided in checkpoint file is rearranged in support of fast compression by Xdelta3-PA, running concurrently on the checkpointing core by means of the `taskset` Linux command for low overhead.

In contrast to performing in-memory delta compression before writing delta to the disk found previously [19], AIC performs delta compression (as a part of remote checkpoint mechanism) after it writes an incremental checkpoint to the local disk. The rationale is that doing delta compression on-the-fly requires buffering the contents of **every** old page in the previous checkpoint, leading to excessively high memory overhead for large applications.

D. AIC Leightweight Predictor

AIC predicts values necessary for its checkpointing decision (i.e., $c_{1(i)}$, $c_{2(i)}$, and $c_{3(i)}$, see Table 2). The predictor for use must be fast to allow fine-grained prediction online (e.g., one prediction per second). Also, no profiling should be required for high usability and wide applicability. AIC predictor achieves these goals by means of **Stepwise regression** [13] and **Online prediction** [1], with lightweight metrics easily computed from gathered measures of taken checkpoints during code execution.

The essence of AIC prediction is to relate a target variable y to n predictor variables, defined as an n-dimensional vector, x. Given N candidates for predictor variables, for $N \geq n$, stepwise regression [13] selects which of them to include in the *linear* model. AIC starts by obtaining four samples to permit stepwise regression with up to three variables (i.e., $n = 3$) plus their initial weights for the prediction model. Its online prediction then adjusts the model weights for subsequent prediction use by adopting a *normalized* version of *Gradient Descent* algorithm [1].

Target Variables and Lightweight Metrics. As stated above, target variables of AIC predictions are $c_{1(i)}$, $c_{2(i)}$, and $c_{3(i)}$. Given delta latency, $dl_{(i)}$, and delta size, $ds_{(i)}$, at i^{th} checkpoint, $c_{2(i)}$ can be calculated by

$$c_{2(i)} = dl_{(i)} + ds_{(i)} / B_2 ,$$

where B_k is the bandwidth of level-k checkpoint. The second term is the time required for transmitting delta to RAID-5 group. Since delta compression has been done by L_2, level-3 checkpoint latency, $c_{3(i)}$ can be calculated by

$$c_{3(i)} = ds_{(i)} / B_3 .$$

B_k of each level is assumed to be known a priori. Hence, target variables for predictor model are $c_{1(i)}$, $dl_{(i)}$, and $ds_{(i)}$.

Metrics related to the AIC predictor are derived easily from gathered measures of taken checkpoints include the number of dirty pages (*DP*), elapsed time since the last local checkpoint ($t_{(i)}$), *Jaccard Distance* (*JD*) [9], and *Divergence Index* (*DI*) [1]. Stepwise regression chooses some of those four metrics plus their composites appropriately to form the desirable AIC predictor for a given code after its very first checkpoint. Precisely, with $\Phi = \{DP, t_{(i)}, JD, DI\}$, stepwise regression considers the candidate metrics in $\{C_1^{\gamma} C_2^{\zeta} \mid C_1, C_2 \in \Phi, 1 \leq \gamma + \zeta \leq 2\}$ for inclusion in the AIC predictor.

Jaccard Distance $JD(P, P')$ represents the degree of dissimilarity between a hot page P and its old version P' resided in the previous checkpoint, as

$$JD(P, P') = 1 - (m/p),$$

where p is the page size and m is the number of bytes in P whose values are equal to those located at corresponding addresses in P'. Next, Divergence Index $DI(P)$, which measures self-dissimilarity of a hot page P, is given by

$$DI(P) = 1 - (v/p),$$

where v is the number of occurrences of the most popular value in Page P, and p is the page size. In general, *JD* (or *DI*) measures the degree of inter-page (or intra-page) dissimilarity. The strength of these two metrics lies in their simplicity for fast calculation. Their values are also normalized, ranging from zero (totally identical) to one (totally different). To lower computation complexity, the AIC predictor calculates the mean of *JD* (or *DI*) for only **selected hot pages** (whose selection process is stated next).

