Ph.D. DISSERTATION

I/O Performance Optimization Schemes for Manycore HPC Systems

매니코어 초고성능 컴퓨팅 시스템 환경에서의 I/O 성능 최적화 방안

August 2023

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Bang Jiwoo

I/O Performance Optimization Schemes for Manycore HPC Systems

매니코어 초고성능 컴퓨팅 시스템 환경에서의 I/O 성능 최적화 방안

지도교수 엄현상

이 논문을 공학박사 학위논문으로 제출함

2023 년 6 월

서울대학교 대학원

컴퓨터 공학부

방지우

방지우의 공학박사 학위논문을 인준함

2023 년 6 월

위 원 장	유 승 주	(인)
부위원장	엄 현 상	(인)
위 원	이 재 욱	(인)
위 원	이 재 환	(인)
위 원	성 한 울	(인)

Abstract

High-performance computing (HPC) systems are composed of thousands of compute nodes, storage systems, and high-speed networks, which provide multiple layers of I/O stacks with high complexity. To meet the increasing demand for data access performance in applications run on HPC systems, an efficient design of the HPC memory management system and storage file system is becoming more important. Moreover, HPC users need to be properly guided with optimal system configuration settings to avoid significant fluctuations in performance.

In this dissertation, our first focus is on reducing lock contention on the memory management system of an HPC manycore architecture. One of the critical sections that causes severe lock contention in the I/O path is the page management system, which uses multiple Least Recently Used (LRU) lists with a single lock instance. To solve this problem, we propose the Finer-LRU scheme, which optimizes the page reclamation process by splitting LRU lists into multiple sub-lists, each with its lock instance. Our evaluation result shows that the Finer-LRU scheme can improve sequential write throughput by 57.03% and reduce latency by 98.94% compared to the baseline Linux kernel version 5.2.8 in the Intel Knights Landing (KNL) architecture.

We also analyze the root cause of low I/O performance on a ZFS-based Lustre file system and propose a novel ZFS scheme, dynamic-ZFS, which combines two optimization approaches. The experimental results show that our approach can improve the sequential I/O performance by an average of 37%. We demonstrate that dynamic-ZFS can deliver I/O performance comparable to that of

ldiskfs-based Lustre while still providing a multitude of beneficial features.

Finally, we employ multiple machine learning approaches to perform an in-depth analysis of I/O behaviors in HPC applications and to search for optimal configuration settings for jobs sharing similar I/O characteristics. Improved by a maximum of 0.07 R-squared score, our overall results show that jobs run on HPC systems can obtain the predicted I/O performance for different configuration parameters with high accuracy using the proposed machine learning-based prediction models.

Keywords: High Performance Computing, Manycore Architecture, Fine-grained Lock, Lustre File System, ZFS, Unsupervised Learning, Prediction Model
Student Number: 2017-25955

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

1.1.1 High Performance Computing Systems

High-performance computing (HPC) systems have become essential for processing complicated scientific tasks involving massive amounts of data. These systems are composed of thousands of compute nodes connected by high-speed networks, and backend servers with parallel file systems are utilized for scalable storage. The distributed parallel computation feature of HPC systems allows for improved energy consumption and computational performance for complex workloads.

Due to the distributed parallel computation feature of this architecture, complicated workloads can benefit from this architecture in terms of energy consumption and computational performance. One way to further upgrade HPC systems and provide higher performance is by increasing the number of cores in each processor. Multi-core processors, which refer to a scale-out architecture,

have two or more cores on a single chip [1]. However, the processing demand of state-of-the-art systems is growing rapidly, and the conventional approach of using a multi-core design is no longer fast enough in the HPC environment. To achieve more powerful HPC systems, manycore designs with tens of cores packed within a single chip are adopted [2]. Compared to the multi-core architecture, manycore architecture can efficiently demonstrate a higher degree of parallelism, thus providing higher performance. In addition, each physical core can be integrated via a simple design with lower power consumption [3–6].

Fast storage devices such as non-volatile memory express (NVMe) solid-state drives (SSD) have also gained attention in recent years in HPC storage systems, as they offer low flash cost [7, 8]. As the I/O capability that storage devices can deliver has grown, the performance that parallel file systems provide has become increasingly important in recent HPC systems. Lustre parallel file system [9] is one of the representative file systems in HPC environments and is widely chosen for the storage layer of the world's leading supercomputing platforms. The powerful supercomputer systems that rank top ten in the TOP500 list [10], such as Frontier [11] and Summit [12] in Oak Ridge National Laboratory, Supercomputer Fugaku [13] in RIKEN Center, and Perlmutter [14] in Lawrence Berkeley National Laboratory NERSC, feature Lustre-based distributed storage systems for high-performance scalable file systems. The benefits of using Lustre, such as being optimized for a large range of various HPC applications, attract great attention in the HPC community.

1.1.2 Problems

There are several challenges associated with running applications on an HPC system platform. The followings are the three problems to be addressed in this dissertation.

Limited I/O Scalability. The first problem is the limited scalability of the manycore architecture. As a result of multiple threads simultaneously accessing a shared data structure during runtime, several critical sections in the kernel I/O path have to be protected to guarantee atomic access and prevent synchronization problems. Although the Linux kernel provides protection for shared data structures using various locks and allows a single thread to access the resource, this ultimately affects the overall system performance due to the long latency of blocked threads. Thus, the actual performance does not scale linearly with an increase in the number of cores [15].

Several studies have proposed solutions to handle this problem, such as lock-free implementation [16, 17] or per-socket locking mechanisms [18]. For example, [16] proposed a new block layer that parallelizes the I/O serving tasks in multi-SSD volumes, avoiding the limitation of locking semantics that prevent scalable parallel I/O performance. The first practical lock-free and wait-free algorithms were implemented in [17] using compare-and-swap (CAS) primitive on lock-based linked lists. [18] designed NUMA-aware locks using per-socket data structures and strategies that can lower the scheduler overhead. They also implemented a new locking mechanism that incorporates various policy enforcement such as NUMA-awareness and an efficient wake-up strategy in lock design. Similar to the previous works, our approach tries to mitigate the locking overhead in the contended data structure in Linux kernel. Above all, we figure out that the critical I/O performance degradation problem shown in HPC system is mainly due to contention in the page reclamation process.

Inefficient Storage I/O Stack. The second problem is the inefficiency of storage I/O stack. Currently, Lustre supports two underlying backend file systems, ldiskfs and ZFS [19]. ZFS, unlike traditional file systems, combines the roles of a volume manager and a file system. Its key features include data

integrity and zettabyte scale storage capacity. To ensure data integrity and minimize risks, ZFS uses a 256-bit secure hash algorithm (SHA) checksum on every ZFS block, thereby enabling a self-healing mechanism. Although ZFS has many features that can potentially improve storage performance, its overhead in maintaining these features results in lower performance than ldiskfs. Therefore, most of the current Lustre file systems deployed in HPC platforms use ldiskfs as their backend file system.

To analyze the core reason for the lower I/O performance of ZFS-based Lustre compared to ldiskfs-based Lustre, we perform an in-depth analysis of the latency of the I/O stacks in ZFS. Based on our analysis, we figure that the checksum feature provided by ZFS to ensure end-to-end data integrity [20] is the primary cause of high CPU usage. We also observe that, to accelerate the checksum calculation, ZFS uses hundreds of ZFS threads, resulting in a high load of CPU context switches. In this dissertation, we assert that an optimized design of the ZFS-based Lustre file system can maximize I/O performance with low latency NVMe SSDs and be the next-generation HPC storage file system solution.

Lack of Guidance for HPC Configuration Settings. HPC applications often experience poor I/O performance because of several reasons. As the massive amount of output and checkpoint files are read or written to the storage system through complex I/O software stacks, the bandwidth can be limited by the contention among multiple layers of software and hardware. In addition to bottleneck in the I/O stack, shared resource contention can adversely affect job execution time. Significant fluctuations in performance can occur occasionally depending on the number of jobs and their computational or I/O loads that are executed in parallel. The periodic software upgrade or malfunctioning hardware can also attribute to the I/O performance degradation [21].

One of the solutions for mitigating the problem is to make use of system configuration settings when the users submit their jobs. The HPC systems provide multiple tunable parameters that users can adjust via the resource scheduler. The parameters include the amount of computation and storage resources that can change the degree of parallelism in application execution. Unfortunately, the users often suffer from lack of knowledge in figuring out the optimal configuration setting, since each parameter has extremely large range of values. The growing diversity of the HPC workloads makes it more difficult to search for the configuration setting that can be highly complementary with the I/O characteristics of the workloads. The proper guidance with which to help the HPC users decide the optimal configuration setting can improve both application performance and resource utilization in the supercomputer systems.

There have been numerous works that make use of configurability in the HPC environment, in order to get the maximum performance of the applications. [22,23] find the optimal configuration settings by using supervised, semi-supervised or unsupervised learning models in HPC environments. [24,25] focus on characterizing the I/O behaviors of the HPC workloads in order to provide better insights on the performance. Specifically, [26–28] predict the I/O performance of petascale file systems using various machine learning techniques.

1.2 Contributions

In this dissertation, our contributions are summarized as follows:

- We present a new approach that reduces lock contention on the critical path of the I/O operation and improves the performance scalability in HPC manycore systems. We observe that in Knights Landing (KNL) processor, one of the most commonly used manycore architecture in the

supercomputer systems, the performance does not scale proportionally with the number of threads issuing I/O operations. Our thorough analysis of the critical path of I/O operations shows that the performance drop is mainly due to the severe lock contention on the memory subsystem when tens to hundreds of threads process the I/O operations concurrently. Among the several critical sections lying on the I/O path, the page reclamation process accounts for large proportion of the high latency. In order to reduce the overhead, we develop Finer-LRU scheme which divides a large critical section into multiple small sections [29].

- We introduce two design principles that can be applied to ZFS and improve I/O performance. First, we add a parallelized scheme to the current ZFS I/O pipeline so that percentage of time taken by checksum calculation to total I/O latency can be reduced. We create new task queue threads that are responsible for executing checksum calculation only and allow the threads to run in parallel with other I/O worker threads. Second, we introduce a dynamic thread control scheme to reduce the CPU overhead and to take advantage of concurrent execution with a reasonable number of ZFS threads. We create a context switch monitoring tool that captures the current CPU load of the Lustre file system servers. Then, the current CPU load is used to determine whether to dispatch another thread that handles the checksum operation and operate in a parallel ZFS I/O pipeline scheme [30].

- We propose machine learning-based approach that can help get a better insight into understanding the I/O characteristics of the HPC applications and to figure out the configurations that can result in the improved performance. We leverage the unsupervised learning model to reveal com-

plex I/O patterns and relationships that the HPC applications have on performance. By building prediction models based on the clustering results, the predicted I/O performance with various configuration settings on different jobs can be obtained with improved prediction accuracy [31].

1.3 Outline

This dissertation is structured as follows:

- **Chapter 2** covers the background about the manycore system architecture and Lustre backend file systems.
- **Chapter 3** introduces `Finer-LRU`, our fine-grained HPC memory subsystem scheme. We first explain the I/O scalability problem of existing manycore architecture and propose our new architecture. We describe the details of design and implementation of our scheme and evaluate our scheme on the Linux kernel version 5.4.8 in the KNL architecture.
- **Chapter 4** introduces `dynamic-ZFS`, an optimized ZFS-based Lustre file system scheme. We start with explaining the problems of existing ZFS-based Lustre file system and analyze the root cause of low I/O performance. We give details of how we can address the challenge and evaluate our scheme compared to ldiskfs-based Lustre file system.
- **Chapter 5** introduces `ML-based methodology`, which predict I/O performance of HPC workloads by performing an in-depth analysis on I/O behaviors of HPC applications. We collect the I/O related information from the real-world log dataset and present our approach to construct I/O performance prediction models.
- **Chapter 6** summarizes and concludes the dissertation.

Chapter 2

Background

This paper covers challenges associated with achieving high I/O performance for applications running on HPC systems (i.e., limited I/O scalability, inefficient storage I/O stack, and lack of guidance for HPC configuration settings). In this chapter, we introduce the HPC manycore system and the Lustre file system, which is the most widely used parallel file system in HPC environments. Additionally, we present the Cori supercomputer system, which we collect the real-world HPC log data from to analyze the I/O characteristics of the HPC jobs. Finally, we summarize related works at the end of this chapter.

2.1 Manycore System

The ongoing trend toward improving concurrency and reducing the power consumption by packing more cores into a chip continues. Intel, the world's largest semiconductor manufacturer [32], recently produced its latest 4th generation Xeon Platinum Scalable processor with 60 cores in a single computing compo-

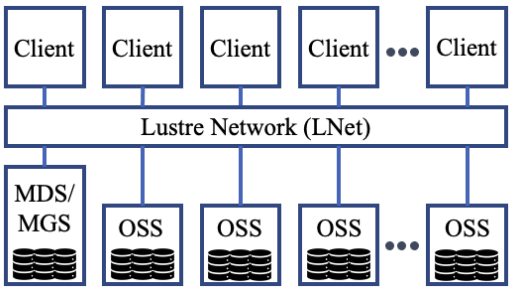

Figure 2.1: Lustre file system architecture.

nent [33]. The Intel Xeon Phi Knights Landing (KNL) is also one of the most widely used x86 manycore processor in HPC systems. The KNL architecture consists of a total of 36 tiles with two cores on each interconnected by a 2D mesh structure. The KNL processor can have up to 72 cores with the ability of running four threads per core, which enables the parallel execution of a total of 288 threads. The KNL is also equipped with a Multi-Channel DRAM (MCDRAM), which is used as the on-package memory for higher bandwidth and capacity. The MCDRAM is configured as a memory side cache and is located between the CPU and DDR DRAM in the default configuration.

2.2 Lustre File System

Lustre file system is one of the most popular distributed file systems that dominates the market share of supercomputer systems nowadays. More than 75% of the world's top 100 supercomputer systems feature Lustre, which shows that Lustre is currently a leading storage file system that meets the high I/O performance demand in HPC systems.

Lustre file system consists of Lustre client, object storage server (OSS), metadata server (MDS), and management server (MGS). Lustre clients main-

tain a namespace of the data stored in the file system to applications using POSIX interface, whereas OSSes provide scalable high-speed I/O performance by distributing data to multiple object storage targets (OSTs). MDS stores metadata of the file system, such as inodes, directories, and file layout, to efficiently handle data allocation and namespace operations on OSSes. Similar to the OSS, the MDS can maintain multiple metadata storage targets (MDTs) that can improve scalability. MGS manages configuration logs for Lustre file systems that servers and clients retrieve when the system starts [34].

Applications can access the data via Lustre client by directly communicating with OSSes after the file is opened and until it is closed. Lustre client interacts with MDS only when the file is opened and closed in order to locate the data. The overall architecture of Lustre is shown in Figure 2.1.

2.3 Cori Supercomputer

Supercomputer Cori system, a Cray X40, has been delivered since 2017 at National Energy Research Scientific Computing Center (NERSC) [35]. Cori is comprised of 2,388 Intel Xeon Hasweell processor nodes and 9,688 Intel Xeon Phi Knights Landing nodes. In addition, Cori also has 1.8TB Cray Data Warp Burst Buffer with a performance of 1.7TB/s, which user can use by specifying APIs. All the nodes are connected with Cray Aries high-speed inter-node network and Dragon fly topology. For efficiently handling parallel I/O, Cori uses Lustre scratch file system as its disk-based storage system. Lustre file system consists of 248 OSSs including 41 HDDs and 248 OSTs, providing total 27TB of storage with peak performance of 744GB/s. When HPC users submit their job using Slurm workload manager, they can specify the amount of resources used to run their applications. For example, they can specify the number of pro-

cessors or storage nodes and whether to use Burst Buffer while running their jobs.

2.4 Related Work

Limited I/O Scalability. This dissertation first addresses the limited I/O scalability problem of the manycore architecture due to the coarse-grained locking mechanism used in several critical sections. Numerous works have attempted to reduce and minimize lock contention in single shared resources in the Linux kernel communities [36]. Studies related to our work take two major approaches: 1) applying a more efficient lock mechanism, 2) optimizing the thread scheduling policy to minimize lock contention and 3) taking lock-free synchronization approach to remove the lock contention.

One of the approaches to ease lock contention is to change the locking mechanism in a fine-grained manner. The trade-off between the fine-grained and coarse-grained locks has been debated ever since the introduction of the lock system. Similarly to our approach, some studies found it more efficient to have fine-grained locks in manycore environments [37, 38] since sequential execution using a coarse-grained lock cannot fully exploit parallelism [39–41]. Zheng et al. [42] partitioned the global cache into many independent page sets in order to eliminate the lock contention in multicore systems. Zhang et al. [43] presented a refactoring tool which can automatically convert coarse-grained locks into fine-grained locks at the application level. The study shows that the throughput of the real-world applications could be improved by using the refactor locks. Our study presents that the I/O performance can be significantly improved by converting the highly contended coarse-grained lock structure into fine-grained multiple locks at the kernel level. Moreover, there have been several studies

to make use of a multiple-granularity locking mechanism [44, 45]. Ganesh et al. [46] proposed a combined locking technique of both fine-grained and coarse-grained locks by identifying each thread's granularity requirements. However, this method can impose another overhead at runtime—traversing the threads to figure out the optimal lock mechanism.

Another approach to decrease lock contention makes use of optimizing the scheduler algorithm to minimize contention. Other than directly modifying the lock structure, researchers chose to change the thread scheduling policy to minimize the contention between multiple threads. If contention is unavoidable, it is better to prioritize a job acquiring a lock over a job that is waiting for a lock for higher throughput. Amer et al. [47] proposed a two-level priority locking protocol, which allows the lock-acquired job to select from several locking protocols to be used at each priority level. Cai et al. [48] suggested tScale, a contention-aware threading framework, to alleviate excessive system calls and contention-unconscious thread scheduler. Nisar et al. [49] introduced Jumbler, a scheduler that mitigates lock contention by mapping the threads of a multi-threaded application into the same socket. Since Jumbler is only optimized for the threads that can share and utilize the last-level cache, I/O-intensive workload with frequent cache miss cannot obtain any performance benefits.

Lock-free algorithms can also enhance throughput and latency. As more threads cause more contention, lock-free algorithms have the advantage of resolving contention with less waits for multi-threads. One approach is to use clock-based approximations instead of using LRU algorithm to reduce the lock contention [50, 51]. However, since the clock-based algorithms can have only limited history information, replacing the data structure in the page management process can degrade the caching performance. SPECK [52] provided a lock-less design with less overhead and scalable predictability. To control access

to the system without a lock, it has to maintain additional resource tables. Inspired by lock-free data structures [17, 53], Son et al. [54], proposed lock-free data structures and operations to reduce lock contention by allowing multiple threads to access the data structures concurrently. In this work, the I/O scheme was introduced for performing parallel I/O operations with multiple threads on page cache layer.

Inefficient Storage I/O Stack. The second problem of this dissertation is to tackle the inefficient storage I/O stack of ZFS, one of backend file systems used in Lustre file system. Several studies have been conducted regarding performance analysis on ZFS [20, 55–60]. Mohr et al. [55] showed the relative benefits of using ZFS on SSD-based Lustre file systems by comparing ZFS and ldiskfs backend file systems, whereas Phromchana et al. [57] focused on the logical volume manager(LVM) feature of ZFS by comparing ZFS and LVM (with ext4) under various configurations. The effect ZFS has on Lustre file system performance was also examined in [61]. Gurjar et al. [59, 60] compared the I/O performance of ZFS and BTRFS using different real-world applications on different frameworks. Zhang et al. [20] analyzed end-to-end data integrity feature provided by ZFS. To further investigate the reliability mechanism of ZFS, Qiao et al. [56] characterized the failure recovery performance varying ZFS configurations and figures out factors that influence ZFS performance. Another feature ZFS provides, data compression, was tested using different compression algorithms in Widianto et al. [58]. Previous studies have analyzed the multitude of features provided by ZFS and have shown the usefulness of layering ZFS on a Lustre file system. In line with the previous studies, our goal is to improve and show the I/O performance benefits of using ZFS on Lustre.

Both ZFS and Lustre provide a large number of user-configurable parameters to give users a chance to further increase the application performance.

In our work, we show that one of the parameters ZFS kernel module provides controls the number of ZFS I/O worker threads, which significantly affects the I/O performance. Several approaches have helped users automatically find best system parameters without the burden of exploring parameter space [62–65]. You et al. [64] presented mathematical models that can find the optimal configuration setting on Lustre for high I/O performance. By combining I/O performance prediction model and run-time monitoring, Bagbaba et al. [65] further improved the I/O bandwidth of the Lustre file system. Behzad et al. [62, 63] demonstrated the effectiveness of applying an autotuning system on HPC platforms and HDF5 applications. Although dynamic-ZFS does not tune ZFS kernel module parameter, we show that the alternative approach of combining thread scheduling mechanism and run-time CPU load monitoring can improve the I/O performance dynamically.

Our approach is partly in line with previous works regarding the thread scheduling [66, 67]. Li et al. [66] applied the dynamic thread scheduling approach on HPC applications to improve parallelism and thread scalability. Goponenko et al. [67] performed real-time utilization of resources when scheduling workloads on HPC clusters. Dynamic-ZFS also uses run-time CPU load information to schedule ZFS I/O tasks to the threads, while the number of threads is adjusted dynamically.

Another contribution of our work is parallelizing ZFS I/O pipeline in a way that checksum operations and rest of the I/O operations overlap together. Previous studies related to saving checksum computation overhead include [68–71]. Liu et al. [68] maximized the data transfer and checksum computation time to reduces the overall data transfer time. By pipelining data transfer and data integrity check in block-level, 70% of overall data transfer time with end-to-end integrity verification could be improved. Globus [70, 71] also pipelined

data transfers and checksum computation to reduce the checksum calculation overhead. The FIVER algorithm implemented by Arslan et al. [69] reduced the cost of integrity verification by overlapping transfer and checksum computation processes. Similarly, dynamic-ZFS reduces the checksum calculation overhead time during I/O operations by modifying the order of I/O stages within the ZFS I/O pipeline.

