Optimal Cooperative Checkpointing for Shared High-Performance Computing Platforms

Thomas Herault*, Yves Robert†*, Aurelien Bouteiller*, Dorian Arnold‡,
Kurt B. Ferreira§, George Bosilca*, Jack Dongarra*,¶
*Innovative Computing Lab. The University of Tennessee, Knoxville, TN, USA
†ENS Lyon, Lyon, France
‡Emory University, Atlanta, GA, USA
§Center for Computing Research, Sandia National Laboratories , USA
¶University of Manchester, UK

***Abstract*—In high-performance computing environments, input/output (I/O) from various sources often contend for scarce available bandwidth. Adding to the I/O operations inherent to the failure-free execution of an application, I/O from checkpoint/restart (CR) operations (used to ensure progress in the presence of failures) place an additional burden as it increase I/O contention, leading to degraded performance. In this work, we consider a cooperative scheduling policy that optimizes the overall performance of concurrently executing CR-based applications which share valuable I/O resources. First, we provide a theoretical model and then derive a set of necessary constraints needed to minimize the global *waste* on the platform. Our results demonstrate that the optimal checkpoint interval, as defined by Young/Daly, despite providing a sensible metric for a single application, is not sufficient to optimally address resource contention at the platform scale. We therefore show that combining optimal checkpointing periods with I/O scheduling strategies can provide a significant improvement on the overall application performance, thereby maximizing platform throughput. Overall, these results provide critical analysis and direct guidance on checkpointing large-scale workloads in the presence of competing I/O while minimizing the impact on application performance.**

I. Introduction

Space-sharing high-performance computing (HPC) platforms for the concurrent execution of multiple parallel applications is the prevalent usage pattern in today's HPC centers. In fact, space-sharing in this fashion is more common than *capability* workloads that span the entire platform [1]. Furthermore, while computational nodes are dedicated to a particular application instance, the interconnect links and storage partition are typically shared amongst application instances. Therefore, without careful consideration, network and storage contention can reduce individual application and overall system performance [2]. On these platforms, checkpoint/restart (CR) is the most common strategy employed to protect applications from underlying faults and failures. Generally, CR periodically outputs snapshots (*i.e.* checkpoints) of the application's global, distributed state to some stable storage device. When an application failure occurs, the last stored checkpoint is retrieved and used to restart it. Typically, concurrently executing applications independently decide when to take their own checkpoints.

There are two widely-used approaches to determine when an application should *commit* a checkpoint: (i) using a fixed checkpoint period (typically one or a few hours) for each application; and (ii) using platform and application-specific metrics to determine its optimal checkpoint period. In the second approach, the well-known Young/Daly formula [3], [4] yields an application optimal checkpoint period, $\sqrt{2\mu C}$ seconds, where C is the time to commit a checkpoint and μ the Mean Time Between Failures (MTBF) for the application: $\mu = \frac{\mu_{\text{ind}}}{q}$, where q is the number of processors enrolled by the application and μ_{ind} is the MTBF of an individual processor [5]. Both μ and C in the Young/Daly formula are application-dependent, and optimal periods can be quite different over the application spectrum.

Independent CR of concurrent application instances can incur significant resource wastage, because they lead to an inefficient usage of an already scarce resource, namely available I/O bandwidth [6]. There are two major reasons for this:

• *Application-CR I/O contention*: On many systems, the I/O subsystem does not have enough available usable bandwidth to meet the requirements of the concurrent application workloads [6]. This congestion is expected to worsen going forward with the increased importance of data intensive workflows in HPC. Let β_{tot} be the total filesystem I/O bandwidth. Concurrently executing applications typically perform regular (non-CR) I/O operations throughout their execution, so that only a fraction β_{avail} of the total bandwidth remains available for checkpoints. This fraction may be insufficient, particularly when some applications perform intensive non-checkpoint I/O and others may write very large checkpoints.

• *CR-CR I/O contention*: Most importantly, there is a high probability of overlapping CR activity amongst concurrent application instances. Consider the simple case where two applications of same size checkpoint simultaneously a file of the same size. Each will be assigned half the fraction β_{avail} to checkpoint, therefore the commits will take twice as long. Such interferences can severely decrease application efficiency and overall platform throughput: when the expected checkpoint commit time used to compute the optimal checkpoint interval differs from the actual checkpoint commit time, efficiency

Sandia National Laboratories is a multi mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525

will decrease. This is consistent with practical observations of interference conducted on various HPC systems [7], [8] and is confirmed by the experiments reported in Section VI.

In this work, we develop and investigate a cooperative CR scheduling strategy for concurrently executing HPC applications. Our objective is to assess the impact of such interferences in the overall platform efficiency, and to design scheduling algorithms that optimize I/O bandwidth availability for CR activity. Using these cooperative algorithms, applications never checkpoint concurrently but always in sequence, with a dynamic, priority-dependent frequency dictated by a cooperative scheduler. It may be counterintuitive to give an I/O token to each application, because one could expect that the aggregated I/O bandwidth provided by the system is always sufficient to allow for several applications to checkpoint concurrently. However, concurrent checkpoints always incur interferences and delays, and our simulations show that these interferences have a tremendous impact on performance in many realistic scenarios. On the contrary, our cooperative scheme eliminates all interferences. There are two cases; (i) When enough I/O bandwidth is available, each application can checkpoint with its optimal, Young/Daly, period. In this case, scheduling applications to checkpoint in sequence is enough to provide an optimal I/O strategy; (ii) When I/O bandwidth is scarce, it is no longer possible to checkpoint each application with its optimal, Young/Daly, period. In this case, our scheduling algorithm provides an optimal checkpoint period that maximizes overall platform throughput. This cooperative checkpoint process is calculated such that there is no I/O interference and minimal re-work to be done when failures occur.

The main contributions of this paper are the following: (i) Development of a model allowing for the quantification of the I/O interference of checkpointing applications sharing a common underlying I/O substrate; (ii) Investigation of the costs of various I/O-aware scheduling strategies through both steady-state analysis as well as detailed simulations; (iii) Survey of a number scheduling strategies: from oblivious algorithms similar to those currently deployed on many large-scale platforms, to ones which exploit application knowledge in an effort to minimize the total system waste by scheduling the application with the most critical I/O needs; and (iv) Extensive set of simulations that assess the dramatic impact of checkpoint interference and demonstrate the usefulness of cooperative strategies for current and forthcoming HPC systems.

The rest of the paper is organized as follows. Our model is described in Section II, followed by a description of the various scheduling strategies in Section III. Section IV presents a theoretical analysis of the model under a steady-state scenario, and provides a lower bound of the optimal platform waste. Section V describes the discrete event simulator used to quantitatively compare the different scheduling strategies. Section VI presents the results of the simulation, providing guidance on the necessary I/O bandwidth for current and future systems. Section VII surveys related work and is followed by a summary and future directions outlined in Section VIII.

II. Model

Computational Platform Model: We consider a shared platform comprised of computational nodes, storage resources in the form of a parallel file system (PFS), and a network that interconnects the nodes and storage resources. Applications are scheduled on the platform by a job scheduler such that computational nodes are space-shared (dedicated) amongst concurrent application instances. The I/O subsystem is time-shared (contended) amongst application instances (*i.e.* multiple applications performing I/O simultaneously result in a per-application reduction in commit speed). Without loss of generality, we consider a linear interference model in which the global throughput remains constant and is evenly shared among contending applications, proportional to their size; a more adversarial interference model can be substituted, if needed.