E. Hot Page Selection

The arrival time of a hot page P is defined as the first time in the current checkpoint interval that a write to an address within P is made. AIC groups hot pages based on their arrival times, assigning two hot pages in different groups if their arrival times apart beyond a threshold T_g. To limit its space and time overhead, AIC buffers only the **first** hot page of each group in a fixed-size Sample Buffer (SB) for *JD* and *DI* computation. It also adjusts T_g in an attempt to hold as many samples as possible in SB at the decision time. This is achieved by doubling (or halving) T_g if SB is full (or more than half empty). Pages in SB are dropped accordingly if SB is full.

[1] We also examined other relevant metrics stated in the literature, like *Cosine Similarity* and the qualitative variation index *M2* [5]. As those metrics were found to be closely similar to *JD* and *DI* under our target applications, we adopted *JD* and *DI* due to their low computational costs, with the computation time of a hot page below 100 μs.

V. EXPERIMENTAL EVALUATION AND DISCUSSION

Experimental evaluation has been performed on our testbed using benchmark codes to assess the delta compressor and to compare AIC with both the non-adaptive counterpart and state-of-the-art multi-level checkpointing (i.e., Moody's model [11]). Based on experiment results, we have the subsequent findings:

- AIC enjoys better reduction in the normalized expected turnaround time (NET^2, by up to 47%) in comparison to its non-adaptive counterpart, while incurring negligibly low runtime overhead (by less than 2.6%),
- Concurrent checkpointing (either static or adaptive) with size reduction techniques yields substantial drops in NET^2 when compared to the earlier Moody model, and
- our page-aligned delta compressor (Xdelta3-PA) enables online checkpoint decision (as detailed in Section IV.C), observed to have similar compression performance (in terms of file size reduction and execution latency) as a conventional compressor (Xdelta3).

A. Experimental Testbed

Hardware and System Software. Our experiment testbed consists of a Dell PowerEdge R610, with two quad-core Xeon E5530 processors running at 2.4 GHz and having 8-MB shared cache. Running CentOS 5.5 64-bit with 2.6.18 kernel, the testbed contains 32 GB of physical memory (with the page size of 4096 bytes) and one 7200-RPM SATA disk.

Three different types of multi-level checkpointing software are installed for the experiment:

- Moody: modified BLCR which periodically does full checkpoints without delta compression. The checkpoint interval is calculated using the Moody multi-level checkpoint model [11].
- SIC (Static Incremental checkpointing with Compression): modified BLCR which periodically does incremental checkpointing, with delta compression performed between two successive checkpoints. The checkpoint interval is computed by our L_2L_3 concurrent multi-level checkpoint model presented in Section III.C.
- AIC: our adaptive checkpointing mechanism.

Both Moody and SIC require the average checkpoint latency beforehand to calculate their optimal checkpoint intervals, while AIC gathers its prediction information online using stepwise regression. The delta compressor and remote checkpointing under SIC and AIC are conducted on a separate idle core to avoid penalizing application execution. Hence, the sharing factor (defined in Section III.D), is always 1.

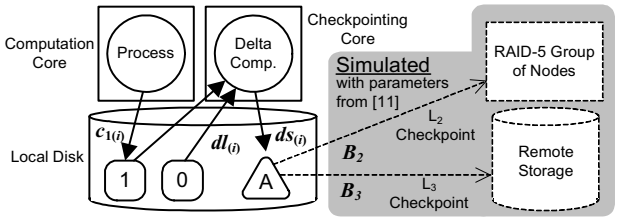


Fig. 10. Testbed setup composed of physical and simulated components, with simulated ones shown in the shaded area.

Applications and Setup. The target applications include six benchmarks from SPEC CPU2006 benchmark suites listed in Table 3. Each of them is the processor-memory intensive benchmark fitting in 1-GB memory. Those benchmarks are chosen as representative RMS applications. The terms of *application* and *benchmark* are used interchangeably.

