Ph.D. DISSERTATION

Optimizing Memory Subsystem for Efficient Resource Utilization of Data-intensive Applications

데이터 집약적 응용의 효율적인 시스템 자원 활용을 위한 메모리 서브시스템 최적화

August 2020

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

Jiwoong Park

Ph.D. DISSERTATION

Optimizing Memory Subsystem for Efficient Resource Utilization of Data-intensive Applications

데이터 집약적 응용의 효율적인 시스템 자원 활용을 위한 메모리 서브시스템 최적화

August 2020

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

Jiwoong Park

Optimizing Memory Subsystem for Efficient Resource Utilization of Data-intensive Applications

데이터 집약적 응용의 효율적인 시스템 자원 활용을 위한 메모리 서브시스템 최적화

지도교수 염헌영

이 논문을 공학박사 학위논문으로 제출함

2020 년 6 월

서울대학교 대학원

전기·컴퓨터 공학부

박지웅

박지웅의 공학박사 학위논문을 인준함

2020 년 6 월

위 원 장	김 진 수	(인)
부위원장	염 헌 영	(인)
위 원	이 재 욱	(인)
위 원	전 병 곤	(인)
위 원	손 용 석	(인)

Abstract

With explosive data growth, data-intensive applications, such as relational database and key-value storage, have been increasingly popular in a variety of domains in recent years. To meet the growing performance demands of data-intensive applications, it is crucial to efficiently and fully utilize memory resources for the best possible performance.

However, general-purpose operating systems (OSs) are designed to provide system resources to applications running on a system in a fair manner at system-level. A single application may find it difficult to fully exploit the system's best performance due to this system-level fairness. For performance reasons, many data-intensive applications implement their own mechanisms that OSs already provide, under the assumption that they know better about the data than OSs. They can be greedily optimized for performance but this may result in inefficient use of system resources.

In this dissertation, we claim that simple OS support with minor application modification can yield even higher application performance without sacrificing system-level resource utilization. We optimize and extend OS memory subsystem for better supporting applications while addressing three memory-related issues in data-intensive applications. First, we introduce a memory-efficient cooperative caching approach between application and kernel buffer to address double caching problem where the same data resides in multiple layers. Second, we present a memory-efficient, transparent zero-copy read I/O scheme to avoid performance interference problem caused by memory copy behavior during I/O. Third, we propose a memory efficient fork-based checkpointing mechanism for

in-memory database systems to mitigate the memory footprint problem of existing fork-based checkpointing scheme; memory usage increases incrementally (up to 2x) during checkpointing for update-intensive workloads.

To show the effectiveness of our approach, we implement and evaluate our schemes on real multi-core systems. The experimental results demonstrate that our cooperative approach can more effectively address the above issues related to data-intensive applications than existing non-cooperative approaches while delivering better performance (in terms of transaction processing speed, I/O throughput, or memory footprint).

Keywords: Operating System, Database, Memory Management, Double Caching, Zero-Copy, TLB Shootdown, Copy-on-Write, Checkpointing

Student Number: 2013-20794

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

1.1.1 Importance of Memory Resources

Memory is an invaluable system resource for application performance, although the extremely large performance gap between memory and storage has become narrow with the rapid advance of non-volatile memory (NVM) technologies [1]; memory is still orders of magnitude faster than disk (HDD or SSD). However, due to the cost effectiveness reason, DRAM has become a critical bottleneck for scaling data centers [2].

There are two trends that make it cost ineffective. First, with the advent of the big data era, there are growing memory demands from applications [2–4]. For example, many big data workloads, such as big data analysis, real-time graph processing, and machine learning, require in-memory computing to quickly respond to rapidly changing environments. Second, DRAM scaling is slowing down in terms of capacity growth [2,4,5] while current processor trends

show an increasing number of cores in each node [6, 7]. These result in less memory per core, which aggravates the memory demand. In recent years, these two trends have caused a global DRAM supply shortage and have resulted in rise in the price of DRAM [4, 8, 9]. These trends seem to be continuing owing to the increasing smart devices in the 5G era [10].

Therefore, efficient utilization of memory resources is more important than ever. In line with this trend, our goal is to optimize the OS memory subsystem so that it allows data-intensive applications to efficiently utilize memory resources while achieving the best performance from a single server.

1.1.2 Problems

Due to current lack of OS support, data-intensive applications, such as Database Management System (DBMS) and In-Memory Database (IMDB), cannot achieve both high performance and efficient resource utilization. The following is the three problems to be addressed in this dissertation. They are different but all related to memory efficiency.

Double Caching. Due to its high performance, memory is often used as a caching layer for slower secondary storage such as hard disk drives (HDDs). For example, many modern operating systems take advantage of unused memory for caching of file contents, so that the next requests to the cached contents can be served without expensive I/O operations. Additionally, many cloud data stores (such as MongoDB [11] and MySQL [12]) have their own user-level caching layer that can be greedily optimized for best performance.