Application Workload Model: Applications can vary in size (computational node count), duration, memory footprint and I/O requirements. *Application I/O* entails loading an input file at startup, performing regular I/O operations during their main execution phase and storing an output file at completion. Because applications are long-running, (typically, several hours or days) and the platform is failure-prone, applications are protected using coordinated CR that incurs periodic *CR I/O*.

To model these behavioral variations with minimal parameters, we make the following simplifying assumptions: (i) There is a large number of applications, but only a small number of application classes, *i.e.*, sets of applications with similar sizes, durations, footprints and I/O needs; (ii) Excluding initialization and finalization I/O, an application's regular (non-CR) I/O operations are evenly distributed over its makespan; and (iii) Job makespans are known a priori. This allows us to ignore all other sources of job disturbance except C/R overheads.

We use specific numbers and characteristics of application classes based on documented production workloads, such as those provided in the APEX workflows report on the Cielo platform [9]. To avoid the side effects induced by hundreds of completely identical jobs, we use a normal distributions for job durations with a mean equal to original APEX value and small (20%) standard deviation. In the rest of the paper, we use the term *job* to denote a specific application instance, and *application class* to denote a set of applications with similar characteristics.

Checkpoint Period and I/O Interference: Both application computation and CR generate I/O requests. In both cases, activity is scheduled using an I/O scheduling algorithm (see Section III). As described above, steady-state application I/O is regular. However, CR I/O periodicity, P, depends upon the CR policy being used. In our model, applications either checkpoint using an application-defined periodicity or using Young and Daly's [3], [4] optimal checkpoint period detailed in Section I. The parameters in this formula are dependent upon application features (checkpoint dataset size) and platform features (system reliability and I/O bandwidth). For fixed, application-defined periods, a common heuristic in compute centers is to take a checkpoint every hour – capping the worst case amount of lost work at one hour. In the reminder of this paper we will refer to the two variants as *Fixed* (with a 1 hour period unless otherwise specified) and *Daly*.

Traditionally, when a job J_i of class A_i completes a checkpoint, its next checkpoint is scheduled to happen $P_i - C_i$ instants later (and the first checkpoint is set at date P_i). With potential CR I/O interference, the checkpoint commit may last longer than C_i, and setting the appropriate checkpointing period can be challenging. Additionally, I/O scheduling algorithms that try to mitigate I/O interference can impose further CR I/O delays. In other words, the traditional strategy of scheduling subsequent checkpoints at $P_i - C_i$ yields the desired checkpointing period P_i only in interference-free scenarios. CR I/O delays (induced by interferences or scheduling delays) dilate the checkpoint duration to $C_{dilated}$, and the effective period differs from the desired period by the difference $C_{dilated} - C_i$. Section III discusses how each I/O scheduling algorithm handles this discrepancy.

Job Scheduling Model: To evaluate the scheduling policies, we consider a finite segment, typically lasting a few days, of a representative schedule where the computing resource usage by each application instance (job) in each class remains nearly constant. Of course, with varying job execution times, we cannot enforce a fixed proportion of each application class at every instant. However, we ensure the proper proportion is enforced on average throughout the schedule execution. Similarly, we enforce that at every instant during the finite segment, at least 98% of the nodes are enrolled for the execution. This allows us to compare actual (simulated) performance with the theoretical performance of a co-scheduling policy that optimizes the steady-state I/O behavior of the job portfolio, assuming that all processors are used. We shuffle and simultaneously present all jobs to the scheduler, which uses a simple, greedy first-fit algorithm. We resubmit failed jobs with a new wall-time equal to the fraction that remained when the last checkpoint commit started. In this case, input I/O becomes recovery I/O; output I/O is unmodified.

The Formal Model: We consider a set $\mathcal{A}$ of $|\mathcal{A}|$ applications classes $A_1, \dots A_{|\mathcal{A}|}$ that execute concurrently on a platform with $\mathcal{N}$ nodes. Application class A_i specifies: (i) n_i: the number of jobs in A_i; (ii) q_i: the number of nodes used by each job in A_i; (iii) P_i: the checkpoint period of each job in A_i; and (iv) C_i and R_i: the checkpoint and recovery durations for each job in A_i when there is no interference with other I/O operations. Jobs inherit their characteristics from their classes. For a job J_j, we use q_j, P_j, C_j and R_j to denote respectfully the number of nodes, checkpoint period, and checkpoint and recovery durations of the application class to which J_j belongs. We let $P_{Daly}(J_j) = \sqrt{2C_j\mu_j}$ be the *Daly period* [3], [4] of a job J_j, where $\mu_j = \frac{\mu_{\text{ind}}}{q_j}$ and μ_{ind} is the MTBF of an individual processor [5]. At each instance, we schedule as many jobs as possible. Jobs that are subject to failures are restarted at the head of the scheduling queue, as to restart immediately on the same compute nodes previously used (in most cases, only one node has failed and is replaced by a hot spare).

III. I/O Scheduling Algorithms

In this section, we present the application I/O scheduling algorithms used in this study. The first algorithm, *Oblivious*, represents the status-quo in which I/O activities are scheduled independently and may incur slowdowns due to I/O contention. The second algorithm, *Ordered*, coordinates I/O activity to eliminate interference: I/O operations are scheduled in a First-Come-First-Serve (FCFS) fashion and only one I/O operation executes at any given time, while other I/O requests are blocked until their turn comes. The third algorithm, *Ordered-NB*, is similar except that jobs that are waiting for the I/O token to checkpoint continue working until their turn comes. Lastly, our new heuristic, *Least-Waste* improves on *Ordered-NB* by giving the I/O token to the I/O operation that will minimize system waste. Note that unlike the blocking approaches (*Oblivious* and *Ordered*), non-blocking optimizations (*Ordered-NB* and *Least-Waste*) may require application code refactoring.

A. Oblivious *I/O Scheduling*

In *Oblivious* I/O scheduling, jobs are executed to fill-up the system based on processor availability, and their I/O workload (including CR activities) are not coordinated by the system. Instead, jobs use the parallel file system assuming they are the sole user – with no modifications made to their access patterns to accommodate for possible interference. One has observed that concurrent I/O resource access can decrease the I/O bandwidth observed by applications [10]. Under the conditions of an under-provisioned I/O substrate, our model gives each I/O stream a decrease in bandwidth linearly proportional to the number of competing operations. We account for the additional delays imposed by this decreased available bandwidth as *waste*. Since subsequent checkpoints are scheduled to start after $P_i - C_i$, and delays may result in checkpoint commit times longer than C_i, the resultant checkpoint period may be longer than P_i. This is consistent with a trivial I/O policy that does not consider potential contention.

B. *Blocking* Ordered *FCFS I/O Scheduling*

A simple optimization to the *Oblivious* scheme is to favor one jobs' I/O over all others. While the overall throughput may remain unchanged (given an efficient filesystem implementation), the favored job completes its I/O workload faster (*i.e.*, in time C_i for a job of class A_i). In the *Ordered* scheme, I/O requests are performed sequentially, in request arrival order. Jobs with outstanding I/O requests are blocked until their requests are completed. Assuming a favorable linear interference model, a simple workload with two jobs can show the potential advantage of the *Ordered* over *Oblivious* strategy. If the two jobs simultaneously request I/O transfers of similar data volume, V, in the *Oblivious* strategy, both jobs take $\frac{V}{\frac{\beta_{\text{avail}}}{2}}$ time to complete their I/O. In the *Ordered* strategy, the first scheduled job takes only $\frac{V}{\beta_{\text{avail}}}$, while the second job waits $\frac{V}{\beta_{\text{avail}}}$ before its own I/O starts, but then executes at full available bandwidth completing in $\frac{2V}{\beta_{\text{avail}}}$. Thus, reducing I/O interference reduces the average I/O completion time (although fairness may be decreased). Once again, however, observed checkpoint durations may increase past C_i, due to I/O scheduling wait time, and the checkpointing period may be, on average, larger than the desired P_i.