SPEC provides a framework to run its benchmarks and to measure the results. We compiled SPEC applications with one of three checkpointing libraries (Moody, SIC, AIC) separately for comparison. During application execution, the following measures are gathered in each interval: L_1 checkpoint latency ($c_{1(i)}$), checkpoint size, delta latency ($dl_{(i)}$), and delta size ($ds_{(i)}$). The latencies for remote checkpoints (L_2 and L_3) are calculated from the checkpoint (or delta) size and predefined bandwidth amounts (B_2 and B_3). Fig. 10 shows AIC setup with its delta compressor and remote checkpointing handled by a separate core. While the incremental checkpoint file 1 and the delta file A are produced under AIC, the measures of $l_{(i)}$, $dl_{(i)}$, and $ds_{(i)}$ are gathered in the (physical) compute node. Being simulated components, L_2 and L_3 do not exist physically, permitting our evaluation on various system characteristics (similar to [12]). Like in Section III.D, our base system is the Coastal cluster [11] with 1024 nodes. Its L_2 bandwidth, B_2, equals 483 GB/s. The aggregate bandwidth of 1024 nodes for writing files to the Lustre distributed file system amounts to 2.1 GB/s. Hence, its L_3 bandwidth per node, B_3, is 2 MB/s. This reflects Lustre performance when there are 1024 nodes concurrently writing, regardless of application types (MPI, RMS, or Sequential programs).

B. Page-aligned Delta Compression

For comparison, we run SIC with Xdelta3 and Xdelta3-PA executed on the checkpointing core. The mean delta latency and file size results under both compression methods were calculated for each benchmark. Let the *mean compression ratio* of a benchmark refers to its average

TABLE 3. TARGET SPEC BENCHMARKS WITH THEIR PERFORMANCE METRICS

Benchmark	Base execution time, t (seconds)	Delta compressor performance				AIC Execution time (seconds)
		Compression Ratio		Delta Latency (seconds)		
		Xdelta3	*Xdelta3 -PA*	*Xdelta3*	*Xdelta3 -PA*	
Bzip2	152	0.63	0.66	1.7	0.9	156 (2.6%)
Sjeng	661	0.51	0.66	10.6	17.2	670 (1.4%)
Libquantum	846	0.65	0.51	1.5	3.0	853 (0.8%)
Milc	527	0.94	0.79	79.8	52.5	533 (1.1%)
Lbm	462	0.91	0.90	63.2	56.9	467 (1.1%)
Sphinx3	749	0.14	0.27	0.12	0.05	754 (0.7%)

Note: The base execution time refers to the application execution time without checkpointing or any failure. Numbers in parentheses denote percentages of execution time increases over base execution times.

compressed delta size to its mean uncompressed checkpoint size. It is desirable to **lower** the delta latency (for faster operations) and the compression ratio (for smaller file sizes) as much as possible. Table 3 shows compression performance metrics (i.e., compression ratios and delta computation latencies) of Xdelta3 and Xdelta3-PA for six benchmarks. As can be seen, the compression performance results of Xdelta3 and Xdelta3-PA are mostly close to each other for a given benchmark, yielding comparable file size reduction and delta computation times. Note that Xdelta3 *cannot support online checkpoint decision* necessary for AIC (as detailed in Section IV.C) and also that delta compression is carried out concurrently on a separate (dedicated) core without penalizing the regular job execution time.

C. AIC Results and Discussion

Experiments have been conducted to assess the performance of AIC, for comparison with its static counterpart (SIC) and the state-of-the-art multi-level checkpoint model (Moody). Our performance metric of interest is normalized expected turnaround time (NET^2, which denotes the expected turnaround time normalized by its base runtime). NET^2 value ranges from 1.0 to ∞ and is desirable to be held as close to 1.0 as possible.

AIC in our evaluation makes a checkpoint decision every second, with 8-MB Sample Buffer (SB). The delta compressor of Xdelta3-PA is adopted by AIC and SIC. Since our benchmarks are short-lived (with the longest base execution time of 846 seconds), we set unusually high failure rates to be able to collect experimental data (since otherwise, no failure is to happen in the course of job execution). In this experiment, we let the failure rate λ equal 1.0×10^{-3}. As wide swings in the delta latency and the size of target benchmark are observed (see Fig. 2), we expect that long running applications with such large swings would especially benefit from adaptive checkpointing. Each level-k failure rate, λ_k, is calculated in proportion to the rates under the Coastal system (i.e., 8.3%, 75%, and 1.67% for λ_1, λ_2, and λ_3, respectively [11]). Other parameters of the Coastal system are described in Section V.A. NET^2 outcomes of AIC and SIC are calculated by Eq. (1) given in Section III.E, while those of Moody is obtained from the code taken from a public source [12].