Lack of Guidance for HPC Configuration Settings. Lastly, this dissertation aims to address the issue of the lack of guidance for HPC users on selecting I/O configuration settings when they run the jobs. Many researchers and HPC users often seek ways to achieve optimal performance on HPC systems for each application they employ [72–76]. Kim et al. [72] developed an integrated database for system logs using Darshan and LMT monitoring tools and dynamically selected the most pertinent features for I/O performance from recent logs. They utilized the selected features to automatically choose the best regression algorithm for accurate performance prediction. In our work, we also utilize similar database logs to identify features highly relevant to I/O performance. However, we enhance the accuracy of the prediction models by employing a clustering algorithm based on these relevant features. Furthermore, our work mainly focus on providing users with suggestions for optimal user-configurable parameters that can enhance I/O performance. In order to enhance overall I/O performance for the applications run on HPC systems, it is crucial to analyze logs from the past jobs and search for optimizable features. [73–75] tried to understand the correlation of various features collected by Darshan logs in HPC environment. Gauge [76] provided a web application that calculates unorganized logs of HPC jobs and showed a interactive visualization of the workloads that ran on the HPC system. However, only 61.4% of the data variances are taken into account with PCA. On the other hand, through Min-max feature

selection and SBS, all the features are thoroughly considered in order to get the best clustering result in our work. Also, by utilizing these methods, most of the applications we analyzed gather into a single cluster, such that users can easily look for the cluster they need.

Also, there have been numerous works that have tried to predict the performance of applications to enhance the overall result. Lux et al. [77] analyzed a benchmark with a set of configurations in order to build a prediction model. The study suggested that the multivariate model can accurately predict I/O performance. Other works [26, 78–80] also predicted I/O performance for HPC-scale clusters using various methods. Our work also targets prediction of the performance of the application using characteristics of the HPC workloads. In contrast, our work is focused on analyzing the system using Darshan logs and makes the prediction based on the historical logs collected in the same HPC environment. This enables our work to make a prediction based on I/O behaviors of HPC applications and give suggestions for input parameters for various applications.

Chapter 3

HPC Scalable Memory System Optimization

3.1 Motivation

3.1.1 I/O Scalability of Existing Manycore Architecture

Even though the manycore architecture provides tens of cores for higher parallelism, the performance does not scale well with the number of cores increased. To evaluate the I/O performance on the manycore architecture, we ran IOR benchmark [81] on the Intel Xeon Phi 7250 KNL node with 68 physical cores. Since the KNL supports hyper-threading technology, which allows each physical processor to execute four processes at a same time, a total number of 272 logical cores can be operated concurrently. Figure 4.1 shows the performance result by changing the number of threads doing sequential write operations from 8 to 256. Each thread writes with contiguous bytes of a block size between 64 MB and 1024 MB while the transfer size is fixed to 1 MB.

The throughput decrease starts with a different block size in each case

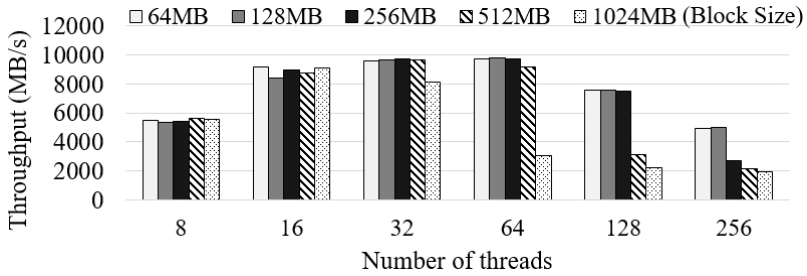

Figure 3.1: The file-per-process sequential write I/O throughput on the Intel Knights Landing node

where more than 32 threads are executing write operations in parallel. The performance degradation is due to the write I/O throttling from the memory management subsystem. In order to preserve free space in memory, the Linux kernel performs the flush operation in advance. When the number of dirty pages in the page cache exceeds 40% of the total number of pages, the data flush daemon and the thread processing write operations start writing out dirty pages to the disk. As we use 96 GB of memory capacity in the KNL node, dirty pages are flushed after writing more than 38 GB of data. This flush policy highly affects the performance, since the write I/O is blocked until the flush operation is finished.

However, in the case of less than 64 threads concurrently processing the write I/O, even though the total I/O size does not exceed the flush threshold, the performance saturates when 16 or more threads are used. Even though the KNL processor allows the concurrent execution of up to 272 threads, there is a severe performance degradation with less than a quarter of the maximum allowed concurrency. While it is a critical problem in HPC environments that the available parallelism is not fully exploited, this problem has not been addressed so far to the best of our knowledge.

LRU lists	Description
LRU_INACTIVE_ANON	Inactive LRU list tracking anonymous and swap pages
LRU_ACTIVE_ANON	Active LRU list tracking anonymous and swap pages
LRU_INACTIVE_FILE	Inactive LRU list tracking file-backed pages
LRU_ACTIVE_FILE	Active LRU list tracking file-backed pages
LRU_UNEVICTABLE	LRU list tracking unevictable pages

Figure 3.2: The description of the LRU lists of tracking page statuses in the memory management system

3.1.2 Page Frame Reclamation Process

When an application issues a write request, the *write()* system call at the user level calls the kernel level *vfs_write()* function to handle the request at the virtual file system. Before going all the way down to the underlying storage device layer where the write request is served, the function first checks whether the requested page is on the cache layer. In order to find whether the desired page exists in the page cache, *pagecache_get_page* is called. If the page is not allocated, *__page_cache_alloc* is executed to allocate a new page frame via the buddy allocator on a proper memory zone. Next, the allocated page is added to the LRU list, which is used to track the statuses of the pages. Finally, the write request can be served on the returned page.

The Linux memory system is dynamically managed by the page frame reclamation process with the LRU algorithm [82, 83]. The main goal of the process is to free the page frame that has not been referenced for a while to make space in memory before it gets exhausted. In order to efficiently keep track of page frames, the kernel groups pages into five different LRU lists as shown in Figure 3.2. If an anonymous page or the swap cache was recently accessed, it is included in *LRU_ACTIVE_ANON*, otherwise, in *LRU_INACTIVE_ANON*. The same ap-

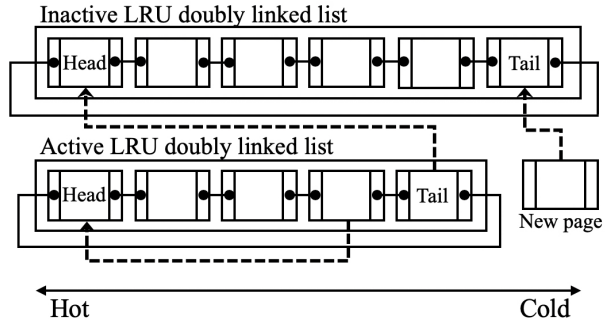

Figure 3.3: The page frame management process on inactive and active LRU lists

plies to the file-backed memory on *LRU_ACTIVE_FILE* and *LRU_INACTIVE_FILE*. If a page cannot be reclaimed, it is included in *LRU_UNEVICTABLE*. Since the LRU lists are frequently accessed in parallel, they are protected by a lock instance by each NUMA node. To ease the contention and reduce the number of accesses to the LRU lists, the pages are buffered in the per-CPU pagevec structure first. Only when the pagevec structure is fully filled, the pages are added or removed from the LRU lists with the lock held.

In the Linux kernel, the LRU list is implemented as a cyclic doubly linked list. A recently accessed page is placed at the head of the list, while one that has not been accessed for a long time is placed at the tail of the list. The current memory management system handles the LRU lists by using the following auxiliary functions:

- add_page_to_lru_list(): Increment the list size and add page to the head of the list.
- add_page_to_lru_list_tail(): Increment the list size and add page to the tail of the list.

- del_page_from_lru_list(): Delete page from the list and decrement the list size.

Using the auxiliary functions, specific operations of the page state transition and page movement across the LRU lists are handled as depicted in Figure 3.3. The pages to be moved are passed to the pagevec data structure, and each of the pages contained in the pagevec structure changes its status by several page movement functions. The function operations include adding a page to the head or tail of the specific LRU list, moving a page from the active list to the inactive list and vice versa. Each of these page movement functions is passed as the callback function to *pagevec_lru_move_fn*.

3.1.3 Problem Analysis

In order to determine the reason of the performance bottleneck, we profile in detail the callback function of *pagevec_lru_move_fn* that lies on the write I/O path. Among several callback functions, *__pagevec_lru_add_fn* is called most frequently in the I/O-intensive workload. This is due to the frequently performed write requests that allocate and assign the new page frames in the proper LRU lists. Figure 3.4 shows the latency and the number of function calls of *__pagevec_lru_add_fn* when executing the file-per-process I/O with the IOR benchmark, varying the block size from 64 MB to 1024 MB. The latency is calculated by adding the function execution time of every thread that executes I/O operations. Similarly, the total latency on the other evaluations is calculated in the same way in the rest of the section. The experiment was conducted on the KNL platform and the number of threads running in parallel was increased from 8 to 256.

The right side of Figure 3.4 indicates that the number of function calls increases linearly with the total I/O size. In contrast, the latency of the page

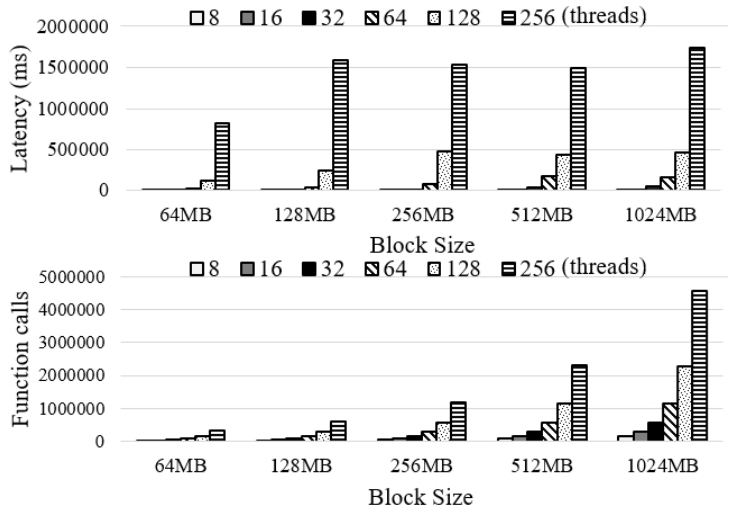

Figure 3.4: The latency and number of function calls of *__pagevec_lru_add_fn* when executing sequential write I/O operations. Each thread writes the block size file in file-per-process mode.

movement process shows exponential growth with increasing the number of concurrently running threads. When each thread writes the same block size, the latency increases up to 8,978 times while the level of parallelism increases. This result is shown when the block size is fixed to 128 MB and the number of threads is increased from 8 to 256, writing the maximum total I/O size of 32 GB. Although there is no flush operation to degrade the overall performance, the latency of adding pages to the LRU lists dramatically increases when running multiple processes simultaneously.

The problem associated with the significantly increased latency is that the whole page movement process across the multiple LRU lists is protected by a single lock instance of the node. In a multiprocessor system, multiple threads concurrently try to allocate and release page frames. As the memory subsystem

frequently accessed by multiple threads is protected by a single lock, it is highly likely that the lock is already acquired by another thread when other threads try to enter the critical section. Since a page frame reclamation process lies on the critical path of the I/O request process, in an HPC environment, it is especially important to overcome the performance bottleneck resulting from severe lock contention.

3.2 Design and Implementation

3.2.1 Design of Scalable Locking Mechanism

The performance degradation in manycore architecture is highly related to the coarse-grained locking mechanism used in the page frame reclamation process. In order to solve this problem, it is necessary to split the coarse-grained lock into multiple fine-grained locks. To meet both the high throughput and low latency goals on the manycore system, we present the Finer-LRU, an advanced memory management scheme designed for achieving higher scalability. The details of the Finer-LRU scheme to handle page frames in the page reclaiming process are as follows:

- The Finer-LRU splits the frequently accessed data structure in the page reclamation process into multiple sub-structures to avoid contention.
- The Finer-LRU also changes the page frame reclamation algorithm from the coarse-grained to the fine-grained locking mechanism by assigning lock instance to each data structure.
- Last, the Finer-LRU can selectively split the contended data structures to efficiently handle the contention.

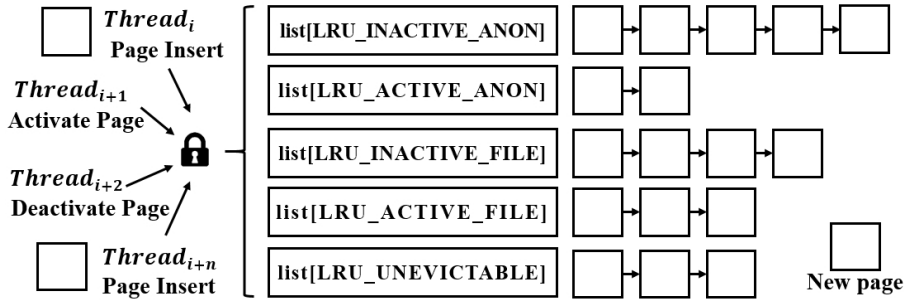

Figure 3.5: The previous design of the LRU list locking mechanism

Figure 3.5 depicts the previous design of the LRU list locking mechanism, while Figure 3.6 shows the Finer-LRU scheme applied to the locking mechanism. Since the Linux kernel manages five LRU lists with a single lock instance, threads have to wait for another thread to release the lock when trying to enter the critical section. The total lock wait time significantly increases, especially in the manycore environment, when hundreds of processes run simultaneously. To ease such an overhead, the Finer-LRU scheme creates a number of sub-list structures in advance. The LRU lists can be selectively split into N sub-lists, with N being a pre-defined value. When the pages to be added or removed from the list come in, the target list is chosen based on the predefined page index. Before entering the critical section, the appropriate lock instance that handles the specific list needs to be held and released after the page movement operation is done. As there are lock instances in charge of protecting each list, threads can easily enter the critical section unless there are other threads operating on the exact same list.

The rest of the section describes the implementation Finer-LRU scheme to improve the I/O scalability based on the Linux kernel version 5.2.8.

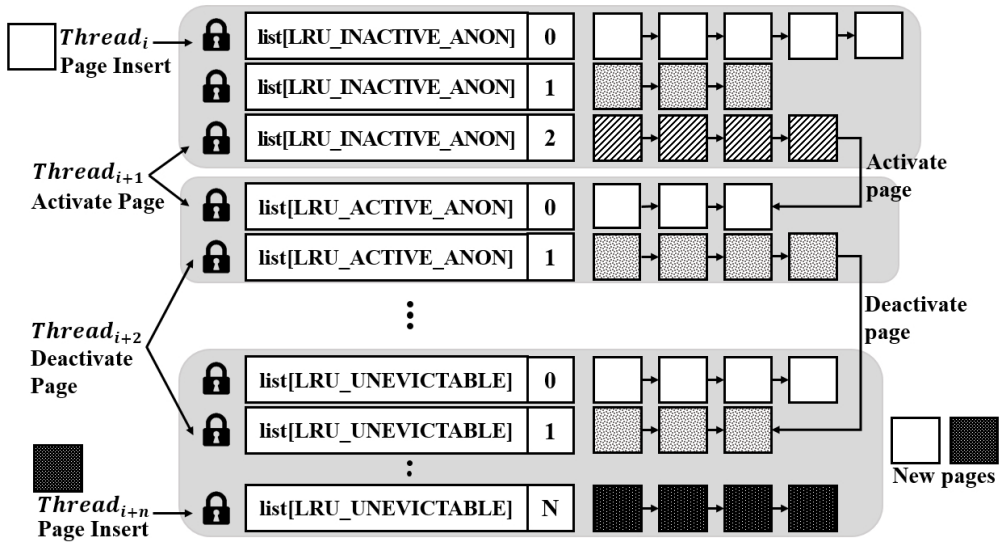

Figure 3.6: The Finer-LRU design of the LRU list locking mechanism

3.2.2 Data Structures

Current Linux kernels manage five LRU lists to keep track of page frame states, protected by a lock on each NUMA node. Since performance degradation is caused by frequent access to the LRU lists, we aim to divide these lists into multiple sub-lists. In order to safely split the lists, we modify two kernel data structures: the lruvec structure and the page structure.

Each node manages all the information about its memory subsystem, including the physical memory status and page distribution. The lruvec structure is one of the data structures used to track the current page status by handling the LRU operations. Whenever the page status changes, the page moves across the LRU lists so that the page reclamation process can easily find the page frame to be reclaimed. In order to handle the page movement between the lists, the lruvec structure maintains the head of each LRU list. In our approach, the lruvec structure has to manage more than five lists. Therefore, we define the

configurable global parameter *LRU_FACTOR*, which represents the multiplication factor of the number of LRU lists. Since the number of current LRU lists is defined as *NR_LRU_LISTS* in the kernel, we change the size of the list head array in the lruvec structure to *LRU_FACTOR* * *NR_LRU_LISTS*. For simplicity, we implicitly use *NR_LRU_LISTS* as five for the rest of the section. If *LRU_FACTOR* is configured to one, it is the same as using the default page management algorithm. In addition to modifying the number of LRU lists, we add spin lock instance for each list so that each LRU list can be protected by its particular lock instance.

After splitting the LRU lists into multiple sub-lists, each page needs to know from which list it has to be added or deleted. To easily determine the source and the destination, the index value is added in the page structure. The index value ranges from zero to *LRU_FACTOR*−1. In order to evenly distribute the number of accesses to multiple lists, the index value is initialized as a random variable generated by modular operation. We first create a random value from the page structure address and then divide it by *LRU_FACTOR*. The remainder is taken as the index value.

3.2.3 Calculation of the LRU list index

When the number of total LRU lists is configured to *LRU_FACTOR* * *NR_LRU_LISTS*, five different LRU lists are allocated sequentially in each iteration during the *LRU_FACTOR* iterations. In order to provide simplicity in getting the target LRU lists where the page needs to be moved, we present an LRU list index function. Given the page index and the LRU index representing the list

type, the function calculates the new LRU index with the following formula :

$$get\_new\_lru\_idx(lru\_idx,\ page\_idx) = \\ lru\_idx + NR\_LRU\_LISTS \times page\_idx \tag{3.1}$$

The page movement functions are executed with the following arguments: the page structure, the lruvec structure, and the type of LRU list. By using these parameters, the LRU list index function is called inside the page movement functions to change the LRU list type parameter into the newly calculated LRU list index. Once it is decided which LRU list is to be handled, the rest of the operations—such as changing the size of the list and relocating the pages—are processed as usual.

3.2.4 Customized Callback Functions

Since we split each LRU list into a number of *LRU_FACTOR* sub-lists and manage them with the proper spin lock instance, the page movement functions need to be redesigned accordingly. The current Linux kernel manages five page movement functions, which are used as callback functions and passed as arguments to *pagevec_lru_move_fn*. To enable a scalable locking mechanism, we remove *pagevec_lru_move_fn* and its callback functions. Instead, we manage each of the page movement functions on its own. In this way, the previous scheme that used the coarse-grained lock to protect the entire function is modified to use multiple fine-grained locks to protect smaller critical sections inside the function. The page movement functions can be divided into two groups, the single-lock scheme and double-lock scheme, based on their algorithms.

The single-lock scheme

The page movement functions that use the single-lock scheme only need a single-lock instance to protect the critical section in their algorithm. Among the current Linux kernel page movement functions, *__pagevec_lru_add_fn* and *pagevec_move_tail_fn* are included in this category. *__pagevec_lru_add_fn* adds a given page to the proper LRU list, while *pagevec_move_tail_fn* relocates a given page to the tail of the LRU list. All the page movement functions are passed with the pagevec structure, which contains a batch of pages as a parameter, and each page movement process is operated in a for loop. In our implementation, the critical section includes the auxiliary functions that handle the LRU list, since the same LRU list can be accessed by multiple threads executing these functions. Therefore, the lock needs to be held right before the auxiliary function is called. However, we do not release the lock immediately after the function is returned. One may think that making critical section smaller can reduce the lock waiting time from other threads, expecting to get a higher performance. But in our algorithm, since each of the pages is checked sequentially in for loop, the lock does not need to be released and held repeatedly if the same lock was already acquired for a previous page. Releasing the lock right after exiting the critical section and acquiring the lock in the next loop also degrades the performance even though the critical section is kept small. The same reasoning applies to the double-lock scheme.

Algorithm 3.1 shows our implementation of the single-lock scheme. The function gets the pagevec structure as its input, and the pages in the structure are sequentially accessed in a for loop. At first, the current status of the page decides to which particular LRU list the page needs to be added. From the given LRU list type, the new LRU index is calculated through the LRU list

Algorithm 3.1: The customized callback function algorithm

```
Input: pagevec structure
1  for i ← 0 to size of pagevec do
2      page = pagevec[i];
3      lruvec = get_lruvec_from_page(page);
       /* previous lru index is decided by the page status        */
4      lru_idx = get_new_lru_idx(lru_idx, page_idx);
5      this_lock = lruvec → lock[lru_idx];
6      if this_lock ≠ previous_lock then
7          unlock(previous_lock);
8          lock(this_lock);
9      end
10     page_movement_auxiliary_function(page);
11     previous_lock = this_lock;
12 end
13 unlock(previous_lock);
14 release_pages(pagevec);
```

index function. Additionally, the proper lock instance is decided by the newly calculated LRU index. Next, the decision of whether to acquire a lock is made by comparing the current lock instance and the previously held lock instance. The previous lock is unlocked, and the thread acquires the current lock instance only when the two instances are different. Finally, the page is moved to the LRU list by calling the auxiliary functions that handle the lists. Since both *__pagevec_lru_add_fn* and *pagevec_move_tail_fn* operate on the same list, the whole process can be protected by a single lock instance, which is responsible for the list.

The double-lock scheme

The double-lock scheme is slightly different from the single-lock scheme, since in the former, the page movement functions have to lock twice in order to protect their operations. The functions *__activate_page*, *lru_deactivate_file_fn*, and *lru_lazyfree_fn* need two kinds of lock instances in their implementation since they move pages from one LRU list to another and each LRU lists needs to be protected. The pages are moved from an inactive to an active list in *__activate_page*, while the opposite process is operated in *lru_deactivate_file_fn*. The function *lru_lazyfree_fn* changes the status of the anonymous page and moves it from the inactive list for anonymous pages to the inactive list for file-backed pages.

The overall double-lock scheme algorithm is similar to the single-lock scheme one, which is delineated in Algorithm 3.1. For each of the pages in the pagevec structure, the two new LRU indexes are calculated using the index and corresponding LRU index. The two lock instances to be acquired in the following critical section are subsequently decided by these newly calculated LRU indexes, respectively, which handle the different LRU lists. The first lock instance is compared to the previously held lock and if they are different, the previously held two locks are released and the two new lock instances are acquired. Finally, in the critical section, the page is moved across the LRU lists. All the functions in the double-lock scheme hold two locks before entering the critical sections and each lock is responsible for the LRU list to which the page is removed or added.