C. *Non-Blocking* Ordered-NB *FCFS I/O Scheduling*

The previous strategy trades the cost of I/O interferences for idle time, as jobs perform a blocking (idle) wait for the I/O token. If the application developer can refactor the program code to continue computing while awaiting I/O request completions, it becomes possible to replace otherwise idle wait time with useful computation. In the *Ordered-NB* algorithm, when the previous checkpoint ends at time t_{now}, a tentative time for the next checkpoint is set at $t_{req} = t_{now} + P_i - C_i$. At time t_{req}, a non-blocking I/O request is made to request the I/O token – the I/O token is still scheduled FCFS according to request arrival time. The job continues its computation until the scheduler informs it that the I/O token is available. At this point, the job must generate its checkpoint data as soon as possible (or after a short synchronization[1]). In most applications, the granularity of the work is small enough for a simple approach to be efficient: applications can use existing APIs in SCR [11] or FTI [12] to regularly poll if a checkpoint should be taken at this time. In this work, we assume that this re-synchronization cost is negligible relative to the checkpoint commit duration. Postponing checkpoint I/O increases a job's exposure to failures. However, if the job successfully commits the postponed checkpoint, upon a subsequent failure, the job would restart from the time at which the postponed checkpoint was taken, not at t_{req} – a fact that may mitigate the increased risk exposure when compared to *Ordered* and *Oblivious* algorithms.

D. *Variants*

The periods P_i of the checkpointing requests are input parameters to the three strategies *Oblivious*, *Ordered* and *Ordered-NB*. In Section V, we instantiate each strategy with two variants: (i) using a fixed checkpointing period for each job; and (ii) using the Young/Daly period of each job.

E. Least-Waste *Algorithm*

Finally, our *Least-Waste* algorithm further refines the *Ordered-NB* algorithm by issuing the I/O token to the job whose I/O request minimizes the total expected waste (explained hereafter), rather than simply based on request arrival order. Given the time-dependent nature of this decision, the selection may not be a global optimum, but only an approximation given currently available information about the system status. The *Least-Waste* algorithm assumes that jobs issue checkpointing requests according to their Daly period[2]. For each I/O scheduling decision, at time t (when a previous I/O operation completes), we consider a pool of $r + s$ candidates from two different categories:

- Category IO-CANDIDATE $\mathcal{C}_{IO}$: Jobs J_i, $1 \leq i \leq r$ with an (input, output or recovery) I/O request of length v_i seconds and enrolls q_i processors. J_i initiated its I/O request d_i seconds ago and has been idle for d_i seconds.
- Category CKPT-CANDIDATE $\mathcal{C}_{Ckpt}$: Jobs J_i, $r + 1 \leq i \leq r + s$, with a checkpoint duration of C_i seconds and enrolls q_i processors. J_i took its last checkpoint d_i seconds ago and keeps executing until the I/O token is available for a new checkpoint. Since J_i is a candidate, $d_i \geq P_{Daly}(J_i)$.

If we select job J_i to perform I/O, the expected waste W_i incurred to the other $r + s - 1$ candidate jobs in $\mathcal{C}_{IO} \cup \mathcal{C}_{Ckpt}$ is computed as follows. Assume first that $J_i \in \mathcal{C}_{IO}$. Then J_i will use the I/O resource for v_i seconds:

- Every other job $J_j \in \mathcal{C}_{IO}$ stays idle for v_i additional seconds, hence its waste $W_i(j)$ is $W_i(j) = q_j(d_j + v_i)$ since there are q_j processors enrolled in J_j that remain idle for $d_j + v_i$ seconds. For $J_j \in \mathcal{C}_{IO}$, the waste $W_i(j)$ is deterministic.
- Every job $J_j \in \mathcal{C}_{Ckpt}$ continues executing for v_i additional seconds, hence will be exposed to the risk of a failure with probability v_i/μ_j, where $\mu_j = \mu_{\text{ind}}/q_j$. The cost of such a failure will be $v_i/2$ seconds on average. Thus, overall, the q_j processors will have to recover and re-execute $d_j + v_i/2$ seconds of work, hence we have $W_i(j) = \frac{v_i}{\mu_j} q_j(R_j + d_j + \frac{v_i}{2}) = \frac{v_i}{\mu_{\text{ind}}} q_j^2(R_j + d_j + \frac{v_i}{2})$, where R_j is the recovery time for J_j. For $J_j \in \mathcal{C}_{Ckpt}$, the waste $W_i(j)$ is probabilistic.

Altogether, the expected waste W_i incurred to the other $r + s - 1$ candidate jobs is $W_i = \sum_{J_j \in \mathcal{C}_{IO}, j \neq i} W_i(j) + \sum_{J_j \in \mathcal{C}_{Ckpt}} W_i(j)$. We obtain

$$W_i = v_i \times \left(\sum_{1 \leq j \leq r, j \neq i} q_j(d_j + v_i) + \sum_{r+1 \leq j \leq r+s} \frac{q_j^2}{\mu_{\text{ind}}}(R_j + d_j + \frac{v_i}{2}) \right) \quad (1)$$

Assume now that the selected job $J_i \in \mathcal{C}_{Ckpt}$. Then J_i will use the I/O resource for C_i seconds instead of v_i seconds for $J_i \in \mathcal{C}_{IO}$. We directly obtain the counterpart of Equation (1) for its waste W_i:

$$W_i = C_i \times \left(\sum_{1 \leq j \leq r} q_j(d_j + C_i) + \sum_{r+1 \leq j \leq r+s, j \neq i} \frac{q_j^2}{\mu_{\text{ind}}}(R_j + d_j + \frac{C_i}{2}) \right) \quad (2)$$

Finally, we select the job $J_i \in \mathcal{C}_{IO} \cup \mathcal{C}_{Ckpt}$ whose waste W_i is minimal.

F. *Feasibility of Cooperative Strategies*

The cooperative strategies (*Ordered*, *Ordered-NB*, and *Least-Waste*) require a form of synchronization to be implemented. This synchronization can happen at the filesystem level for *Ordered*: metadata servers in the filesystem can select which I/O stream is given the priority, and in the extreme case, give access to the I/O storage nodes only to a given application (*e.g.* by using the technology proposed in CALCioM, [10]); however, *Ordered-NB* and *Least-Waste* cannot be implemented without modifying the applications, as work must continue until the access is granted. Operating and runtime systems should provide these strategies at the level of the checkpoint library (*e.g.* inside SCR or FTI). These libraries already provide APIs for the applications to get informed when a checkpoint is desirable, and applications that use these libraries regularly poll the system to decide if a checkpoint should be started.

[1]In user-level checkpointing, the job typically finishes its current computing block before generating its checkpoint data.

[2]Fixed checkpointing makes little sense in the *Least-Waste* strategy, it is designed to optimize checkpoint frequencies across all jobs.