However, ironically, when every performance-critical software tries to retain the data in their own layer, there can be a negative impact on performance due to inefficiency in memory utilization. It is generally known as *double caching*. It could occur when several hierarchical software layers cache the same data

redundantly due to the lack of coordination among them. It matters for performance because it reduces the effective memory size. Unfortunately, it is a rather common issue, mainly but not limited to, between datastore and OS.

Most datastores [13–15] try to solve this problem by skipping all caching layers except one (either user-level or kernel-level). Having only one caching layer for the entire system stack absolutely eliminates double caching at the expense of losing an opportunity to utilize both strengths of each caching layers. In this dissertation, we claim that utilizing both caching layers with appropriate coordination can lead to higher performance than disregarding one or the other.

Copy-based I/O. Performance interference has been an important issue for cloud computing where resource sharing is the key to yield cost benefits [16–19]. In such an environment, multiple applications that demand different system resources (e.g., CPU, memory, storage) can be co-located within a server, under the assumption that they do not interfere with each other [20, 21]. For example, memory-intensive workloads can be co-located with I/O-intensive ones. However, in such a case, there can be serious performance interference. This is because copy-based I/O, which modern operating systems (OSs) use by default, involves an additional memory copy between user and kernel memory during I/O, consuming CPU cycles and memory resources. This extra use of memory resources for I/O can be problematic in cloud computing due to contention on shared resources such as the last level cache (LLC) and memory controller.

To address this problem, many zero copy I/O schemes have been proposed, but none of them simultaneously provide 1) transparent copy avoidance via read/write system calls and 2) the benefits of kernel-level caching. In this dissertation, we claim that practical transparent zero-copy read I/O operation can be realized by addressing two main challenges: 1) page remapping overhead and

2) Copy-on-Write (CoW) fault overhead.

Fork-base Checkpointing. Consistent checkpointing plays a crucial role in improving fault tolerance of long-running applications. In particular, many in-memory databases (IMDBs) exploit it to give persistence to in-memory data [22–25]; data can be restored from the consistent point-in-time backup in case of a crash. The process of consistent checkpointing in IMDBs involves two steps. The first step is to take a *in-memory consistent snapshot* of point-in-time data and the next step is to write them to a file for persistence. A `fork()` system call can be leveraged to quickly create a virtual memory snapshot of the database and to allow the child process (*checkpointer*) to write the records (or key-value pairs) to a file in background while the parent process (*servicer*) handles client requests. Several industrial IMDBs, such as Redis [23] and HyPer [24], use this simple fork-based checkpointing model.

However, there is a well-known problem—but unsolved—in the fork-based checkpointing. It is the increase in the memory footprint during checkpointing [26]. As the checkpointing continues, the actual total size of memory consumed by *servicer* (*parent*) and *checkpointer* (*child*) will increase with frequent data updates on *servicer*-side, owing to copy-on-write (CoW) page duplication. In *checkpointer*'s point of view, there is no reason for the data that have been written to a file to be preserved in memory. However, the duplicated pages cannot be reclaimed until *checkpointer* finishes writing all data set into a file and exits. Depending on the speed of storage device and the update intensity of workloads, this behavior potentially double the memory footprint of the IMDB during checkpointing. In this dissertation, we claim that physical memory dump-based scheme with minor OS support can effectively mitigate the memory footprint issue.

1.2 Contributions

The contributions are summarized as follows:

- We present a new approach that exploits cooperative caching between application buffer and kernel buffer for efficient memory utilization. It utilizes OS page cache as a victim cache, which can be easily applied to self-caching user applications with only few lines of changes (less than 500 LOCs) to the target applications. We also show that choosing the I/O behavior when issuing I/Os has a higher opportunity for optimization than when opening the file. There are such cases where sometimes buffered I/O is preferred, and at other time, direct I/O is the best. With the new read/write function call, user-level caching layer can make a smarter decision whether it bypasses OS page cache or not.

- We introduce an efficient, transparent, and practical zero copy read I/O for preventing memory interference caused during I/O. We propose several optimization techniques to overcome two main challenges that other previous studies have not addressed: 1) page remapping and 2) CoW fault overhead.

- We propose a new fork-based checkpointing scheme that relies on memory dump and minor operating system supports to mitigate the memory footprint problem of the existing fork-based checkpointing scheme. We implement our scheme as an user-level library providing high-level APIs. As a result, MDC can be applied to existing software with minimal modifications (only few tens of lines of code for Redis).

1.3 Outline

This dissertation is structured as follows:

- **Chapter 2** covers the background about the Linux kernel memory management and Linux support for efficient resource utilization of applications.

- **Chapter 3** introduces `DBIO`, our memory-efficient cooperative caching scheme. We first explain the problems of existing datastore architecture and propose our new architecture. We describe the details of design and implementation of our scheme and evaluate our scheme on the cloud key-value store workloads and the traditional OLTP workloads.

- **Chapter 4** introduces `z-READ`, our memory-efficient zero-copy I/O scheme. We start with explaining the problems of copy-based I/O and introduce the two main challenges for zero-copy I/O. We give details of how we can address the challenges and evaluate our scheme in a mixed workloads scenario.