3.3 Evaluation

3.3.1 Experimental Setup

We evaluate our proposed Finer-LRU scheme on the Intel Xeon Phi CPU 7250 @ 1.40 GHz code-named KNL, having a total of 68 physical cores and 272 virtual cores (with hyper-threading enabled) on a single socket. The KNL is equipped with a 16-GB on-package high bandwidth MCDRAM memory with up to 450 GB/s bandwidth and a 96-GB DDR4 main memory with up to 90 GB/s bandwidth. We run all the experiments with the default configuration of the KNL machine, in Quadrant cluster mode and Cache memory mode. For the storage device, we use an 800-GB Intel DC P3700 SSD, with speeds up to 2,800 MB/s and 1,900 MB/s for sequential read and write throughput, respectively. The XFS file system is mounted on the raw device, and all the experiments are run in a buffered I/O mode. On every evaluation, we compare the baseline Linux kernel v5.2.8 with the Finer-LRU scheme applied to the same kernel, varying the *LRU_FACTOR* to split the LRU lists with different granularity. All the experiments are conducted in a file-per-process mode to reduce the I/O bottleneck caused by multiple threads accessing a single shared file protected by a file lock.

3.3.2 I/O Path Latency Evaluation

In order to directly evaluate the impact of applying the Finer-LRU scheme, we profile the latency of the functions on the write I/O path. When the write operation is served, *vfs_write()* is the first function to be called in the kernel level. Next, multiple functions—including *__pagevec_lru_add*, one of the page movement functions—are invoked to handle the write procedure. Figure 3.7 presents the percentage of *__pagevec_lru_add* latency to the *vfs_write()* execution

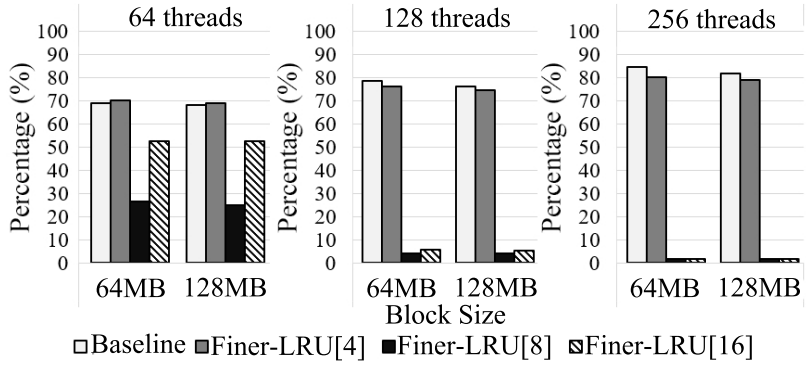

Figure 3.7: The percentage of the page movement latency against the total write operation time

time. We vary the number of threads executing write operations to 64, 128, and 256, and each thread to write a 64-MB and 128-MB block size file with a 1-MB transfer size. We compare the ratio on the baseline Linux kernel with the Finer-LRU scheme applied kernels, changing the *LRU_FACTOR* to 4, 8, and 16, which split all five LRU lists into *LRU_FACTOR* sub-lists.

On the baseline kernel, the latency of multiple pages to be added in the LRU lists occupies most of the time in the write operation time. On every evaluation, the *__pagevec_lru_add* latency takes up more than 68% of the *vfs_write()* execution time. When all 256 cores perform I/O concurrently, the page movement time occupies up to 84.3% of the total write time since a large number of threads have to wait longer to acquire the lock. Even when the lists are split and managed by four different locks each, the ratio is similar to that of the default kernel. This happens because splitting the list lowers the contention level but also increases the locking overhead due to the increased number of locks. When the lists are split into eight different sub-lists, the *__pagevec_lru_add* latency reduces significantly and takes up only 1.48% of the total *vfs_write()* time. The latency in the case where the list is split in 16 sub-lists takes a larger proportion

of the write time than when the list is split in 8. This is because the lock release and acquisition counts increase, as there are too many locks to be handled in the page movement process. Overall, up to 65.85% of the total *vfs_write()* time is saved on average when using the fine-grained locking mechanism from our approach.

3.3.3 I/O Evaluation with IOR

Sequential I/O evaluation

In this section, we evaluate the throughput and latency of running sequential I/O operations on the kernels with the Finer-LRU scheme applied. We run the IOR benchmark using the MPI-IO interface for parallel execution. In order to adjust the parallelism on each evaluation, we change the number of threads doing I/O operations from 32 to 256. Each thread accesses files in the file-per-process mode, reading and writing a block size of 16 MB to 1024 MB depending on the total I/O size. To avoid the performance degradation caused by memory throttling, we keep the total I/O size lower than 40% of the total memory size, which is the threshold value of the flush operation. Since the memory size is 96 GB in our environment, the maximum amount of system memory that can be filled with dirty pages is about 38 GB. Therefore, we select the total write size to 4 GB, 8 GB, 16 GB, and 32 GB and adjust the block size of each thread accordingly.

Figure 3.8 shows the throughput of the threads, each sequentially writing its own block size data. Increasing *LRU_FACTOR* can reduce the contention of threads accessing the same list but also increase the number of releasing lock and new lock acquisition because more locks are being used. Consequently, there is a trade-off between reducing lock contention and increasing the number

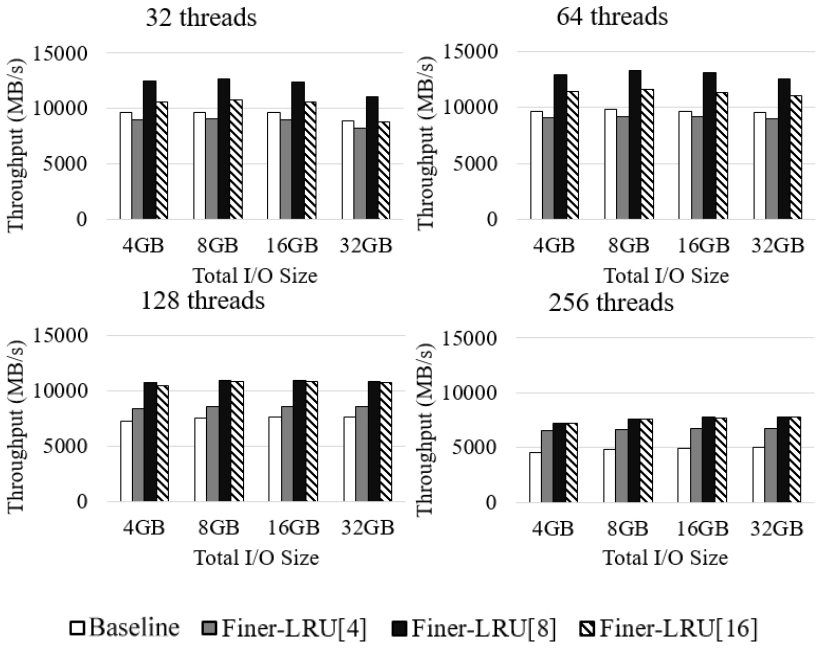

Figure 3.8: The throughput of sequential write I/O on the baseline kernel and the Finer-LRU scheme applied kernels. The number in brackets is *LRU_FACTOR*, equal to the number of sub-lists.

of lock acquisitions. In the case of 32 and 64 threads running the write operations, the performance increases when *LRU_FACTOR* increases from 4 to 8 and then drops when it is 16. When compared to the baseline kernel, only Finer-LRU[4] shows 6.3% lower throughput, while Finer-LRU[8] and Finer-LRU[16] show 21.89% higher throughput. When 128 or more threads write simultaneously, all three Finer-LRU schemes show higher throughput than the baseline kernel. Additionally, the performance is enhanced as more threads execute I/O operations. In other words, splitting the LRU list can efficiently ease the lock contention in the multi-threaded environment of large number of threads concurrently issuing the requests. The performance improves by up to 57.03% compared to the baseline kernel in the case of a total of 256 threads performing

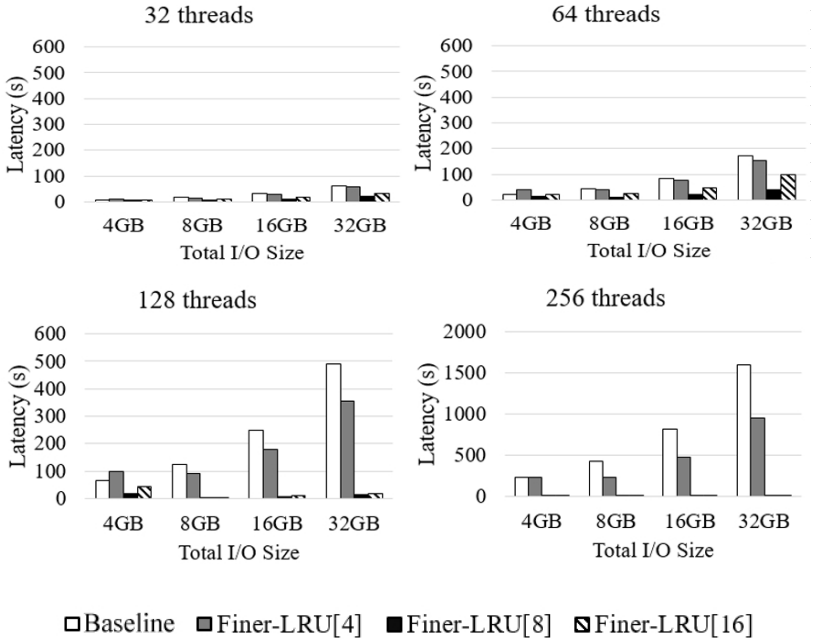

Figure 3.9: The latency of the page movement function while processing the sequential write I/O on the baseline kernel and the Finer-LRU scheme applied kernels. Note that only the scale of the y-axis in the bottom right figure is different because the range of the latency is too large.

write operations.

In the same experiment as in Figure 3.8, the latency of adding a page to the LRU lists is shown in Figure 3.9. The difference between baseline kernel latency and that of the Finer-LRU scheme becomes greater when the total I/O size increases or when the number of threads performing I/O operations increases. In the case of less than 64 threads performing write operations, splitting the list by 16 does not reduce the latency compared to the default kernel because the contention is not heavy enough. In fact, handling multiple locks increases the lock acquisition overhead in this case. In more extreme cases, such as more than 128 threads executing I/O operations, a latency reduction by more than

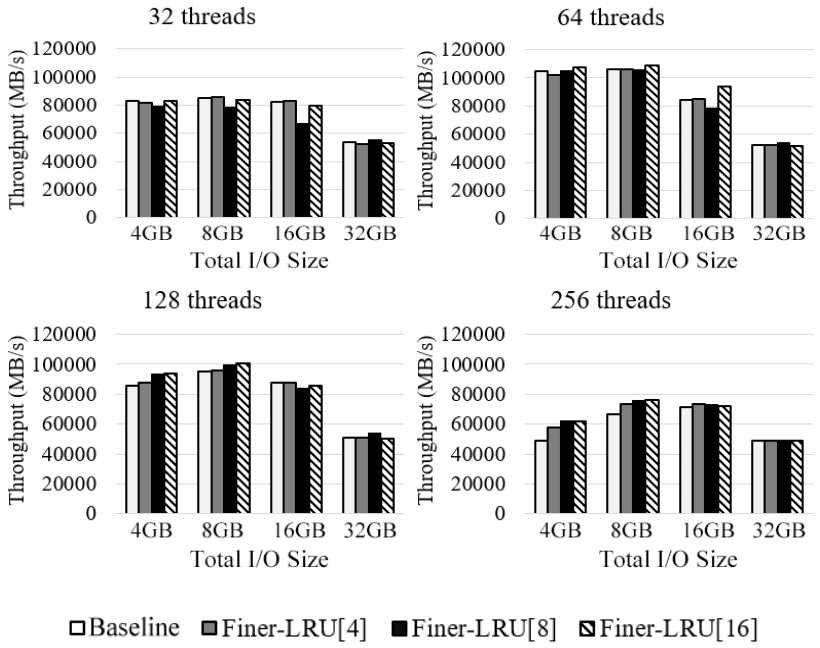

Figure 3.10: The throughput of sequential read I/O on the baseline kernel and the Finer-LRU scheme applied kernels. The number in brackets is *LRU_FACTOR*, equal to the number of sub-lists.

90% is achieved in our approach. When 256 threads perform I/O concurrently, the latency is decreased by up to 98.94%, which greatly reduces the time the processes spent waiting for the lock.

We also evaluate the read performance of the Finer-LRU scheme. Figure 3.10 shows the sequential read throughput on the same evaluation setting as the write I/O experiment. In most cases, the read performance is similar for the different kernels. Since the read I/O is operated on already allocated page cache, the LRU lists are accessed by the threads with low latency compared to the latency in the case of write I/O operation. As a result, the performance does not improve much like in the case of the write I/O evaluation even though the Finer-LRU scheme is applied. The read throughput increases for every case by

up to 25.95% only when 256 threads in total perform I/O operations. The result indicates that the Finer-LRU scheme does not slow down the read I/O path and slightly improves performance when a large number of threads perform I/O simultaneously.

Random I/O evaluation

To evaluate the Finer-LRU scheme on different I/O access patterns, we run the IOR benchmark configured to perform random access I/O. Using the same evaluation setting as for the sequential write performance evaluation, the throughput and the latency of random I/O came out similar to those of sequential I/O, as shown in Figure 3.8 and Figure 3.9, respectively. Finer-LRU[4], Finer-LRU[8], and Finer-LRU[16] show a 7.73%, 37.29%, and 29.99% throughput improvement on average compared to the baseline kernel, respectively, with a 57.08% improvement when 256 threads process random write operations on the Finer-LRU[16] scheme. Moreover, the latency is reduced by 7.82%, 72.4%, and 63.59% on average, with a 99.02% reduction exhibited by the Finer-LRU[8] scheme when 256 threads run concurrently.

Read/write mixed I/O evaluation

Real HPC workloads generally perform both read and write operations simultaneously. To evaluate the performance of the Finer-LRU scheme under a mixed I/O pattern, we run sequential read and write operations together using the IOR benchmark. In this evaluation, a total of 128 and 256 threads are run with total I/O sizes of 8 GB and 16 GB in a file-per-process mode. We adjust the read to write ratio by differentiating the number of threads issuing read and write operations in order to consider both read-intensive and write-intensive workloads. The x-axis of Figure 3.11 shows the ratio of threads issuing read

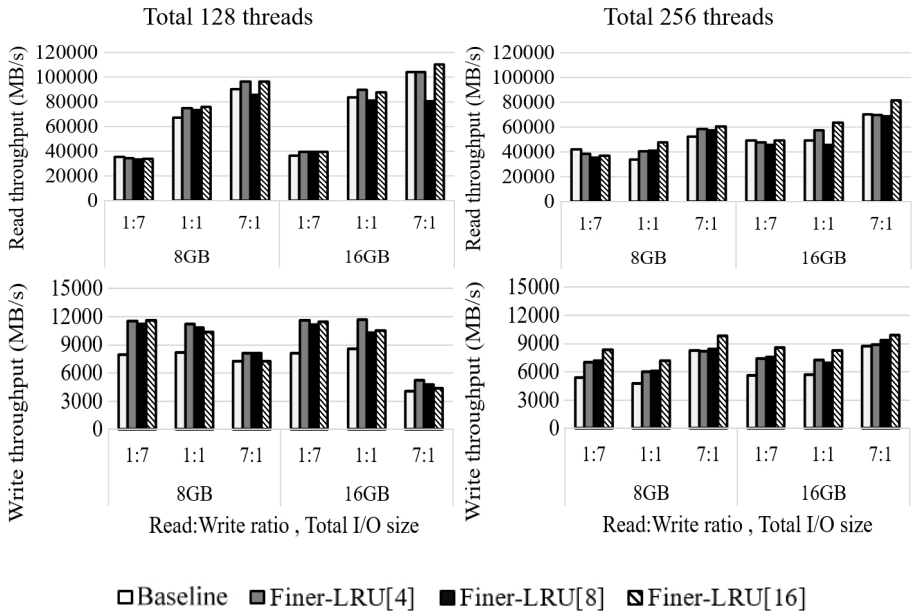

Figure 3.11: The throughput of the sequential read and write I/O in a mixed evaluation varying the read-to-write ratio and total I/O size

operations to threads issuing write operations and total I/O size. We choose the ratios of 1:7, 1:1, and 7:1, which represent the read-intensive, mixed, and write-intensive workloads, respectively.

Figure 3.11 shows the read and write throughput of 128 and 256 threads performing I/O operations. Since these are large numbers of threads issuing I/O operations, the write throughput improves in most cases of the Finer-LRU scheme applied kernels. When compared to the write-only throughput result in Figure 3.8, the performance improvement ratio is similar to the write-intensive scenario where seven out of eight threads are performing write operations. In the read-intensive scenario, the throughput does not increase much and even decreases in some cases when compared to the baseline kernel. The evaluation

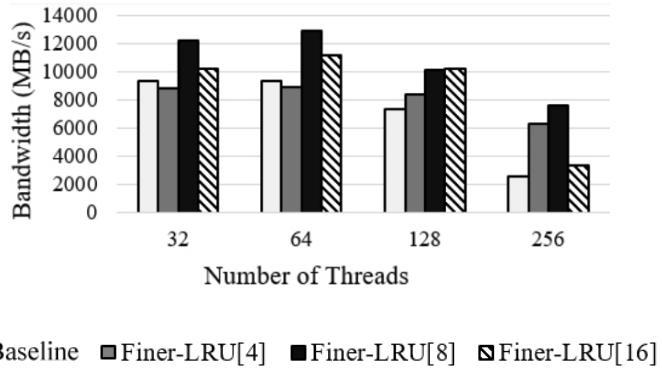

Figure 3.12: The throughput of HACC-IO writing a checkpoint file of 2 760 000 particles per thread

result with mixed I/O indicates that the Finer-LRU scheme can improve the total I/O performance, especially in write-intensive workloads, while the write performance gain is greater than the read performance one.

3.3.4 I/O Evaluation with HACC-IO

The HACC-IO [84] benchmark is an I/O kernel of the cosmology framework HACC (Hardware/Hybrid Accelerated Cosmology Code), which uses N-body techniques to simulate the evolution of the universe. The benchmark is modified to perform only the checkpoint phase of the workload and to write one file per thread in our evaluation. The throughput of the HACC-IO using different schemes and for different numbers of threads is shown in Figure 3.12. Each thread is configured to manage 2 760 000 particles, writing the same number of checkpoint files accordingly. Inevitably, the overall performance decreases when more than 64 threads execute I/O operations, as multiple threads have to share a limited number of physical cores. On the other hand, applying the Finer-LRU scheme provides a higher performance than the baseline kernel when the lists are split into more than 8 sub-lists in every case. In fact, Finer-LRU[4]

increases the overhead of handling multiple numbers of locks in the cases with low contention levels. Finer-LRU[4], Finer-LRU[8], and Finer-LRU[16] show a 36.78%, 75%, and 23.94% throughput improvement on average compared to the baseline kernel, respectively. The performance is improved by up to 194.02% with the Finer-LRU[8] scheme when the LRU lists are most contended—by 256 threads trying to access the shared resources.

3.3.5 Memory Consumption

Finer-LRU scheme reduces contention on kernel memory data structure at the cost of increased memory consumption. In this section, we further examine the extra memory consumption when applying the Finer-LRU scheme. When each LRU list is split into *LRU_FACTOR* number of sub-lists, the lruvec structure maintains total *LRU_FACTOR* * *NR_LRU_LISTS* number of the list head structures and same number of spin lock instances to protect each list. Since the lruvec structure of the baseline kernel already contains *NR_LRU_LISTS* number of the list head structures, the extra memory required by Finer-LRU scheme can be calculated as follows:

$$\{LRU\_FACTOR * (NR\_LRU\_LISTS - 1)\} * L + (LRU\_FACTOR * NR\_LRU\_LISTS) * S$$

where:

L = size of list_head structure

S = size of spinlock_t

After simplifying the equation, the extra memory needed for each lruvec structure in a byte-scale can be calculated as follows:

$$100 * LRU\_FACTOR - 80$$

Note that the lruvec structure exists for every node in memory controller group. We find that when the Finer-LRU scheme is used with *LRU_FACTOR* of 8 and 16, about 150KB and 320KB of extra memory is consumed respectively, which we consider acceptable.

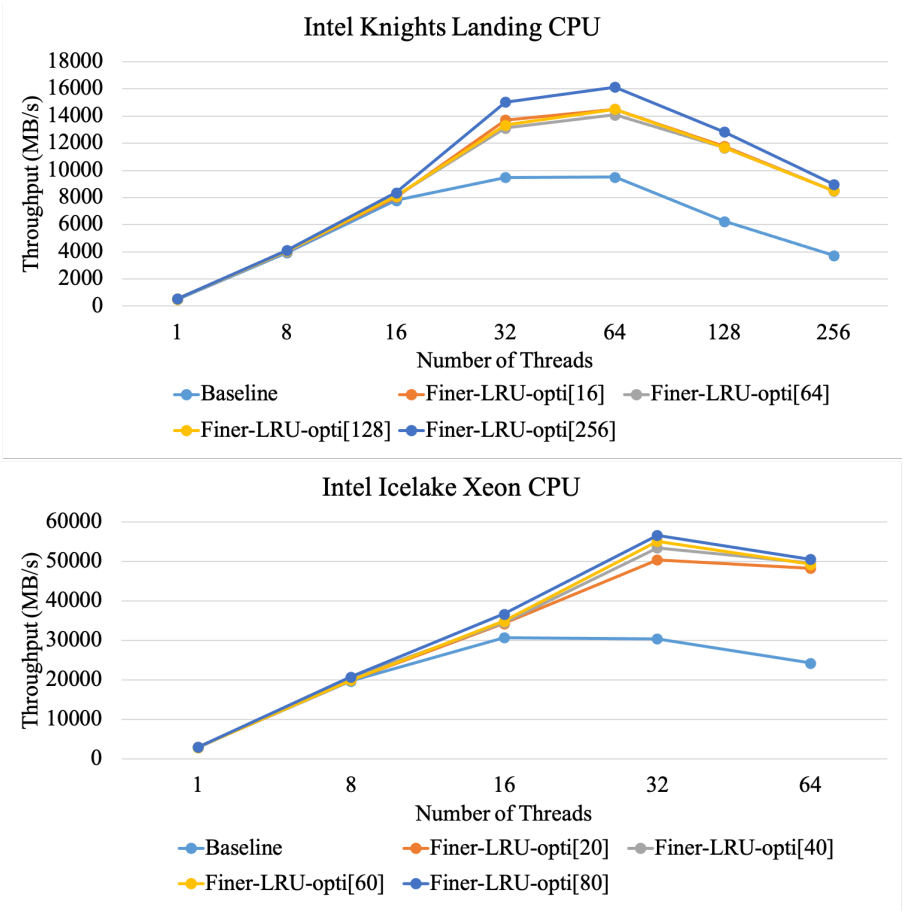

Figure 3.13: The throughput of sequential write I/O on the baseline kernel and the Finer-LRU-opti scheme applied kernels on Intel Knights Landing CPU and Intel Icelake Xeon CPU. The number in brackets is *LRU_FACTOR*, equal to the number of lock instances.