Fig. 11 compares NET^2 outcomes of AIC, SIC, and Moody under six target benchmarks, where the multi-level *concurrent* checkpoints (AIC and SIC) are seen to yield markedly better NET^2 values over those of Moody. The NET^2 reduction amount under AIC depends on applications, ranging from 8.5% (Sphinx3) to some 40% (Milc). In addition, NET^2 is always less under AIC than under SIC, and the gap is larger for applications with higher NET^2 (i.e., Milc and Lbm). Note that all AIC and SIC results presented in this section are with sharing factor (SF) = 1, (i.e., there is one dedicated core for concurrent checkpointing and delta compression).

Next, the effect of the system size is examined. As described in Section III.D, when the processes of a program involve limited communications, the system scale has little

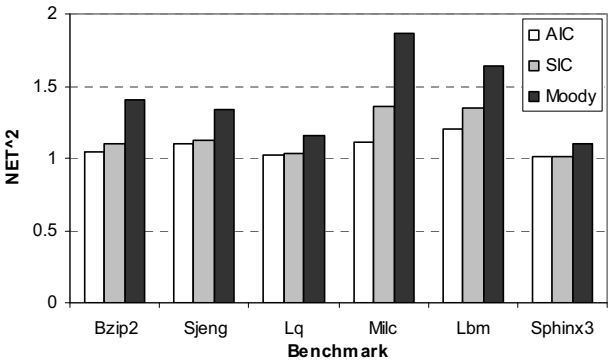


Fig. 11. NET^2 (normalized expected turnaround time) of six benchmarks under AIC, SIC, and Moody.

impact on the failure rate but has a proportional effect on the bandwidth per node available to remote storage (i.e., B_3). B_2 remains unchanged since it scales with the system size. As illustrated in Fig. 12, AIC and SIC performance results for the Milc benchmark are compared over the system scale of 0.25× to 4×, aiming to reveal the effect of *adaptive checkpointing* under growing and shrinking system sizes. Clearly, the NET^2 difference gap widens when the system grows (from 14% to 47%). While not shown here, Sphinx3 exhibits the least reduction for AIC versus SIC over same scaling, by less than 0.5% at 4×. This is due to the extremely small file size of Sphinx3 (in an order of half MB), not large enough to have a measurable benefit for delta compression and its adaptive companion. NET^2 reduction amounts under other benchmarks lie within the range between those of Milc and Sphinx3.

The benchmark execution times under AIC are listed in Table 3, revealing AIC execution time overhead to range from 0.7% to 2.6%. Note that this is the pure overhead when there is no failure present, mostly due to the AIC Predictor and Checkpoint Decider. When failures do occur in practice, the overhead time is included in NET^2 results demonstrated in Figs. 11 and 12. AIC indeed reduces the expected turnaround time, when compared to its static counterpart and best known (Moody) multi-level checkpointing [11].

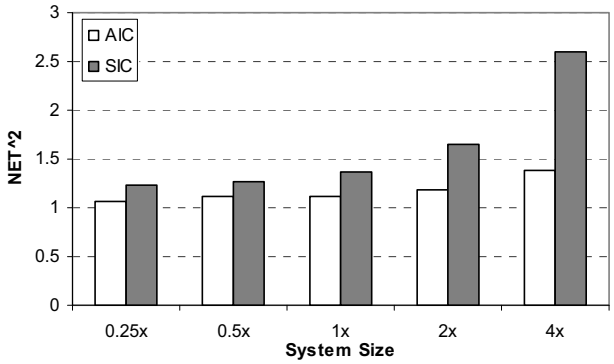


Fig. 12. NET^2 of Milc under adaptive and static concurrent checkpoint scheme.