Moreover, checkpointing libraries try to take advantage of the memory hierarchy to checkpoint first the process memory on unreliable (but fast) media, and then to upload the checkpoints in the background, while the application proceeds to compute. As the I/O Interference scheduling strategies rely on knowing when a checkpoint is started and when it is complete, implementing that strategy at the checkpointing library level is thus the natural place.

IV. Lower Bound

We now derive a lower bound for optimal platform waste. When we assess the performance of the scheduling algorithms presented in Section III, we also compare their relative performance to this lower bound (in Section VI). We envision a (theoretical) scenario in which the platform operates in steady-state, a constant number of jobs per application class spanning the entire platform. We also assume that the I/O bandwidth β_{avail} available for CR operations remains constant throughout execution. This amounts to ignoring initial input and final output I/O operations, or more precisely, to assuming these operations span the entire execution of the jobs. Without this assumption, we would need to account for job durations; this renders the steady-state analysis intractable. We determine the optimal checkpointing period for each application class with the objective of minimizing the total waste of the platform; or equivalently, of maximizing the total throughput. To complicate this analysis, these optimal periods may not be achievable, hence we derive a lower bound of the optimal waste.

In steady-state operation, there are n_i jobs of class A_i, each using q_i nodes, and with checkpoint time C_i. Because we orchestrate checkpoints to avoid CR-CR interferences, we have $C_i = \frac{size_i}{\beta_{\text{avail}}}$, where $size_i$ denote the size of the checkpoint file of all jobs of class A_i. The waste of a job is the ratio of time the job spends doing resilience operations by the time it does useful work. The time spent performing resilience operations include the time spent during each period to checkpoint; and in case of failure, the time to rollback to the previous checkpoint and the time to recompute lost work. We express the waste W_i of a job J_i of class A_i that checkpoints with period P_i as [5]:

$$W_i = W_i(P_i) = \frac{C_i}{P_i} + \frac{q_i}{\mu}(\frac{P_i}{2} + R_i) \tag{3}$$

Let W be the waste of the platform, defined as the weighted arithmetic mean of the W_i for all applications, where each application is weighted by its number of computing nodes:

$$W = \sum_i \frac{n_i q_i}{\mathcal{N}} W_i \tag{4}$$

In the absence of I/O constraints, the checkpointing period can be minimized for each job independently. Indeed, the optimal period for a job of class A_i is obtained by minimizing W_i in Equation (3). Differentiating and solving $\frac{\delta W_i}{\delta P_i} = -\frac{C_i}{P_i^2} + \frac{q_i}{2\mu} = 0$, we readily derive that

$$P_i = \sqrt{2\frac{\mu}{q_i}C_i} = \sqrt{2\mu_i C_i} \tag{5}$$

where μ_i is the MTBF of class A_i applications, and we retrieve the Daly period $P_i = P_{Daly}(J_i)$.

I/O constraints may impose the use of sub-optimal periods. If each job of class A_i checkpoints in time C_i during its period P_i (hence without any contention), it uses the I/O device during a fraction $\frac{C_i}{P_i}$ of the time. The total usage fraction of the I/O device is $F = \sum_i \frac{n_i C_i}{P_i}$ and cannot exceed 1. We have to solve the following optimization problem: find the set of values P_i that minimize W in Equation (4) subject to the I/O constraint:

$$F = \sum_i \frac{n_i C_i}{P_i} \leq 1 \tag{6}$$

Hence the optimization problem is to minimize:

$$W = \sum_i \frac{n_i q_i}{\mathcal{N}} \left(\frac{C_i}{P_i} + \frac{q_i}{\mu}(\frac{P_i}{2} + R_i)\right) \tag{7}$$

subject to Equation (6). Using the Karush-Kuhn-Tucker conditions [13], we know that there exists a nonnegative constant λ such that $-\frac{\delta W}{\delta P_i} = \lambda \frac{\delta F}{\delta P_i}$ for all i. We derive that $\frac{n_i q_i C_i}{\mathcal{N} P_i^2} - \frac{n_i q_i^2}{2\mu\mathcal{N}} = -\lambda \frac{n_i C_i}{P_i^2}$ for all i. This leads to:

$$P_i = \sqrt{\frac{2\mu\mathcal{N}}{q_i^2}\left(\frac{q_i}{\mathcal{N}} + \lambda\right) C_i} \tag{8}$$

for all i. Note that when $\lambda = 0$, Equation (8) reduces to Equation (5). Because of the I/O constraint in Equation (6), we choose for λ the minimum value such that Equation (6) is satisfied. If $\lambda \neq 0$, this will lead to periods P_i larger than the optimal value of Equation (5). Note that there is no closed-form expression for the minimum value of λ, it has to be found numerically. Altogether, we state our main result:

Theorem 1. *In the presence of I/O constraints, the optimal checkpoint periods are given by Equation* (8)*, where* λ *is the smallest non-negative value such that Equation* (6) *holds. The total platform waste is then given by Equation* (7)*.*

The optimal periods may not be achievable, because Equation (6) is a necessary, but not sufficient condition. Even though the total I/O bandwidth is not exceeded, meaning there is enough capacity to take all the checkpoints at the given periods, we would still need to orchestrate these checkpoints into an appropriate, periodic, repeating pattern. In other words, we only have a lower bound of the optimal platform waste.

V. Simulation Framework

We use discrete event simulations to evaluate the performance of the proposed approaches. The simulator is publicly available from *https://github.com/SMURFSorg/InterferingCheckpoints*. Simulations are instantiated by a set of initial conditions that define a set of application classes, the distribution of resource usage between application classes, and the main characteristics of the platform on which application instances will execute.

High level parameters: Application classes are characterized by: initial input and output sizes, checkpoint size, quantity of work to execute, number of nodes to use, volume of I/O to execute during job makespan, and job compute time.

Platforms are characterized by the number of nodes, a system Mean Time Between Failures, and an aggregated I/O subsystem bandwidth that is shared among the nodes. For simplicity, we assume symmetric read and write filesystem bandwidths, hence $C_i = R_i$ for each application class, A_i.

A simulation first randomly selects a list of jobs that are instances of the different application classes. This list is ordered by job priority (*i.e.*, arrival time for our FCFS algorithms) and constrained by two parameters: the minimum simulated time to consider, and the relative proportion of platform resources used by each application class (based on the APEX report [9]). As an example, we consider the subset of application classes given by the APEX workflows report for the subset of application classes of LANL (EAP, LAP, Silverton and VPIC), simulated as if executed on the Cielo supercomputer, for a minimal execution time of 60 days. A simulation will randomly instantiate one of the four classes, assigning a work duration uniformly distributed between $0.8w$ and $1.2w$, where w is the typical walltime specified for the chosen application class, and count the resource allocated for this application class, until 1.) the simulated execution would necessarily run for at least 2 months, and 2.) resources used by the selected class is within 1% of the target goal of the representative workload percentage defined in the APEX workflows report (see Table I).

In addition to the jobs list, we generate a set of node failure times according to an exponential distribution with the specified MTBF. At the chosen times, we randomly choose which of the nodes fail. These jobs list and failure times constitute the initial conditions of a simulation.

Job Scheduling: We compute a job schedule (start and end times for all jobs in the list) using a simple first-fit strategy considering: job characteristics, job priority and resource availability. We simulate online scheduling; whenever a job ends at a date different than the initially planned end date (because of failures, or because the I/O interference made the job extend after its planned end date), the schedule is amended by rescheduling all jobs that were not started yet.