3.3.6 Optimized Finer-LRU scheme

Finer-LRU has limited scalability due to the overhead in handling multiple lock instances. As shown in Figure 3.8, the I/O throughput increases with *LRU_FACTOR* of 8 but drops when the LRU lists are further split. Moreover, a recent version of the Linux kernel includes changes in page movement functions, making direct application of Finer-LRU scheme difficult. For these reasons, we further optimize Finer-LRU scheme based on kernel v5.18.11, which we refer to Finer-LRU-opti for the rest of this section [85].

The Finer-LRU-opti scheme reduces locking overhead by decreasing the number of lock instances. In the Finer-LRU scheme, the number of lock instances is *LRU_FACTOR* * *NR_LRU-_LISTS*, which is the same as the total number of LRU lists. However, we have confirmed that the page movement process always occurs between lists with the same *lru_idx*. Therefore, the Finer-LRU-opti scheme manages a single lock instance for five LRU lists with the same *lru_idx*, instead of managing a lock instance for each individual LRU list. With the Finer-LRU-opti scheme, the page movement functions that utilize the double-lock scheme, such as *__activate_page*, *lru_deactivate_file_fn*, and *lru_lazyfree_fn*, can be handled with a single lock instance as, two LRU lists are managed using the same lock instance.

Figure 3.13 shows the write throughput on two different CPU processors: Intel Knights Landing and Intel Ice Lake Xeon CPU. The single-core performance of the Ice Lake Xeon processor is significantly higher than that of the KNL processor. The Intel Xeon Gold 6338 processor, codenamed Ice Lake, features a total of 32 physical cores and 64 virtual cores. The number in brackets represents *LRU_FACTOR*, which is equal to the number of LRU sub-lists. Since the Finer-LRU-opti scheme manages a single lock instance for four LRU sub-

lists (excluding the *LRU_UNEVICTABLE* list in Figure 3.5), *LRU_FACTOR* * 4 number of LRU sub-lists are handled.

The upper graph in Figure 3.13 illustrates the I/O performance on the KNL node. In contrast to the evaluation results in Figure 3.8, the I/O performance of the Finer-LRU-opti scheme improves with an increased number of *LRU_FACTOR*, demonstrating the effectiveness of the Finer-LRU-opti scheme. The I/O performance on the Intel Ice Lake Xeon CPU is shown at the bottom of Figure 3.13 and exhibits similar results to the performance on the KNL node. As *LRU_FACTOR* increases, the I/O throughput increases by up to 140%.

3.4 Discussion

The Finer-LRU scheme has been developed to address the issue of lock contention on LRU lists utilized by the Linux kernel memory system for tracking page statuses. In the context of reducing locking overhead in the kernel memory system, the fine-grained approach and the non-blocking wait-free approach represent distinct strategies for managing concurrent access to shared resources. While this dissertation focuses on the fine-grained approach for reducing lock contention, we also discuss the performance implications of employing the wait-free approach to alleviate this contention.

The fine-grained approach utilizes a more granular locking mechanism, allowing for individual resources or sections of code to be protected by their own locks [42,43]. This enables simultaneous access by multiple threads or processes to different resources, reducing contention and minimizing wait time for locks. As a result, the overall system throughput is improved. However, it is important to implement the fine-grained locking carefully, as the increased overhead of acquiring and releasing locks at a finer granularity can introduce synchro-

nization overhead and potentially degrade performance. On the other hand, the non-blocking approach aims to minimize or eliminate the need for locks by employing lock-free or wait-free algorithms. These algorithms ensure that threads or processes can progress even in the presence of contention, without relying on traditional locking mechanisms. Consequently, the non-blocking algorithms offer better scalability compared to the fine-grained approach by avoiding lock contention entirely. Furthermore, they can enhance system responsiveness by enabling threads to make progress even if they need to retry operations.

To apply the non-blocking wait-free approach to LRU lists, a lock-free doubly linked list needs to be constructed. However, constructing such a list poses challenges and maintenance difficulties. The fundamental idea behind a lock-free doubly linked list is to design operations that can be executed concurrently by multiple threads without compromising data integrity or the list's consistency. This is achieved through the use of atomic operations, typically compare-and-swap (CAS), which enable modifications to be performed atomically and offer a means to handle concurrent updates. However, the doubly linked list necessitates multiple pointer assignments and is susceptible to threads observing an inconsistent view of the list. Sundell et al. [86] was the first to present a lock-free doubly linked list utilizing CAS operations. Nevertheless, due to the need for traversing deleted nodes, frequent updates on the list lead to significant overhead. Consequently, the current Linux kernel still restricts threads to only move forward to prevent them from perceiving an inconsistent sequence of items in the list.

Implementing a lock-free doubly linked list correctly requires in-depth understanding of concurrency, memory models, and low-level synchronization primitives. It is a complex task that demands meticulous design and consideration of edge cases and potential race conditions, particularly when integrating it into

the Linux kernel. It is anticipated that the utilization of lock-free algorithms can enhance the overall system throughput by enabling greater concurrency without blocking operations.

3.5 Summary

In this chapter, we present Finer-LRU, a memory subsystem scheme in response to the limited scalability of the HPC manycore architecture. In an environment where hundreds of tasks are running in parallel, a large number of threads have to access the shared memory resource simultaneously, causing heavy lock contention. In order to reduce the contention level, the Finer-LRU scheme splits the contended resources used in the page frame reclamation process in a fine-grained manner. A lock instance handling multiple contended data structures is divided into several instances, and each of them handles a split data structure. The Finer-LRU framework is also capable of selectively splitting the data structure with severe contention to efficiently handle the memory subsystem with reduced memory space overhead. Finally, we evaluate the Finer-LRU scheme with IOR and HACC-IO benchmarks on the 68-core KNL system. By applying the Finer-LRU scheme on the Linux kernel, the throughput is improved by up to 57.03% and the latency is reduced by up to 98.94%. Furthermore, we address scalability concerns by further optimizing Finer-LRU. The Finer-LRU-opti scheme reduces the number of lock instances and manages five LRU lists with a single lock instance, while still increasing the total number of LRU lists. This optimization enhances I/O performance, leading to an increased throughput of up to 140% on both Intel KNL and Intel Ice Lake Xeon CPUs.

Chapter 4

HPC Storage I/O Stack Optimization

4.1 Motivation

4.1.1 Lustre Backend File Systems: ldiskfs and ZFS

Lustre maintains object storage device (OSD), which is the server software interface that controls access to underlying local file systems on OSTs and MDTs. Ldiskfs, one of the backend local file systems that are supported in Lustre, is an enhanced version of ext4 file system [87], with an improved performance and additional functionality for the Lustre file system. Another backend file system that Lustre supports is ZFS which originally stands for Zettabyte file system. As the name implies, ZFS can handle up to 256 ZB data size. Other features that ZFS provides include pooled storage, end-to-end data integrity, copy-on-write, and native compression [88]. ZFS works as both a file system and a volume manager, which enables multiple physical devices to be aggregated as a single storage pool and improves overall file system scalability. ZFS also offers

checksum feature on every data written to ZFS to avoid and recover from data corruption. Furthermore, the compression feature that ZFS provides can save space of the storage devices and reduce data access time.

Because of the multiple features that ZFS supports to improve reliability and data integrity, ldiskfs-based Lustre is more lightweight and shows higher performance compared with ZFS-based Lustre. Conseqeuntly, current supercomputer systems prefer ldiskfs as the backend file system of Lustre. Nevertheless, the advent of artificial intelligence and big data analytics has resulted in a substantial increase in the amount of data stored in HPC storage systems in recent years [89–91]. Consequently, Lustre must be scalable enough to process petabyte range data generated by large-scale scientific workloads. To achieve scalability without contraints imposed by the backend file system, ZFS, which supports zettabyte-scale storage is a better option compared to ldiskfs, which only permits maximum volume size of 50 TB [92]. For this reason, we mainly focus on analyzing the I/O performance of Lustre with a ZFS backend file system and introduce optimization designs to enhance the performance of ZFS.

To compare I/O performance of ldiskfs and ZFS, we run FIO microbenchmark with various configurations to conduct I/O operations on a small-scale testbed. The experimental environment of Lustre cluster includes three nodes: Lustre client, OSS, and MDS/MGS. OSS node is equipped with an Intel Xeon CPU E5-2620 v4 with 16 physical cores and a 64 GB DDR4 RAM. As flash prices are getting lower, the next-generation flash storage systems begin to deploy all-flash storage devices and remove the burden from Burst Buffer [93, 94]. In line with current trend, we use four 280 GB Intel Optane SSD 900P on OSS and configure each device to be an OST. MDS node has same system specification with Lustre client node, and a 960 GB Samsung PM963 NVMe SSD is configured as MDT. Lustre client, OSS, and MDS/MGS nodes are connected

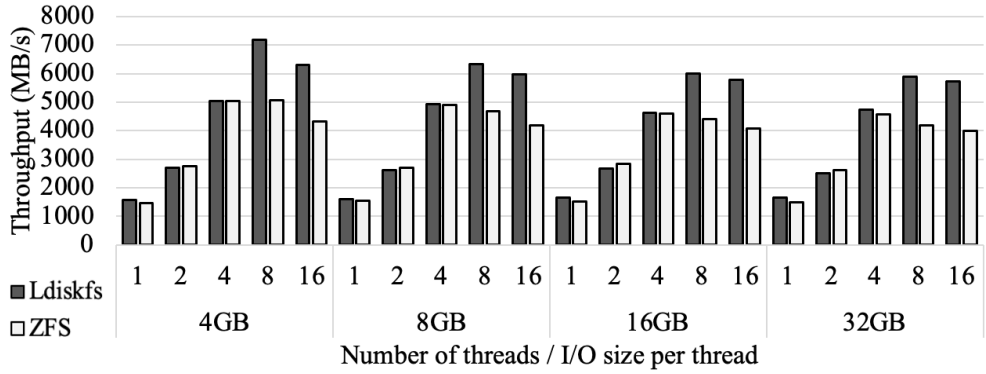

Figure 4.1: File-per-process sequential write I/O throughput of ldiskfs-based Lustre and ZFS-based Lustre.

by Mellanox ConnectX-5 MT27800 100 Gb/s EDR Infiniband fabric.

Figure 4.1 shows sequential write performance of ldiskfs-based Lustre and ZFS-based Lustre varying the number of threads concurrently executing I/O operations and the I/O size of the file each thread writes. The number of threads increases from 1 to 16 to differ the concurrency level. Each thread writes contiguous bytes of a file size between 4 GB to 32 GB whereas the transfer size is fixed to 1 MB. All experiments are conducted using FIO benchmark with various configurations. When ZFS is used as backend file system for Lustre, ZFS zpool is created for each MDT and OSTs.

The result shows that throughput scales nearly up to 7,000 MB/s with Lustre-ldiskfs when the number of threads concurrently executing I/O operations increases. When the number of threads exceeds eight threads, the performance of Lustre-ldiskfs starts to decrease because of the limited scalability of the processing units on Lustre client node. Compared with that of ldiskfs-based Lustre, the performance of ZFS-based Lustre starts to saturate when more than four threads perform I/O operations concurrently. The maximum throughput

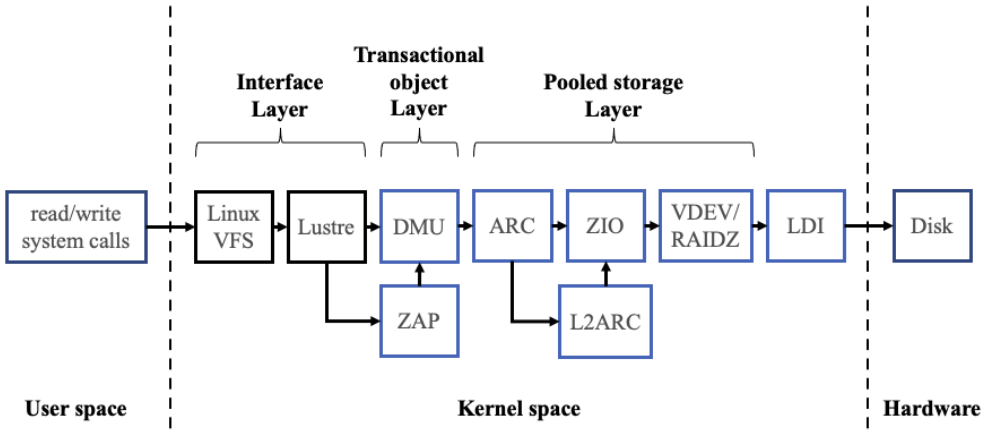

Figure 4.2: I/O stack of ZFS-based Lustre file system.

that ZFS-based Lustre shows is slightly higher than 5,000 MB/s with a small total I/O size. The performance of ZFS-based Lustre is lower than that of ldiskfs-based Lustre by 27% when eight or more threads are used. The experimental result shows large performance difference between ldiskfs and ZFS with high-performance storage devices used as OSTs and MDT. The main reason for performance degradation needs to be identified to fully exploit the functional advantages that ZFS can offer while achieving high I/O performance.

4.1.2 I/O stack of ZFS-based Lustre

Figure 4.2 shows the simplified I/O stack diagram of a ZFS-based Lustre file system. ZFS provides different modules that are related to each other within the kernel space. When the user issues write system calls, the kernel invokes according VFS write function that Lustre provides. Then, Lustre OSD layer directly interact with ZFS using the data management unit (DMU) interface. DMU module is the transactional object model in ZFS and is responsible for committing transactions to the underlying storage layer. Adaptive replacement

cache (ARC) and L2ARC modules serve memory management function within ZFS and maintain caching layer for data stored in ZFS. All write operations issued to the ZFS pool are first accumulated in ARC cache and assigned to a specific transaction group (TXG). Accumulated operations are handled at once by sync operation when a specific time has passed or a total of 64 MB of data blocks are accumulated. Among different modules of ZFS, the ZFS I/O (ZIO) module is the core part that is responsible for reading and writing data block to disk. The accumulated data blocks on an ARC buffer is passed into a ZIO pipeline that includes multiple I/O function stages, such as compression, checksum calculation, and data redundancy. After the data virtual address (DVA), which specifies the block address of the data block, is identified in the ZIO module, the data block is written to a physical device in the VDEV module [95].

4.1.3 ZFS I/O Pipeline

The ZIO module provides a pipelined I/O engine so that I/O operations can be executed in pipelines. Figure 4.3 depicts the diagram of a logical write I/O pipeline. The pipeline first starts with initializing the ZFS block pointer that stores metadata of the data block. The contents of the block pointer include physical size, logical size, DVA, and the 256-bit checksum of the data. After the initialization, the rest of the I/O pipeline is operated asynchronously by dispatching additional threads, given that compression and checksum operations in the following stages yield expensive CPU overhead. ZFS supports checksums for both metadata and data block for data integrity. Various checksum algorithms, such as fletcher2, fletcher4 [96], and SHA-256 [97] hashes, are supported by default to avoid silent data corruption. Finally, when the DVA allocation of the data block that block pointer references are finished, the actual physical I/O is executed through the VDEV module.

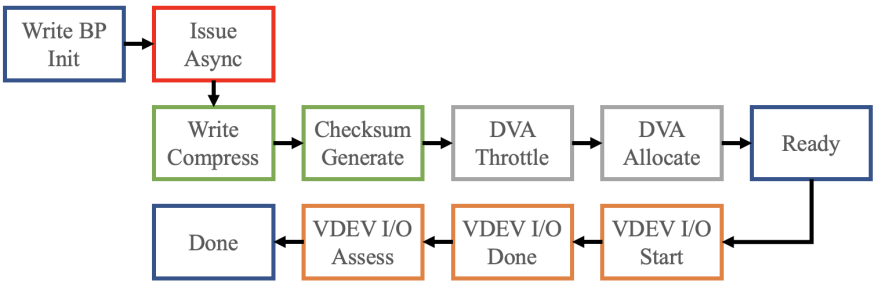

Figure 4.3: Logical write I/O pipeline in ZIO module.

The execution of the ZFS I/O pipeline is handled by threads from several task queues. Each I/O task is added and deleted from corresponding task queues depending on the type of operations, which is determined by the combination of ZIO type (read, write, free, claim, ioctl) and ZIO task queue type (issue, interrupt). For instance, the task going through the logical write I/O pipeline shown in Figure 4.3 is enqueued to one of two task queues, *z_wr_iss* or *z_wr_int*, depending on the type of the task. Each task queue has multiple dedicated threads to process the tasks in the queue.

ZFS supports multiple kernel parameters that can be tuned by users to improve the performance. *zio_taskq_batch_pct* controls the percentage of online CPUs that run I/O worker threads for the rest of the pipeline stages in the ZFS I/O pipeline. The default value is set to 75% so that 75% of the cores run a number of threads that can participate in executing compression and checksum operations asynchronously. Another ZFS kernel parameter that controls the number of threads is *zfs_sync_taskq_batch_pct*. The parameter determines the percentage of online CPUs that run the threads that participate in synchronizing dirty in-memory DMU objects to disk. Similar to the *zio_tqskq_batch_pct*, the default value of *zfs_sync_taskq_batch_pct* is set to 75%.

Table 4.1: Latency of write I/O pipeline stages of ZFS

Write I/O Pipeline Stages	**ZFS (checksum on)**	**ZFS (checksum off)**	**Ratio**
Write BP Init	29,299,253	28,201,044	0.96
Issue Async	2,186,845,778	2,368,581,658	1.08
Write Compress	16,199,056,718	10,599,544,753	0.65
Checksum Generate	143,340,468,139	682,601,521	0.01
DVA Throttle	833,687,951	347,673,284	0.42
DVA Allocate	3,570,880,925	3,539,767,862	0.99
Ready	128,008,807,689	173,464,632,853	1.36
VDEV I/O Start	11,030,463,390	10,636,164,802	0.96
VDEV I/O Done	15,456,183,161	13,559,295,143	0.88
VDEV I/O Assess	1,321,798,672	1,144,140,313	0.87
Done	9,125,897,261	8,018,373,192	0.88
Total pipeline stages	334,534,110,488	227,475,458,797	0.68
Throughput	4,446MB/s	4,995MB/s	1.12

4.1.4 Problem Analysis

To further investigate the root cause of low I/O performance shown with ZFS-based Lustre, we profile the latency of each function stage in the ZFS I/O pipeline. The first column of Table 4.1 shows the write I/O pipeline stages, and the second column shows the time taken by each pipeline stage in nanoseconds when a total of 128 GB write I/O operations are executed concurrently with 16 threads. The two bottom rows show the total time taken in the ZFS I/O pipeline by profiling *zio_execute* function that recursively executes each pipeline stage and the write throughput. Under the default configuration with checksum feature enabled, checksum calculation takes up approximately 43% of the total time taken in ZFS I/O pipeline stages. We also observe that 28% of CPU core utilization by average is used by the checksum function execution according to Linux perf tool.

To identify the amount of checksum computation overhead on ZFS, we profile latency of the same pipeline function stages of ZFS with checksum configuration turned off, the result of which is shown in the third column of the table. The fourth column of the table shows the ratio of latency taken in the ZFS I/O pipeline with checksum disabled to latency taken in the ZFS I/O pipeline with the default configuration. The result shows that not only the checksum function but almost every other functions take slightly less amount of time when the checksum feature is disabled. Specifically, the time taken in compression function, which is a computationally intensive task similar to checksum function, is reduced by 35%. Overall, 32% of the total ZFS I/O pipeline latency is saved, and write throughput is increased by 12% when the checksum configuration is turned off. The fact that disabling checksum generation affects the time taken in rest of the pipeline functions indicates that CPU overhead has been a bottleneck in the ZFS pipeline.

One of the I/O pipeline functions, *Issue Async* stage (*zio_issue_async* function), dispatches I/O worker threads to accelerate computationally intensive functions, such as compression and checksum operations, in the next stages. The maximum number of ZFS I/O worker threads dispatched in the stage is controlled by the ZFS kernel parameter *zio_taskq_batch_pct*, with a default value of 75%. In our testbed environment, OSS node is equipped with 32 logical cores with hyper-threading enabled, and a zpool is configured for each OST. In other words, each zpool can have maximum of 24 I/O worker threads responsible for handling tasks in each type of task queues, which is 75% of 32 online cores. Given that four zpools exist in OSS server, a maximum of 96 I/O worker threads responsible for *z_wr_iss* task queue can be dispatched in the testbed environment. The other I/O worker threads handling different types of task queues are dispatched in the same way. Similar to the ZFS I/O worker thread, the number

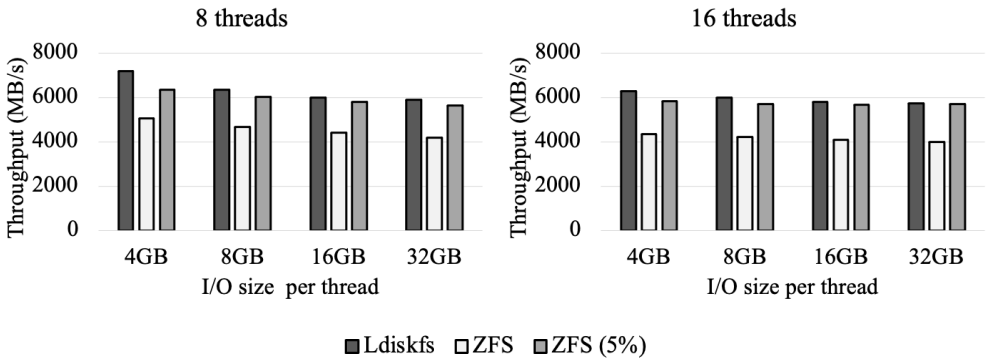

Figure 4.4: File-per-process sequential write I/O throughput of ldiskfs-based Lustre, ZFS-based Lustre, and ZFS-based Lustre with 5% *zio_taskq_batch_pct*.

of threads handling DSL pool synchronization operation is controlled by the ZFS kernel parameter *zfs_sync_taskq_batch_pct*, with a default value of 75%. As a result, another 96 sync threads are dispatched and used for synchronous I/O operations. Considering that Lustre manages its own service threads as well, hundreds of threads are created in a single server node. While there are a number of adjustable kernel module parameters provided by ZFS [98], we only focus on two parameters in this work as they are the only parameters that directly controls the number of ZFS threads.

