# Design and Modeling of an Asynchronous Checkpointing System

Kento Sato[1,2] Adam Moody[3] Kathryn Mohror[3] Todd Gamblin[3] Bronis R. de Supinski[3]
Naoya Maruyama[4,5] Satoshi Matsuoka[1,5,6]

***Abstract:*** As the capability and component count of high performance computing systems increase, the mean time before failure correspondingly decreases. Typically, applications tolerate failures with checkpoint/restart, where an application writes its state in checkpoints on the parallel file system (PFS); upon failure the application restarts from the last checkpointed state. While simple, this approach suffers from high overhead due to contention for PFS resources. A promising solution to this problem is multi-level checkpointing. However, while multi-level checkpointing is successful on today's machines, it is not expected to be sufficient for exa-scale class machines, where the total memory sizes and failure rates are predicted to be orders of magnitude higher. Our solution to this problem is a system that combines the benefits of asynchronous and multi-level checkpointing. In this paper, we present the design of our system and a model describing its performance. Our experiments show that our system can improve efficiency by 1.1 to 2.0 × on future machines. Additionally, applications using our checkpointing system can achieve high efficiency even when using a PFS with lower bandwidth.

## 1. Introduction

The computational power of High Performance Computing (HPC) systems is growing exponentially, which enables researchers to conduct fine-grained scientific simulations. However, the overall failure rate of HPC systems increases with the size of the system. For example, in the year and a half from November 1st 2010 to April 6th 2012, the TSUBAME2.0 cluster experienced 962 node failures due to a variety of failures ranging from memory errors to whole rack failures [1]. This means that a failure occurred every 13.0 hours on average. Moreover, in future systems, the MTBF (mean time between failures) is projected to shrink to only tens of minutes [2]. Without a viable resilience strategy, it will be impossible for an application to run for even one day on such a large machine. Thus, resilience in HPC has become more important than ever as we plan for future systems.

Checkpointing is an indispensable fault tolerance technique, commonly used by HPC applications that run continuously for hours or days at a time. A *checkpoint* is a snapshot of application state that can be used to restart execution if a failure occurs. However, when checkpointing large-scale systems, tens of thousands of compute nodes write checkpoints to a parallel file system (PFS) concurrently, and the low I/O throughput becomes a bottleneck. Although simple, this straightforward checkpointing scheme can impose huge overheads on application run times.

1 Tokyo Institute of Technology
2 Research Fellow of the Japan Society for the Promotion of Science
3 Lawrence Livermore National Laboratory
4 RIKEN Advanced institute for Computational Science
5 Global Scientific information and Computing Center
6 National Institute of Informatics

Multi-level checkpointing [3], [4] is a promising approach for addressing these problems. This approach uses multiple storage levels, such as RAM, local disk, and the PFS, according to the different degrees of resiliency and the cost of checkpointing in those storage levels. Multi-level checkpointing systems typically rely on node-local storage levels for restarting from more common failures, such as single-node failures, and the PFS for more catastrophic failures. By taking frequent, inexpensive node-local checkpoints, and less frequent, high-cost checkpoints to the PFS, applications can achieve both high resilience and better efficiency.

However, the bandwidths of PFSs do not keep pace with increases in computational capabilities, and the imbalance in performance means applications can be blocked for tens of minutes for a single checkpoint [3]. Thus, the overhead of checkpointing to the PFS can dominate the overall application run time even with infrequent PFS checkpoints. Moreover, the huge number of concurrent I/O operations from large-scale jobs burden the PFS and are themselves a major source of failures. Thus, it is critical to achieve both high reliability and efficiency while reducing the load on the PFS.

Our solution to this problem is an asynchronous checkpointing system in which agents running on additional nodes asynchronously transfer checkpoints from the compute nodes to the PFS. Our approach has two key advantages. It lowers the overhead of checkpointing on the application by allowing the overlap of computation and writing checkpoints to the PFS. Also, it reduces the load on the PFS by reducing the number of concurrent writers and moderating the rate of I/O operations to the PFS.

The major contributions of this paper are listed as follows:

- the design of an asynchronous checkpointing system

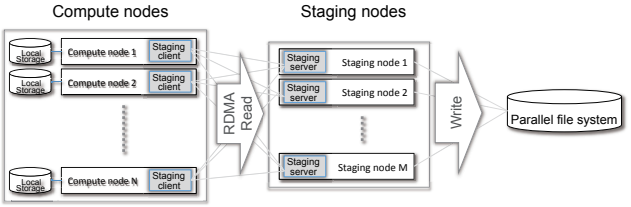



**Fig. 1** The asynchronous checkpointing system

- a Markov model of asynchronous checkpointing on top of any multi-level checkpointing systems
- a comprehensive exploration of asynchronous checkpointing on current and future systems.

Our experimental results show that our asynchronous checkpointing system can improve efficiency by 1.1 to 2.0 times on future systems. Additionally, we found that combining asynchronous and multi-level checkpointing results in highly efficient application runs with low requirements for PFS bandwidth.

## 2. Asynchronous Checkpointing System

Overlapping I/O with computation by delegating operations to dedicated I/O nodes is known to improve application performance and to mitigate workload imposed on the PFS [5], [6]. Furthermore, as aforementioned, an application using multi-level checkpointing does not require frequent checkpoints to a PFS. By combining long intervals between consecutive PFS checkpoints with asynchronous flushes (which we refer to as asynchronous checkpoints), data can be copied to the PFS at a slow rate to reduce impact on the application as well as the PFS. Our asynchronous checkpointing system is developed as an extension to the SCR library [7]; it asynchronously transfers node-local checkpoints written by SCR to the PFS. In this section, we cover the design of our asynchronous checkpointing system.

