The impact of asynchronous I/O in checkpoint-restart workloads.

Hariharan Devarajan[1], Adam Moody[1], Donglai Dai[2], Cameron Stanavige[1],
Elsa Gonsiorowski[1], Marty McFadden[1], Olaf Faaland[1], Greg Kosinovsky[1], and Kathryn Mohror[1]

hariharandev1@llnl.gov, moody20@llnl.gov, d.dai@x-scalesolutions.com, stanavige1@llnl.gov
gonsiorowski1@llnl.gov, mcfadden8@llnl.gov, faaland1@llnl.gov, kosinovsky1@llnl.gov, and kathryn@llnl.gov
[1]Lawrence Livermore National Laboratory
[2]X-ScaleSolutions

***Abstract*—With increasing data production from scientific checkpoint-restart (CR) workloads, data management libraries utilize asynchronous I/O operations to hide the cost of accessing global file systems in high-performance computing (HPC) systems. However, the characteristics of asynchronous operation, such as software and hardware architecture and workload features, dictate the impact of asynchronous I/O operations on checkpoint-restart application resources and asynchronous I/O performance. In this work, we aim to study the impact of these design choices on CR performance, such as CPU, memory, and checkpointing, while also considering the performance of asynchronous I/O operation and the performance variability introduced in the workload. We observe the following three main findings. First, the architecture used for asynchronous I/O must be flexible to manage its resources to reduce application interference while maximizing asynchronous I/O performance. Second, modern MPI-aware schedulers apply an implicit core affinity to each workload, which is helpful for compute-bound applications but can significantly reduce checkpointing performance. Finally, asynchronous I/O designs need to consider the cost of communication between the service and application to reduce the overhead of checkpointing operations. Through this study, we aim to pave the path for future middleware libraries to adopt different designs based on their target workload and HPC system architecture.**

***Index Terms*—Asynchronous I/O, background threads, interference study, IBM burst buffer, checkpoint-restart, SCR**

I. Introduction

Scientific simulation workloads generate an unprecedented amount of data in the form of checkpoints on high-performance computing (HPC) systems. For instance, cosmology applications such as HACC, which simulates several trillions of particles, checkpoints 48 TB (for six coordinates each with 1.1 T particles) of data per time step [1]. In another instance, the Community Earth System Model (CESM) produces 12 PB of raw output data in CMIP6 [2]. However, the performance improvements in CPU, memory, and network throughput are not matched by the I/O subsystem in HPC systems [3]. This results in I/O bandwidth being a major bottleneck in achieving scalability and performance for large-scale simulation applications in HPC systems.

To reduce the I/O bottleneck, modern I/O subsystems are based on deep memory and storage hierarchies that utilize non-volatile memories, node-local or shared burst buffers, and global parallel file systems [4]. Typically, node-local devices have a smaller I/O capacity but larger I/O bandwidth. They are temporarily bound to a job allocation and have low interference among I/O operations performed on them. To effectively utilize the storage hierarchy, data management services use an asynchrnous (async) I/O strategy where the data from the applications is temporarily stored in a faster-but-volatile node-local device and then transferred to the slow-but-durable storage in the background. This approach improves the effective I/O bandwidth as the checkpoint-restart (CR) workloads can continue immediately after interacting with node-local devices [5]. This approach is utilized by a wide range of data management services and middleware libraries such as Scalable Checkpoint/Restart (SCR) [6], Hermes [4], VeloC [7], and Univistor [8] and has shown significant improvements in I/O throughput for many large-scale simulation applications.

However, employing async I/O strategies could potentially introduce resource contention by running background threads to perform async data transfer. A recent study characterized the interference of async operations on memory and CPU on a single node [9]. The study demonstrated initial but promising findings about the trade-off between async I/O and application performance. However, more extensive analysis is required for CR workloads that perform large-scale simulations and frequent checkpointing on the HPC systems. Specifically, we need to study the impact of async I/O in the following three main aspects: a) the impact of async architecture on the application's computation; b) the impact of interference among the async I/O operations performed on different types of node-local devices; and c) the extent of application jitter introduced by async I/O operations. This study would provide empirical evidence for building future async I/O software.

In the context of this study, we use the Scalable Checkpoint/Restart library (SCR), a state-of-the-art software tool, to perform scalable checkpointing. The SCR library is ported to a wide range of large-scale HPC systems and supports async data movement with multiple backend implementations, node-local storage targets, and efficient memory data structures. This makes SCR an ideal candidate for this study. In this study, we aim to bridge the gap in the literature by diving deeper into the impact of async I/O on resources used by

large-scale CR applications. Specifically, we study the impact of async I/O operations on the application itself, node-local devices used for checkpointing, and the performance variability in the underlying system. To this end, we explore three directions on the Lassen machine with up to 128 nodes. First, we explore two different async I/O architectures: per-core background threads and global per-node service. Second, we explore different core-binding policies for applications to understand the slow-down caused by oversubscribing compute nodes for async operations. Finally, we explore different computation and checkpointing configurations to understand the benefits of async checkpointing for various applications. These directions provide critical insights into the best practices for designing async I/O optimizations within CR workloads. The key findings of this study are:

1) Global per-node async service has a lower interference on application resources such as CPU and memory than background per-core threads.
2) Using kernel-space file copying instead of user-space file copying can significantly reduce memory interference.
3) Increasing the number of threads allocated for per-core background threads increases CPU interference and jitter but does not impact memory performance.
4) Faster caching targets such as shared memory (SHM) would give less time for async operations to finish and thus are more susceptible to resource interference.
5) Inefficient communication between the application and async service reduces checkpointing performance.
6) Imbalance between the checkpointing rate and the async I/O rate reduces checkpointing performance.
7) Using core affinity and hyperthreading improves CPU and memory performance for CR applications.
8) Using no core affinity maximizes the performance of checkpointing phase and async I/O operations.
9) Lowering checkpoint frequency reduces resource interference and improves async I/O performance.