Figure 4.4 compares the I/O performance of ldiskfs-based Lustre, ZFS-based Lustre, and ZFS-based Lustre with 5% *zio_taskq_batch_pct* when 8 and 16 threads concurrently execute file-per-process sequential write operations. A single I/O worker thread is created for each zpool when ZFS-based Lustre is configured with 5% *zio_taskq_batch_pct*. As a result, four I/O worker threads are responsible for executing ZFS I/O pipeline stages and handling all accesses on the four zpools. The experimental result shows that reducing the maximum number of ZFS I/O worker threads from 96 to 4 significantly improves performance by 19% relative to the default ZFS. The performance of ZFS-based Lustre

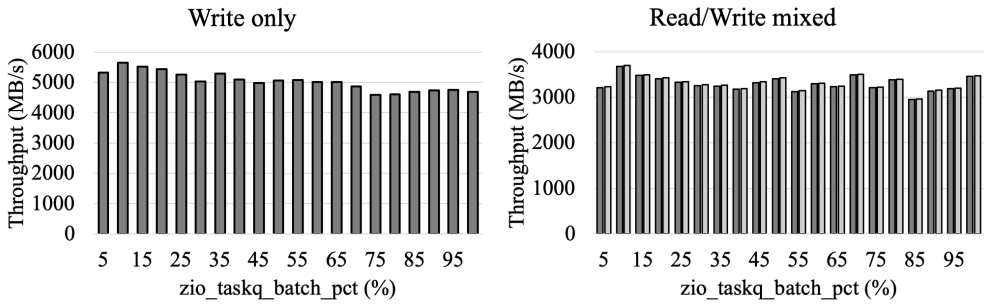

Figure 4.5: Sequential write and read/write mixed I/O throughput of ZFS-based Lustre with different *zio_taskq_batch_pct*.

with less threads is similar to that of ldiskfs-based Lustre, which indicates that using a large number of threads rather decreases the performance due to excessive context switching overhead. In other words, ZFS uses too much threads in default configuration on the 16-core system when running write-heavy workload. ZFS loaded on manycore system in HPC environment dispatches even more ZFS threads in default because of a large number of cores used, which eventually worsens the performance degradation.

Although adjusting *zio_taskq_batch_pct* to 5% significantly improves I/O performance, 5% is not the magic number for ZFS-based Lustre. Figure 4.5 shows the I/O performance of write-only (left), and read/write mixed (bottom) workloads with 16 threads concurrently executing 8GB-size I/O operations. We configure *zio_taskq_batch_pct* from 5% to 100% and measure the performance. The result shows that using small number of threads improves the I/O performance on write-heavy workload while the relatively low I/O performance is shown with 5% *zio_taskq_batch_pct* on read/write mixed workload. The performance of the two different workloads demonstrate that the optimal number of ZFS I/O worker threads varies depending on the workload I/O patterns. While using small number of threads can reduce the context switch overhead, low degree

of parallelism can lead to I/O bottleneck at the same time. The number of ZFS I/O worker threads is controlled by the number of CPU cores used in the system with *zio_taskq_batch_pct*. Thus, dynamic adjustment of ZFS I/O threads considering both I/O patterns and system configuration setting can improve the performance.

To sum up, we determine that performance difference between ldiskfs and ZFS is mainly due to the large amount of time taken in managing the checksum feature, which ldiskfs does not support. Also, we find that hundreds of threads are created throughout the ZFS modules for handling checksum calculation and synchronous tasks, which increases the context switching overhead and degrades the I/O performance. This motivates our work of designing a parallel checksum calculation pipeline and a dynamic thread control scheme.

4.2 Design and Implementation

4.2.1 Parallel Checksum Calculation Pipeline

We have observed that the time taken in checksum calculation takes up a large portion of the total ZFS I/O time in Table 4.1. Two of ZFS task queues that ZFS uses, *z_wr_iss* and *z_wr_int*, are responsible for handling write I/O tasks, including checksum operations. *z_wr_iss* threads are dispatched in one of ZFS I/O pipeline functions, *zio_issue_async*, and increase the number of concurrent operations to speed up the compression and checksum calculation. To separate computationally intensive tasks from the rest of the ZFS I/O stages, dynamic-ZFS creates and uses a new task queue named *z_wr_cks*. The threads handling the tasks in *z_wr_cks* are responsible for executing the checksum operation only, and those threads can be executed in parallel with *z_wr_iss* or *z_wr_int* threads.

ZFS Checksum Threads

ZFS manages I/O threads responsible for every task queue in the storage pool when the pool is created. Some of the threads handle the compression and checksum tasks of the I/O accessing the pool. To reduce the I/O overhead in CPU-intensive checksum calculation, we first create a new task queue that is responsible for handling the checksum tasks only. The new task queue, *z_wr_cks*, is created with the combination of ZIO type (write) and ZIO task queue type (cksum). In other words, only write operations are handled with three kinds of task queues: *z_wr_cks*, which is responsible for checksum calculation task only, and *z_wr_iss* or *z_wr_int*, which is responsible for the rest of write I/O-related function tasks with a parallel checksum calculation pipeline. The number of threads handling the tasks in *z_wr_cks* task queue is determined by another kernel module parameter that we implement, *cksum_zio_taskq_batch_pct*, with a default value of 75%. Given that the number of active checksum task queue threads is dynamically adjusted using the algorithms in Section 4.2.2, the kernel parameter value is for configuring the maximum number of threads that can be created and does not affect the performance. Similar to threads handling *z_wr_iss* task queue, the checksum task queue threads are created when the ZFS pool is created and dispatched in the ZFS I/O pipeline stage, *zio_issue_async* function (*Issue Async* stage in Figure 4.3). As having too many ZFS threads results in poor performance, we redesign the function to not dispatch extra *z_wr_iss* task queue threads, but to only dispatch *z_wr_cks* task queue threads.

In summary, write operations going through a parallel ZFS I/O pipeline are handled by three kinds of thread sets: *z_wr_iss* and *z_wr_int* task queue threads, which handle every pipeline function operations, except the checksum calculation function; and *z_wr_cks* task queue threads, which handle only the

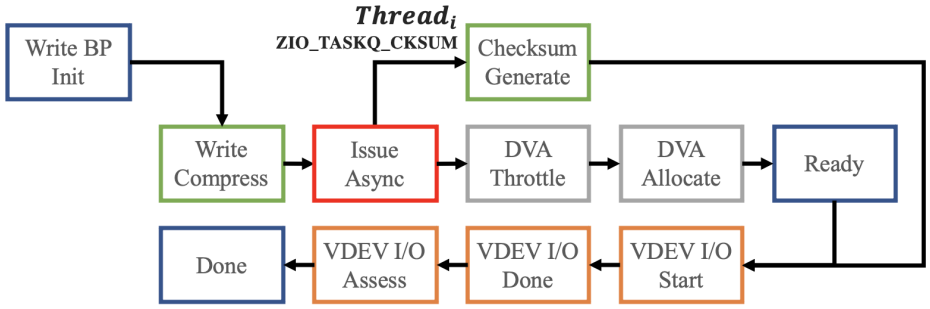

Figure 4.6: Parallel logical write I/O pipeline with parallel checksum calculation in the ZIO module.

checksum calculation of the ZFS data block. The number of *z_wr_iss* queue threads does not increase in the parallel ZFS I/O pipeline, whereas the number of *z_wr_cks* queue threads is dynamically adjusted by using the dynamic thread control scheme presented in Section 4.2.2.

Parallel ZFS I/O Pipeline

We redesign the ZFS write I/O pipeline to enable parallel checksum calculation in the ZIO module. By using the additional threads dedicated to handle only the checksum operation, the parallel execution of pipeline functions can be achieved. Figure 4.6 shows the parallel logical write I/O pipeline design of dynamic-ZFS. We carefully design dynamic-ZFS in a way that no data structure is shared between two parallel stages. Note that dynamic-ZFS always chooses whether to operate in parallel I/O pipeline or original I/O pipeline, depending on the decision made in the dynamic thread control algorithm.

Different from Figure 4.3 in Section 4.1.4, *Issue Async* stage is located after the compression function in parallel I/O pipeline. This is because the purpose of dynamic-ZFS is to reduce the checksum calculation latency, which occupies

a large portion of the total latency, as shown in Table 4.1. By positioning *Issue Async* stage just before *Checksum Generate* stage, we allow *Issue Async* stage to be responsible for making a decision of whether to dispatch new threads that handle checksum calculation tasks in *z_wr_cks* task queue. If the decision is made to dispatch new *z_wr_cks* task queue thread, the checksum calculation function is removed from the mainline pipeline stages and executed in parallel with the rest of the stages (*DVA Throttle*, *DVA Allocate*, and *Ready* stages). In this case, we add a barrier function in *Ready* stage to wait for a slow checksum task to make sure that the checksum operation is finished before the data are written to physical devices. As a result, the time taken in checksum operation can be saved by executing the ZFS I/O pipeline in parallel.

To validate that the block checksum is successfully calculated in our scheme, we conduct a simple experiment. In the experiment setting, we include extra *Checksum Generate* stage in the main pipeline in Figure 4.6 and let *z_wr_iss* task queue thread executes checksum operation as in default ZFS. The data block is duplicated right after *Write Compress* stage and we let each block to go through original checksum stage and parallelized checksum stage. Then, the checksum values calculated in the two checksum stages are compared in the *Ready* stage. Through the experiment, we confirm that the checksum values of default and parallel ZFS I/O pipelines are always the same.

4.2.2 Dynamic Thread Control Scheme

We implement algorithms that can adjust the number of ZFS I/O worker threads dynamically by considering current CPU loads. Even though ZFS already has *TASKQ_DYNAMIC* feature enabled by default that dynamically controls the number of ZFS worker threads, the current CPU load is not considered in the adjustment process. As we observe that spawning additional thread

Algorithm 4.1: Context Switch Monitoring Algorithm

```
1  open File;
   /* open Monitoring File                                          */
2  mmap File;
   /* map file into memory                                          */
3  while do
4      pid_pool ← ZFS process id;
       /* get process ids of currently running ZFS processes        */
5      for i ← 0 to size of pid_pool do
6          sum ← 0;
7          pid ← pid_pool[i];
8          acc_ctxt ← ctxt(pid);
           /* get accumulated number of process's context switches  */
9          sum ← sum + acc_ctxt;
10     write File ← sum;
       /* write total number of context switches                    */
11 munmap File;
   /* unmap file                                                    */
12 close File;
   /* close Monitoring File                                         */
```

whenever an I/O task is to be processed rather decreases the overall performance, the number of threads needs to be controlled considering the current CPU utilization.

CPU usage Monitoring

To dynamically adjust the number of ZFS threads, we first implement a CPU usage monitoring framework to identify the current CPU load used by ZFS I/O

worker threads. Given that a large number of threads used in ZFS results in poor performance, we assume CPU usage with the number of context switches. Algorithm 4.1 shows the context switch monitoring procedure. In our implementation, the monitoring file is opened and memory mapped in order to reduce the overhead of using a complex software I/O stack and enable fast file accesses (line 1-2). Then, process IDs of currently running ZFS processes are retrieved and stored in *pid_pool* data structure (line 4). The process ID is retrieved one by one from *pid_pool*, and the accumulated number of context switches of the process so far is calculated (line 5-9). The numbers of context switches of the ZFS processes are summed up and written to the monitoring file (line 10). The process of getting an accumulated number of context switches of ZFS processes is repeated during the time ZFS module is loaded.

In dynamic-ZFS, the numbers of ZFS threads in two different thread sets are controlled: *z_wr_cks* task queue threads and *dp_sync_taskq* task queue threads. The reason for adjusting the two thread sets is because two kernel parameters that control the number of ZFS threads on default ZFS already exist: *zio_taskq_batch_pct* and *zfs_sync_taskq_batch_pct*. The task queues storing read tasks can also be adjusted, but we only focus on optimizing the write path in this work. As a result, the number of ZFS I/O worker threads handling checksum calculation in write operations (*z_wr_cks*) and the number of ZFS threads responsible for synchronous writes (*dp_sync_taskq*) are controlled.

Dynamic Thread Control Algorithm

Using the context switch monitoring framework, we implement a dynamic thread control algorithm embedded in the ZFS I/O module. Dynamic-ZFS controls the number of threads that execute the tasks by monitoring the current CPU load in real time. Algorithm 4.2 shows the algorithm of controlling the

Algorithm 4.2: Dynamic Thread Control Algorithm

Global variables : *prev_acc_ctxt*, *prev_ctxt*

```
1  Function ThreadControl:
2      acc_ctxt ← get_ctxt();
       /* get accumulated number of context switches                */
3      ctxt ← acc_ctxt − prev_acc_ctxt;
       /* get number of context switches occurred during last
          interval                                                  */
4      ctxt_diff ← ctxt − prev_ctxt;
       /* difference of number of context switches                  */
5      prev_ctxt ← ctxt;
6      prev_acc_ctxt ← acc_ctxt;
7      return ctxt_diff;
8  Function ZfsFunction:
9      if ctxt_diff >= 0 then
           /* dispatch another thread to execute function           */
10     else if ctxt_diff < 0 then
           /* let current thread to execute function                */
```

number of ZFS threads dynamically. Two global variables, *prev_acc_ctxt* and *prev_ctxt*, represent the accumulated number of context switches and the number of context switches calculated in the previous iteration, respectively. The thread control procedure first gets *acc_ctxt*, which is the current accumulated number of context switches of the ZFS threads (line 2). The value is calculated from Algorithm 4.1. To determine the number of context switches that occurred during the last timestamp interval (from the last time when thread control procedure is called until the current time), we calculate the *ctxt* variable by determining the difference between *acc_ctxt* and *prev_acc_ctxt* (line 3). Then,

we compare the current CPU load and the previous CPU load by calculating the difference between *ctxt* and *prev_ctxt*. The difference is stored in variable *ctxt_diff* (line 4). The current context switch indicator values are updated to the global variables for next function call (line 5-6).

The calculated *ctxt_diff* variable is used to determine whether to wake up another task queue thread and operate in parallel I/O pipeline to execute the next procedure. In default ZFS, an additional thread is always dispatched whenever there exist a new task to be handled. However, this increases the number of threads excessively and eventually leads to overabundant CPU usage. To avoid using unnecessarily large number of ZFS threads, we control the number of threads by checking whether the current CPU usage load is higher than the load in the last timestamp interval. When *ctxt_diff* is positive, which indicates that the current CPU usage load is higher than the previous load in the last interval, we dispatch another thread to lower the current CPU load and let the thread handle the task in the parallel I/O pipeline (line 9). On the contrary, when the current CPU usage is the same or lower than the previous load and *ctxt_diff* being zero or negative, we do not dispatch another thread and let current thread execute the task in the original I/O pipeline (line 10).

The dynamic thread control algorithm is applied to two ZFS functions, *cksum_zio_taskq_dispatch* and *dmu_objset_sync*. *cksum_zio_taskq_dispatch* function is the new function that we implemented, which combines *zio_issue_async* and *zio_checksum_generate* functions (*Issue Async* and *Checksum Generate* stages in Figure 4.6). The function controls the number of checksum task queue threads using the algorithms and let the thread execute a checksum calculation task in the parallel I/O pipeline. When the algorithm determines not to dispatch another thread and let the current thread to execute the function, current write task queue thread executes checksum function in the original I/O pipeline.

Meanwhile, *dmu_objset_sync* also adds the dynamic thread control algorithm to control the number of ZFS threads handling synchronization tasks.

4.3 Evaluation

4.3.1 Experimental Setup

We evaluate dynamic-ZFS on three different system configurations.

Cluster A, The *Cluster A* consists of one client, one OSS, and one MDS/MGS node. Lustre client node is equipped with Intel Xeon CPU E5-2650 v3 with 20 physical cores and 80 GB DDR4 RAM. OSS node is composed of Intel Xeon CPU E5-2620 v4 with 16 physical cores and 64 GB DDR4 RAM, equipped with four 280 GB Intel Optane 900P. Each Intel Optane SSD is configured as an OST and has sequential read and write throughput by up to 2,500 MB/s and 2,000 MB/s, respectively. MDS/MGS node has the same CPU and memory specification with Lustre client node and uses a single 960 GB Samsung PM963 NVMe SSD configured as MDT.

Cluster B, The *Cluster B* consists of two clients, two OSSes, and one MDS/MGS node to evaluate the scalability of dynamic-ZFS. The additional Lustre client node and OSS node have the same system hardware specification with the Lustre client used in the *Cluster A*. Two OSS nodes are equipped with four Intel Optane 900P devices each, having a total of eight OSTs on the Lustre file system.

Cluster C, The *Cluster C* consists of two clients, four OSSes, and one MDS/MGS node. The additional two OSS nodes have the same system hardware specification with the Lustre client in the *Cluster A*. Four OSSes are equipped with two Intel Optane 900P devices each, having a total of eight

OSTs on the Lustre file system.

All nodes are connected by Mellanox ConnectX-5 MT27800 100 Gb/s EDR Infiniband fabric to avoid network bottleneck.

All the experiments are run using parallel I/O benchmark, FIO, with buffered I/O mode. We vary the number of threads concurrently running I/O operations and file size each thread writes, while fixing the transfer size to 1 MB. We configure FIO to run in file-per-process mode to avoid I/O bottleneck of every threads accessing the same shared file protected by a single file lock. We compare the Lustre performance with three different backend file systems: ldiskfs, ZFS, and dynamic-ZFS.

4.3.2 ZFS I/O Pipeline Latency

To evaluate the reduced latency of ZFS I/O pipeline by applying parallel ZFS I/O pipeline and dynamic thread control schemes, we measure the latency of every stage function that participated in ZFS write I/O pipeline. Figure 4.7 shows the latency of functions when 16 threads concurrently write a total of 128 GB I/O size on a Lustre file system with dynamic-ZFS as backend file system. All the latency of functions measured on Lustre file system with default ZFS is set to 1, and the latency of dynamic-ZFS is normalized accordingly.

The result shows that the latency of every function is reduced significantly with dynamic-ZFS even though the I/O load to handle is the same. The reason why no latency is measured on *Issue Async* stage (*zio_issue_async* function) is that dynamic-ZFS chooses not to dispatch additional ZFS I/O thread that handles the tasks in write task queues during its I/O pipeline process. Instead, whether to dispatch an additional ZFS I/O thread that handles the checksum tasks is determined in *cksum_zio_taskq_dispatch* function. The reason for reduced latency of every stage functions is that context switching overhead is

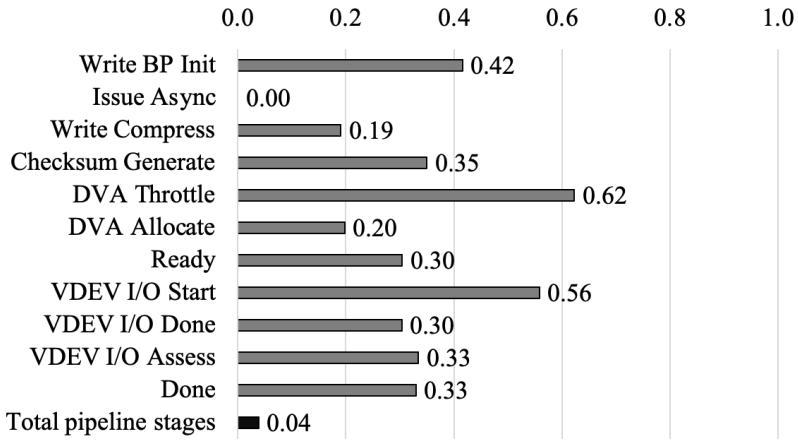

Figure 4.7: Normalized latency of write I/O pipeline stages of dynamic-ZFS compared to ZFS.

reduced from a moderate number of threads dispatched. Overall, the time taken in the total pipeline stages is reduced by 96% when Lustre uses its backend file system as dynamic-ZFS.

4.3.3 CPU Utilization

We first evaluate the performance of the dynamic-ZFS-based Lustre file system on a single server cluster, *Cluster A*. The cluster consists of one client, one OSS, and one MDS/MGS node. We compare the context switching overhead by measuring CPU utilization on a OSS server during I/O operations. We use FIO benchmark to issue a total of 128 GB file-per-process I/O operations using 16 threads each. To measure CPU utilization in real time, we use sar monitoring tool [99] to run in background with FIO benchmark. Figure 4.8 shows application-level and system-level CPU utilization when Lustre is configured with ZFS and dynamic-ZFS. We configure FIO benchmark to run in write-only mode (top), and read/write mixed mode (bottom). In a read/write

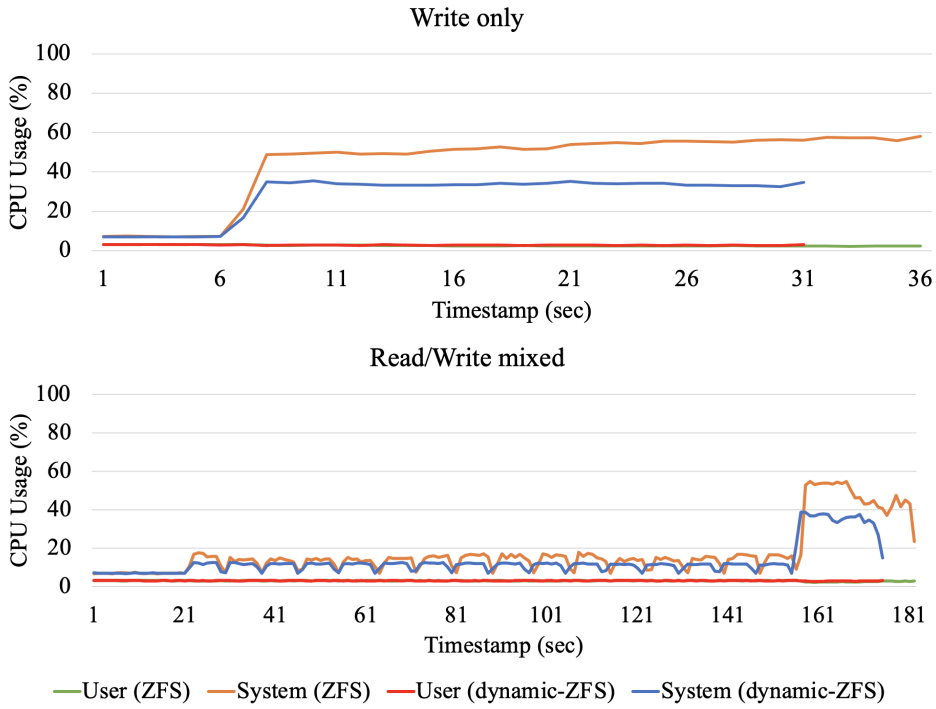

Figure 4.8: CPU utilization of OSS under different I/O patterns with 16 threads.

mixed mode, 16 files (8 GB each) are sequentially created first, and then all 16 threads perform read operations and write operations equally together within each file. As a result, a slight increase in CPU usage rate is shown 16 times before the actual read/write mixed operations are performed. Both graphs show that using dynamic-ZFS decreases the system level CPU load significantly. Also, the application-level CPU utilization does not increase much with dynamic-ZFS even though context switch monitoring tool runs in background. Average system-level CPU load is decreased from 53% to 34% when dynamic-ZFS is used under write-only mode, whereas the load is decreased from 48% to 35% on average when read and write are performed together. We assume that more CPU load utilization can be saved under a system configuration of a large number of

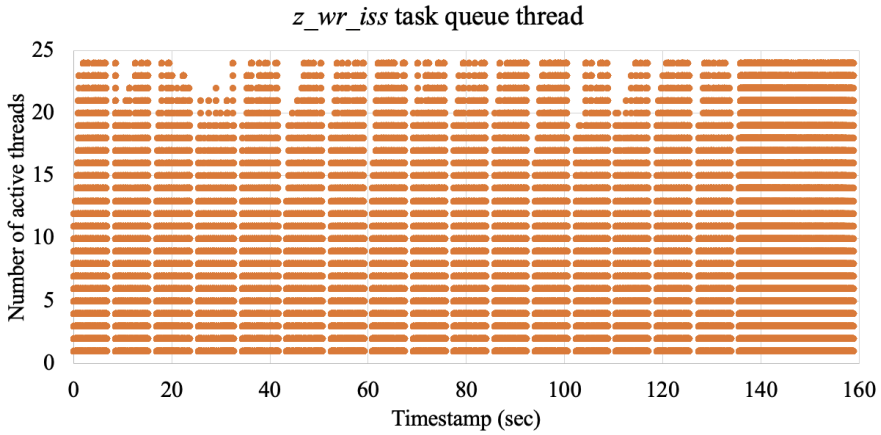

Figure 4.9: Number of active ZFS I/O worker threads in ZFS.

flash storage devices used together within a server.