- **Chapter 5** introduces `MDC`, our memory-efficient fork-based checkpointing scheme. We explain the problems of existing fork-based checkpointing and present our approach to mitigate the problem. We describe details of our scheme for checkpointing and restoring. Then, we evaluate our scheme on the cloud key-value store workloads.

- **Chapter 6** summarizes and concludes the dissertation. It also points out directions for future work.

Chapter 2

Background

Our approaches heavily rely on many functions supported by OS. In this chapter, we explain the Linux kernel memory management and the current Linux support for efficient resource utilization of applications to help understand the rest of the dissertation.

2.1 Linux Kernel Memory Management

The basic unit of memory Linux kernel handles is *page* [27]. There are two types of pages in Linux. File pages are backed by a file whereas anonymous pages are not. An example of the latter is pages in the heap or stack of a user process while the page cache is an example of the former.

2.1.1 Page Cache

The page cache plays the role of the kernel-level caching layer to hide the slowness of the underlying non-volatile secondary storage such as HDDs, giving

user applications an illusion that underlying storage is as fast as memory. As long as there is enough memory available, kernel aggressively tries to use the remaining memory to store the file contents in a page unit as page cache. Whenever there is memory pressure, page cache is more preferable to be selected as a victim page for reclamation.

2.1.2 Page Reclamation

When memory is full, some pages should be evicted to make a room to deal with the new memory allocation requests. In order to determine which page is evicted first, Linux kernel uses a simplified Least Recently Used (LRU)-based algorithm as a page reclamation policy [27]. For the simplified LRU, Linux kernel maintains four linked lists of pages: active/inactive lists for anonymous pages and file pages respectively.

When the page is allocated, it is first placed in the inactive list. It is moved to the active list when accessed again. Only pages in the inactive lists are selected as candidates for eviction. Active lists shrink when the number of pages in inactive lists is lower than the number of pages in active lists. In that case, a certain number of pages in active lists are moved to inactive lists. Some of them will remain in active lists if they have been accessed more than once after being inserted in active lists. When kernel chooses an anonymous page as a victim, the page cannot be discarded without storing its contents somewhere else (swap area) unlike a file page that can be discarded either, immediately for clean pages or after being flushed to disk for dirty pages. Kernel also can *deactivate* a file page to make it a good reclaim candidate by moving it to the inactive list.

Swapping out anonymous pages is a more expensive operation than evicting file pages because swapping always requires one I/O to write the page to

disk(swap device) and another I/O to read the page from disk at next access. Although this cost seems similar to the cost of evicting dirty file page, user-perceived latency can be very different because user applications do not expect disk-level latency when accessing those pages swapped out. In the other hand, user applications already assume that it takes some time for their explicit I/O requests to finish. In this case, even though there are no cached pages in memory, user-perceived latency is the same as expected, thus no problem arises. Furthermore, kernel might swap out the clean pages that have not been modified in the user buffer, generating unnecessary I/Os, which can be avoided if user applications could just discard them.

Linux provides a manual configuration of *vm swappiness* that determines how aggressively the kernel swaps out anonymous pages. While swappiness ranges from 0 to 100, the default value is 60, which is reasonable for desktop machines. Setting this value to 0 for server machine allows user applications to experience no unexpected performance degradation by preventing swapping. However, it could result in the forced termination of the applications by kernel Out-Of-Memory(OOM) killer in case of memory pressure.

2.1.3 Page Table and TLB Shootdown

A page table is a kernel data structure that holds page table entries (PTEs), each of which stores the mapping between a virtual address and a physical address. Because a CPU uses a virtual address for all instruction fetches and data access, the memory management unit (MMU) on each core is tasked with the translation of a virtual address to a physical address, which is accomplished by traversing the hierarchical page table (a so-called *page table walk*). A TLB is a per-core cache for fast virtual-to-physical address translation. Since the coherency of TLBs is not preserved by modern CPUs, the OS must keep the

TLB synchronized at the software level.

TLB shootdown is an operation for this purpose, which removes stale TLB mappings from TLBs. TLB shootdown must be performed after manipulating the PTE mappings for the page remapping technique. However, the problem is that TLB shootdown is known to be very costly on the latest x86 multicore systems with several tens of cores because it involves sending inter-processor interrupts (IPIs) to other cores in the system [28]. Generally, the TLB shootdown cost becomes larger when the system has more cores.

2.1.4 Copy-on-Write

Copy-on-write (CoW) is an optimization technique that was originally designed to defer the expensive copying of data until the first write attempts. It is a "lazy" optimization that becomes effective only when it can completely avoid copying by being lazy. This technique allows the page to be safely shared by multiple processes without worrying the shared page will be updated. Zero-copy I/O and fork-based checkpointing both rely on the kernel CoW mechanism.

After setting a write protection bit in the PTE corresponding to the target page, any attempt to modify the contents of the page triggers a CoW fault. Upon a CoW fault, the kernel page fault handler allocates a new page, maps the faulting virtual address to the physical address of the new page, copies the contents of the old page to the new page, and performs TLB shootdowns to invalid the stale mappings in TLB so that the application can continue modifying the data at the faulting address. That is to say, handling a CoW fault is apparently more costly than "eager" page copying.