II. Background & Related Work

The Scalable Checkpoint/Restart (SCR) library enables large-scale applications to utilize hierarchical storage for achieving high I/O bandwidth for checkpointing and restarting on high-performance computing (HPC) systems. SCR API calls are used to indicate the start and completion of checkpointing phases. The SCR API redirects application I/O to the SCR cache. During the completion API call, SCR initiates transfers from cache to the global file system using the AXL library [10]. One can configure the maximum number of checkpoints SCR can hold in cache using `SCR_CACHE_SIZE`. We can configure SCR such that if we checkpoint fast enough, we can create n outstanding transfers where $n =$ `SCR_CACHE_SIZE`. This mechanism creates multiple outstanding requests for async transfers. Finally, SCR provides users with the capability to asynchronously transfer data to global parallel file systems, such as per-core background threads using pthreads and to IBM's burst buffer [11] APIs.

Middleware libraries perform async I/O to hide the cost of data movement from the workload. Examples of middleware libraries that use async I/O mechanisms are DataElevator [12], Async I/O in HDF5 [13], Async flushing in NetCDF and HDF5 [14], Univistor [8], Hermes [4], HFetch [15], VeloC [7], TensorFlow input pipeline [16], and PyTorch DataLoader [17]. These libraries have demonstrated the benefit of using async I/O on application performance. However, they have not studied the impact of async transfer mechanisms on application resources.

Even though many middleware libraries utilize async I/O, their impact on the workloads is not well studied. For instance, the authors of SCR [7] conclude the need to investigate parallel writers to further improve the I/O performance of async checkpointing. However, they do not study the impact of their async architecture on the application resources. The work close to ours is an interference study by Tseng et. al [9], where the authors investigate the impact of background threads on CPU, memory, and I/O performance. The main difference of our study from this work is the dimensions for exploration. We investigate the impact of different async I/O architectures, different core affinity, and different computation to checkpointing ratios. Additionally, we study the impact of these tests on application resources such as CPU operations, memory operations, checkpointing phase, async I/O performance, and the application's jitter.

III. Experiment Methodology

In this section, we discuss the method, the hardware, and the software setup used in our experiment study in detail.

A. Methodology

Testbed We run the experiments on the Lassen supercomputer at Lawrence Livermore National Laboratory (LLNL) [18]. Lassen is a 23-petaflop IBM Power9 system of which we use 128 nodes. Each node has two 44-core IBM POWER9 CPUs, 256GB RAM, and 1.6 TB NVMe SSD (write bandwidth of 2 GB/s). For global storage, Lassen has a 24 PiB IBM Spectrum Scale file system with a write bandwidth of 160 GB/s.
Software We use SCR version 3.0.1 [19] compiled with GCC 8.3.1 with IBM Spectrum MPI version 2020.08.19. Additionally, we use the IBM Burst Buffer (BB) API to asynchronously move data from the node-local SSD to IBM Spectrum Scale. The pthread implementation and the interface to IBM's BB within SCR are provided by AXL version 0.8.0 [10]. We improve the implementation of AXL to use copy_file_range instead of user-level copy to reduce the memory interference.
Workload We built our own microbenchmark to evaluate and test various aspects of async I/O with SCR. The rationale behind the microbenchmark is to create a controlled environment to test various aspects in isolation and understand its impact in detail. The microbenchmark consists of a simulation step (*ie., timestep*) that is divided into a computation phase and a checkpointing phase. Each checkpointing phase in SCR waits for all processes to reach the phase in that step and then writes a checkpoint in a file-per-process (FPP) fashion.

Each checkpointing phase writes 128MB checkpoint file per process. The microbenchmark consists of the following three features. First is the ability to simulate different computation kernels, namely, CPU and memory kernels. For the CPU kernel, we perform N computation operations on a single array at every step. Similarly, we copy a mutated buffer N times per step for the memory kernel. The N parameter enables us to simulate long vs. short computation cycles by the application. Second, the ability to choose different async I/O backends within SCR. We can switch the checkpointing cache target of SCR using `SCR_CACHE_BASE` environment variable and async I/O backends using `SCR_FLUSH_TYPE` set to `BBAPI` or `PTHREAD` implementation of AXL. Additionally, we can disable async flushing to understand the application's baseline performance without async flush. Third, the ability to set cache size within SCR. We can set different cache sizes within SCR to simulate the pressure on async flushing to meet the application's I/O bandwidth demands.

Measurement We measure the time the application takes in each phase (ie., compute and checkpoint) in a step and the time taken for asynchronously flushing the checkpoint for a step. These timing measurements help us understand the impact of async flushing within our workload. To measure the non-overlapped async I/O time, we add up each async I/O phase and subtract with computation and checkpoint phases of the workload. To measure the application's variability, we add an extra synchronization operation in the computation phase of the benchmark to see the slowest computation across the application nodes. Additionally, we calculate a box plot (noting, min, max, 25, and 75 percentile) for our analysis with the central line showing the average. The comparative percentages used in the analysis are captured using the maximum cost of operations across multiple runs and take an average across scale. We provide a Jupyter notebook to extract detailed results from our microbenchmark logs and analyze the data. In all the results reported in this study, we consider performance metrics for system and workload components, namely, CPU, memory, checkpointing, async transfer, and application variability. We repeat each test case fifteen times, calculate the mean, plot the percentiles as the wiskers of the graph, and note the standard deviations of the top ten samples in the figures. Finally, in these observations, we focus on the corner cases more than average to identify the worst and best possible performance for a given configuration.