Execution Simulation: Once a job is started, it executes its initial input. It then, 1.) executes some work for a certain period, and 2.) checkpoints. These two steps are repeated until all planned work is executed, after which the final output is executed by the job, before it ends. At any time during the execution, a node hosting the job may be subject to a failure (according to the pre-computed failure times and location). When that happens, the job is terminated and a new job is added to the list of jobs to schedule. That new job represents the restart of the failed one; it has similar characteristics except its initial input corresponds to the restart size, and its work time corresponds to the remaining work from the last successful checkpoint. To reflect a common job scheduling policy on shared platforms, restarted jobs are set to the highest priority, maximizing their chances of obtaining an immediate allocation and continuing what was the original (failed) job execution.

Interference Models: Our simulations implement each of the interference models and avoidance strategies defined in Section III: for *Oblivious*-Fixed and *Oblivious*-Daly, interfering I/O and checkpoints get a portion of the available aggregated bandwidth proportional to the number of nodes they use, and inversely proportional to the number of nodes involved for all jobs doing I/O; for *Ordered*-Fixed and *Ordered*-Daly, I/O requests and checkpoints are ordered in a first-come first-served basis, and when they are selected, obtain the full bandwidth; for *Ordered-NB*-Fixed and *Ordered-NB*-Daly, I/O requests and checkpoints are served in order, but the simulation adds all the time waiting for a checkpoint to start as progress in the computation for the job; and for *Least-Waste*, the same is implemented, but I/O is ordered to minimize the waste in Equations (1) and (2).

Note that in the scheduled I/O methods (*Ordered-NB* and *Least-Waste*), initial inputs and final outputs are blocking (the job cannot progress during the I/O until it is served), but checkpoints are non-blocking.

Method of statistics collection from simulations: We compute the distribution of performance of each strategy using the Monte Carlo method: a large set of initial conditions (at least a thousand) is randomly chosen, and we simulate the execution of the system over each element of this set for each strategy. Since simulations for the various scheduling strategies have different initial conditions (including job mix), it would be misleading to compare simple averages of the time spent doing useful work (or time wasted) across simulation instances. Instead, we collect performance statistics over a fixed length segment of each simulation and extract and compare waste/work ratios that can be compared appropriately. The segment excludes the first and last days of the simulation: during the first day, jobs may be synchronized artificially because a subset starts at the same date, and during the last day, large amounts of resources may not be used as new jobs are not added to the workload. For each aggregate measurement, we compute and show mean, first and ninth decile, and first and third quartile statistics.

VI. Results

A. LANL APEX Simulation Workflows on Cielo

We consider the workload from LANL found in the APEX Workflows report [9] that consists of four applications classes: EAP, LAP, Silverton and VPIC. The main characteristics of these classes are reported in Table I. We simulate the behavior of these applications on the Cielo Platform. Cielo was a 1.37 Petaflops capability system operated from 2010 to 2016 at the Los Alamos National Laboratory. It consisted of 143,104 cores, 286 TB of main memory, and a parallel filesystem with a theoretical maximum capacity of 160GB/s. Cielo was chosen for this initial analysis due to the availability of the report [9], something not available for other platforms. Later on, we consider similar workloads on a more modern platform.

The baseline in this comparison comprises a set of simulations with neither faults, nor checkpoints, nor regular I/O interference. For these simulations, we selected a 60-day execution

Workflow	EAP	LAP	Silverton	VPIC
Workload percentage	66	5.5	16.5	12
Work time (h)	262.4	64	128	157.2
Number of cores	16384	4096	32768	30000
Initial Input (% of memory)	3	5	70	10
Final Output (% of memory)	105	220	43	270
Checkpoint Size (% of memory)	160	185	350	85

TABLE I
LANL WORKLOADS FROM THE APEX WORKFLOWS REPORT.

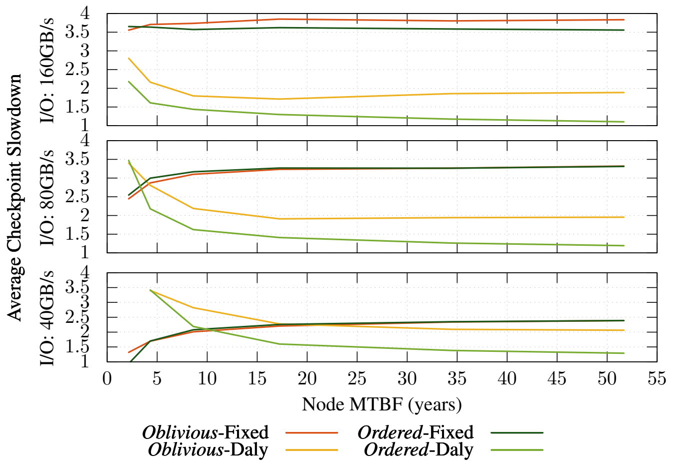

Fig. 1. Slowdown of checkpoints due to interference for the APEX Workshop workflow when simulating the Cielo platform with an effective bandwidth of 160GB/s, 80GB/s and 40GB/s.

segment, and computed the resources used by the jobs during this period, *i.e.* the total time each node spent on (non-CR) I/O and computation in a failure-free environment. For the I/O scheduling techniques presented in Section III, we compute the resource waste as the total time nodes spend not progressing jobs. In the figures, we represent the performance of each strategy by computing the waste, *i.e.* the ratio of the resource waste over a segment of 60 days divided by the application resource usage over that same segment for the baseline simulation. Each simulation is conducted over 1,000 times; the candlestick extremes represent the first and last decile of the measures, while the boxes represent the first and last quartile, and the center the mean value.

a) Slowdown of Checkpoints due to Interference: To illustrate how independent checkpoints interfere with each other, we measured how many resource applications spend their time checkpointing, relative to the same checkpointing strategy, but without interference. Applications either apply the fixed checkpoint period of one hour, or the optimal period give by the Young/Daly Formula. When two or more checkpoints overlap, we consider two scenarios: either they are slowed down proportionally to the amount of processes that share the bandwidth (*Oblivious*), or the first checkpoint completes before the next one can start (*Ordered*).

Depending on the filesystem availability and the MTBF of the machine (which impacts the results by introducing more or less failures, but also by changing the optimal checkpoint interval), applications spend more time to commit their checkpoint than they would if there was no interference. Figure 1 illustrates how much the average checkpoint commit is slowed

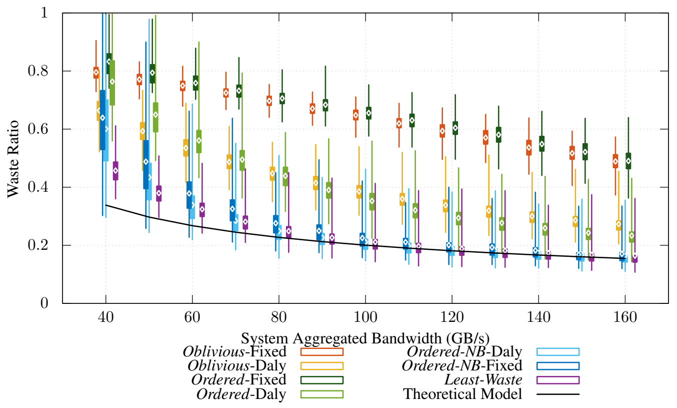

Fig. 2. Waste as a function of the system bandwidth for the seven I/O and Checkpointing scheduling strategies.