### 2.1 Architecture

As shown in Figure 1, our asynchronous checkpointing system has two types of nodes: *compute nodes* and *staging nodes*. The compute nodes are the nodes on which an application is executed. The staging nodes are a group of nodes that are used to transfer checkpoints from the compute nodes to the PFS. The staging nodes asynchronously read checkpoint data from the compute nodes and write data to the PFS while the application continues to execute and write other checkpoints to cache as node-local checkpoints. Generally, each staging node handles multiple compute nodes, and the exact ratio is determined by modeling and experimental testing (See Sections 3 and 4).

A *staging client* process runs on each compute node, and a *staging server* process runs on each staging node. When SCR finishes caching a checkpoint (node-local checkpoint) that is to be flushed to the PFS, it signals the staging client process via a library function call. The staging client then sends a request to the staging server and the two processes execute a protocol to transfer the checkpoint; we give details in Section 2.2. The staging server reads checkpoints from the compute nodes using Remote Direct Memory Access (RDMA) to minimize CPU usage on the compute nodes.

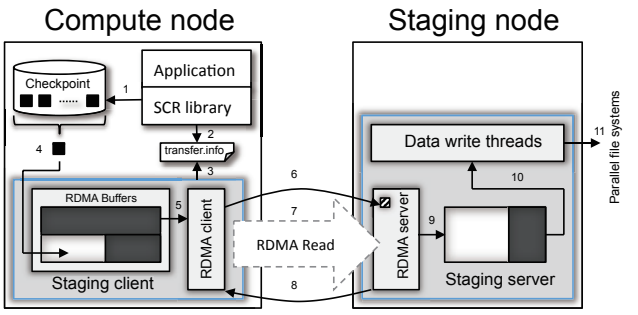



**Fig. 2** Proposed asynchronous checkpointing client/server using RDMA

By using additional nodes to transfer checkpoint data with RDMA, our asynchronous checkpointing system can drain a checkpoint from compute nodes to a PFS while minimizing the impact on the application runtime.

### 2.2 RDMA checkpoint transfers

We implemented an RDMA checkpoint transfer system for asynchronous checkpointing based on the SCR library [7]. The existing SCR asynchronous flush reads a checkpoint from a local storage to a buffer and directly writes the checkpoint from the buffer to the PFS. We extended this implementation to drain a checkpoint from compute nodes to the PFS through a staging node using RDMA transfers. The other checkpoint management, such as version, checkpoint location and redundancy scheme, relies on the original SCR library. Figure 2 describes the architecture. The staging client and server processes run on compute nodes and staging nodes, respectively. These processes transfer and write checkpoints to the PFS.

Here, we explain how a checkpoint on a compute node is written to the PFS through a staging node. For simplicity, we assume that the SCR library writes checkpoints to local storage according to a particular redundancy scheme (Step 1 in Figure 2). After several checkpoints are written to local storage, SCR writes transfer information into a file called *transfer.info* to request the staging client to transfer the checkpoint according to specified flush frequency (Step 2). The transfer.info file includes the source and destination paths from and to which the checkpoint should be flushed. The staging client process periodically checks the transfer.info file to see if SCR has issued the request (Step 3). If the staging client detects a request in transfer.info, it reads the checkpoint from the source path and writes to local RDMA buffer space (Step 4). Once the staging client fills the buffer, it calls an RDMA client function to send out the chunk of checkpoint data in the buffer (Step 5). Since the RDMA client function returns control immediately, the staging client process can read the next chunk to one of the buffer entries in the buffer pool (Figure 2 shows the double-buffering case) while the RDMA client transfers checkpoint chunks to the staging server. The RDMA client issues a request for the chunk transfer to an RDMA server (Step 6). When the RDMA server running on the staging node receives a transfer request, an RDMA read request is issued to the staging client to read the remote buffer space into a local buffer, which then sends an acknowledgement to the RDMA client. The RDMA client issues RDMA read requests until all chunks are sent to the staging

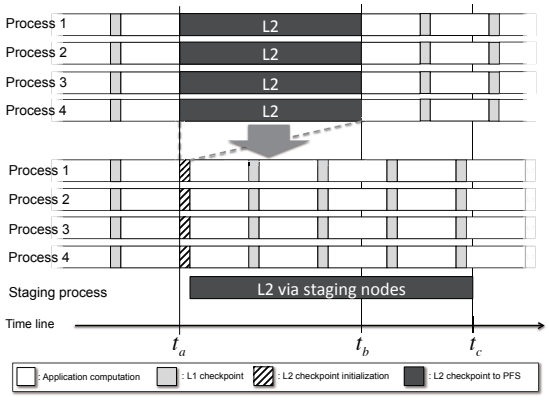



**Fig. 3** Asynchronous checkpointing can hide L2 checkpoint overhead

node (Step 7-9). Finally, the data writer threads write the checkpoint to the PFS in parallel with RDMA reads by the RDMA server (Step 10-11).

To transfer checkpoints from thousands of compute nodes with fewer staging nodes, a staging server can concurrently handle RDMA requests from multiple staging clients. However, a large amount of incoming checkpoint data can cause a buffer overflow on a staging node. To avoid this, if buffered checkpoint data exceeds a specified buffer size limit, the staging nodes throttle the RDMA read rate to balance incoming and outgoing checkpoint data. Thus, to avoid imbalance in incoming and outgoing data, We discuss determination of the appropriate number of staging nodes.