B. Asynchronous I/O Support in SCR

In the literature, two different strategies are adopted for performing async I/O operations: per-core background threads and global per-node async service. In the per-core background threads, one or more threads are spawned to handle async operations from each application process on a core. Example middlewares adopting this strategy are DataElevator [12], Async I/O in HDF5 [13], and Univistor [8]. This strategy greatly simplifies the design of the async I/O mechanism, as each background thread only caters to the I/O from the underlying core. However, spawning multiple threads (ie., the application process/thread and async background thread) per core could lead to oversubscription of the compute node, leading to increased CPU and memory operations interference. Alternatively, in the global per-node async service, a process with multiple threads is spawned on every node to serve the async I/O operations from all application processes running on a node. Example middlewares adopting this strategy are Hermes [4], HFetch [15], TensorFlow input pipeline [16], VeloC [7], and PyTorch DataLoader [17]. This strategy improves the resource utilization of the async I/O operations by controlling the number of threads in the service. Additionally, it enables I/O optimizations, such as data aggregations, reduced redundancy of I/O calls, and collective I/O operations across multiple nodes efficiently. However, this strategy is harder to design as various demands on CPU, memory, and I/O capability from the application processes and the background async I/O service must be properly balanced on a per-node basis. However, no quantitative study of overhead on application and checkpointing performance has been reported in the literature yet.

SCR supports both async I/O strategies. The per-core background threads are implemented using `pthread`, where each file transfer spawns a new thread on the core. The global per-node async service is supported via an API for the IBM Burst Buffer software deployed on Lassen. Finally, we can set appropriate cache targets: SHM and SSD within the SCR library to study the impact of async I/O on these targets.

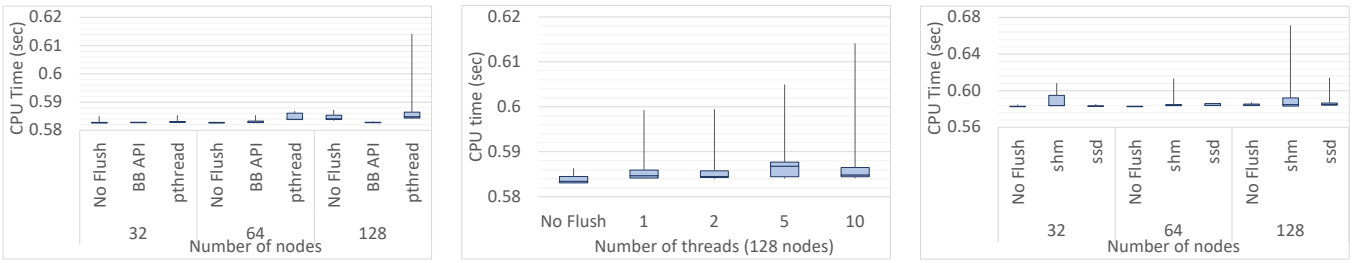
C. Microbenchmarks

We developed three microbenchmark tests to help comprehensively understand the effects of async I/O on application resources from the two strategies. **Test-1**: In this test, the application performs ten async transfers per core. We measure the impact of the async transfer on our test components. Tests are scaled from 32 to 128 nodes with 40 cores per node. We compare the performance of critical resource components with a baseline where no async I/O occurs. In the figures, no async I/O is referred to as "No Flush", IBM's burst buffer is referred to as "BBAPIs", and the per-core background threads implementation is referred to as "pthrd". **Test-2**: This test measures the impact of oversubscribing cores in the per-core background threads strategy. In SCR implementation, if the cache target is full, the application process would be blocked until more space is available in the cache before storing its checkpoint. In this test, the application runs for ten steps, each containing a computation and checkpoint phase. It uses 128 nodes with a varying pool of 1, 2, 5, or 10 threads per core. **Test-3**: This test measures the impact of cache targets on the performance of critical resource components. Either SHM or node-local SSD is used as the cache target. In this test, the application performs ten checkpoints, each issuing an async transfer using the per-core background threads strategy.

IV. Evaluation Results and Discussions

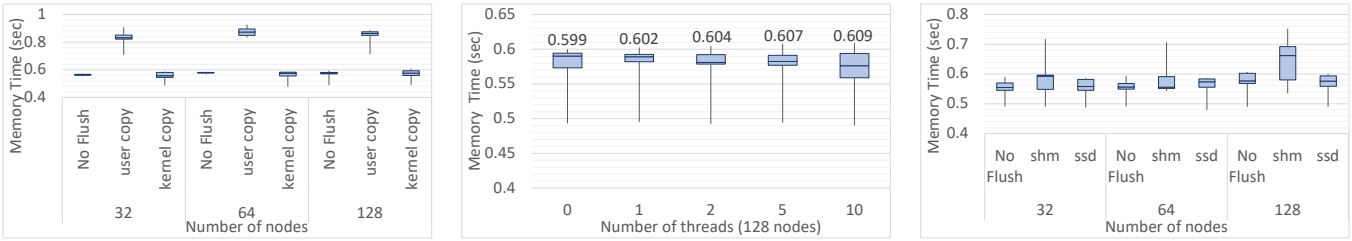
A. Impact of asynchronous I/O architecture

1) CPU interference on application process: In *Test-1*, the maximum interference across all scales on CPU in *pthread* is



(a) BBAPI vs pthreads (b) oversubscribing by pthread (c) SHM vs SSD in pthread

Fig. 1. a) demonstrates that BBAPIs have lower CPU interference than per-core pthreads. b) increasing per-core threads increases CPU interference for applications. c) async I/O transfers from faster cache target increases CPU interference.



(a) User space vs Kernel space (b) oversubscribing by pthread (c) SHM vs SSD in pthread

Fig. 2. a) Copying files through user-space memory introduces significant memory interference. b) Increase threads does not impact memory performance. c) Using SHM for checkpointing reduces memory performance.