However, in Linux kernel, there is an interesting optimization that can mitigate this latency for a special case where the faulting page is private[1]. In such a

[1] A private page is a page that is mapped by only one process.

case, the CoW fault handler allows the process to in-place update the faulting page without the page duplication process. Besides the reduced latency, this also prevents memory usage from growing.

2.2 Linux Support for Applications

Linux kernel provides several functions to applications for efficient resource utilization. Applications can use these functions to accelerate their performance or to prevent wasting system resources.

2.2.1 fork

`fork()` is originally designed to allow a process to efficiently create a new process. When a process (*parent*) calls `fork()`, the kernel creates copies of all the process-related kernel data for the new process (*child*). For duplicating address space, only page table entries (PTEs) are copied. As a result, the *child* process has an exact copy of the *parent*'s address space, mapping to the same set of physical pages. Since the pages are shared between two processes, the PTEs of the both processes are made write protected during `fork()`. The time complexity of `fork()` is $O(n)$ where n is the actual memory size of the calling process because the number of page table entries to be copied is proportional to the memory size. Therefore, the latency of `fork()` can become problematic as the *parent* process uses more memory.

2.2.2 madvise

The purpose of a `madvise()` system call is to allow applications to give some advice to the kernel about the pages for the given address range. Although there are many advice types predefined in the kernel, we give details of only two advice: `MADV_DONTNEED` and `MADV_FREE`. For `MADV_DONTNEED`, the kernel im-

mediately releases the target pages so that they can be freed when there is no reference to them. The kernel also updates the associated PTEs as if a physical page frame is not yet assigned. Subsequent access to these pages results in either a zero-filled page for anonymous private mappings or repopulating the contents from the mapped file for shared mappings [29]. Similar to `MADV_DONTNEED`, `MADV_FREE` is used to let the kernel know that the pages are no longer required. However, it differs that the kernel only handles the PTEs, but delays freeing of the pages until memory pressure happens.

2.2.3 Direct I/O

Some applications do not benefit from kernel-level file caching. For example, self-caching applications such as DBMS manage their own memory buffer pool to cache file contents, thus duplicating the kernel-level cache. In this case, the caching behavior in the kernel wastes CPU and memory resources. Furthermore, because the page cache has limited space for caching and is shared across other applications and the kernel, this useless caching of data can cause other useful file data to be pushed out of the page cache.

`Direct I/O` is designed to allow applications to avoid such inefficiency by completely bypassing the kernel cache. When a file is exclusively opened with the `O_DIRECT` flag, all of the I/O of the file always results in physical disk I/O, as it is directly served from/to the user buffer by DMA. Because `direct I/O` uses the same interfaces as buffered I/O, applications can be easily converted to use `direct I/O` with minor changes (e.g., the alignments of the file offset and memory buffer). `Direct I/O` may help increase the overall system-wide performance because the elimination of memory copy leads to the reduced CPU utilization during I/O. However, `direct I/O` may not improve the I/O performance of the calling application. For most cases, giving up kernel-level caching

significantly lowers the user-perceived I/O performance.

2.2.4 mmap

An `mmap` system call is originally designed for memory mapped I/Os. `mmap` allows applications to map the region of a file into their address space. Once this preparation process is completed, the application must access the mapped file data through the returned address using a normal memory operation instead of traditional read/write system calls. Upon memory access, the kernel populates the page cache associated with the access if the related region of the file is not yet cached. Otherwise, the application can access the data without a switch to kernel mode as if it is in the user buffer. In this way, `mmap` enables applications to utilize kernel-level caching without incurring memory copy costs.

However, `mmap` is not almighty. It is difficult to use correctly for performance-critical applications because, aside from the disparate interfaces, the functionality of the `mmap` interface is quite different from what read/write interfaces provide [30]. For example, unlike read/write calls, `mmap` does not guarantee that the mapped region of a file resides in memory; this means that I/O can occur in the middle of a critical section of applications. To avoid this, applications have to make extensive use of other system calls: `mlock` to pin data (page) in memory and `munlock` to allow the data to be evicted from memory. Furthermore, the performance of `mmap` can be dramatically decreased if the working set size of the mapped file is larger than the physical memory size owing to the high cost of page mapping/unmapping, which are as part of kernel page reclamation [31]. In these regards, the application of `mmap` to existing applications that use read/write system calls is difficult; it may require substantial redesign of the applications.

Chapter 3

Memory Efficient Cooperative Caching

3.1 Motivation

3.1.1 Problems of Existing Datastore Architecture

Although the detailed architecture of datastore varies, there is always a software component responsible for in-memory data caching. In Figure 3.1, we have classified the architectures of existing datastores into three categories according to the caching layers that they rely on for data caching.

Both User and Kernel Firstly, Figure 3.1(a) shows the architecture that utilizes both kernel and user caching layers (but not in an efficient way) with buffered I/O. However, double caching problem occurs here because user buffer and kernel buffer do not cooperate.