### 2.3 Synchronous and Asynchronous Checkpointing

While intuitively an asynchronous checkpointing is expected to be efficient, an asynchronous one have advantages and disadvantages. Figure 3 shows the difference between synchronous and asynchronous checkpointing. While an asynchronous one writes checkpoints in the background of an application execution, synchronous one blocks the application until the copy has completed. To clarify the differences, we characterize both schemes with two metrics, *checkpoint overhead* and *checkpoint latency*. Checkpoint overhead ($C$) is the increased execution time of an application because of checkpointing. Checkpoint latency ($L$) is the duration of time it takes to complete a checkpoint.

During synchronous checkpointing, each process writes its own checkpoint data to the PFS, and blocks until the checkpoint operation completes. Thus, the checkpoint overhead is identical to the checkpoint latency, i.e., $C_{blk} = L_{blk} = t_b - t_a$. $N$ iterations of synchronous checkpointing result in $N \times C_{blk}$ increase in application runtime.

During asynchronous checkpointing, since each process continues computation during the PFS checkpoint, the checkpoint overhead is generally smaller than the checkpoint latency, $C_{nblk} \leq L_{nblk} = t_c - t_a$. Here, $C_{nblk}$ and $L_{nblk}$ are determined by the application characteristics. If an application is computation or network bound, $C_{nblk}$ and $L_{nblk}$ increase due to resource contention, and $L_{nblk}$ $(= t_c - t_a)$ becomes larger than $L_{blk}$ $(= t_b - t_a)$. Actually, to initiate an asynchronous checkpoint, an application need to write its checkpoint data to local storages, such as RAM disks, SSDs. During the write operations, an application is blocked, and the overhead is added to $C_{nblk}$.

Asynchronous checkpointing has advantages over synchronous checkpointing. With asynchronous checkpointing, $C_{nblk}$ can be minimized by slowly writing checkpoint data to the PFS, thereby alleviating resource contention. Because lower-level checkpoints can continue to be cached on the compute nodes during an asynchronous checkpoint, the application can take more frequent checkpoints and increase resiliency with low checkpoint overhead. In contrast, when an application takes a synchronous checkpoint to the PFS, the application loses $C_{blk}$ potential computation time, and it is significantly more vulnerable to failure, as heavy load on the PFS increases the likelihood of failure of the PFS.

Thus, intuitively we expect asynchronous checkpointing to be more efficient than synchronous checkpointing. However, asynchronous checkpointing has a disadvantage. In Figure 3, the synchronous checkpoint completes at $t_b$ while the asynchronous checkpoint finishes at $t_c$. If a failure which requires PFS checkpoint occurs in the period between $t_b$ and $t_c$, an asynchronous checkpointing system incurs a catastrophic rollback to much older checkpoint, because the PFS checkpoint is not available at the time of the failure. On the other hand, synchronous checkpointing only rollbacks to $t_b$. Therefore, with asynchronous checkpointing, the checkpoint interval, $C_{nblk}$, $L_{nblk}$, and the frequency of each level of checkpoint must be optimized to lower the risk of the catastrophic rollback.

## 3. Asynchronous Checkpointing Model

As mentioned previously, with asynchronous checkpointing, several factors are critical to performance: checkpoint interval, $C_{nblk}$, $L_{nblk}$, and frequency of each level of checkpoint. To determine the optimal values, we extend an existing model of a multi-level checkpointing system [3] to support our asynchronous checkpointing system.

### 3.1 Assumptions

For simplification, we make several assumptions in our asynchronous checkpointing model. Because our model is constructed on a top of an existing one, we include the assumptions made in the existing model [3]. We highlight the important assumptions here.

We assume that failures are independent across components and occur following a Poisson distribution. Thus, a failure within a job does not increase the probability of successive failures. In reality, some failures can be correlated. For example, failure of a PSU (Power supply unit) can take out multiple nodes. The XOR encoding can actually handle failures category 2 and even higher. In fact, using topology-aware techniques, the probability of those failures affecting processes in the same XOR set is very low. In such cases you don't need to restart from the PFS. SCR also exclude problematic nodes from restarted runs. Thus, the assumption implies that the average failure rates do not change and dynamic checkpoint interval adjustment is not required during application execution.

We also assume that the costs to write and read checkpoints are constant throughout the job execution. In reality, I/O per-

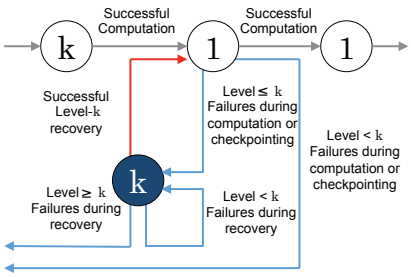



**Fig. 4** The basic structure of the asynchronous checkpointing model

formance can fluctuate because of contention for shared PFS resources. However, staging nodes serve as a buffer between the compute nodes and the PFS. Thus, our system mitigates the performance variability of the PFS.

If a failure occurs during asynchronous checkpointing, we assume that checkpoints cached on failed nodes have not been written to the PFS. Thus, we need to recover the lost checkpoint data from redundant stores on the compute nodes, if possible, and if not, locate an older checkpoint to restart the application. This could be an older checkpoint cached on compute nodes, assuming multiple checkpoints are cached, or a checkpoint on the PFS.