down as a function of the machine MTBF, for three system I/O bandwidths: 160GB/s, which is the theoretical peak of the machine, 80GB/s, and 40GB/s, which represent a degraded but realistic value of the achievable bandwidth when there is no interference. When the system bandwidth is at its peak (top of the figure), the Young/Daly formula provides a critical tool to minimize interference; still, when no cooperation between the applications is enforced, a significant interference is observed, and applications spend in average twice the time in checkpointing that they did expect. A simple interference management policy, like *Ordered*, reduces this dramatically, for all values of the MTBF. However, when the bandwidth is more constrained (bottom of the figure), and when the node MTBF is shorter, interferences introduce a significant slowdown for all strategies.

b) The Impact of Available System Bandwidth: First, we explore the performance of each approach in a failure-prone environment. Figure 2 represents the waste on the simulated platform, assuming the node MTBF μ_{ind} of 2 years (*i.e.* a system MTBF of 1h). We vary the filesystem bandwidth from 40 GB/s to 160GB/s in order to evaluate the impact of this parameter. We observe three classes of behavior: *Oblivious*-Fixed and *Ordered*-Fixed exhibit a waste that decreases as the bandwidth increases, but remains above 40% even at the maximum theoretical I/O bandwidth; *Ordered-NB*-Daly, *Ordered-NB*-Fixed, and *Least-Waste* quickly decrease to below 20% of waste, and reach the theoretical model performance[3]; and *Oblivious*-Daly and *Ordered*-Daly start at the same level of efficiency as *Oblivious*-Fixed and *Ordered*-Fixed, and slowly reach 20% of waste as the bandwidth increases. Note, in some cases the error bars dip below the theoretical lower bound. In the simulations, failures have an exponential probability distribution centered around the desired MTBF. For some runs, a lower number of failures experienced during the simulation results in a larger MTBF than the average used in the lower-bound formula; such instances can experience a waste lower than the theoretical model.

This figure shows that with a high frequency of failures, providing each job with the appropriate checkpoint interval

[3]Maple code to compute the performance predicted by the theoretical model is available at https://github.com/SMURFSorg/InterferingCheckpoints.

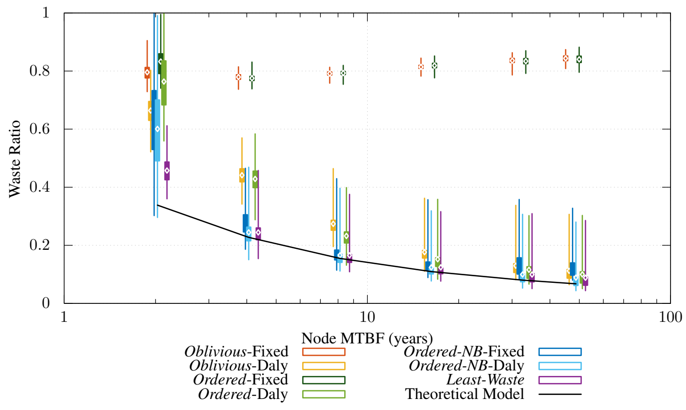

Fig. 3. Waste as a function of the system MTBF for the seven I/O and Checkpointing scheduling strategies.

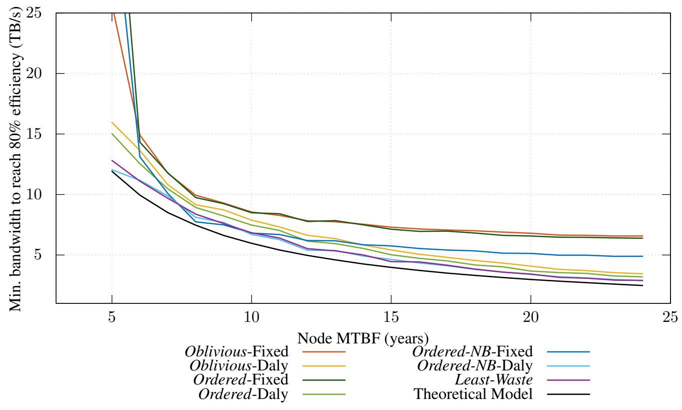

Fig. 4. Minimum aggregated filesystem bandwidth to reach 80% efficiency with the different approaches on the prospective future system.

is paramount to preventing unnecessary (or even detrimental) checkpoints: the two strategies that render high waste despite high bandwidth rely on a fixed 1h interval. However, it also shows that this is not the sole criteria that should be taken into account, nor a necessary condition to extract performance. Even with favorable bandwidth, *Oblivious*-Daly and *Ordered*-Daly experience nearly twice the waste of the other strategies with the same checkpointing period. All strategies that decouple the execution of the application from the filesystem availability (*Ordered-NB*-Daly, *Ordered-NB*-Fixed, *Least-Waste*) exhibit considerably better performance despite low bandwidth.

Notably, *Least-Waste* remains the most efficient technique in this study, and reaches the theoretical performance given by Equation (7) for steady-state analysis. This illustrates the efficiency of the proposed heuristic (Equations (1) and (2)) to schedule checkpoints and I/O in a way that avoids interferences, allowing the system to behave as if no interference is experienced, in most cases. The high variation shows that a minority of the runs experienced a significantly higher waste, but such is the case for all algorithms.

c) The Impact of System Reliability: Next, we explore the performance of each approach under low bandwidth (and thus high probability of interference). A scenario with such low bandwidth is not unrealistic. As shown in Luu et al [6], practical bandwidth can be considerably lower than theoretical. Figure 3 represents the waste on the simulated platform, assuming the aggregated filesystem bandwidth of the system is 40GB/s. We vary the node MTBF μ_{ind} from 2 years (1h of system MTBF) to 50 years (24h of system MTBF) in order to evaluate the impact of this parameter. Similar to Figure 2, we observe three classes of behavior: *Oblivious*-Fixed and *Ordered*-Fixed exhibit a waste that remains constant around 80% for all values of the MTBF. These approaches are critically dependent on the filesystem bandwidth, and a lower frequency of failures does not significantly improve their performance. The I/O subsystem is saturated, and the applications spends most of their time waiting for it. *Oblivious*-Daly and *Ordered*-Daly, see poor efficiency for small MTBF values, but steadily improve to come close to the theoretical bound for higher MTBF values. Lastly, *Ordered-NB*-Daly, *Ordered-NB*-Fixed, and *Least-Waste* quickly reach the theoretical model performance, even with a low MTBF (4 year node MTBF or 2h of system MTBF).

For all the strategies that use the Daly checkpointing period, increasing the MTBF reduces the amount of I/O required and thus relieves the pressure of a constrained bandwidth. All strategies that schedule the bandwidth are successful at increasing the efficiency close to the theoretical model. Similarly, *Ordered-NB*-Fixed, despite its fixed checkpoint interval is capable of reaching a performance comparable to the Daly-based strategies (which reduce the number of total checkpoints). The rapid improvement of the *Ordered-NB*-Fixed approach can be explained by a combination of 2 factors. Foremost, the non-blocking aspect of the checkpoint provide the I/O subsystem with enough flexibility to order the checkpoint without imposing an additional wait. Delayed checkpoints only translate in additional waste if that application itself is subject to failure. Additionally, for lower MTBFs, the more frequent restarts of interfering jobs, despite the fact that they delay the checkpointing operation, do not introduce additional waste.