4.3.4 Dynamic Thread Control

Dynamic-ZFS adjusts the number of ZFS I/O worker threads to achieve low CPU utilization and high I/O performance. To demonstrate the dynamic aspect of dynamic thread control scheme, we track the number of active threads run in ZFS and dynamic-ZFS while running the same I/O workload. The active thread refers to the thread that got woken up to execute tasks, such as executing I/O pipeline functions. Although there are other task queue threads involved in ZFS I/O pipeline, we focus on *z_wr_iss* and *z_wr_cks* task queue threads in this evaluation. There are four zpools exist in the OSS node in *Cluster A* and each zpool creates 24 I/O worker threads, which is 75% of the number of online CPUs. Even though there are total 96 threads run in the server node, the number of active threads is tracked per zpool. Thus, the maximum number of active threads is 24 in the graphs.

Figure 4.9 and Figure 4.10 show the changes in the number of active threads

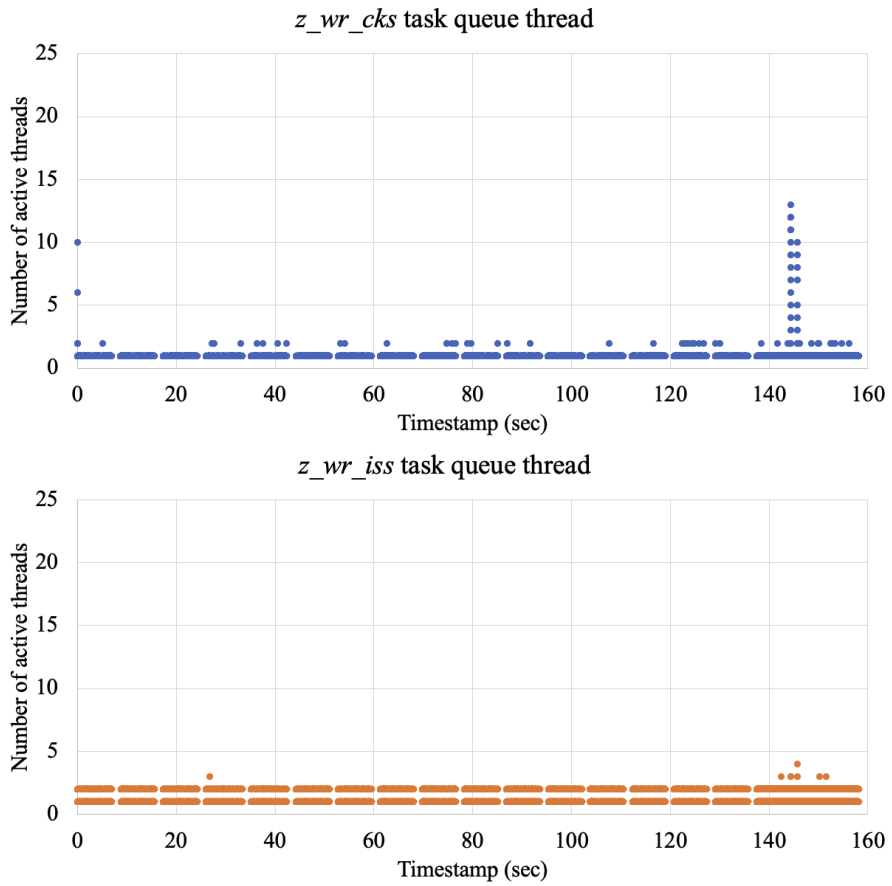

Figure 4.10: Number of active ZFS I/O worker threads in dynamic-ZFS.

while running FIO benchmark in read/write mixed mode. Same with the read-/write workload run in Section 4.3.3, the benchmark first sequentially creates 16 files of 8 GB in size. Then, all 16 threads perform read and write operations equally together within each file. Each dot in the graph represents the number of active threads run in each zpool at a timestamp. In default ZFS, ZFS I/O worker threads (*z_wr_iss* taskq threads) are always dispatched to the maximum extent regardless of the current I/O loads as shown in Figure 4.9. Even when the single file is created sequentially, all 96 threads got woken up to handle

the tasks. On the contrary, dynamic-ZFS does not dispatch additional ZFS I/O worker threads in the I/O pipeline and controls the number of checksum threads (*z_wr_cks* taskq threads). Figure 4.10 shows that dynamic-ZFS determines to use only 2 to 4 threads in total when I/O load is low, which is regarded to be sufficient to perform ZFS I/O pipeline. When the I/O load increases, the active number of checksum threads increases by the decision made from dynamic thread control algorithm. To sum up, our evaluation shows that dynamic-ZFS captures the I/O load in real time and dynamically controls the number of ZFS threads to reduce CPU overhead.

4.3.5 Sequential I/O Performance

In this section, we compare sequential write throughput and latency of Lustre file system when ldiskfs, ZFS, and dynamic-ZFS backend file systems are used. Figure 4.11 shows the throughput and latency when 8, 16, and 32 threads concurrently write total I/O size of 64, 128, 256, 512, and 960 GB. Every thread is configured to write the same file size (i.e. Each thread writes 60 GB file sequentially when 16 threads write a total I/O size of 960 GB). For every file to be equally striped over every OST, we set the stripe count to 4.

The result shows that dynamic-ZFS can improve the I/O performance as the number of ZFS I/O worker threads is dynamically adjusted by monitoring the run-time CPU load. Dynamic-ZFS-based Lustre improves I/O performance by 37% and reduces latency by 27% on average compared with default ZFS-based Lustre when eight or more threads concurrently issue I/O operations. When compared with ldiskfs-based Lustre, I/O performance is slightly degraded by 2% and latency is increased by only 3% with dynamic-ZFS-based Lustre. Considering that dynamic-ZFS provides a multitude of features that ldiskfs does not, slight performance degradation is negligible. Moreover, the performance

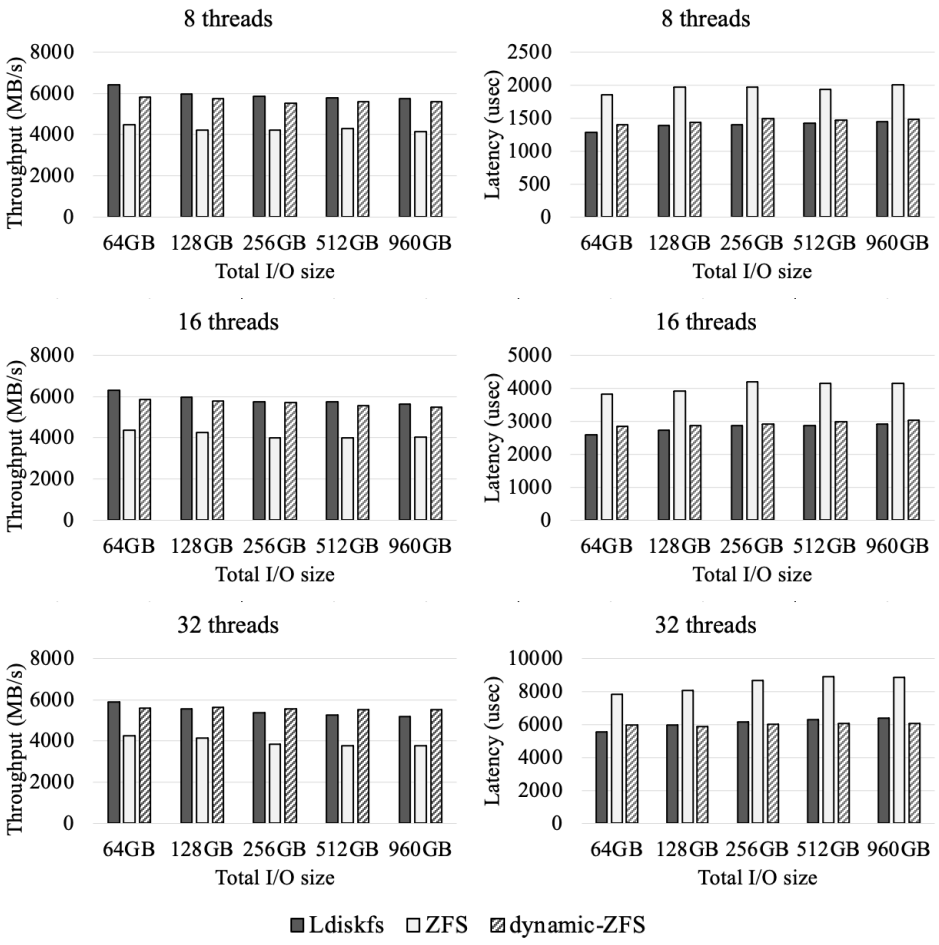

Figure 4.11: Throughput and latency of the sequential write I/O varying the number of threads and total I/O size in *Cluster A*

.

improvement with dynamic-ZFS increases as the concurrency level gets higher with an increased number of threads participating in I/O. The I/O throughput is increased by 33%, 38%, and 41%, and the latency is reduced by 25%, 28%, and 29% when 8, 16, and 32 threads issue I/O operations with dynamic-ZFS, respectively. This indicates that under the heavy CPU load, dynamic-ZFS can further improves the I/O performance. As it is common in HPC platforms that each OSS is composed of more than four storage devices and multiple jobs share storage nodes concurrently reading and writing large amount of files, we can assume that the performance gain by using dynamic-ZFS as backend file system for Lustre is even higher in real-world HPC systems.

4.3.6 Scalability

To evaluate the scalability of dynamic-ZFS, we run FIO benchmark on *Cluster B* and *Cluster C*. One includes two Lustre clients, two OSS nodes, and one MDS/MGS node, and the other includes two Lustre clients, four OSS nodes, and on MDS/MGS node Figure 4.12 shows the sequential write throughput in *Cluster B*, and Figure 4.13 shows the same throughput result in *Cluster C*. The write throughput is measured and summed up in two Lustre clients on both evaluations. We vary the number of threads run in each Lustre client from 1 to 32 and the file size each thread writes from 4 GB to 32 GB. Eight OSTs are used in two system clusters: four OSTs per OSS node in *Cluster B*, and two OSTs per OSS node in *Cluster C*. For all the files to be equally striped into every OST, we set the stripe count to 8 on every evaluations.

Similar to the result shown with *Cluster A* in Figure 4.11, dynamic-ZFS-based Lustre achieves higher I/O performance than default ZFS-based Lustre in Figure 4.12. The performance peak shown with ldiskfs-based Lustre when eight threads concurrently writes 4 GB of file per thread is due to the effect

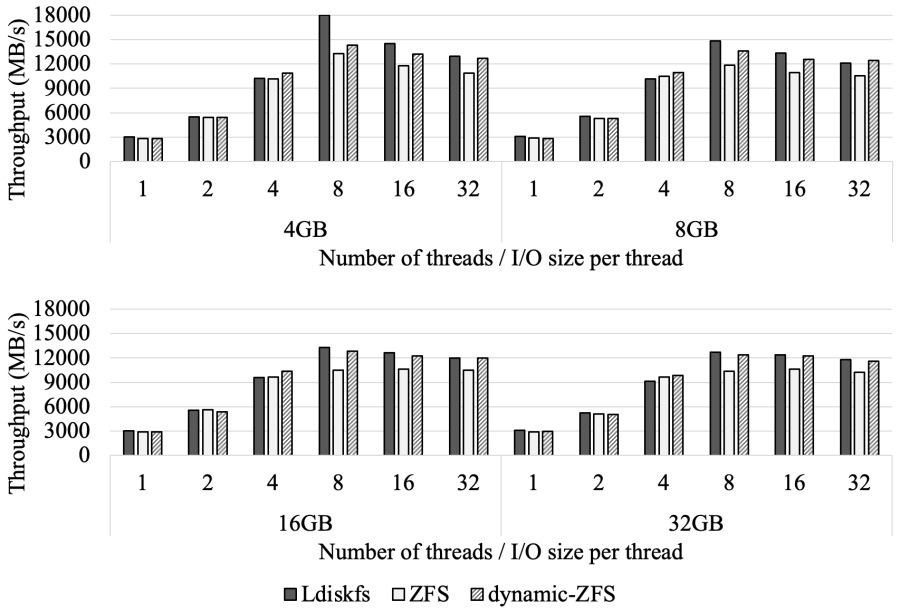

Figure 4.12: Throughput of sequential write I/O varying the number of threads and I/O size per thread in *Cluster B*

.

of buffered I/O mode. Other than small size I/O evaluation, dynamic-ZFS-based Lustre shows similar performance with ldiskfs-based Lustre. The write throughput of dynamic-ZFS is improved by 8% on average relative to ZFS. Under the heavy I/O load of 8, 16, and 32 threads concurrently issuing I/O operations per Lustre client node, 15% of the I/O performance on average is improved.

When only two OSTs are on each OSS node, the performance still improves with dynamic-ZFS, but the performance improvement is less than that of *Cluster B*. Figure 4.13 shows the I/O throughput with the same FIO configuration settings with Figure 4.12. The reason why not much difference is observed among the performance between ldiskfs, ZFS, and dynamic-ZFS-based Lustre

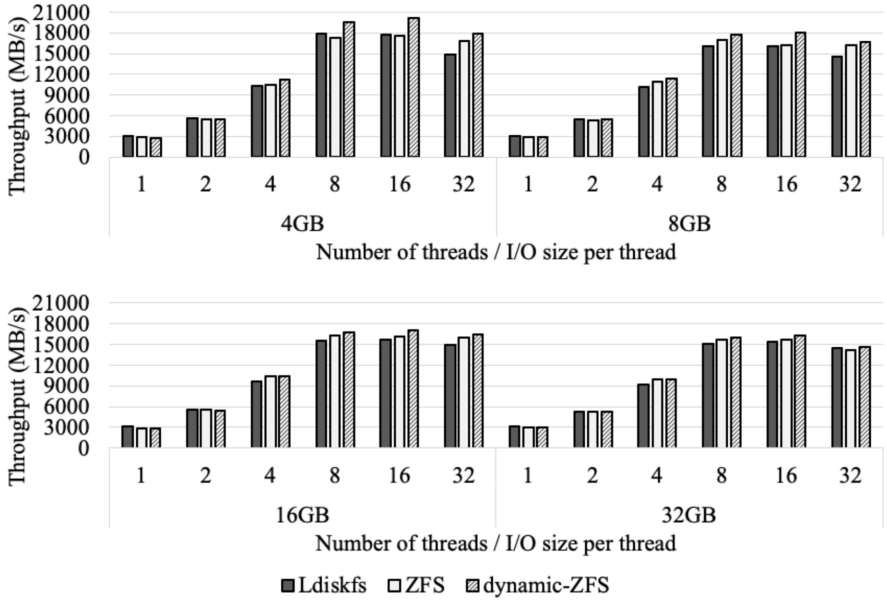

Figure 4.13: Throughput of sequential write I/O varying the number of threads and I/O size per thread in *Cluster C*

.

is that two NVMe devices on each server node does not incur CPU usage overhead. As a result, ZFS can show high performance comparable to ldiskfs even with an excessive number of ZFS I/O worker threads. The I/O performance of dynamic-ZFS-based Lustre is slightly higher than that of ldiskfs-based Lustre with 5% on average. Our evaluation shows that dynamic-ZFS can still improve I/O performance, even under the environment where CPU utilization is not high.

4.4 Summary

ZFS, a backend file system used by Lustre, delivers a number of beneficial features that can improve I/O performance. However, ZFS rather incurs high

CPU usage load and results in low performance because of a large number of worker threads handling end-to-end data integrity checksum computation jobs. To overcome the performance limitation, we designed dynamic-ZFS, a combination of parallel execution scheme of ZFS I/O pipeline and dynamic ZFS thread control scheme. The evaluation results showed that dynamic-ZFS can successfully control the high CPU load of ZFS by interacting with CPU monitoring framework in run-time. The sequential I/O throughput is improved by 37%, and latency is reduced by 27% on average, whereas CPU utilization during I/O operations is decreased by 20%. We also showed that dynamic-ZFS can be effectively applied to scalable HPC platforms with a multitude of feature configurations and a high performance comparable to that of ldiskfs. As next-generation HPC platforms begin to deploy all-flash storage systems, we expect the performance benefit of using dynamic-ZFS on Lustre to increase in the near future.

Chapter 5

HPC System Configuration Optimization

5.1 Motivation

The Cori Supercomputer system at the National Energy Research Scientific Computing Center (NERSC) is one of the most powerful HPC systems in the world. HPC users can run their applications on the Cori system by submitting a job script to the Slurm workload manager. The term *job* refers to a computational task that a user requests to run via the Slurm scheduler, while *application* refers to an executable program. Users can specify configurations on the job script, such as the amount of computational resource, the type of cores in the compute node to use, or the number of MPI processes with which to run the job. The limit of execution time and the storage system-related parameters can also be included in the script. Using the information that the user defined on the job script, the Slurm workload manager allocates the available resources to the users so that the jobs can be run accordingly.

We perform our analysis using the real-world user log data from Cori system from January to June, 2019 (PDT). In order to collect the HPC log data, we use the Darshan I/O profiling tool [25]. The Darshan module is a lightweight I/O instrumentation library that can capture various I/O behaviors, including application access patterns, number of I/O operations, or access sizes using multiple interfaces. Since we have to get the aggregated database of the logs to apply the machine learning models, we use the python3-based parser developed by Kim et al [72]. The parser can extract I/O-related data not only from the logs that the Darshan module provides, but also from the Slurm workload manager and Lustre Monitoring Tool. By utilizing both application-level, file system-level, and scheduler-level statistics, a total of 112 I/O related features are collected from 134,069 jobs in the analysis period.

Among the features given by the parser, there are five features that are user-configurable. The number of compute nodes($numNode$), CPUs($numCPU$), and processes($numProc$) are the features that control the amount of computational resource, while the stripe count($numOST$) and the stripe size($stripeSize$) are the ones used to configure the file striping settings in the Lustre file system. Figure 5.1 shows the distribution of the user-configurable features on the jobs from the top five applications that issue the largest amount of I/O operations. The graph shows that all the jobs do not change stripe size setting in the Lustre file system, although they have a chance to improve the performance by changing the parameter. Also, only one application (marked as purple) out of five changes the stripe count dynamically, while another four use a relatively fixed number of OSTs. Another thing to notice is that 93.42% of jobs run from the other application (green) use the same configuration setting while I/O throughput varies from 14.3 to 215.6GB/s, which is approximately 15 times higher performance.

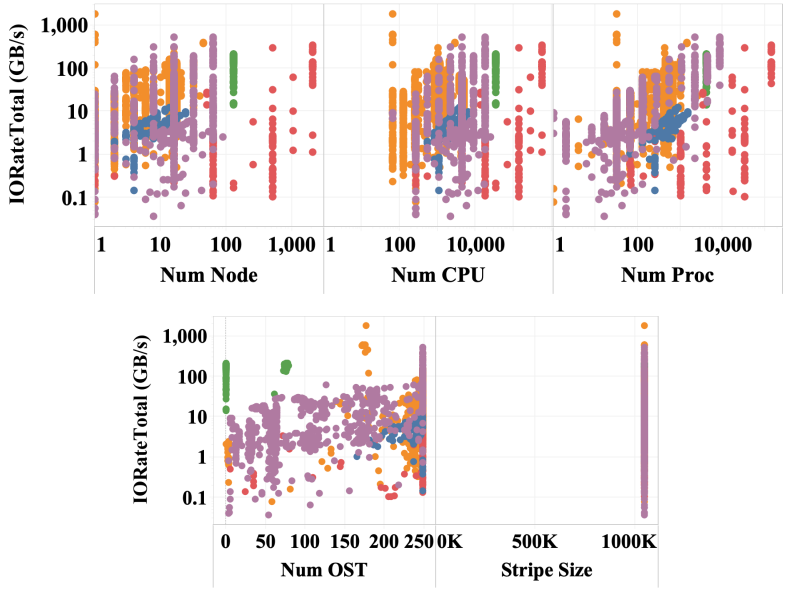

Figure 5.1: The distribution of user-configurable parameters on the top five applications that have large amounts of I/O during the job execution time. The applications have different colors in the graph

Our observations show that most of the users choose to use fixed amounts of resources to run the jobs and there are multiple factors affecting I/O performance in the HPC environment, other than the amount of resources the jobs use. When the job increases the I/O parallelism by configuring the Lustre file system to use multiple OSTs, it also increases the shared resource contention on the network and file system layer. The performance can even change due to the input parameters that are used to run the job, the information of which the profiling tools cannot capture. It is an exhaustive work to manually figure out the exact I/O related features that lead to poor performance.