### 3.2 Basic model structure

As employed in the existing model [3], we use a Markov model to describe run time states of an application. We construct the model by combining the basic structures shown in Figure 4. The basic structure has *computation* (white circle) and *recovery* (blue circle) states labeled by a checkpoint level. The computation states represent periods of application computation followed by a checkpoint at the labeled level. The recovery state represents the period of restoring from a checkpoint at the labeled level.

If no failures occur during a compute state, the application transitions to the next right compute state (gray arrow). We denote the checkpoint interval between checkpoints as $t$, the cost of a level $c$ checkpoint as $c_c$, and rate of failure requiring level $k$ checkpoint as $\lambda_k$. The probability of transitioning to the next right compute state and the expected time before transition are $p_0(t + c_c)$ and $t_0(t + c_c)$ where:

$$p_0(T) = e^{-\lambda T}$$
$$t_0(T) = T$$

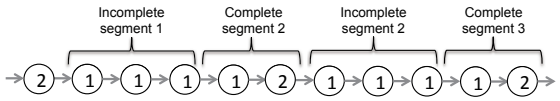
We denote $\lambda$ as the summation of all levels of failure rates, i.e., $\lambda = \sum_{i=1}^{L} \lambda_i$ where $L$ represents the highest checkpoint level. If a failure occurs on during a compute state, the application transitions to the most recent recovery state which can handle the failure (bule arrow). If the failure requires level $i$ checkpoint or less to recover and the most recent recover state is at level $k$ where $i \leq k$, the application transitions to the level $k$ recovery state. The expected probability of $i$ level failure in an interval $t + c_c$, and run time before the transition from the compute state $c$ to the recovery state $k$ are $p_i(t + c_c)$ and $t_i(t + c_c)$ where:

$$p_i(T) = \frac{\lambda_i}{\lambda}(1 - e^{-\lambda T})$$

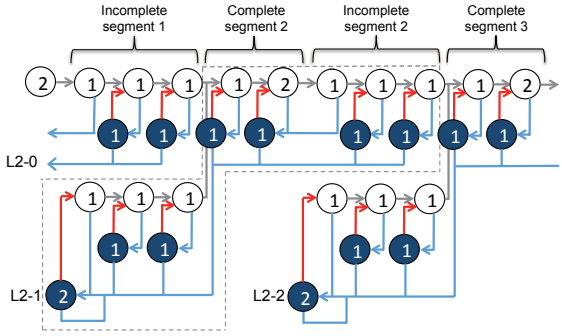
$$t_i(T) = \frac{1 - (\lambda T + 1) \cdot e^{-\lambda T}}{\lambda \cdot (1 - e^{-\lambda T})}$$

During recovery, if no failures occur, the application transitions to the compute state that took the checkpoint that was used for recovery (red arrow). If cost of recovery from a level $k$ checkpoint is $r_k$, the expected probability of the transition and the expected run time are given by $p_0(r_k)$ and $t_0(r_k)$. If a failure requiring $i$ level checkpoint occurs while recovering, and $i < k$, we assume the current recovery state can retry the recovery. However, if $i \geq k$, we assume the application must transition to a higher-level recovery state. The expected probabilities and times of failure during recovery are $p_i(r_k)$ and $t_i(r_k)$. We also assume that the highest level recovery state (level $L$) that uses checkpoints on the PFS, can be restarted in the event of any failure $i \leq L$.

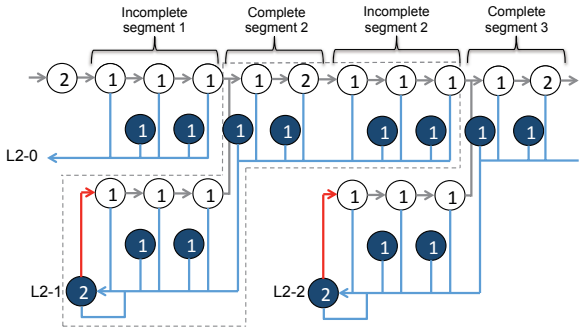
### 3.3 Asynchronous checkpoint model




(a) No failures and no recoveries




(b) Level-1 (L1) failures and recoveries




(c) Level-2 (L2) failures and recoveries

**Fig. 5** Transition diagram of the asynchronous checkpointing

We describe our model of asynchronous checkpointing by combining the basic structures from Figure 4. We show a two level example in Figure 5. If no failures occur during execution, the application simply transitions across the compute states in sequence (Figure 5(a)). In this example, level 1 (L1) checkpoints (e.g., XOR checkpoints) are taken as synchronous checkpoints, and level 2 (L2) checkpoints (e.g., PFS checkpoints) are taken