4.5% in 128 nodes vs. 0.4% in 64 nodes in IBM's BBAPIs (Fig 1(a)). This is because the pthread implementation spawns up to ten concurrent threads per core. Therefore, there are up to 400 threads on each node leading to constant context switching between the threads. In the case of BBAPI, the service runs on 4 dedicated system cores with a thread pool of four threads per node and thus have minimal interference to the application cores [11]. Additionally, as the scale of the application increases, the variability of CPU time on pthreads increases by 4.9%. This is because a typical checkpointing application synchronizes between every step. Thus, the impact of interference would compound as we increase the scale of the application.

In *Test-2*, increasing thread-pool for pthread implementation increases the interference on CPU by 2.2% with one thread to 4.7% with ten threads (Fig 1(b)). As we increase threads, more parallel async operations get scheduled onto the CPU, thus increasing CPU interference.

In *Test-3*, a slower cache target can reduce interference on CPU by up to 9.3% as seen in the 128 node case between SSD and pthread (Fig 1(c)). Specifically, checkpointing in SHM reduces CPU performance by up to 14.3% as compared to 4.7% for checkpointing in SSD in the 128 node case. This is because slower cache targets give more time for CPU operations while the async threads wait for I/O operations to complete. This reduces the interference incurred by the async threads on the application process. Additionally, as the scale of the application increases from 32 to 128 nodes, such interference increases from 2.2% to 4.7%. Like in *Test-1*, the synchronization between steps compounds the interference cost for more nodes.

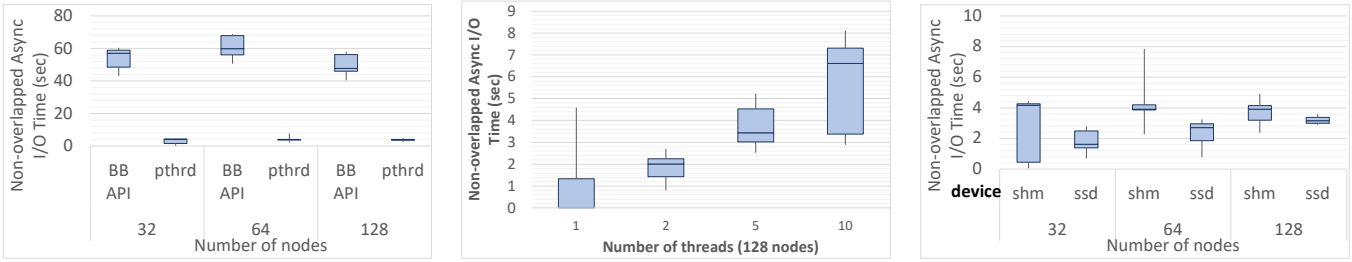
Findings 1: *Global per-node dedicated async service has a lower interference on application's CPU resource than per-core background threads.*

Findings 2: *Increasing number of threads allocated for the per-core approach increases CPU interference.*

Findings 3: *Faster caching targets reduce the I/O wait time in async threads, increasing application's CPU interference.*

2) Memory interference on application process: POSIX supports two copy mechanisms: user-space copy and kernel-space copy (Fig 2(a)). We performed a test with the pthread implementation to use both approaches. We observed that the user space copy is 48% in the 128 node case to 59% in the 64 node case, which is more expensive than the kernel space copy. This is because the pthread implementation uses `copy_file_range` that transfers the file's content in the system kernel memory space without moving it to the user memory space. This approach avoids two extra memory copying operations, resulting in less memory interference. In the case of BBAPI, the transfer happens using NVME over fabric issues from a dedicated BB server. The BB server moves the data from the compute to the IBM Spectrum Scale using PCI point-to-point protocol. The NVMeoF uses host memory to transfer data to the BB server, which introduces some memory interference. Here, the NVMeoF protocol allows for lower variability of interference but higher average interference by 3.7% than the pthread. Additionally, when the scale of the application increases, the variability in the performance of all implementations grows, as made apparent by the synchronization added in each step of the test.

In *Test-2*, as we increase the number of threads for the pthread implementation, the interference on memory is



(a) BBAPI vs pthreads (b) oversubscribing by pthread (c) SHM vs SSD in pthread

Fig. 3. a) The async flush rate should match the application's checkpointing rate to maximize performance while minimizing interference. b) More threads (through increasing cache size) lead to more I/O interference. c) Slower cache targets easily balance the flushing rate with application checkpointing.

between 0.3% with one thread to 1.5% with ten threads (Fig 2(b)). This is because of the efficient implementation of file transfer within pthreads.

In *Test-3*, using SHM for checkpointing would incur an interference of up to 23% for the 128 node case on memory operations (Fig 2(c)). This is expected as both checkpointing and async flushing utilize SHM to transfer checkpoint files. This can be significantly reduced to almost no memory interference if we use SSD checkpointing. However, this would slow down the checkpointing performance.

Findings 4: *User-space file copies significantly increase memory interference for the application.*

Findings 5: *Changing the number of threads used in the per-core background threads strategy does not affect memory interference if we avoid user-space file copying.*

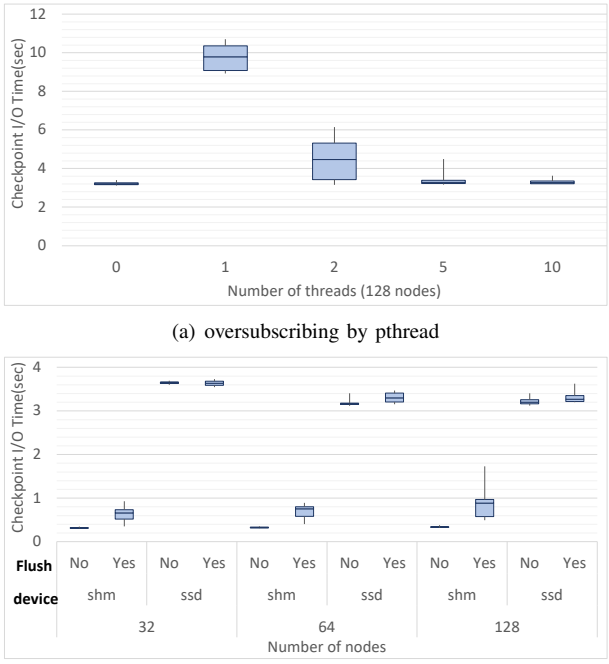
Findings 6: *Using SHM for checkpointing may noticeably reduce memory performance for the application.*