VI. Prior Work

In past decades, several techniques for checkpointing have been pursued, aiming to reduce execution overhead and increase system availability and reliability. Meanwhile, on-line delta compression has been treated as well. Brief reviews on previous checkpointing and delta compression work are provided below in sequence.

A. Checkpointing

A Plethora of work has been on finding the optimal periodic checkpoint interval where checkpoint latency is known a priori [4, 24], calling for profiling before actual job execution. AIC, however, requires *no profiling* beforehand.

Based on simulation results, checkpointing which dynamically ignores application-initiated checkpoints was introduced [14]. On the other hand, AIC dynamism is resulted from prediction of the delta file size, able to uncover better checkpointing times through finer granularity (i.e., every second). A variety of adaptive single-level checkpointing work based on the dynamic checkpoint cost was considered previously [23, 25], whereas AIC addresses two-level checkpointing with its cost prediction functions including such key parameters as delta compression overheads and network traffic. Unlike [25], AIC does not assume that the program state changes according to a stochastic process. Multi-level checkpointing [11, 21] involves multiple checkpoint types for tolerating different failure categories. Being a two-level checkpointing mechanism, AIC employs fast prediction to realize online checkpoint decision during job execution, in contrast to off-line decision common to prior multi-level checkpointing. Hence, AIC soundly outperforms its state-of-the-art multi-level checkpointing counterpart [11] in terms of NET^2.

Effort was attempted earlier to reduce overhead involved in incremental checkpointing [6, 16]. The model for concurrent (or forked) checkpointing (which lets the process continue its execution upon writing the checkpoint file to the local disk [16]) is presented in [22]. By contrast, our model supports multi-level checkpointing and adopts concurrent checkpointing only for the remote levels (to avoid interfering with computation processes).

B. Delta Compression

Delta compression has been considered to reduce the checkpoint cost regarding local disk accesses [17, 19], favoring simple algorithms (like the XOR [19]) to contain its overhead. As AIC tracks dissimilarity in the working sets to identify the desirable point with a low compression overhead, it can afford more aggressive compression [10] for better compressed results. Unlike an earlier in-memory compression approach which buffers contents of *every* modified page fully [19], our AIC buffers only a small number of selected pages for its, involving far less space overhead to permit large applications.

Various delta compressors are devised for version control, software updates, and remote file synchronization [20]. Among delta compressors available in the public domain, *Xdelta3* [10] is adopted by AIC because it has the lowest latency while yielding quality compression results.

VII. Conclusion

This article introduces AIC (adaptive incremental checkpointing), a two-level concurrent checkpointing mechanism with delta compression for networked multicore systems. According to our study on production system's logs, idle cores often exist for checkpointing use in realizing AIC. We develop a new multi-level concurrent checkpoint model, which shows performance improvement when compared with recent multi-level checkpointing under various circumstances (i.e., system sizes, application types, sharing factor). The developed model serves as the basis for AIC, which further reduces overhead by means of adaptive checkpointing with delta compression. In support of fast online prediction without profiling (for predictor establishment), AIC significantly reduces checkpoint sizes with only negligible increases in benchmark code execution times (ranging from 0.7% to 2.6% in comparison to those without checkpointing at all), according to evaluation results collected on a real testbed. This leads to lower application expected turnaround time (by up to 47%), when compared to its static counterpart with fixed checkpointing intervals. AIC thus enjoys better execution performance.

Acknowledgments

The authors would like to thank anonymous reviewers for their insightful comments. They also thank Adam Lewis for his discussion on Linear regression. This work was supported in part by the U.S. National Science Foundation under Aware Number: CCF-0916451.

References

[1] N. Cesa-Bianchi, P. M. Long, and M. K. Warmuth, "Worst-case quadratic loss bounds for prediction using linear functions and gradient descent," *IEEE Trans. on Neural Networks*, vol. 7, no. 3, pp. 604 - 619, May 1996.

[2] Y.-K. Chen et al., "Convergence of recognition, mining, and synthesis workloads and its implications," *Proc. of IEEE*, vol. 96, no. 5, pp. 790–807, May 2008.