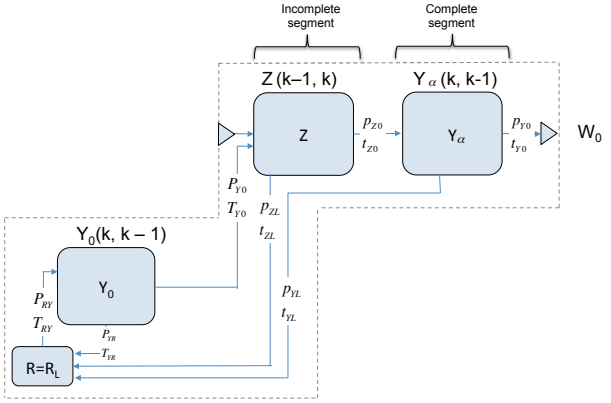



**Fig. 6** General asynchronous model structure: $W(k)$ state

as asynchronous checkpoints. With synchronous checkpointing, the checkpoint becomes available at the completion of the corresponding compute state. Thus, if an L1 failure occurs, the application transitions to the most recent L1 recovery state (Figure 5(b)). On the other hand, with asynchronous checkpointing, the L2 checkpoint is not finished when the L2 compute state completes. Therefore, compute states can be divided into two segment types, *incomplete segments* and *complete segments*. A computation state in an incomplete segment represents a period where the L2 checkpoint has started but is not yet complete. For example, if an L2 failure occurs in incomplete segment 2, the application transitions to the recovery state for the last completed L2 checkpoint (L2-1 in Figure 5(c)). When an L2 failure occurs in a complete segment 3, the application transitions to the recovery state for the completed L2 checkpoint (L2-2 in Figure 5(c)).

In Figure 6, we show the general structure of our asynchronous checkpointing model. Our model is composed of hierarchical states, with the outermost state denoted as a $W(k)$ *state*. The definition of $Z(k,c)$ and its expected probabilities and times for transition to other states were defined in the original exisiting model [8]. In incomplete segments, interference from competing asynchronous operations can inflate the time in compute states before transitions; therefore, we extend the definition of $Y(k,c)$ to reflect the interference. We introduce an overhead factor, $\alpha$, which quantifies the overhead induced upon compute states in incomplete segments by asynchronous checkpointing. We define the time spent in compute states in an incomplete segment as $(1+\alpha)t$, where $t$ is the sum of the computation time and the time to complete an L1 checkpoint. Thus, the expected probability and time for compute states in $Y_\alpha(k,c)$, become $p_0(T)$, $t_0(T)$, $p_i(T)$ and $t_i(T)$ where $T = (1+\alpha)t + c_c$. Using the model, we can compute the *expected time* to complete a given number of compute states with an arbitrary number of checkpointing levels.

## 4. Evaluation

In this section, we compare our asynchronous checkpointing system to the synchronous, multi-level checkpointing system presented in [3]. For illustration, we model a two-level system where the first level uses SSD on the compute nodes with a type of RAID5 redundancy scheme and the second level is a PFS.

### 4.1 Tuning of asynchronous checkpointing

Checkpoint efficiency highly depends on I/O throughput, so it is important to tune I/O operations such that the staging nodes fully exploit the available I/O performance. Generally, I/O throughput to a PFS can be accelerated by writing with multiple threads, and so we designed the staging server process to be multi-threaded. Our goal is to find the optimal numbers of threads and staging nodes such that we can obtain near peak performance from the PFS. As a target PFS, we use Lustre file system [9] on TSUBAME2.0. A TSUBAME2.0 node has two sockets of Intel Xeon X5670, 58GB of DDR3 1333MHz memory, 120GB of local SSD and three Tesla M2050 GPUs. The nodes are connected through Dual-Rail QDR Infiniband(x4). Figure 7 shows write throughput of one staging node in with different numbers of *data writer threads*. The result shows that we achieve the highest performance of 1.6 GB/s on a single stager node when using 16 threads. We then explore how many staging nodes can exploit the Lustre file system. Figure 8 presents aggregate write throughput with different numbers of staging nodes all using 16 data writer threads. The result shows that aggregate write throughput rapidly increases from 1 to 32 staging nodes, but then quickly saturates around 8 GB/s beyond 32 staging nodes. Based on these results, we choose 32 staging nodes and set the staging server to run with 16 data writer threads as an optimal configuration. Under this condition, checkpoint data can be transferred to the PFS at a rate of 6.4 GB/s via 32 staging nodes, which is only 2.3% of TSUBAME2.0 thin nodes (1408 nodes).

Whenever the staging client and server processes read checkpoint data from compute nodes in the background, a measurable amount of overhead is added to the application runtime due to resource contention, and the degree of this overhead depends on the read rate. To estimate this overhead, we transferred checkpoints while running the Himeno benchmark [10] as a target application. This benchmark solves Poisson's equation using the Jacobi iteration method. The Himeno benchmark is a stencil application in which each grid point is repeatedly updated using only neighbor points in a domain. Such a computational pattern frequently appears in numerical simulation codes for solving partial differential equations. Many fluid dynamics phenomena can be described by partial differential equations over multi-dimensional Cartesian grids, including weather, seismic waves, heat flow, and electric charge and magnetic field distribution in a domain.

Figure 9 shows the overhead factor imposed on the Himeno benchmark while varying the checkpoint read rate of a staging node. The result shows that the overhead factor roughly increases linearly with the read rate. Based on the result, we model the overhead factor ($\alpha$) of the Himeno benchmark as $\alpha = cx$ where $c$ is 0.008768, and $x$ represents the checkpoint read rate of a staging node in units of GB/s. The parameter $c$ is derived from the slope of the fitting line in Figure 9. With 32 staging nodes, we calculate the read rate per staging node to be 209.5 MB/seconds, which is derived by dividing the aggregate write throughput when using 32 nodes in Figure 8 by 32, the number of staging nodes. Thus, the overhead factor model gives us the overhead factor of 0.00184 (= 0.008768×0.20905). We add this overhead to the cost of computation in our model when computing efficiency whenver transferring checkpoint data in the background.