3) Asynchronous I/O performance on application process: In *Test-1*, the non-overlapped async flush time is higher for BBAPI than pthread (Fig 3(a)). This is because the BBAPI on Lassen was configured to throttle its data copy rate to interfere less with a running application, whereas the pthread implementation attempts to copy data as fast as possible. Additionally, scaling the applications has no visible effect on async I/O performance as the I/O pattern mostly writes independent files and scales well with the application. This performance of BBAPIs can potentially be fixed by removing the throttling.

In *Test-2*, the non-overlapped async flush time increases as we increase the number of threads available for flushing (Fig 3(b)). This is because the application gives async flush less time for performing I/O with more outstanding transfer requests. This shows that a mismatch between checkpointing and flushing rates can lead to delays in application completion. With fewer threads, the async I/O is easily hidden behind the waiting of checkpoints, as shown in previous tests.

In *Test-3*, the non-overlapped flush time is smaller on SSD checkpointing than on SHM checkpointing (Fig 3(c)). This is because the checkpointing is slower on SSD, giving the async I/O more time to flush to the parallel file system. Additionally, scaling the application has a limited effect on the non-overlapped I/O time on each cache target.

Findings 7: *Speeding up flushing with threads also increases mismatch between the flushing rate and checkpointing rate.*



(a) oversubscribing by pthread

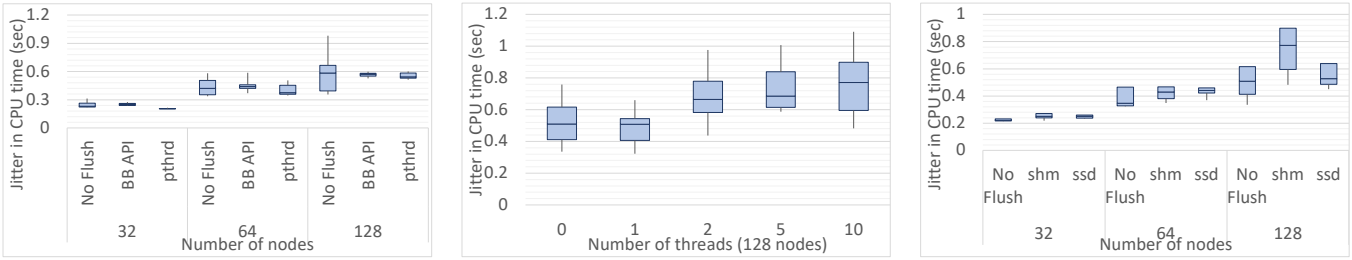
(b) SHM vs SSD in pthread

Fig. 4. a) limiting the checkpointing rate increases the application's checkpoint delays. b) SHM checkpointing has a larger variance but is faster than SSD checkpointing.

Findings 8: *Higher checkpointing rate as compared to flushing rate increases checkpointing wait time required to finish outstanding async operations.*

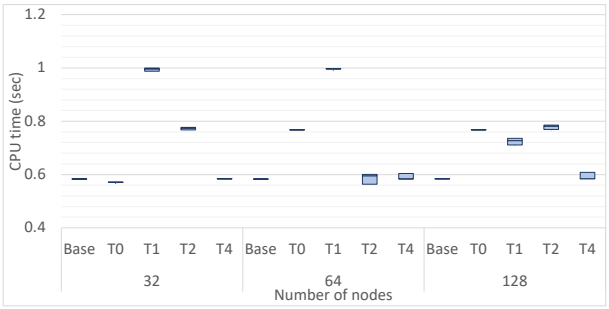
4) Checkpointing I/O interference on application process: In *Test-2*, limiting the number of threads available for performing async I/O slows down the application's checkpointing performance (Fig 4(a)). This is because the application has to wait for async flush to make the cache available for checkpointing. This demonstrates the need for matching the application's checkpointing rate to the async flushing rate, as increasing async flushing can increase interference in the performance of the application.

In *Test-3*, checkpointing in SSD is slower than checkpointing in SHM, as expected (Fig 4(b)). However, the performance variability is much higher for SHM than for

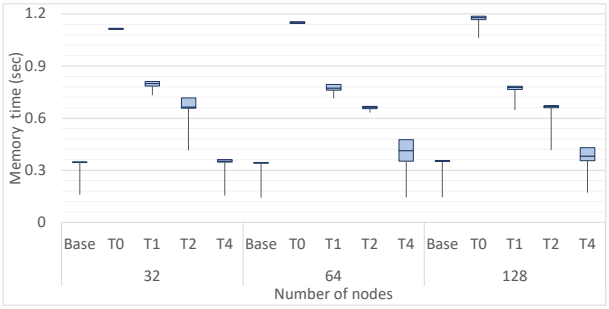


(a) BBAPI vs pthreads (b) oversubscribing by pthread (c) SHM vs SSD in pthread

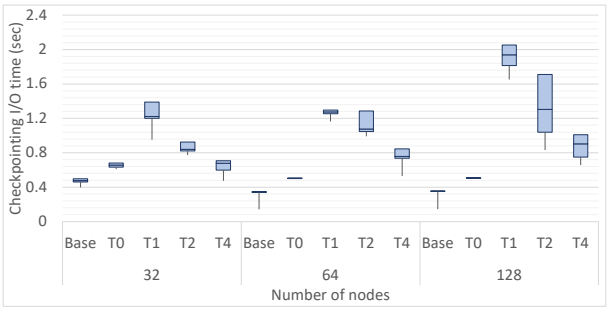
Fig. 5. a) The CPU jitter increases with user-level file transfer than for kernel-level file transfer. b) Increasing background threads increase variability. c) checkpointing on the faster cache target has more application jitter than slower cache target.