Kernel Only Figure 3.1(b) describes the architecture that relies on kernel buffer only for caching persistent data and exploits the mmap system call and the madvise system call, which gives hints about future memory access patterns

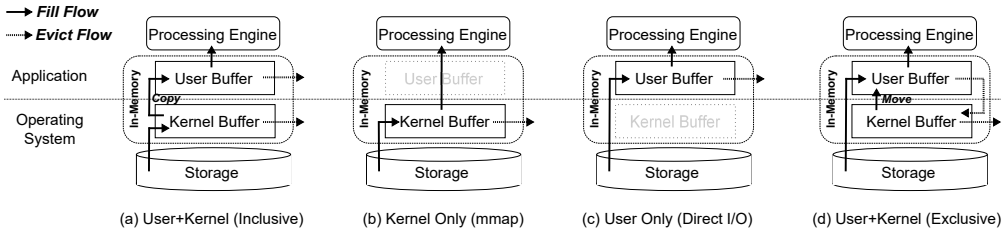


Figure 3.1: Existing datastore architecture and our proposed architecture

to kernel.

Having a caching layer in kernel-level only, mmap eliminates the copy overhead between user and kernel space, but requires page table manipulation whose cost sometime exceeds the copy overhead. Moreover, in fact, mmap is easy to use but hard to use correctly. Most applications usually expect that the target data to compute already presents in memory. In order to satisfy the expectation, aggressive use of system calls is required; `madvise` with `MADV_WILLNEED` to prefetch the given range of data into kernel buffer, `mlock` to guarantee that the data is in memory, and `munlock` to allow kernel to evict them if necessary. With the appropriate use of these system calls, it can mimic the user-level buffer but sometimes make the system design more complex than just using read/write system calls. Furthermore, with mmap, we cannot do fancy things such as data compression.

User Only Figure 3.1(c) shows the architecture that uses direct I/O to maintain its own caching layer without intervention of kernel buffer. User-level caching layer has two primary strengths compared to kernel-level caching layer.

First, with swapping disabled, data remains in memory once it is loaded into the user buffer unless explicitly discarded by the application. This is quite important for application side optimization. For example, with this guarantee, we can ensure that I/O does not occur while in the middle of a critical section

which should be kept as short as possible.

Secondly, self-caching applications can directly manage user-level buffer with their own replacement policy based on their explicit knowledge of data while there is no other option for kernel buffer except simplified LRU replacement policy.

However, maintaining the caching layer only in user-level has several downsides.

First, it is hard to fully utilize the memory resources. In order to maximize the memory utilization, self-caching applications should allocate as much memory as possible for user-level buffer. However, it is undesirable that one user process occupies the entire memory because some of the memory is needed to run not only other applications in the system but also Linux kernel itself. Therefore, in order to avoid memory pressure causing the unpredictable performance degradation or the forced termination of the process at worst, a self-caching application should leave enough amount of memory for the others in a conservative manner. For example, for a dedicated MySQL/InnoDB server, it is recommended to use only 80% of total amount of memory.

Second, without swapping, memory allocated for user buffer can be used solely by the application itself. When an application greadily allocates huge memory for its own buffer then does not fully make use of them, it is such a waste of system resources. Swapping is designed to resolve the problem, enabling the system to steal the currently unused memory from the applications. However, when disabling swapping for performance reasons, memory allocated for user buffer cannot be reclaimed unless the application explicitly frees the memory.

Third, more memory might be needed to manage the user-level buffer. Self-caching applications require extra metadata for managing their user-level buffer while kernel reuses the page structure for page cache, which is needed anyway

Table 3.1: Relationship between Buffer Pool Size and RSS for MySQL/InnoDB

Buffer Pool Size(GB)	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0
Resident Set Size(GB)	1.8	3.1	4.4	5.7	6.9	8.2	9.4	10.7

for metadata of a chunk of memory. The more complex the replacement policy is, the more extra memory user-level buffer is likely to need for metadata. Table 3.1 shows the relationship between the buffer pool size and the resident set size (RSS) of MySQL server, which is the real memory usage by the MySQL server. In our system setup, we found that about 300MB of metadata is needed for 1GB of user-level buffer. Therefore, given the same memory size, kernel-level buffer usually can hold more data than user-level buffer.

3.1.2 Proposed Architecture

To combine the strengths of both user and kernel buffer, we propose a two-tier cache hierarchy in memory; an isolated, reserved user-level cache that is fast but consumes more memory for the same amount of data and a shareable kernel-level cache that is relatively slow due to the copy overhead but requires less memory. Figure 3.1(d) shows the proposed architecture with our scheme, called `DBIO`, that exploits the kernel buffer as a victim cache for user-level buffer. With this architecture, one can expect the strengths of the user-level buffer as well as the full utilization of system memory resources that is hard to achieve with user-level buffer alone.

3.2 Related Work

Linux kernel VFS layer optionally provides a cleancache [32] that is a page-granularity victim cache for kernel page cache. The mechanism of cleancache is very similar to `DBIO` *evict clean*. However, they work at different layers; clean-

cache is a victim cache for kernel page cache whereas DBIO uses page cache as a victim cache for user buffer. Moreover, cleancache works transparently with respect to applications and kernel while with DBIO, applications can be explicitly involved in cache management. Furthermore, applications that use direct I/O do not gain the benefits of cleancache because they do not utilize page cache. However, DBIO and cleancache can be utilized together, being complementary with each other.