With the SCR library, an application can adjust a degree of resiliency by changing the number of processes in each XOR

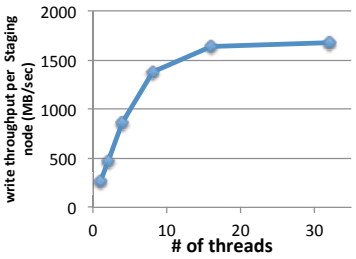



**Fig. 7** Write throughputs under varying number of data write thread

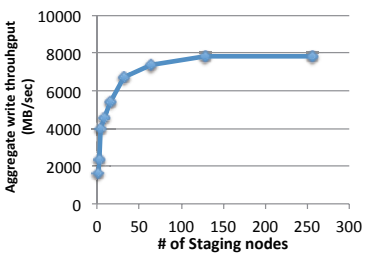



**Fig. 8** Aggregate write throughputs under different number of nodes

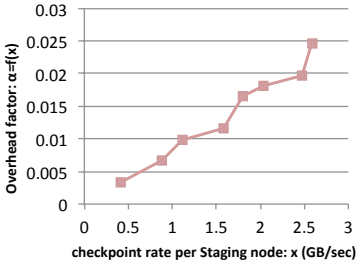



**Fig. 9** overhead factor under varying checkpoint rate

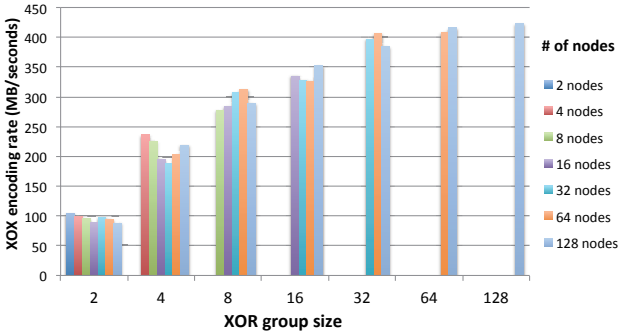



**Fig. 10** XOR encoding performance under varying # of nodes and XOR group size

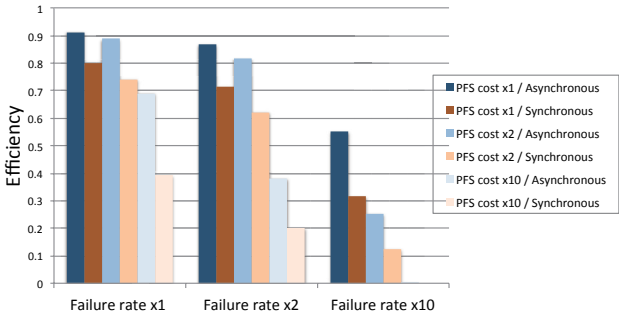



**Fig. 11** Efficiency comparison: Synchronous vs. Asynchronous checkpointing

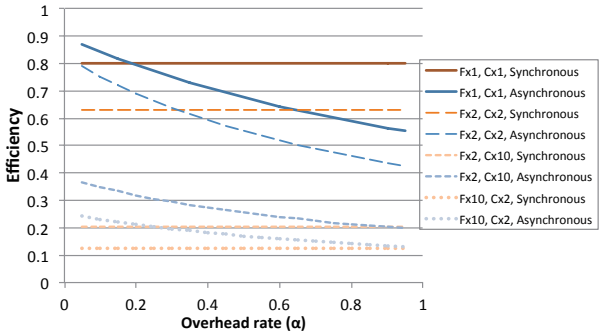



**Fig. 12** Efficiency under varying the overhead factor: $\alpha$

group used to compute redundancy data. Figure 10 shows encoding throughput for different XOR group sizes. Resiliency is improved with smaller XOR groups, but at the cost of decreased XOR encoding throughput. On a XOR checkpoint, SCR compute a parity of each block in the same way as RAID-5 [11], [12], and $S = B + \frac{B}{N-1}$ bytes of encoded checkpoint data is created from $B$ bytes of original checkpoint data within $N$ members of a XOR group. Since the encoding time increase linearly with the encoded checkpoint data size, $S$, XOR encoding rate is saturated in the large XOR group size, $N$. Actually most of failures affect just one node[1]. Therefore, we use XOR checkpoint to handle one node failure failure, and we handle the rest failures by a PFS checkpoint. Thus, we set XOR encoding rate (L1 checkpoint rate) as the saurated maximal rate, 400MB/s.

### 4.2 Efficiency comparison

As future systems become larger and have more memory size, failure rates and checkpoint size are expected to increase. To explore the effects, we increase failure rates and checkpoint costs by factors of 1, 2, and 10, and compare efficiency between both a synchronous checkpointing and an asynchronous one. As for base checkpoint size per compute node, we employ 29GB, which is just a half of memory size of TSUBAME2.0 thin nodes. As show in Figure 10, a XOR encoding rate is constant regardless of the number of compute nodes, which means XOR encoding scales with system size. Thus, when we increase checkpoint costs, we increase only PFS checkpoint cost.