(a) CPU Interference



(b) Memory Interference



(c) Checkpoint Interference

Fig. 6. a) For compute-bound applications, hyperthreads reduce CPU interference. b) For memory-bound applications, hyperthreads reduce memory interference. c) for I/O bound applications, no core affinity would maximize I/O performance.

SSD. The fast SHM is more susceptible to interference than a slow caching target like SSD. Additionally, this variability increases as we increase the scale of the application.

Findings 9: *When the application checkpoints at a rate higher than async flush, it causes declined I/O performance and severe interference, resulting in a long delay for checkpointing.*

Findings 10: *Using SHM for checkpointing is more susceptible to performance variability than SSD.*

5) Variability/Jitter on application process: In *Test-1*, we observe the jitter in the computation time of the application increase as we increase the scale of the application (Fig 5(a)). Additionally, the jitter in computation time is up to 2.5× in the 64 node case for per-core background threads strategy (labeled as pthreads) than global per-node async service strategy (labeled as BBAPI). This is because the interference caused by oversubscribing is much higher in pthreads. In *Test-2*, we observe the jitter in computation time of the application increase as we increase the number of threads per core by up to 2.2× for ten threads as compared to one thread per core (Fig 5(b)). Again this is due to the oversubscribing of core in pthreads. In *Test-3*, the jitter is higher by 2.3× in 64 nodes for checkpointing in SHM as compared to checkpointing in SSD (Fig 5(c)). This is because faster cache targets are more susceptible to jitter than slower ones.

Findings 11: *Oversubscribing the cores for async flush increases the jitter in computation time of the application.*

Findings 12: *Faster the checkpointing cache target more the application jitter introduced by async flush.*

B. Impact of core affinity

Scientific applications use MPI job launchers, such as `mpiexec` in MPICH, `srun` in Slurm, and `jsrun` in IBM's LSF scheduler, to scale their applications on multiple nodes. While allocating resources for the application, these scripts assign cores for every `rank` in the application, called as core-affinity of the process or rank. Core affinity defines how processor cores or hyperthreads in a compute node are assigned to the processes (i.e., ranks) of an application. For instance, on the Lassen supercomputer, we have two wrappers for launching an MPI job `jsrun` and `lrun`. The `jsrun` script assigns four hyperthreads per MPI-rank by default, whereas `lrun` assigns one physical core per MPI-rank. As the pthread implementation spawns multiple threads per application MPI rank, the number of hyperthreads available per rank would change the impact of async I/O operations on the application. Note the hyperthreads would only impact per-core background thread architecture as they are spawned within the scope of

the application. The global threads are separate processes with their core allocation and can be changed without changing the application's affinity. Using microbenchmark *Test-2*, we compare the component performance with a baseline where no async I/O occurs. The tests are conducted with different core affinity configurations: 1) **No core affinity:** The application process has core affinity disabled and can be assigned to any core as the system scheduler sees fit. 2) **Core affinity with a given number of hyperthreads enabled:** Each core has either 1, 2, or 4 hyperthreads enabled. Each application process (i.e., MPI-rank) is assigned to one core. In Fig 6, "Base" refers to the case where hyperthread is set to default (four threads per core by `jsrun`) and checkpoint cache is not flushed. This provides an upper bound for best performance. "T0" refers to the case where core-affinity is disabled and async I/O flushing is used. "Tn" refers to the case where n hyperthreads are activated per core and async I/O flushing is used.

1) CPU interference on application process: In the test, increasing hyperthreads enabled per core improves CPU performance but also increases the variability in CPU performance (Fig 6(a)) as each CPU core is shared by multiple hyperthreads. Not using core affinity (i.e., T0), the application is 35% slower than using four hyperthreads (i.e., T4) but has much lower variability in performance. However, even with variability, the slowest T4 is faster than T0.

2) Memory interference on application process: Setting core affinity reduces memory interference compared to not using core affinity (Fig 6(b)). This is because having a fixed mapping makes the cost of using memory banks more consistent and allows the kernel to allocate the memory closest to the core. Additionally, increasing hyperthreads per core reduce memory interference close to almost no interference when using all four hyperthreads (i.e., T4).

Findings 13: *Using core affinity with hyperthreads reduces CPU and memory interference.*

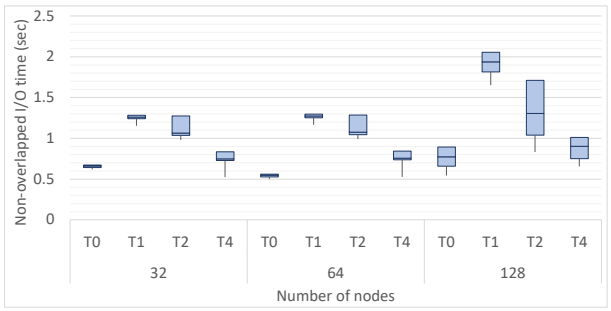
3) Checkpointing I/O interference on application process: Setting core affinity has higher interference on checkpointing cost than not setting any core affinity (Fig 6(c)). This is because not having a core affinity would allow the system kernel to schedule computation tasks when a thread waits for an I/O operation. Additionally, having more hyperthreads reduces the cost of checkpointing as the async flushing has more resources to schedule its operations.