Recently, using flash storage as an extension of memory has been researched in [33–37]. All these works target the workload whose working set is much larger than memory. On the other hands, DBIO is not effective at all when working set is so much larger than memory that no in-memory cache hit occurs because DBIO only expands the memory capacity somewhat by utilizing the memory hidden in kernel space and by keeping only one copy of data as possible. However, memory is much faster than flash storage thus fully efficiently utilizing the given memory resource is always valuable. Moreover, DBIO does not lower the I/O performance even where no in-memory cache hit occurs, thus it is always worth trying.

Li et al. [38] use write hints to minimize the memory inefficiency caused by lack of coordination between storage client cache and storage server cache. The intention of the write request from storage client is delivered to storage server in the form of write hint, which is simply a tag. DBIO works similar way but on different layers: server and client vs user and kernel. DBIO also allows clean pages in first-tier cache (user buffer) to be moved to second-tier cache (kernel buffer) without generating unnecessary physical I/Os.

Li et al. [39] show that providing applications with an ability to pass information to file systems, can greatly boost application performance, helping kernel make better decisions. They utilize and extend the existing hint mech-

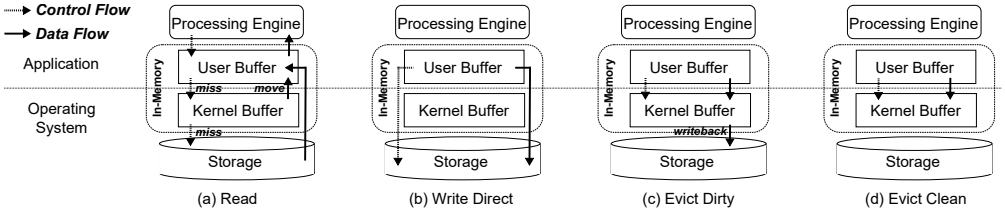


Figure 3.2: Overview of DBIO

anism such as posix_fadvise, so that applications can give kernel more information on metadata cache control, file open/creating, and readahead. On the other hand, `DBIO` integrates the hint mechanism with each read/write request, allowing kernel handles each I/O differently depending on the page cache policy hints. Their work aims to accelerate the performance of the underlying file system especially for Ceph [40] whereas `DBIO` is designed to maximize the performance of self-caching applications such as datastore.

3.3 Design and Implementation

3.3.1 Overview

The high-level design of `DBIO` is shown in Figure 3.2. It supports four distinct I/O paths, each of which is for different user context: *read*, *write-direct*, *evict-dirty*, and *evict-clean*.

Read `DBIO` *read* is kind of a hybrid version of buffered read and direct read, which closely resembles the buffered read for accessing the cached data but exploits direct read for the uncached data in order to prevent the pollution in kernel buffer.

When the application issues the read request on the file opened with `DBIO` I/O mode, it firstly tries to find the requested data in kernel buffer. If succeeded, it copies the found data into the user buffer, then deactivate the page cache in order for the page to be preferentially evicted in case of memory pressure. Note

that we do not free the page cache until needed, thus there might be many duplicate pages before page reclamation triggered by kernel evicts those pages from memory asynchronously in batch. This *relaxed* design choice is advantageous for performance because it excludes the task of freeing the page from the critical path of the read and allows those tasks to be handled asynchronously in batch only when needed. With this design, our two-tier cache does not follow strict exclusion, behaving more like non-inclusive cache rather than exclusive cache.

If it fails to find the data in kernel buffer, it follows the direct I/O path to read the data directly from storage into the user buffer. Therefore, no unnecessary copy overhead is incurred and the duplicate copy of the data, which is deactivated right away anyway, can be avoided. Note that for self-caching applications such as datastores when the data is in the user buffer, the request to access the data is served from the user buffer without generating I/Os to the storage. Keeping a copy of data in kernel buffer could reduce effective memory size for caching especially when memory is full of useful data because some pages containing useful data must be evicted for the new data that will not be used after being copied into the user buffer. This results in lower kernel buffer hit ratio and lower I/O performance.

Looking up kernel buffer for every read request will add a little overhead especially when the data is found to be not in memory. However, the extra lookup overhead will be hidden by the longer disk latency. On the other hand, at success in finding the data in kernel buffer there is a chance to change the disk-level latency to the memory-level latency thus it is always worth looking into the kernel buffer before going to disk.

The application can use `posix_fadvise` system call with `POSIX_FADV_DONTNEED` to force kernel to free the cached page for the specified region. It can be used

with buffered I/O mode in order to imitate `DBIO` *read* by calling it once the application reads the data from kernel buffer. However, in comparison with `DBIO` *read* it needs unnecessary data copy between user and kernel layer when loading the data from storage. Futhermore, it adds an extra system call overhead involving context switch between user and kernel space.

Write-Direct `DBIO` *write-direct* works exactly the same as direct write. When there is a demand of writing the data in the user buffer to persistent storage, it is more desirable to bypass kernel buffer because the data will still remain in the user buffer even after writing. If buffered write is used for those data, memory inefficiency occurs as a result of two duplicate copies of the data.