Figure 11 shows that *efficiency* of both checkpointing methods under different failure rates and checkpoint costs. We define the *efficiency* as $\frac{ideal\_time}{expected\_time}$. Here, *ideal_time* is the runtime assuming the application encounters no failures and take no checkpoints, while *expected_time* is the expected runtime computed from our model for an asynchronous method and an existing model [3] for a synchronous one. When we compute the efficiency, we optimize (1) Level 1 counts between Level 2 checkpoints, and (2) the interval between checkpoints, given failure rates and checkpoint costs. The efficiency can be *maximal* efficiency. We found that the asynchronous method achieves higher efficiency than a synchronous method in any cases. Especially, the efficiency gap become more apparent in higher failure rate and higher checkpoint cost because longer PFS checkpoint time on a synchronous checkpointing is easy to encounter a lower level failure during the PFS checkpoint, and rollback to the beginning, while an asynchronous method can rollback to the recent XOR checkpoint. Moreover, since overhead of a synchronous checkpoint is identical to checkpoint latency, which is directlly added to an application runtime, the efficiency become lower than an asynchronous checkpointing.

Because an asynchronous checkpointing overlaps with an application computation, the checkpointing method can imapct the application runtime depending on overhead factor, $\alpha$, in different applications. If the overhead factor becomes larger, our asyn-

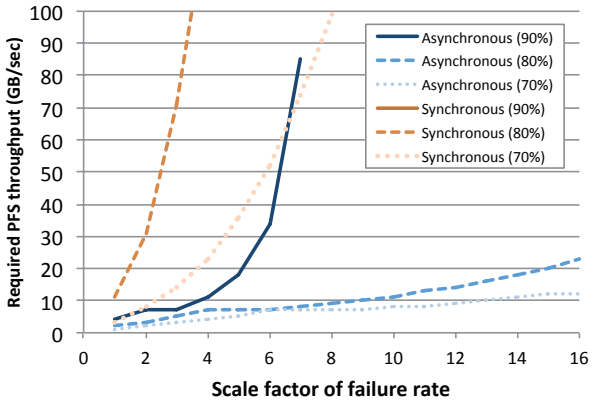



**Fig. 13** Required PFS throughput at different failure rates

chronous checkpointing can introduce lower efficiency than a synchronous checkpointing. Figure 12 shows efficiency of systems with increasing overhead rate in different failure rates factor and PFS checkpoint cost factor. $F$ and $C$ denote the base failure rate and PFS checkpoint cost. We found that a synchronous checkpointing can become more efficient than our asynchronous with larger overhead factor in current failure rates and cost. However, in future systems where the failure rates and cost become larger, an asynchronous checkpointing can be effective even with large overhead factor. In large failure rate and checkpoint cost factors, checkpoint interval become short and the checkpoint overhead dominate to the overall runtime in a synchronous checkpointing. Especially, since an application is blocked with a synchronous checkpointing, the checkpiont latency impacts an application runtime rather than an asynchronous one in future systems.

### 4.3 Building an efficient and resilient system

When building a reliable data center or supercomputer, two major concerns are cost of the PFS and how much throughput a PFS should have to maintain high efficiency. Generally, we want to minimize cost, but not sacrifice performance. Using our model, we can predict the required PFS bandwidth for achieving high system efficiency when using our checkpointing system.

Figure 13 presents the required PFS bandwidth to maintain 90%, 80%, and 70% efficiency under increasing failure rates. The failure rates are scaled from 1× up to 16× today's rates. Because synchronous checkpointing requires extremely high PFS bandwidth to achieve 90% efficiency, the line is not shown in the figure. On the other hand, our asynchronous checkpointing system achieves 90% efficiency with a mere 10 GB/second bandwidth for failure rates that are 1× and 4× today's rates.

Overall, our checkpointing system outperforms synchronous checkpointing. However, at 90% efficiency, the bandwidth requirement rises sharply after a factor of 5. This is because the time for L1 checkpoints begins to dominate application run time due to shortened optimal checkpoint intervals. Here, improvement of PFS bandwidth cannot increase efficiency. However, we found that current levels of PFS throughput are adequate for maintaining 80% and 70% efficiency.

With synchronous checkpointing, systems require higher PFS throughput to minimize the risk of failure during a PFS checkpoint. Moreover, synchronous checkpointing uses the PFS only when a PFS checkpointing is being taken, which means the PFS is not utilized during most of the application run. With our checkpointing system, we not only hide PFS checkpoint overhead, but the PFS is utilized throughout the application execution.

## 5. Related Work

Multi-level checkpointing [3], [4] is a promising technique for fault-tolerant execution. Moody et al. [3] modeled a multi-level checkpointing system and optimized the checkpoint frequency based on collected failure rates and checkpointing costs with a Markov model. We extend their model in this work. Bautista-Gomez et al. [4] proposed multi-level checkpointing using local SSDs and a PFS. They use Reed-Solomon (RS) encoding for highly resilient cached checkpoints to reduce usage of the PFS. Generally, usage of a PFS is costly when compared to local storage, and the PFS is accessed less often in multi-level checkpointing. However, increasing failure rates require checkpoints to a PFS more frequently. Thus, even with multi-level checkpointing, checkpointing to a PFS is crucial for future systems.

Asynchronous I/O has long been used to hide I/O bottlenecks [6], [13], [14] . These techniques enable applications to parallelize I/O with computation, resulting in increased CPU utilization and enhanced I/O performance. Patrick et al. [13] presented a comprehensive study of different techniques of overlapping I/O, communication, and computation, and showed the performance benefits of asynchronous I/O. Nawab et al. [14] asynchronously transfer multiple striped TCP data streams to increase I/O performance in Grid environments. An asynchronous staging service using RDMA proposed by Hasan et al. [6] is the closest research to this work. The authors achieved high I/O throughput by using additional nodes. As we observed, it is necessary to determine the proper number of staging nodes for a given number of compute nodes in order to optimize performance. However, the comprehensive study on the problem is not shown nor do they present their solution.