To ease computational complexities, we utilize multiple machine learning models together to identify factors that have a significant impact on I/O perfor-

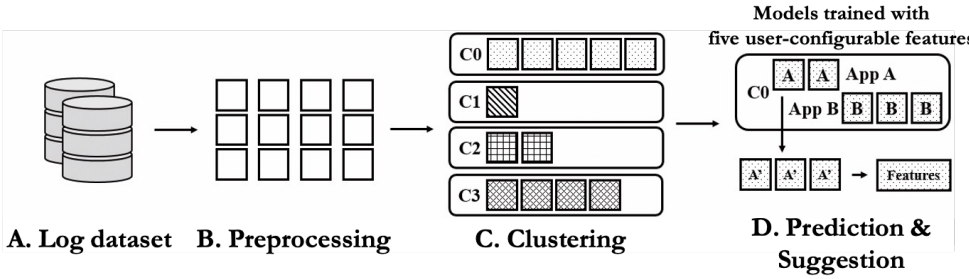

Figure 5.2: The machine learning process for user-configurable parameter suggestion

mance. By grouping real-world logs with similar I/O characteristics, the prediction models built on each cluster can achieve improved prediction accuracy. The methodology we introduce aims to assist HPC users in gaining better insights into configuring HPC resources for enhancing I/O performance. The overall machine learning process for suggesting optimal user-configurable parameters is depicted in Figure 5.2.

5.2 Design and Implementation

5.2.1 Dataset Preprocessing

The log data from multiple profiling tools has to be refined using several data preprocessing steps before applying machine learning models. We use the jobs that issue more than 1GB I/O operations, in order to focus on the data that have enough I/O related information. Also, only the applications that have been executed more than 100 times during the analysis period are included in the dataset. To overcome the data imbalance, we sample the data to make the number of job executions of the applications to be the same by performing random over-sampling and under-sampling [100]. After the data preprocessing

Table 5.1: Features extracted on the parser

Category	**Features**
Application Statistics (MPIIO / POSIX / STDIO)	totalFile
	[Read,Write]Req / OpenReq / SeekReq / StatReq
	seq[Read,Write]Pct
	[read,write][Less,More][1k,1m]
	runTime
Computation Statistics	**numNode**
	numCPU
	numProc
Storage Statistics	**numOST**
	stripeSize
	mdsCPU[Min,Mean,95]
	mdsOPS[Min,Mean,95]
	oss[Read,Write][Min,Mean,95]
	oss[Read,Write][Min,Mean,95]Used
	oss[Read,Write]LargestUsed
	oss[Read,Write]Higher[1g,4g]
Performance Statistics (MPIIO / POSIX / STDIO)	IORateTotal
	[Read,Write]Bytes
	[Read/Write/meta]Time
	slow[Read/Write]Time

steps are completed, the dataset includes a total of 122,640 jobs from 84 different applications, each with the same 1,460 jobs. There are 35 features remaining in the dataset, and the target feature of clustering and prediction models is the sum of the read and write I/O throughput, also termed *IORateTotal*. Table 5.1 summarizes some of the features that the parser generates, with five user-configurable features emphasized.

5.2.2 Feature Selection and Clustering Models

A clustering method can reveal hidden information in I/O behaviors in the HPC system and yield a better understanding of the I/O workload characteristics. The clustering algorithm calculates the similarity of the features for each data point, in order to group the jobs together. Since there are too many features that are related to I/O characteristics, it is necessary first to eliminate the irrelevant features for dimensional reduction before clustering the jobs. We seek the best feature subset that includes features having the most dominant impact on the target feature in two steps: the Min-max mutual information feature selection, and the SBS process combined with clustering algorithm.

In order to select the features that can best represent the I/O characteristics of the data, we implement a new feature selection algorithm that chooses the features based on the correlation coefficient value. The new method, named Min-max mutual information feature selection, first selects the feature that is most strongly correlated with the *IORateTotal*, which is the target feature. Then, the second feature is selected from among the top ten least correlated features with the previously selected one, having the highest correlation value with the *IORateTotal*. In this way, not only the features that have a strong relationship with the target feature can be selected, but also the redundancy among the selected features is minimized. The process of selecting the features iterates until the desired number of features are selected.

The top ten features selected from the Min-max mutual information algorithm in our dataset are as follows: *runTime*, *OpenReqSTDIO*, *mdsOPSMin*, *writeLess1k*, *ossWriteLargestUsed*, *slowWriteTimePOSIX*, *numOST*, *totalFileSTDIO*, *writeMore1m* and *readLess1k*.

Among the ten features that are selected via the Min-max mutual informa-

tion algorithm, naively selecting some of the features correlated to I/O performance on clustering may not group data with similar I/O characteristics. It is also important to determine the best number of features to select for efficient clustering. To find the best feature subset that can give the highest clustering performance, we use the parallel Sequential Backward Selection (SBS) algorithm [101], which can reduce the computational cost by searching for the best feature set in parallel.

The SBS algorithm works along with the KMeans clustering algorithm and two cluster evaluation metrics. Starting from a set of ten features, one feature at a time is removed, making ten feature subsets, each consisting of nine features. Then, the KMeans clustering algorithm is performed on each of the ten feature subsets in parallel, and the clustering result is evaluated using the Silhouette coefficient [102] and the Davies-Bouldin index (DBI) [103] metrics. The two validity metrics evaluate the cluster performance by calculating the inter-cluster variance and the inner-cluster variance. When the Silhouette coefficient score is high and the DBI score is low, the two scores indicate that the inter-cluster distance is short and the nearest-cluster distance is long, which can also be regarded as the quality of the clustering result is good. In order to combine the two validity metrics into one, we make use of the *Combined Score* validity metric, which is calculated by dividing the Silhouette score by the DBI score.

After the ten different clustering results from the ten different feature subsets are evaluated in parallel, the subset with the highest *Combined Score* is selected and becomes the starting feature set in the next iteration. The SBS process continues until three features are left in the feature set. Since the clustering algorithm is executed and evaluated in parallel, the computation cost of searching for the optimal result with the best feature set can be reduced dramatically. The KMeans clustering algorithm runs using each feature subset and

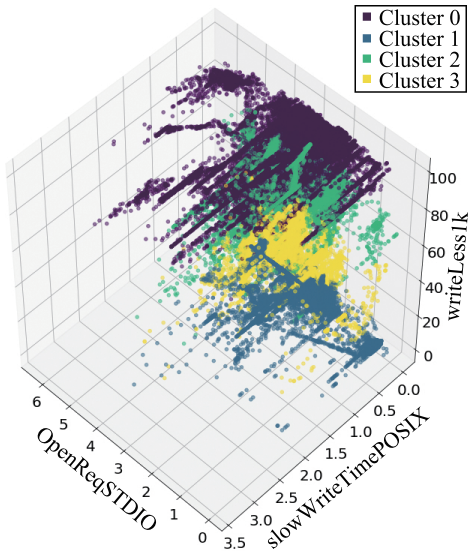

Figure 5.3: Three-dimensional feature space diagram of four clusters

the results are evaluated on the dedicated compute nodes in parallel, reducing the computation time by approximately 82%, compared to the time taken in a serial search process.

The result shows that regardless of the number of clusters, the *Combined Score* increases as the feature set size decreases, which indicates that the SBS process eliminates the feature that is most irrelevant to I/O performance and has a negative impact on the cluster performance in every iteration. Since the objective of the clustering is to group the applications with similar I/O characteristics together and build the prediction model on each of the clustered datasets, clusters need to have a sufficient number of jobs. Considering both the *Combined Score* and the size of each clusters, we select the clustering result of four clusters by automatically calculates the cluster size variability, in the case when every cluster consists of at least 10,000 jobs.

The KMeans clustering algorithm clusters the jobs based on three features,

OpenReqSTDIO, *writeLess1k*, and *slowWriteTimePOSIX*, which are considered to be the most relevant variables to I/O performance and can provide the best clustering performance. Figure 5.3 shows the three-dimensional diagram of the clustering results, with *OpenReqSTDIO* and *slowWriteTimePOSIX* being in logarithmic format. *slowWriteTimePOSIX* represents the time taken by the slowest POSIX write operation, while *writeLess1k* is the percentage of the number of write operations with an access size less than 1KB to the total number of write operations. *OpenReqSTDIO* represents the number of open requests using the STDIO interface. The diagram shows that jobs are grouped mainly based on *writeLess1k*, which represents that *writeLess1k* value highly influences the I/O performance of the job.

5.2.3 Clustered Datasets

In the previous section, we clustered jobs based on the features highly correlated to I/O performance, from which we obtained four clusters, each with similar I/O characteristics. Our next step is to train the prediction models using each of the clustered datasets, and evaluate the prediction performance. Since multiple prediction models are trained from each of the clusters, which model to use can be identified by the application name of the job. However, the internal I/O behaviors of two jobs from the same application can be completely different in some cases. Assuming that the parallel I/O benchmark is run on the system, depending on the command line options the users specify, one job may perform the random write operations with a 4KB block size, while another job may perform sequential read operations with a 4GB block size. Then, the profiling logs would show completely different I/O patterns, even though the two jobs are run using the same application. Therefore, the clustering algorithm would certainly place the two jobs in different clusters, considering that they have

Table 5.2: Evaluation of Prediction models using different datasets

Dataset	Number of Apps	Number of Jobs	R-squared score
Cluster 0	35	51,100	0.86
Cluster 1	22	32,120	0.91
Cluster 2	6	8,760	0.82
Cluster 3	3	4,380	0.20
Un-clustered Data	84	122,640	0.84

opposite I/O behaviors from each other. In this case, I/O performance cannot be predicted for the same I/O benchmark job, since it is impossible to know into which cluster the job will be grouped, and thus the appropriate prediction model cannot be distinguished.

In order to identify the proper prediction model from the application name, we restrict the clustered dataset to include applications that show similar I/O behaviors across multiple runs. For example, when 80% of the jobs from application A belong to Cluster 1 and the rest of the jobs belong to Cluster 2, we consider every future job from the application A would belong to Cluster 1. Interestingly, the clustering result shows that 66 out of 84 applications have more than 80% of the jobs grouped into the same cluster. This indicates that most of the HPC applications are run repeatedly with similar I/O characteristics, which is in line with previous works [72, 104].

5.2.4 Prediction Models

We use the KNN prediction algorithm, which is one of the most commonly used regression methods, to train the prediction models. KNN calculates the similarity between the given feature values of trained data and test data to find

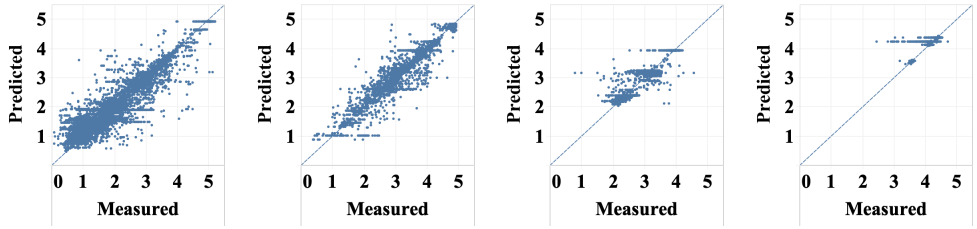

Figure 5.4: Prediction plots of the measured versus predicted I/O performance (MB/s) of jobs being in logarithmic format (from left to right are: Cluster 0, Cluster 1, Cluster 2, Cluster 3)

the K-Nearest neighbors, from which the average of the neighbors becomes the predicted value. We aim to improve the prediction performance by using the training dataset that includes jobs having common I/O characteristics. To evaluate our clustering-based prediction models, we train the models and measure the R-squared score using five different datasets: the four clusters we get in Section 5.2, and the un-clustered dataset that has undergone only the data preprocessing step for baseline evaluation. The training set and test set are created by randomly dividing the jobs of each dataset into an 8:2 ratio, respectively. The input parameters to the model are the five user-configurable features, the number of compute nodes($numNode$), CPUs($numCPU$), and processes($numProc$), stripe count($numOST$) and the stripe size($stripeSize$), information of which can be obtained before the job execution time.

Table 5.2 shows the number of applications, jobs and the R-squared score, which can represent the predictive accuracy, on five types of datasets. The number of applications and jobs in the un-clustered dataset is larger than the sum of those in the four clusters, because we only include applications that have similar I/O characteristics across the runs in the four clusters. The evaluation result shows that the R-squared score is higher with Clusters 0 and 1, when

compared to the score of the un-clustered dataset. One of the reasons why the score of Cluster 3 is extremely poor is because only three applications are included in the cluster, which is considered to be too small to train the model. Also, since most of the jobs in Cluster 3 use a fixed value of user-configurable parameters, the prediction model cannot accurately predict I/O performance with different configurations. Figure 5.4 plots the measured and predicted I/O performance of the jobs in logarithmic scale in the four clusters. The horizontal dots in the bottom right graph indicate that the prediction model trained by jobs in Cluster 3 predicts the same I/O performance most of the time, due to the lack of training data. In contrast to Cluster 3, the prediction model created from Cluster 1 shows the highest R-squared score, due to the dynamic configuration settings the users configured in running the jobs. The average R-squared score with different weights for Cluster 0, 1, and 2 based on the cluster size is calculated as 0.87, which is higher than the score of un-clustered dataset. Overall, our result shows that clustering the jobs with similar I/O behaviors can help increase the R-squared score and makes it possible to predict I/O performance of the jobs with improved accuracy.

5.3 Evaluation

In this section, we search for the optimal system configuration settings for HPC workloads, using the trained prediction model. We only show the Cluster 0-based trained prediction model and omit the other models due to page limitation. Other than the Cluster 3-based model, which has poor performance because of the lack of information in the training data, Cluster 0, 1 and 2-based models show the similar performance. We create the test dataset by varying the five user-configurable parameters in different ranges, which are de-

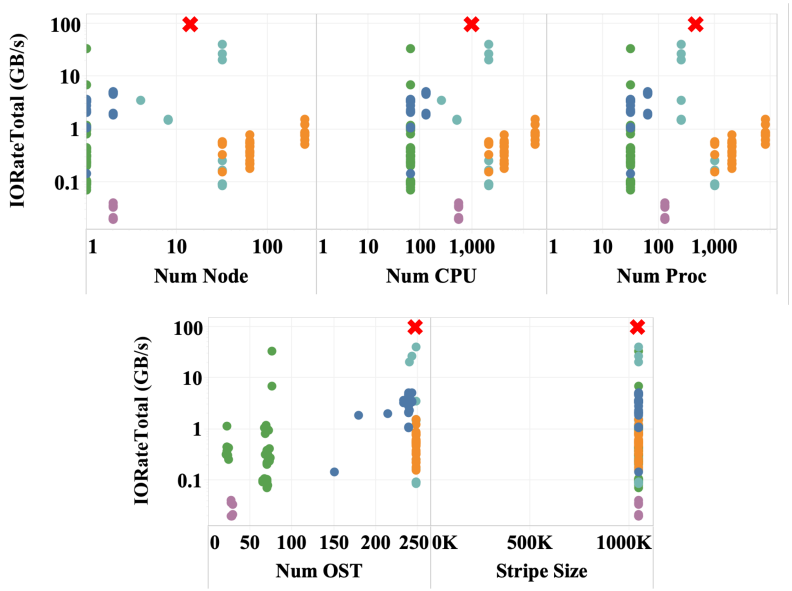

Figure 5.5: The optimal user-configurable parameters obtained from the Cluster 0-based prediction model. The applications have different colors in the graph

termined based on the Cori system user log data, run from July to September 2019 (PDT). Specifically, *numNode* has the range between 1 to 3,850 and *numCPU* is determined as the product of 64 or 272 times *numNode*, which is the number of cores in the Haswell and KNL nodes respectively. Also, the product of *numNode* times 2 to the power of 0 to 8 is determined as *numProc*, referring to the log data. *numOST* ranges from 1 to 248, which is same as the number of OSSs in the Cori system, and *stripeSize* is determined to have one of 1, 2, 4, 8, 16, and 32MB values. I/O performance is predicted for every combination of the five parameters in the prediction model. Then the configuration parameters with the highest I/O performance in the model are selected.

Figure 5.5 shows the distribution of the user-configurable parameters of the jobs and the optimal configurations that give the highest I/O performance, us-

ing the prediction models based on Clusters 0. Among the applications included in the cluster, we plot the top five I/O-intensive applications, run from July to September 2019 (PDT), which is the time period that starts right after our training period. The red marks in the graphs represent the optimal configuration settings that are searched from the prediction models. Compared to the performance of the jobs in all five applications, the red marks show the highest I/O throughput. Although we omit the figures showing the results of the Cluster 1 and 2-based models, the optimal user-configurable parameters that are searched using the two models also give the highest I/O performance compared to the performance of the jobs that run between the test time period. By using the predicted performance for various configuration settings, not only can users get a better understanding of configuring the jobs and achieve higher performance, but also the system can efficiently schedule the limited amount of resource in the HPC environment.

While we show that I/O performance can be predicted with different combinations of configuration parameters, only the methodology for building the prediction model based on the clustering results is applicable to other HPC systems. We obtain the optimal configuration settings from the regression models that are trained with jobs run in the specific analysis period, January to June, 2019 (PDT). Considering that the supercomputer system changes significantly over time for multiple reasons, such as periodic maintenance, hardware replacement, and software upgrades, the results that we observe are not expected to be found in general. The objective of our work is to provide the methodology that can predict I/O performance of HPC workloads with high accuracy, using various machine learning models. By periodically searching for the optimal user-configurable parameters using our methodology, HPC users can be provided with the configuration setting guidance that can give improved I/O

performance.

5.4 Summary

We used multiple machine learning techniques together to efficiently discover hidden information in the complex I/O behaviors of scientific workloads with low computational cost. By clustering the unlabeled user log data, applications having similar I/O characteristics with each other can be grouped together. Our results showed that the R-squared scores on the prediction models built on each of the clusters are higher than the score of the un-clustered dataset, indicating that the clustering can help improve the prediction accuracy. Using the methodology that we introduced, I/O performance can be predicted for any system configuration setting in advance, which can help users find the best configuration settings for improved I/O performance of the jobs. Also, efficient resource scheduling in the HPC environment can be achieved by using the constructed prediction models.

Chapter 6

Conclusion

HPC systems consist of numerous compute nodes, high-speed networks, and storage systems, all of which have complex I/O stacks. It is increasingly crucial to design efficient memory management and storage file systems for HPC systems to satisfy the growing demand for data access performance in applications. In addition, HPC users must be provided with optimal system configuration settings to prevent significant variations in performance. In this dissertation, we focused on three optimization schemes to improve I/O performance in HPC systems. We first minimized lock contention in the memory management system of an HPC manycore architecture, while optimizing storage I/O stack in ZFS-based Lustre file system, which is the commonly used PFS in HPC environments. We also searched for optimal configuration settings to maximize the I/O performance of HPC applications using multiple machine learning models.

We implemented Finer-LRU, a memory subsystem scheme in response to the limited scalability of the HPC manycore architecture. In an environment where hundreds of tasks are running in parallel, a large number of threads have

to access the shared memory resource simultaneously, causing heavy lock contention. In order to reduce the contention level, the Finer-LRU scheme splits the contended resources used in the page frame reclamation process in a fine-grained manner. A lock instance handling multiple contended data structures is divided into several instances, and each of them handles a split data structure. The Finer-LRU framework is also capable of selectively splitting the data structure with severe contention to efficiently handle the memory subsystem with reduced memory space overhead. Finally, we evaluated the Finer-LRU scheme with IOR and HACC-IO benchmarks on the 68-core KNL system. By applying the Finer-LRU scheme on the Linux kernel, the throughput is improved by up to 57.03% and the latency is reduced by up to 98.94%.

ZFS, a backend file system used by Lustre, delivers a number of beneficial features that can improve I/O performance. However, ZFS rather incurs high CPU usage load and results in low performance because of a large number of worker threads handling end-to-end data integrity checksum computation jobs. To overcome the performance limitation, we designed dynamic-ZFS, a combination of parallel execution scheme of ZFS I/O pipeline and dynamic ZFS thread control scheme. The evaluation results showed that dynamic-ZFS can successfully control the high CPU load of ZFS by interacting with CPU monitoring framework in run-time. The sequential I/O throughput is improved by 37%, and latency is reduced by 27% on average, whereas CPU utilization during I/O operations is decreased by 20%. We also showed that dynamic-ZFS can be effectively applied to scalable HPC platforms with a multitude of feature configurations and a high performance comparable to that of ldiskfs. As next-generation HPC platforms begin to deploy all-flash storage systems, we expect the performance benefit of using dynamic-ZFS on Lustre to increase in the near future.

We used multiple machine learning techniques together to efficiently discover hidden information in the complex I/O behaviors of scientific workloads with low computational cost. By clustering the unlabeled user log data, applications having similar I/O characteristics with each other can be grouped together. Our results showed that the R-squared scores on the prediction models built on each of the clusters are higher than the score of the un-clustered dataset, indicating that the clustering can help improve the prediction accuracy. Using the methodology that we introduced, I/O performance can be predicted for any system configuration setting in advance, which can help users find the best configuration settings for improved I/O performance of the jobs. Also, efficient resource scheduling in the HPC environment can be achieved by using the constructed prediction models.

This dissertation shows performance improvement by optimizing the existing HPC system on memory, storage, and user-level components. Rather than optimizing the existing system, designing a HPC system from scratch would require careful consideration of several factors [105]. Followings are some key aspects to consider in such design:

- **Scalability** Design the kernel to be scalable and NUMA aware to efficiently handle the increased number of cores. Data structures, synchronization mechanisms, and algorithms should be carefully designed to minimize contention and improve parallelism across multiple cores. Review the memory management system to ensure strong NUMA-awareness. Implement memory allocation policies, page migration algorithms, and memory placement strategies to improve locality and reduce remote memory access penalties in multi-socket systems.

- **Scheduling Algorithms** Enhance the kernel scheduling and resource

management mechanisms to efficiently handle the increased number of cores. Explore techniques such as load balancing, task migration, and affinity management to distribute workloads evenly and reduce contention. Consider gang scheduling and hierarchical scheduling to improve parallelism and resource utilization. Reduce unnecessary context switches by considering factors such as cache locality and thread affinity when making scheduling decisions.