4) Asynchronous I/O performance on application process: Using no core affinity has the best async I/O performance and lowest variance in performance (Fig 7(a)). This is because the async I/O puts the CPU on I/O wait, which allows the CPU to perform other computation tasks. With core affinity, increasing hyperthreads improves the performance of async I/O as more resources are used to perform checkpoint flushing.

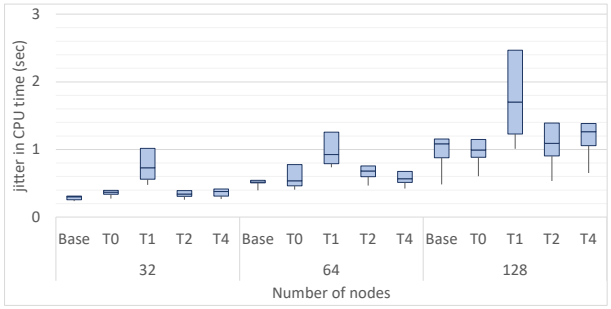
Findings 14: *Using no core affinity reduces checkpointing I/O interference and improve async I/O performance.*

5) Variability/Jitter on application process: The jitter in application performance is the lowest when we use no core affinity (Fig 7(b)). Enabling core affinity increases the jitter in application performance as we increase the scale, with the smallest jitter for four hyperthreads. This is because having more resources available for the system kernel allows processes that are waiting for pending operations to get deprioritized over processes with active computation.

Findings 15: *Using no core affinity reduces the jitter in the application's CPU performance.*

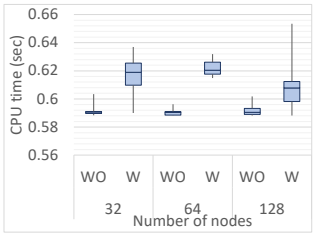


(a) Async I/O Performance

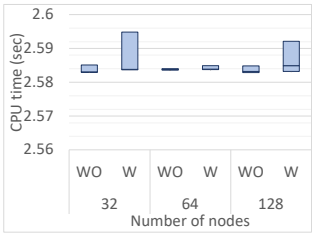


(b) Application Jitter

Fig. 7. a) For improving async I/O performance, the service should not utilize core affinity. b) Having core affinity compounds CPU jitter.



(a) 1:1 ratio

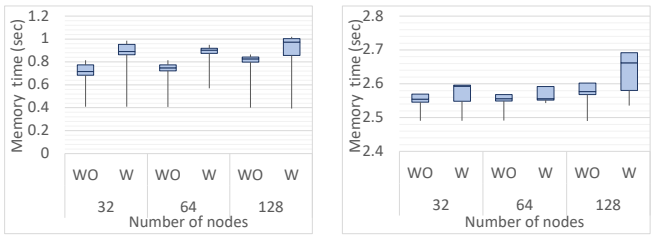


(b) 2:1 ratio

Fig. 8. Higher ratio of compute (i.e. 2:1) reduces the CPU interference.

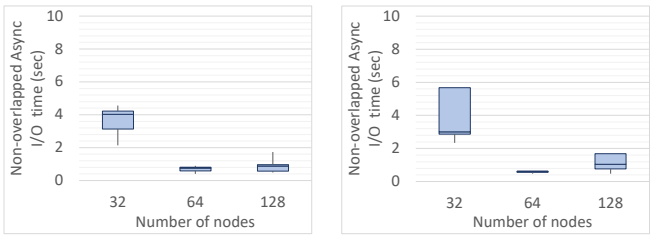
C. Impact of computation to checkpointing ratio

Checkpoint-Restart applications often have two distinct phases: a computation phase and a checkpointing phase. The ratio between computation and checkpointing determines how much time that an async flush has before the next checkpointing phase arises. Ideally, there is enough time between two checkpoints to perform all async I/O operations in the flush. However, this may not be true as the combined time spent in both the computation phase and checkpointing phase (checkpoint to the cache) may be less than the time spent in moving data files from the cache to the global parallel file system. This section studies the impact of async



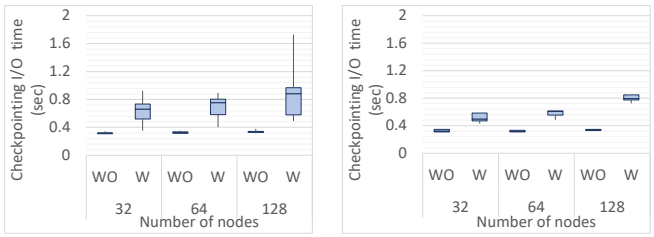
(a) 1:1 ratio (b) 2:1 ratio

Fig. 9. Higher ratio of compute (i.e. 2:1) reduces the memory interference.



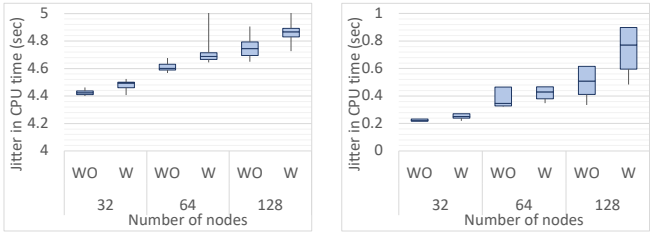
(a) 1:1 ratio (b) 2:1 ratio

Fig. 11. Higher ratio of compute (i.e. 2:1) improves async I/O performance.



(a) 1:1 ratio (b) 2:1 ratio

Fig. 10. Higher ratio of compute (i.e. 2:1) improves checkpoint performance.



(a) 1:1 ratio (b) 2:1 ratio

Fig. 12. Higher ratio of compute (i.e. 2:1) reduces application jitter.

I/O operations on the application with different ratios of the computation phase and checkpointing phase.