[3] C. Constantinescu, "Impact of deep submicron technology on dependability of VLSI circuits," *Proc. Int'l Conf. on Dependable Systems and Networks* (*DSN*), pp. 205-209, June 2002.

[4] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Fut. Gen. Comput. Syst*., vol. 22, pp. 303-312, 2006.

[5] J. P. Gibbs and D. L. Poston Jr., "The division of labor: conceptualization and related measures," *Social Forces*, vol. 53, no. 3, pp. 468-476, March 1975.

[6] R. Gioiosa, J. C. Sancho, S. Jiang and F. Petrini, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," *Proc. of the IEEE/ACM Int'l Conf. on High Performance Computing, Networking, Storage and Analysis* (*SC*), pp. 9-23, November 2005.

[7] I. Goiri, F. Julià, J. Guitart, and J. Torres, "Checkpoint-based fault-tolerant infrastructure for virtualized service providers," *Proc. of IEEE Network Operations and Management Symp.* (*NOMS*), pp.455-462, April 2010.

[8] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," *J. Physics: Conf. Series*, vol. 46, pp. 494-499, 2006.

[9] P. Jaccard, "Étude comparative de la distribution florale dans une portion des Alpes et des Jura," *Bull. Soc. Vaudoise Sci. Nat.*, vol. 37, pp. 547-579, 1901.

[10] J. MacDonald, "File system support for delta compression," M.S. thesis, Univ. California, Berkeley, May 2000.

[11] A. Moody, G. Bronevetsky, K. Mohror, and B.R. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," *Proc. of the IEEE/ACM Int'l Conf. on High Performance Computing, Networking, Storage and Analysis* (*SC*), pp. 1-11, November 2010.

[12] A. Moody, G. Bronevetsky, K. Mohror, and B.R. de Supinski, "Detailed modeling, design, and evaluation of a scalable multi-level checkpointing system," Lawrence Livermore National Laboratory, Livermore, CA, Tech. Rep. LLNL-TR-440491, July 2010.

[13] W. E. Navidi, Principles of Statistic for Engineers and Scientists. Columbus, OH: McGraw Hill, 2010.

[14] A. J. Oliner, L. Rudolph, and R. K. Sahoo, "Cooperative checkpointing: a robust approach to large-scale systems reliability," *Proc. of the 20th Annual Int'l Conf. on Supercomputing* (*ICS*), pp. 14-23, June 2006.

[15] Operational data to support and enable computer science research [Online]. Available: http://institutes.lanl.gov/data/fdata/

[16] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: transparent checkpointing under Unix," *Proc. of Usenix Annual Technical Conf.*, pp. 213-224, January 1995.

[17] J. S. Plank and K. Li, "Ickp: a consistent checkpointer for multicomputers," *IEEE Parallel & Distributed Technology*, vol. 2, pp. 62-67, Summer 1994.

[18] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, pp. 972-986, October 1998.

[19] J. S. Plank, J. Xu, and R. H. B. Netzer, "Compressed differences: an algorithm for fast incremental checkpointing," Univ. of Tennessee, Tech. Rep. CS-95-302, August 1995.

[20] A. Tridgell, "Efficient algorithms for sorting and synchronization," Ph.D. Dissertation, Australian National Univ., Canberra, 2000.

[21] N. H. Vaidya, "A case for two-level recovery schemes," *IEEE Trans. Computers*, vol. 47, pp. 656-666, June 1998.

[22] N. H. Vaidya, "Impact of checkpoint latency on overhead ratio of a checkpointing scheme," *IEEE Trans. Computers*, vol. 46, no. 8, pp. 942-947, August 1997.

[23] S. Yi, J. Heo, Y. Cho, and J. Hong, "Adaptive page-level incremental checkpointing based on expected recovery time," *Proc. of the ACM Symp. on Applied Computing* (*SAC*), pp. 1472-1476, April 2006.

[24] J. Young, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, vol. 17, pp. 530–531, 1974.

[25] A. Ziv and J. Bruck, "An on-line algorithm for checkpoint placement," *IEEE Trans. Computers*, vol. 46, no. 9, pp. 976-985, September 1997.