B. *Evaluating a Prospective System*

To understand the impact of the I/O contention on future platforms, we explore a prospective system and assess the impact of I/O and checkpoint scheduling when the problem size and the machine size will increase. We consider a future system with 7PB of main memory and 50,000 compute nodes (*e.g.* Aurora *https://aurora.alcf.anl.gov/*). Based on the APEX workflow report, we extrapolate the increase in problem size expected for the application classes considered previously, and project these applications on the prospective system. We simulate the workload of Table I, scaling the problem size proportionally to the change in machine memory size. The waste is computed, as previously, by dividing the amount of resource used for checkpoints and lost due to failures by the amount of resource used in a fault-free and resilience-free run with the same initial conditions. We vary system MTBF; and for each strategy, we find the required aggregated practical bandwidth necessary to provide a sustained 80% efficiency of the system. This 80% target efficiency is viewed by many programs (*e.g.* The Exascale Computing Project *https://exascaleproject.org*) as a reasonable cost for resilience activities. Figure 4 shows the impact of MTBF and strategies on this prospective system.

When failures are frequent (less than 10 year node MTBF), the most critical element is to reduce the I/O pressure: all strategies that use a fixed and frequent checkpoint interval require greater available bandwidth to reach the target efficiency. In this case, strategies that combine an optimal checkpointing period with I/O and checkpoint scheduling (*Least-Waste* and *Ordered-NB*-Daly) perform similarly, consistently better than all other approaches. These two approaches exhibit a strong resilience to failures, with a bandwidth requirement that only increases by a factor of three between a very unstable system (less than one hour system MTBF), and a stable one (an 8 hour system MTBF). In contrast, the other strategies are much more dependent upon the frequency of failures; the *Oblivious*-Fixed strategy requires up to 50 times the bandwidth of *Least-Waste* to reach the same efficiency.

When failures are less frequent (*i.e.* a node MTBF is at least 15 years and a system MTBF of 2.6 hours), the hierarchy of different approaches stabilizes. The two blocking strategies relying on frequent checkpoints (*Oblivious*-Fixed and *Ordered*-Fixed) remain expensive, requiring the highest bandwidth to reach the target efficiency. The next contender, *Ordered-NB*-Fixed, requires a quarter of the bandwidth to reach the same efficiency. Despite using the same fixed checkpoint interval as the previous methods, it benefits from not blocking when the filesystem is not available. This is sufficient, when failures are rare, to obtain a significant performance gain. All Daly-based strategies benefit from reduced I/O pressure, and reach the target efficiency with around half the bandwidth needed by *Oblivious*-Fixed. We also observe that *Ordered-NB*-Daly and *Least-Waste* remain the most efficient strategies for the whole MTBF spectrum. These results highlight that checkpoint-based strategies can scale to satisfy the need of future platforms, whether by integrating I/O-aware scheduling strategies or by significantly over-provisioning the I/O partition.

VII. Related Work

We first discuss research regarding checkpoint-induced I/O pressure, followed by works that regard avoiding I/O interference. These techniques are not necessarily independent: generally, reducing I/O pressure will reduce the likelihood of interference. Therefore, we focus our I/O interference discussion to those techniques which consider the global scheduling of checkpoints and/or application I/O across a platform.

Checkpointing and I/O: For a single application, the Young/Daly formula [3], [4] gives the optimal checkpointing period. This period minimizes platform waste, defined as the fraction of job execution time that does not contribute to its progress. Arunagiri et al. [14] studied longer, sub-optimal periods with the intent of reducing I/O pressure and showed, both analytically and empirically using four real platforms, that a decrease in the I/O requirement can be achieved with only a small increase in waste.

Reducing I/O Pressure: There are two general strategies for reducing I/O pressure from a single application: hiding or reducing checkpoint commit times without reducing checkpoint data volumes, and reducing commit times by reducing checkpoint data volumes. Strategies that attempt to hide checkpoint times include Diskless [15] and remote checkpoint protocols [16]. Multi-level checkpoint protocols like SCR [11], [17] attempt to hide checkpoint commit times by writing checkpoints to RAM, flash storage, or local disk on the compute nodes [18] in addition to the parallel file system thereby improving checkpoint or general I/O bandwidth.

Strategies that attempt to reduce checkpoint sizes include *memory exclusion*, which leverage user-directives or other hints to exclude portions of process address spaces from checkpoints [19]. Additionally, incremental checkpointing protocols reduce checkpoint volumes by utilizing the OS's memory page protection facilities to detect and save only pages that have been updated between consecutive checkpoints [20], [21], [22], [23], [24]. Similarly, page-based hashing techniques can also be used to avoid checkpointing pages that have been written to but whose content has not changed [25]. Finally, compression-based techniques use standard compression algorithms to reduce checkpoint volumes [26] and can be used at the compiler-level [27] or in-memory [28]. Tanzima et al. show that similarities amongst checkpoint data from different processes can be exploited to reduce checkpoint data volumes [29]. Lossy compression methods are studied in [30], [31].

Avoiding I/O interference: Most closely related to our work, a number of studies have considered the global scheduling of checkpoints and other I/O across a platform to reduce overall congestion, thereby increasing performance. Aupy et al. [32] presented a decentralized I/O scheduling technique for minimizing the congestion due to checkpoint interference by taking advantage of the observed periodic and deterministic nature of HPC application checkpoints and I/O. This technique allows the job scheduler to pre-define each applications I/O behavior for their entire execution. Similarly, a number of works have investigated the efficiency of online schedulers for data intensive [33], [34] and HPC workload I/O [10], [35]. Finally, a number of works have investigated utilizing recorded system reliability information [36] and the statistical properties of these failures [37] to determine effective checkpoint intervals for the portion of the system used by the workload.

Summary: Unlike a number of the previous studies, our technique considers the interaction between existing non-CR application I/O and CR I/O. The proposed non-blocking approaches that we propose leverage the capability of applications to continue working (to the increased risk of having to re-execute more work) while they wait for the checkpoint token to be granted. Additionally, our approach is agnostic to the I/O patterns of the considered applications as long as they are known. Also, we attempt at optimizing the efficiency of the entire platform, with the changing workloads and failures running on that platform, rather than just considering one workload. Finally, and most importantly, this approach provides optimal checkpointing periods in environments where I/O is highly constrained and Daly/Young's formula is less appropriate, a common scenario on many leadership-class systems.

VIII. Conclusion and Future Work

As we design larger, likely more error-prone platforms, effectively protecting applications from platform faults becomes critical. Current fault-protection techniques available on production platforms rely on checkpoint/restart to ensure fault protection. However, these techniques, by their very nature, regularly save the application state to stable storage, and therefore increase the burden of the already overtaxed I/O subsystem.

Considering a comprehensive I/O interference model for platforms susceptible to I/O contention, we designed multiple I/O scheduling algorithms that target improving overall platform job throughput via waste minimization. We also theorized a lower-bound for platform waste for I/O constrained checkpointing workloads. We use this theoretical lower-bound to demonstrate the effectiveness of our *Least-Waste* I/O scheduling and to compare its performance with other I/O scheduling strategies. Our strategy invariably outperforms the others with respect to the platform efficiency. Unsurprisingly, the biggest gains are rendered on the platforms with a lower MTBF or greater degrees of under-provisioned I/O. Through simulation, we also show a path to supporting C/R on a prospective system while maintaining a 80% platform efficiency, all without a large investment in the I/O subsystem.