One would think that when it comes to user perceived latency, buffered write is better than direct write, because for buffered write users are notified when the data is written to kernel buffer whereas for direct write users have to wait for the data to be written to storage. Therefore, they might be willing to take some memory inefficiency for better user perceived latency.

This is, however, not the case in production-level system. Many production-level systems have a hardware RAID controller to aggregate the multiple inexpensive disks into a single logical disk for either performance, reliability, or capacity. H/W RAID controller usually has caching capability with battery backup unit(BBU), so that in the case of power failure the data in RAID cache has enough time to be safely written back to the disks. Moreover, solid state drive (SSD) also has a DRAM cache for write buffer and read cache. With the existence of fast cache, storage layer also could lie to kernel as if the data is persistently written to the underlying disk but in fact the data only succeeds in being buffered in a physical cache of the underlying disk. Therefore, there is no difference in performance between buffered and direct write as long as the cache of the disk is capable of buffering all the incomming data while doing

writeback. In fact, many applications such as DBMS explicitly call the *fsync* or *fdatasync* system call whenever it needs to make sure that their data is really persistent, thus disk-level latency cannot be avoidable both for buffered and direct write anyway. Besides, buffered write always incurs the small overhead for copying the data to kernel buffer.

Evict-Dirty In order to benefit from using the kernel buffer as a victim cache, we need to insert something useful into it. `DBIO` *evict-dirty* and `DBIO` *evict-clean* play this role.

We reused the existing buffered write I/O path for `DBIO` *evict-dirty* I/O path. It can be used when the victim page, which is chosen for evicition in the user buffer, needs writeback due to changes in its content.

Although a 4KB buffered write might contribute to one page cache eviction in kernel buffer for buffering the write in memory pressure, it is worth more than bypassing kernel buffer because data in higher level cache (user buffer) is likely to be more valuable than anything in lower level cache (kernel buffer) according to temporal locality.

Note that, in this case (unlike the previous case of `DBIO` *write-direct*) the application will remove data in user buffer after writeback finishes, thus there will be only one copy of data in kernel buffer throughout the whole software stack at the end. With this copy-and-delete mechanism, the victim data is logically moved to lower level cache, being demoted. `DBIO` puts the demoted page into the active list in kernel so that it has a higher priority than any other pages already in kernel buffer.

Evict-Clean Last but not least, in fact, `DBIO` *evict-clean* is the biggest contributor that makes it a complete bidirectional data transfer between the user layer and the kernel layer in our two-tier cache.

When a storage engine feels the necessity of vacating pages in its user buffer, unlike dirty pages, the clean pages selected for victim are discarded immediately in the sense that they would be able to be read from storage anytime. This is, however, not optimal if there is spare memory in kernel buffer to store the data. It is very common because even the storage engine utilizing a very large user buffer leaves some memory for kernel and other applications in order to avoid the risks of swapping and forced termination. The big problem is that there is no such a semantic for a write into kernel buffer only, ensuring that no writeback to storage happens. Buffered write, which is currently the only way of utilizing kernel buffer for write, will eventually generate physical I/O even if we know that the target data has not been modified since it was read from storage.

To provide such a semantic, we break down buffered write I/O path into two parts, caching (buffering) and writeback, and then write the new I/O path that performs only caching part of buffer write. `DBIO` *evict-clean* copies the data into the page cache then manipulates the state of the page cache as if it have just been read from storage. Therefore, it skips unnecessary file system activity, such as logging, metadata update, write, etc. In the case that there has already been the page cache for the data, `DBIO` skips the task of copying the data as well. Regardless of the existence of the page cache, `DBIO` moves the page cache to the active list to keep the priority of the pages in memory.

By providing a way that the lowest priority page in user buffer can become the highest priority page in kernel buffer, it allows two different caching layers to behave like a single layer.

3.3.2 Kernel Support

In this section, we explain the other kernel support besides the `DBIO` I/O paths. Fistly, we add new file open flag `O_DBIO` like `O_DIRECT` so that all the next I/O read/write requests on the file opened with `O_DBIO` follow the proper `DBIO` I/O path.

To pass the user I/O context to the existing non-modified system call, we utilize some bits within the request I/O size passed as input parameter. Specifically, we utilize the upper four bits to encode the `DBIO` user contexts: *read*, *write-direct*, *evict-dirty*, and *evict-clean*. Note that this trick does not affect the correctness of kernel I/O behavior at least in 64-bit Linux kernel because Linux kernel limit the I/O size to something that fits in the 32-bit signed *int*. Therefore, upper 32 bits in the size parameter are ignored anyway. Moreover, we do not think there is a need for supporting very large I/O request because `DBIO` is designed for self-caching application that manages their own user buffer in a unit of multiple pages (4KB, 8KB, 16KB, etc), which is relatively small. However, although our prototype of `DBIO` uses the trick, we could implement the same thing in a cleaner way by utilizing *ioctl*.

Having received the I/O request from the application, kernel extracts the `DBIO` user context from the encoded size parameter only when the target file is opened with `DBIO` I/O mode so that it can follow the proper `DBIO` I/O path.