Using microbenchmark *Test-2* with up to ten background threads per core, we test two different cases: 1) **1:1 ratio:** the time spent in the computation phase is similar to the time spent in checkpointing phase and **2:1 ratio:** the time spent in the computation phase is twice of the time spent in checkpointing phase. We compare the component performance of the above two cases using a common baseline without cache flush. The baseline provides an ideal upper bound for performance. In Figs 8-12, label "WO" refers to the case of "without flush" and label "W" refers to the case of "with flush".

1) CPU interference on application process: There exists interference on the CPU performance by async I/O for both ratios (Fig 8). However, the variability of CPU performance is much more significant in the case of 1:1 ratio than in the case of 2:1 ratio. This is because a longer computation phase allows more time for async I/O and less contention for CPU.

2) Memory interference on application process: Memory interference is present in the case of 1:1 ratio only (Fig 9). Additionally, similar to CPU interference, the variability of memory performance is much larger in the case of 1:1 ratio than in the case of 2:1 ratio.

3) Checkpointing interference on application process: In both cases, the interference in the checkpointing phase is present (Fig 10). However, the variability in the checkpointing phase is much higher for the case of 1:1 ratio. Additionally, the case of 2:1 ratio has a faster checkpointing phase than the case of 1:1 ratio, as there is much lower interference.

4) Asynchronous I/O performance on application process: Asynchronous I/O operations perform better in the case of 2:1 ratio than in the case of 1:1 ratio, as the former has more time for async I/O operations in the background (Fig 11). Additionally, we see that the variability of async I/O operations is higher in the case of 1:1 ratio.

5) Variability/Jitter on application process: As seen with computation and memory interference, the jitter is significantly higher in the case of 1:1 ratio than in the case of 2:1 ratio (Fig 12). Also, the jitter increases as we increase the number of nodes for both cases.

Findings 16: *Higher compute to checkpoint ratio improves the performance of background async I/O operations and reduces interference on application performance.*

V. Conclusions

Checkpointing with async I/O support can benefit the performance of an application by increasing the effective I/O bandwidth. The characteristics of the checkpoint-restart workload, such as computation to checkpoint ratio, async I/O architecture used, and node-local device characteristics, can significantly impact the overall performance and scalability of the workload. This study identifies three key design considerations to minimize interference while maximizing I/O performance for async I/O operations. First, global async service would have the least interference on the workload with the flexibility of dynamically tuning the rate of async I/O operations. Second, the orchestration and scheduling of async I/O operations are crucial for an application's checkpointing performance. Finally, CPU core affinity increases the performance of compute-bound applications but reduces the performance of I/O-bound checkpoint-restart workloads. The findings in this study provide empirical evidence on implementation choices for future async I/O-based middleware on large-scale HPC systems.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Labo-

ratory under Contract DE-AC52-07NA27344. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under the DOE Early Career Research Program (LLNL-CONF-853236).

REFERENCES

[1] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann, "HACC: Extreme scaling and performance across diverse architectures," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2013, pp. 1–10, iSSN: 2167-4337.

[2] K. Paul, S. Mickelson, J. M. Dennis, H. Xu, and D. Brown, "Lightweight parallel Python tools for earth system modeling workflows," in *2015 IEEE International Conference on Big Data (Big Data)*, Oct. 2015, pp. 1985–1994.

[3] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, Jun. 2015. [Online]. Available: https://dl.acm.org/doi/10.1145/2699414

[4] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 219–230.

[5] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka, "Design and modeling of a non-blocking checkpointing system," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2012, pp. 1–10, iSSN: 2167-4337.

[6] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2010, pp. 1–11, iSSN: 2167-4337.

[7] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2019, pp. 911–920, iSSN: 1530-2075.

[8] T. Wang, S. Byna, B. Dong, and H. Tang, "UniviStor: Integrated Hierarchical and Distributed Storage for HPC," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 134–144, iSSN: 2168-9253.

[9] S.-M. Tseng, B. Nicolae, F. Cappello, and A. Chandramowlishwaran, "Demystifying asynchronous I/O Interference in HPC applications," *The International Journal of High Performance Computing Applications*, vol. 35, no. 4, pp. 391–412, Jul. 2021, publisher: SAGE Publications Ltd STM. [Online]. Available: https://doi.org/10.1177/10943420211016511

[10] "ECP-VeloC/AXL at v0.8.0." [Online]. Available: https://github.com/ECP-VeloC/AXL/tree/v0.8.0

[11] IBM, "CSMD Configuration — CAST 1.5.0 documentation." [Online]. Available: https://tinyurl.com/y6cwj782

[12] B. Dong, S. Byna, K. Wu, Prabhat, H. Johansen, J. N. Johnson, and N. Keen, "Data Elevator: Low-Contention Data Movement in Hierarchical Storage System," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, Dec. 2016, pp. 152–161.

[13] H. Tang, Q. Koziol, S. Byna, J. Mainzer, and T. Li, "Enabling Transparent Asynchronous I/O using Background Threads," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, Nov. 2019, pp. 11–19.

[14] O. Tatebe, K. Hiraga, and H. Ohtsuji, "I/O-Aware Flushing for HPC Caching Filesystem | IEEE Conference Publication | IEEE Xplore." Santa Fe, NM: IEEE, Oct. 2023, pp. 11–17. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10321937

[15] H. Devarajan, A. Kougkas, and X.-H. Sun, "HFetch: Hierarchical Data Prefetching for Scientific Workflows in Multi-Tiered Storage Environments," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2020, pp. 62–72, iSSN: 1530-2075.

[16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," 2016, pp. 265–283.

[17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., 2019.

[18] "Lassen High Performance Computing." [Online]. Available: https://hpc.llnl.gov/hardware/platforms/lassen

[19] LLNL, "LLNL/scr at v3.0.1." [Online]. Available: https://github.com/LLNL/scr