Optimization of checkpoint interval is critical, because checkpointing is an expensive operation. Several optimization techniques have been studied. Young [15] proposed a method to determine the optimal checkpoint interval for single-level, synchronous checkpoints. Vaidya extended the model to support asynchronous checkpoints, called *fork checkpoint* [16] , and two-level checkpoints [17]. Vaidya also combined both methods to support a two-level asynchronous checkpoint system to achieve higher efficiency [18]. Vaidya's model assumes that at most one fast-level checkpoint is taken between each slow-level checkpoint. However, in current multi-level checkpointing systems, the fastest checkpoints, which are often saved to node-local storage, are orders of magnitude faster than the slowest checkpoints saved to the PFS. To account for this, we extend prior work to model an arbitrary number of node-local checkpoints between consecutive PFS checkpoints.

## 6. Conclusion

We have designed and modeled a asynchronous checkpointing system as an extension to an existing multi-level checkpointing

system. Our asynchronous checkpointing system enables applications to save checkpoints to fast, scalable storage located on the compute nodes and then continue with their execution while dedicated staging nodes copy the checkpoint to the PFS in the background. This capability simultaneously increases machine efficiency and decreases bandwidth required from the PFS. Since applications spend less time in defensive I/O, we find that our asynchronous checkpointing system can improve machine efficiency by 1.1 to 2.0 times on future systems. Furthermore, our model predicts that asynchronous checkpointing significantly reduces the bandwidth required from PFS to maintain 80% of machine efficiency on systems whose faiure rate is 10 times higher than current peta-scale systems

## Acknowledgements

This work was supported by Grant-in-Aid for Research Fellow of the Japan Society for the Promotion of Science (JSPS Fellows) 24008253.

### References

[1] TSUBAME 2.0 - MONITORING PORTAL, http://mon.g.gsic.titech.ac.jp/.
[2] Schroeder, B. and Gibson, G. A. Understanding failures in petascale computers, *Journal of Physics: Conference Series*, Vol. 78, No. 1, pp. 012022+ (online), DOI: 10.1088/1742-6596/78/1/012022 (2007).
[3] Moody, A., Bronevetsky, G., Mohror, K. and de Supinski, B. R. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System, *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, Washington, DC, USA, IEEE Computer Society, pp. 1–11 (online), DOI: 10.1109/SC.2010.18 (2010).
[4] Bautista-Gomez, L., Komatitsch, D., Maruyama, N., Tsuboi, S., Cappello, F. and Matsuoka, S. FTI: high performance Fault Tolerance Interface for hybrid systems, *Proceedings of the 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, WS, USA (2011).
[5] Borrill, J., Oliker, L., Shalf, J. and Shan, H. Investigation of leading HPC I/O performance using a scientific-application derived benchmark, *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, New York, NY, USA, ACM, pp. 1–12 (online), DOI: 10.1145/1362622.1362636 (2007).
[6] Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K. and Zheng, F. DataStager: scalable data staging services for petascale applications, *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, New York, NY, USA, ACM, pp. 39–48 (online), DOI: 10.1145/1551609.1551618 (2009).
[7] Scalable Checkpoint / Restart Library, http://sourceforge.net/projects/scalablecr/.
[8] Moody, A., Bronevetsky, G., Mohror, K. and de Supinski, B. R. Detailed Modeling, Design, and Evaluation of a Scalable Multi-level Checkpointing System, Technical report, Lawrence Livermore National Laboratory (2010).
[9] Lustre: A scalable, high-performance file system, http://wiki.lustre.org/index.php/Main_Page.
[10] Himeno, R. Himeno benchmark, http://accc.riken.jp/HPC_e/himenobmt_e.html.
[11] Patterson, D., Gibson, G. and Katz, R. A Case for Redundant Arrays of Inexpensive Disks (RAID), *Proceedings of the 1988 ACM SIGMOD Conference on Management of Data* (1988).
[12] Gropp, W., Ross, R. and Miller, N. Providing Efficient I/O Redundancy in MPI Environments, *Lecture Notes in Computer Science, 3241:7786, September 2004. 11th European PVM/MPI Users Group Meeting* (2004).
[13] Patrick, C. M., Son, S. and Kandemir, M. Comparative evaluation of overlap strategies with study of I/O overlap in MPI-IO, *SIGOPS Oper. Syst. Rev.*, Vol. 42, pp. 43–49 (online), DOI: 10.1145/1453775.1453784 (2008).
[14] Ali, N. and Lauria, M. Improving the Performance of Remote I/O Using Asynchronous Primitives, pp. 218–228 (online), DOI: 10.1109/HPDC.2006.1652153.
[15] Young, J. W. A first order approximation to the optimum checkpoint interval, *Commun. ACM*, Vol. 17, pp. 530–531 (online), DOI: 10.1145/361147.361115 (1974).
[16] Vaidya, N. H. On Checkpoint Latency, Technical report, College Station, TX, USA (1995).
[17] Vaidya, N. H. A case for two-level distributed recovery schemes, *SIGMETRICS Perform. Eval. Rev.*, Vol. 23, No. 1, pp. 64–73 (online), DOI: 10.1145/223586.223596 (1995).
[18] Vaidya, N. H. Another Two-Level Failure Recovery Scheme, Technical report, College Station, TX, USA (1994).