As burst-buffers and other NVRAM storage mechanisms become more common, a natural extension of this work would consider their impact on I/O contention/interference. Increasing the available I/O bandwidth leads to reduced waste (due to the decrease in checkpoint duration but also an increase in the optimal checkpoint frequency and therefore a decrease in the restart time), while providing relief to the shared I/O subsystem to better absorb additional checkpoint information. We speculate that scheduling parallel filesystem I/O with a heuristic that prioritizes jobs to minimize failure impact can help to improve overall burst-buffer efficiencies. Such a heuristic would build upon the strategies discussed in this work and extend them to the new framework.

References

[1] O. Weidner, M. Atkinson, A. Barker, and R. Filgueira Vicente, "Rethinking high performance computing platforms: Challenges, opportunities and recommendations," in *Proc. Data-Intensive Distributed Computing DIDC*. ACM, 2016.

[2] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: Performance degradation due to nearby jobs," in *Proc. Supercomputing (SC '13)*. ACM, 2013.

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

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

[5] T. Herault and Y. Robert, Eds., *Fault-Tolerance Techniques for High-Performance Computing*. Springer Verlag, 2016.

[6] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A multiplatform study of I/O behavior on Petascale supercomputers," in *HPDC*. ACM, 2015.

[7] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the root causes of cross-application I/O interference in HPC storage systems," in *IPDPS*, 2016, pp. 750–759.

[8] M. Mubarak, P. H. Carns, J. Jenkins, J. K. Li, N. Jain, S. Snyder, R. B. Ross, C. D. Carothers, A. Bhatele, and K. Ma, "Quantifying I/O and communication traffic interference on dragonfly networks equipped with burst buffers," in *IEEE Cluster*, 2017.

[9] APEX, "APEX Workflows," LANL, NERSC, SNL, Research report SAND2016-2371 and LA-UR-15-29113, 2016.

[10] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination," in *IPDPS*. IEEE, 2015, pp. 156–163.

[11] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proc. Supercomputing (SC '10)*. ACM, 2010.

[12] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High performance fault tolerance interface for hybrid systems," in *Proc. SC'11*, 2011.

[13] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[14] S. Arunagiri, J. T. Daly, and P. J. Teller, "Modeling and Analysis of Checkpoint I/O Operations," in *Analytical and Stochastic Modeling Techniques and Applications: 17th International Conference*. Springer, 2010, pp. 387–399.

[15] J. Plank, K. Li, and M. Puening, "Diskless checkpointing," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, oct 1998.

[16] J. Cornwell and A. Kongmunvattana, "Efficient System-Level Remote Checkpointing Technique for BLCR," in *Eighth Int. Conf. on Information Technology: New Generations (ITNG)*, 2011, pp. 1002–1007.

[17] N. H. Vaidya, "A case for two-level distributed recovery schemes," in *ACM SIGMETRICS Joint Int. Conf. on Measurement and Modeling of Computer Systems*. ACM, 1995.

[18] A. Kougkas, M. Dorier, R. Latham, R. Ross, and X. H. Sun, "Leveraging burst buffer coordination to prevent I/O interference," in *IEEE 13th Int. Conf. on e-Science (e-Science)*. IEEE, 2017, pp. 372–379.

[19] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley, "Memory Exclusion: Optimizing the Performance of Checkpointing Systems," *Software – Practice & Experience*, vol. 29, no. 2, pp. 125–142, 1999.

[20] G. Bronevetsky, D. Marques, K. Pingali, S. McKee, and R. Rugina, "Compiler-enhanced incremental checkpointing for OpenMP applications," in *IPDPS*, 2009.

[21] E. N. Elnozahy, D. B. Johnson, and W. Zwaenpoel, "The Performance of Consistent Checkpointing," in *11th IEEE Symposium on Reliable Distributed Systems*, Houston, TX, 1992.

[22] K. Li, J. F. Naughton, and J. S. Plank, "Low-Latency, Concurrent Checkpointing for Parallel Programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 874–879, August 1994.

[23] M. Paun, N. Naksinehaboon, R. Nassar, C. Leangsuksun, S. L. Scott, and N. Taerat, "Incremental Checkpoint Schemes for Weibull Failure Distribution," *Int. J. Computer Science*, vol. 21, no. 3, pp. 329–344, 2010.

[24] S. Al-Kiswany, M. Ripeanu, S. Vazhkudai, and A. Gharaibeh, "stdchk: A Checkpoint Storage System for Desktop Grid Computing," in *ICDCS*, 2008, pp. 613–624.

[25] K. B. Ferreira, R. Riesen, R. Brightwell, P. G. Bridges, and D. Arnold, "Libhashckpt: Hash-based Incremental Checkpointing Using GPUs," in *Proceedings of the 18th EuroMPI Conference*, September 2011.

[26] D. Ibtesham, D. Arnold, P. G. Bridges, K. B. Ferreira, and R. Brightwell, "On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance," *2012 41st International Conference on Parallel Processing*, vol. 0, pp. 148–157, 2012.

[27] C.-C. Li and W. Fuchs, "CATCH-compiler-assisted techniques for checkpointing," in *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, jun 1990, pp. 74–81.

[28] J. S. Plank and K. Li, "ickp: A Consistent Checkpointer for Multicomputers," *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 2, no. 2, pp. 62–67, 1994.

[29] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. De Supinski, and R. Eigenmann, "MCRENGINE: A Scalable Checkpointing System Using Data-Aware Aggregation and Compression," in *SC'12)*. ACM, 2012.

[30] N. Sasaki, K. Sato, T. Endo, and S. Matsuoka, "Exploration of lossy compression for application-level checkpoint/restart," in *IPDPS*. IEEE, 2015.

[31] X. Ni, E. Meneses, N. Jain, and L. V. Kalé, "ACR: Automatic Checkpoint/Restart for Soft and Hard Error Protection," in *SC'14*. ACM, 2014.

[32] G. Aupy, A. Gainaru, and V. Le Fevre, "Periodic I/O Scheduling for Super-computers," INRIA, Research report RR-9037, Feb. 2017.

[33] S. Groot, K. Goda, D. Yokoyama, M. Nakano, and M. Kitsuregawa, "Modeling I/O Interference for Data Intensive Distributed Applications," in *Proc. 8th Annual ACM Symp. on Applied Computing (SAC)*. ACM, 2013, pp. 343–350.

[34] H. Sim, Y. Kim, S. S. Vazhkudai, D. Tiwari, A. Anwar, A. R. Butt, and L. Ramakrishnan, “AnalyzeThis: an analysis workflow-aware storage system,” in *Proc. Supercomputing (SC’15)*. ACM, 2015.

[35] S. Herbein, D. H. Ahn, D. Lipari, T. R. Scogland, M. Stearman, M. Grondona, J. Garlick, B. Springmeyer, and M. Taufer, “Scalable I/O-Aware Job Scheduling for Burst Buffer Enabled HPC Clusters,” in *Proceedings HPDC*. ACM, 2017, pp. 70–79.

[36] A. J. Oliner, L. Rudolph, and R. K. Sahoo, “Cooperative checkpointing: A robust approach to large-scale systems reliability,” in *ICS*. ACM, 2006.

[37] D. Tiwari, S. Gupta, and S. S. Vazhkudai, “Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems,” in *DSN*. IEEE, 2014, p. 2536.