- **Thread and Process Management** Analyze the thread and process management subsystem to improve efficiency and reduce overheads. Explore improved context switching mechanisms, such as minimizing the amount of state that needs to be saved and restored during a context switch. This can involve optimizing register save and restore operations. Also explore lightweight threading models and optimizations like thread pinning or thread affinity to enhance parallelism and cache locality.

- **File System Design** Design file system for high-speed storage devices. Implement striping, caching, and prefetching within file system I/O stacks to improve I/O performance. Adapt the I/O request processing and explore asynchronous I/O mechanisms to overlap I/O operations with computation. Consider optimizing metadata operations and improving caching mechanisms. Additionally, leverage non-volatile storage technologies to exploit high-speed and low-latency characteristics. Incorporate high-speed interconnects and technologies like RDMA to reduce CPU involvement and high latency in data transfers and improve overall I/O performance.

- **Performance Analysis and Monitoring** Introduce or enhance performance monitoring and analysis capabilities within the kernel. This may involve adding performance counters, profiling mechanisms, or tracing fa-

cilities to enable real-time monitoring and fine-grained analysis of system behavior and performance characteristics.

Bibliography

[1] S. Mittal, "A survey of techniques for architecting and managing asymmetric multicore processors," *ACM Comput. Surv.*, vol. 48, Feb. 2016.

[2] J. Shalf, J. Bashor, D. Patterson, K. Asanovic, K. Yelick, K. Keutzer, and T. Mattson, "The manycore revolution: Will HPC lead or follow?," *Journal of SciDAC Review*, pp. 40–49, 01 2009.

[3] C. Pohl, "Exploiting manycore architectures for parallel data stream processing.," in *Grundlagen von Datenbanken*, pp. 66–71, 2017.

[4] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," *SIGPLAN Not.*, vol. 47, p. 37–48, Mar. 2012.

[5] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi, "Database servers on chip multiprocessors: Limitations and opportunities," 01 2007.

[6] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-out

processors," *SIGARCH Comput. Archit. News*, vol. 40, p. 500–511, June 2012.

[7] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia, *et al.*, "Accelerating science with the nersc burst buffer early user program," 2016.

[8] K. S. Hemmert, M. Rajan, R. J. Hoekstra, S. Dawson, M. Vigil, D. Grunau, J. Lujan, D. Morton, H. A. Nam, P. Peltz Jr, *et al.*, "Trinity: Architecture and early experience.," tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2016.

[9] "Lustre file system." `https://www.opensfs.org/lustre/`.

[10] "TOP500 List - November 2022 — TOP500 Supercomputer Sites."

[11] "Frontier supercomputer." `https://www.olcf.ornl.gov/frontier/`.

[12] "Summit supercomputer." `http://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/`.

[13] "Supercomputer fugaku." `https://www.r-ccs.riken.jp/en/fugaku/project`.

[14] "Perlmutter supercomputer." `https://www.nersc.gov/systems/perlmutter/`.

[15] A. Vajda, *Programming Many-Core Chips*. Springer US, 2011.

[16] P. Kumar and H. H. Huang, "Falcon: Scaling IO Performance in Multi-SSD Volumes," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, (USA), p. 41–53, USENIX Association, 2017.

[17] K. Zhang, Y. Zhao, Y. Yang, Y. Liu, and M. Spear, "Practical Non-blocking Unordered Lists," in *Distributed Computing* (Y. Afek, ed.), (Berlin, Heidelberg), pp. 239–253, Springer Berlin Heidelberg, 2013.

[18] S. Kashyap, I. Calciu, X. Cheng, C. Min, and T. Kim, "Scalable and practical locking with shuffling," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, (New York, NY, USA), p. 586–599, Association for Computing Machinery, 2019.

[19] "Zettabyte file system." `https://zfsonlinux.org/`.

[20] Y. Zhang, A. Rajimwale, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "End-to-end data integrity for file systems: A zfs case study.," pp. 29–42, 02 2010.

[21] E. Ates, Y. Zhang, B. Aksar, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "Hpas: An hpc performance anomaly suite for reproducing performance variations," in *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, (New York, NY, USA), Association for Computing Machinery, 2019.

[22] H. Menon, A. Bhatele, and T. Gamblin, "Auto-tuning parameter choices in hpc applications using bayesian optimization," pp. 831–840, Institute of Electrical and Electronics Engineers Inc., 5 2020.

[23] S. Robert, S. Zertal, and P. Couvee, "Shaman: A flexible framework for auto-tuning hpc systems," vol. 12527 LNCS, pp. 147–158, Springer Science and Business Media Deutschland GmbH, 11 2021.

[24] P. J. Pavan, J. L. Bez, M. S. Serpa, F. Z. Boito, and P. O. A. Navaux, "An unsupervised learning approach for i/o behavior characterization," in

2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 33–40, 2019.

[25] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular hpc i/o characterization with darshan," in *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, pp. 9–17, 2016.

[26] B. Xie, Y. Huang, J. S. Chase, J. Y. Choi, S. Klasky, J. Lofstead, and S. Oral, "Predicting output performance of a petascale supercomputer," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, (New York, NY, USA), p. 181–192, Association for Computing Machinery, 2017.

[27] S. Madireddy, P. Balaprakash, P. Carns, R. Latham, R. Ross, S. Snyder, and S. Wild, *Machine Learning Based Parallel I/O Predictive Modeling: A Case Study on Lustre File Systems*, pp. 184–204. 01 2018.

[28] D. Li, Y. Wang, B. Xu, W. Li, W. Li, L. Yu, and Q. Yang, "Pipuls: Predicting i/o patterns using lstm in storage systems," in *2019 International Conference on High Performance Big Data and Intelligent Systems (HPBD IS)*, pp. 14–21, 2019.

[29] J. Bang, C. Kim, S. Kim, Q. Chen, C. Lee, E.-K. Byun, J. Lee, and H. Eom, "Finer-lru: A scalable page management scheme for hpc many-core architectures," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 567–576, 2021.

[30] J. Bang, C. Kim, E.-K. Byun, H. Sung, J. Lee, and H. Eom, "Accelerating i/o performance of zfs-based lustre file system in hpc environment," *The Journal of Supercomputing*, 12 2022.

[31] J. Bang, C. Kim, K. Wu, A. Sim, S. Byna, H. Sung, and H. Eom, "An in-depth i/o pattern analysis in hpc systems," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 400–405, 2021.

[32] "Worldwide Semiconductor Ranking."

[33] "4th Gen Intel Xeon Scalable Processors."

[34] "Introduction to lustre." `https://wiki.lustre.org/Introduction_to_Lustre`.

[35] "Cori supercomputer," 2020.

[36] K. Elphinstone, A. Zarrabi, A. Danis, Y. Shen, and G. Heiser, "An evaluation of coarse-grained locking for multicore microkernels," *CoRR*, vol. abs/1609.08372, 2016.

[37] B. C. Ward and J. H. Anderson, "Supporting nested locking in multiprocessor real-time systems," in *2012 24th Euromicro Conference on Real-Time Systems*, pp. 223–232, 2012.

[38] B. C. Ward and J. H. Anderson, "Fine-grained multiprocessor real-time locking with improved blocking," in *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, (New York, NY, USA), p. 67–76, Association for Computing Machinery, 2013.

[39] B. McCloskey, F. Zhou, D. Gay, and E. Brewer, "Autolocker: Synchronization Inference for Atomic Sections," *SIGPLAN Not.*, vol. 41, p. 346–358, Jan. 2006.

[40] S. Cherem, T. Chilimbi, and S. Gulwani, "Inferring locks for atomic sections," *ACM SIGPLAN Notices*, vol. 43, pp. 304–315, jun 2008.

[41] G. Golan-Gueta, N. Bronson, A. Aiken, G. Ramalingam, M. Sagiv, and E. Yahav, "Automatic Fine-Grain Locking Using Shape Properties," *SIGPLAN Not.*, vol. 46, p. 225–242, Oct. 2011.

[42] D. Zheng, R. Burns, and A. S. Szalay, "A parallel page cache: Iops and caching for multicore systems," in *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'12, (USA), p. 5, USENIX Association, 2012.

[43] Y. Zhang, S. Shao, J. Zhai, and S. Ma, "Finelock: Automatically refactoring coarse-grained locks into fine-grained locks," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, (New York, NY, USA), p. 565–568, Association for Computing Machinery, 2020.

[44] S. Kalikar and R. Nasre, "Domlock: A New Multi-Granularity Locking Technique for Hierarchies," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[45] S. Kalikar and R. Nasre, "Numlock: Towards optimal multi-granularity locking in hierarchies," in *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, (New York, NY, USA), Association for Computing Machinery, 2018.

[46] K. Ganesh, S. Kalikar, and R. Nasre, "Multi-granularity Locking in Hierarchies with Synergistic Hierarchical and Fine-Grained Locks," in *Euro-Par 2018: Parallel Processing* (M. Aldinucci, L. Padovani, and M. Torquati, eds.), (Cham), pp. 546–559, Springer International Publishing, 2018.

[47] A. Amer, H. Lu, P. Balaji, M. Chabbi, Y. Wei, J. Hammond, and S. Matsuoka, "Lock Contention Management in Multithreaded MPI," *ACM Trans. Parallel Comput.*, vol. 5, Jan. 2019.

[48] M. Cai, S. Liu, and H. Huang, "tscale: A Contention-Aware Multithreaded Framework for Multicore Multiprocessor Systems," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 334–343, 2017.

[49] U. B. Nisar, M. Aleem, M. A. Iqbal, and N. Vo, "Jumbler: A lock-contention aware thread scheduler for multi-core parallel machines," in *2017 International Conference on Recent Advances in Signal Processing, Telecommunications Computing (SigTelCom)*, pp. 77–81, 2017.

[50] S. Jiang, F. Chen, and X. Zhang, "Clock-pro: An effective improvement of the clock replacement," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, (USA), p. 35, USENIX Association, 2005.

[51] S. Bansal and D. S. Modha, "Car: Clock with adaptive replacement," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, (USA), p. 187–200, USENIX Association, 2004.

[52] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: a kernel for scalable predictability," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 121–132, 2015.

[53] J. D. Valois, "Lock-free linked lists using compare-and-swap," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, (New York, NY, USA), p. 214–222, Association for Computing Machinery, 1995.

[54] Y. Son, S. Kim, H. Y. Yeom, and H. Han, "High-Performance Transaction Processing in Journaling File Systems," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, (Oakland, CA), pp. 227–240, USENIX Association, Feb. 2018.

[55] R. Mohr and P. Peltz, "Benchmarking ssd-based lustre file system configurations," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, XSEDE '14, (New York, NY, USA), Association for Computing Machinery, 2014.

[56] Z. Qiao, S. Fu, H.-b. Chen, and M. Lang, "Incorporating proactive data rescue into zfs disk recovery for enhanced storage reliability," 11 2017.

[57] V. Phromchana, N. Nupairoj, and K. Piromsopa, "Performance evaluation of zfs and lvm (with ext4) for scalable storage system," in *2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pp. 250–253, 2011.

[58] E. D. Widianto, A. B. Prasetijo, and A. Ghufroni, "On the implementation of zfs (zettabyte file system) storage system," in *2016 3rd International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE)*, pp. 408–413, 2016.

[59] D. Gurjar and S. S. Kumbhar, "File i/o performance analysis of zfs & btrfs over iscsi on a storage pool of flash drives," in *2019 International Conference on Communication and Electronics Systems (ICCES)*, pp. 484–487, 2019.

[60] D. Gurjar and S. S. Kumbhar, "A review on performance analysis of zfs & btrfs," in *2019 International Conference on Communication and Signal Processing (ICCSP)*, pp. 0073–0076, 2019.

[61] R. Mohr and A. P. Howard, "Provisioning zfs pools on lustre," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, PEARC '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[62] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir, "Taming parallel i/o complexity with auto-tuning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), Association for Computing Machinery, 2013.

[63] B. Behzad, S. Byna, Prabhat, and M. Snir, "Optimizing i/o performance of hpc applications with autotuning," *ACM Trans. Parallel Comput.*, vol. 5, mar 2019.

[64] H. You, Q. Liu, Z. Li, and S. Moore, "The design of an auto-tuning i/o framework on cray xt5 system," in *Cray User Group meeting (CUG 2011)*, 2011.

[65] A. Bagbaba, X. Wang, C. Niethammer, and J. Gracia, "Improving the i/o performance of applications with predictive modeling based auto-tuning," in *2021 International Conference on Engineering and Emerging Technologies (ICEET)*, pp. 1–6, 2021.

[66] X. Li, L. Xu, and J. Zhang, "Improving the thread scalability and parallelism of bwa-mem on intel hpc platforms," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 1858–1865, 2019.

[67] A. V. Goponenko, R. Izadpanah, J. M. Brandt, and D. Dechev, "Towards workload-adaptive scheduling for hpc clusters," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 449–453, 2020.

[68] S. Liu, E.-S. Jung, R. Kettimuthu, X.-H. Sun, and M. Papka, "Towards optimizing large-scale data transfers with end-to-end integrity verification," in *2016 IEEE International Conference on Big Data (Big Data)*, pp. 3002–3007, 2016.

[69] E. Arslan and A. Alhussen, "A low-overhead integrity verification for big data transfers," in *2018 IEEE International Conference on Big Data (Big Data)*, pp. 4227–4236, 2018.

[70] I. Foster, "Globus online: Accelerating and democratizing science through cloud-based services," *IEEE Internet Computing*, vol. 15, no. 3, pp. 70–73, 2011.

[71] B. Allen, J. Bresnahan, L. Childers, I. Foster, G. Kandaswamy, R. Kettimuthu, J. Kordas, M. Link, S. Martin, K. Pickett, and S. Tuecke, "Software as a service for data scientists," *Commun. ACM*, vol. 55, p. 81–88, feb 2012.

[72] S. Kim, A. Sim, K. Wu, S. Byna, Y. Son, and H. Eom, "Towards hpc i/o performance prediction through large-scale log analysis," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, (New York, NY, USA), p. 77–88, Association for Computing Machinery, 2020.

[73] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-wide profiling: A continuous profiling infrastructure for data centers," *IEEE Micro*, vol. 30, no. 4, pp. 65–79, 2010.

[74] Y. Lv, B. Sun, Q. Luo, J. Wang, Z. Yu, and X. Qian, "Counterminer: Mining big performance data from hardware counters," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 613–626, 2018.

[75] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 35–44, 2014.

[76] E. del Rosario, M. Currier, M. Isakov, S. Madireddy, P. Balaprakash, P. Carns, R. B. Ross, K. Harms, S. Snyder, and M. A. Kinsy, "Gauge: An interactive data-driven visualization tool for hpc application i/o performance analysis," in *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*, pp. 15–21, 2020.

[77] T. Lux, L. Watson, T. Chang, J. Bernard, B. Li, L. Xu, G. Back, A. Butt, K. Cameron, and Y. Hong, "Predictive modeling of i/o characteristics in high performance computing systems," 01 2018.

[78] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, (Santa Clara, CA), pp. 363–378, USENIX Association, Mar. 2016.

[79] A. Ali-Eldin, J. Tordsson, E. Elmroth, and M. Kihl, "Workload classification for efficient auto-scaling of cloud resources," 2013.

[80] Z. Hou, S. Zhao, C. Yin, Y. Wang, J. Gu, and X. Zhou, "Machine learning based performance analysis and prediction of jobs on a hpc cluster," in *2019 20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pp. 247–252, 2019.

[81] "GitHub - hpc/ior: IOR and mdtest."

[82] D. Bovet and M. Cesati, *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[83] M. Gorman, *Understanding the Linux Virtual Memory Manager*. USA: Prentice Hall PTR, 2004.

[84] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka, V. Vishwanath, Z. Lukić, S. Sehrish, and W. K. Liao, "HACC: Simulating sky surveys on state-of-the-art supercomputing architectures," *New Astronomy*, vol. 42, pp. 49–65, aug 2016.

[85] E.-K. Byun, G. Gu, K.-J. Oh, and J. Bang, "Optimizing lru lock management in the linux kernel for improving parallel write throughout in many-core cpu systems," *KIPS TRANSACTIONS ON COMPUTER AND COMMUNICATION SYSTEMS*, forthcoming 2023.

[86] H. Sundell and P. Tsigas, "Lock-free and practical doubly linked list-based deques using single-word compare-and-swap," in *Proceedings of the 8th International Conference on Principles of Distributed Systems*, OPODIS'04, (Berlin, Heidelberg), p. 240–255, Springer-Verlag, 2004.

[87] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," *Proceedings of the Linux Symposium*, 01 2007.

[88] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum, "The zettabyte file system," in *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, vol. 215, 2003.

[89] B. Settlemyer, G. Amvrosiadis, P. Carns, and R. Ross, "It's time to talk about hpc storage: Perspectives on the past and future," *Computing in Science & Engineering*, vol. 23, no. 6, pp. 63–68, 2021.

[90] M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, H. Bobarshad, V. C. Alves, and N. Bagherzadeh, "Computational storage: an efficient and scalable platform for big data and hpc applications," *Journal of Big Data*, vol. 6, 2019.

[91] J. Do, V. C. Ferreira, H. Bobarshad, M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, D. Souza, B. F. Goldstein, L. Santiago, M. S. Kim, P. M. V. Lima, F. M. G. França, and V. Alves, "Cost-effective, energy-efficient, and scalable storage computing for large-scale ai applications," *ACM Trans. Storage*, vol. 16, oct 2020.

[92] "Comparison of lustre with ldiskfs and zfs." `https://wiki.lustre.org/Introduction_to_Lustre_Object_Storage_Devices_(OSDs)`.

[93] G. K. Lockwood, A. Chiusole, and N. J. Wright, "New challenges of benchmarking all-flash storage for hpc," in *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*, pp. 1–8, 2021.

[94] D. Koo, J. Lee, J. Liu, E.-K. Byun, J.-H. Kwak, G. K. Lockwood, S. Hwang, K. Antypas, K. Wu, and H. Eom, "An empirical study of i/o separation for burst buffers in hpc systems," *Journal of Parallel and Distributed Computing*, vol. 148, pp. 96–108, 2021.

[95] N. Behrmann, *Support for external data transformation in ZFS*. PhD thesis, Master's Thesis. Universität Hamburg, 2017.

[96] J. Fletcher, "An arithmetic checksum for serial transmissions," *IEEE transactions on Communications*, vol. 30, no. 1, pp. 247–252, 1982.

[97] U. NIST, "Descriptions of sha-256, sha-384 and sha-512," 2001.

[98] "Zfs module parameters." `https://openzfs.github.io/openzfs-docs/Performance\%20and\%20Tuning/Module\%20Parameters.html#zfs-admin-snapshot`.

[99] "System activity report." `https://en.wikipedia.org/wiki/Sar_(Unix)`.

[100] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, p. 321–357, Jun 2002.

[101] J. Wang, W. Yoo, A. Sim, P. Nugent, and K. Wu, "Parallel variable selection for effective performance prediction," pp. 208–217, 05 2017.

[102] P. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *J. Comput. Appl. Math.*, vol. 20, p. 53–65, Nov. 1987.

[103] D. Davies and D. Bouldin, "A cluster separation measure," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. PAMI-1, pp. 224 – 227, 05 1979.

[104] R. G. Tirthak Patel, "Gift: A coupon based throttle-and-reward mechanism for fair and efficient i/o bandwidth management on parallel storage systems.," *Proceedings of the 18th USENIX Conference on File and Storage Technologies, 2020*.

[105] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

요약

고성능 컴퓨팅 시스템은 수천 개의 계산 노드와 스토리지 시스템, 그리고 이를 연결하는 고속의 네트워크로 구성되어 높은 복잡성을 가진 여러 계층의 I/O 스택을 제공한다. 고성능 컴퓨팅 시스템에서 응용의 데이터 접근에 대한 높은 I/O 성능을 제공하기 위해 메모리 관리 시스템 및 스토리지 파일 시스템의 효율적인 설계가 더욱 중요해지고 있는 추세이다. 또한 고성능 컴퓨팅 시스템 사용자는 지속적인 높은 성능을 제공받기 위해 응용 수행 시 적합한 시스템 구성 설정을 해주어야 한다.

본 논문에서는 초고성능 컴퓨팅 시스템에서 주로 쓰이는 매니코어 아키텍처를 사용 시 메모리 관리 시스템에서의 락 경합을 감소시키는 것을 첫 번째 목표로 한다. I/O 경로에서 심각한 락 경합을 일으키는 부분은 페이지 관리 시스템이며, 이는 단일 락 인스턴스를 사용하여 메모리 페이지들을 트래킹하는 LRU 리스트를 관리한다. 이 문제를 해결하기 위해 각각의 락 인스턴스를 여러 개의 하위 리스트로 분할하는 최적화 기법인 Finer-LRU 기법을 소개하였으며, 실험 결과 Linux 커널 버전 5.2.8에 적용한 Finer-LRU 기법이 순차 쓰기 성능을 57.03% 향상시키고 지연 시간을 98.94% 감소시킬 수 있었다.

본 논문의 두 번째 목표는 대부분의 고성능 컴퓨팅 시스템에서 사용하고 있는 Lustre 파일 시스템의 backend 파일 시스템으로 ZFS를 사용하였을 때 성능 최적화이다. ZFS 기반 Lustre 파일 시스템에서 낮은 I/O 성능의 근본적인 원인을 분석하고, 두 가지 최적화 접근 방식을 결합하여 dynamic-ZFS를 소개하였다. Dynamic-ZFS는 I/O 파이프라인을 병렬화하고 I/O 스레드의 수를 동적으로 조절함으로써 순차 쓰기 성능을 평균 37% 향상시켰다. 이에 본 연구를 통해 dynamic-ZFS는 ldiskfs 기반의 Lustre 파일시스템이 제공하지 못하는 다양한 기능을 제공하면서도 유사한 I/O 성능을 제공할 수 있음을 입증하였다.

마지막으로 응용의 높은 I/O 성능을 제공하기 위해 고성능 컴퓨팅 시스템에서 제공하는 여러 I/O 설정 값을 다양한 머신 러닝 기법을 사용하여 탐색하는 기법을 구현하였다. 실제 운용되고 있는 고성능 컴퓨팅 시스템에서 실제 응용의 I/O 설정 값과 성능을 포함한 데이터베이스를 구축하고 상관 관계를 분석하여 성능 예측 모델을 구현하였고, 이를 통해 높은 정확도로 다양한 I/O 설정 값에 따른 I/O 성능을 예측할 수 있음을 보였다.

주요어: 초고성능 컴퓨팅, 매니코어 아키텍처, 파인 그레인드 락, Lustre 파일 시스템, ZFS, 비지도 학습, 예측 모델

학번: 2017-25955