We also implement a user library that provides the `DBIO` function calls replacing the read/write system call. These functions take the `DBIO` flag as one of the input parameters and internally perform automatic encoding for the given `DBIO` user context, issuing the read/write system call with the encoded size parameter.

3.3.3 Migration to DBIO

In this section, we explain the application changes needed for utilizing `DBIO`. We believe that `DBIO` fits for many self-caching applications managing their own file contents cache in a unit of fixed-size block, and that improved performance can be easily achieved with only minimal modification on the applications. `DBIO` requires the application to implement a lock mechanism in order to ensure that only one I/O request is issued for the same region of the same file at the same time because mixing of direct I/O and buffered I/O might result in data loss or corruption. However, it is not enough if other applications access the data using buffered I/O while the self-caching application is writing the data using `DBIO` *write-direct*. To prevent such cases, we can simply modify the kernel to make sure that once a file is opened with the `DBIO` open flag, the file cannot be opened unless the file is closed by the last opener.

To show the effectiveness of `DBIO`, we have analyzed and modified the InnoDB storage engine, which is used for MySQL by default. The followings are the user contexts when InnoDB issues I/O.

Open InnoDB can configure the flush method for data and log files. When setting this option to `O_DIRECT`, InnoDB uses `O_DIRECT` to open only the data files not the log files. Then, InnoDB sets the file open flag to `O_DIRECT` using fcntl with `F_SETFL` just before each I/O request. This is all that InnoDB does for `O_DIRECT`. Kernel automatically handles the I/O request differently depending on whether the open flag is set to `O_DIRECT` or not. We add an extra configuration parameter `O_DBIO`, as one of the InnoDB flush method so that InnoDB sets the file open flag to `O_DBIO`. Note that by doing this, `DBIO` will work only for DB data files.

Read In InnoDB, all accesses to the data that are not in buffer pool generate

I/Os to the underlying file system by issuing the read/write system calls. All the reads for the file opened with `O_DBIO` follow the `DBIO_READ` I/O path in kernel. Therefore, no modification is required on application-side for reads.

Write Flushing is the activity that writes pages from buffer pool to disk. InnoDB maintains two lists for flushing: flush list and LRU list. The flush list is used when the ratio of dirty pages in the buffer pool reaches *innodb_max_dirty_pages_pct*. This flushing activity is triggered for making the changes durable. Therefore, these pages do not have to be evicted from buffer pool even after flushing. In this case, buffered write will create the unnecessary copy of the data in kernel buffer. In order to avoid this, the application should directly write the data to storage, bypassing the page cache. When the target file is opened with `O_DBIO`, kernel performs the `DBIO_WD` I/O for the file if no `DBIO` flag is passed with the `DBIO` write function call. Therefore, simply replacing the existing write system call with the `DBIO` write function call is enough for the write user context.

Evict LRU list, another list for flushing, maintains all the used pages in LRU order. When there is no page in free list and InnoDB needs to read data from storage, the least recently used page is selected as a victim.

Since every victim page will be evicted from the buffer pool after ensuring that the modified page has been synchronized with storage, we can use `DBIO_ED` and `DBIO_EC` for this context. When the victim page is dirty, InnoDB calls a write system call to writeback the modified data to storage. For this case, we make InnoDB to call `DBIO` write function call with the `DBIO_ED` flag instead.

On the other hand, InnoDB originally discards clean pages to quickly meet the demand for new reads. To give a second life to these pages, we modified the InnoDB to call a `DBIO` write function call with `DBIO_EC` so that those pages are copied into kernel buffer. Because as the result of `DBIO` *evict-clean*, the

clean data will be cached in memory without affecting the original data, we can skip *doublewrite* that InnoDB provides for data integrity in case of partial page writes. *doublewrite* forces innodb to write data twice for table space writes. We chose to write the data synchronously for evict clean because we already know that it will take only the page copy latency and that vacating the buffer pool is very urgent job.

3.4 Evaluation

In this section, we evaluate the effectiveness of `DBIO` by comparing throughput and page cache read hit ratio of `DBIO` with direct I/O and buffered I/O.

3.4.1 System Configuration

Hardware Setup All experimental evaluations are conducted on a server machine with two quad-core Intel Xeon E5606 2.13GHz, 12GB of RAM, one RAID 0 comprised of 4 1TB HDDs for DB data files, and one RAID 0 comprised of 2 1TB HDDs for DB log files. For our hardware RAID controller, LSI MegaRAID SAS, we set its configuration as following: no read-ahead for read policy, write-back for write policy, and direct I/O for cache policy. Therefore, RAID controller does not prefetch the sequential sectors of the logical drive, notifies the kernel of the completion of the write as soon as the data is written to disk, and makes sure that any reads are not buffered in its cache memory. In short, the cache of RAID controller can be used only for write cache not for read cache.

The client machine is a DELL R715 with two 16-core AMD Opteron 6282SE 2.6GHz. The server and client machines are directly connected with 10GbE so that the network bandwidth does not become a performance bottleneck. The server runs the MySQL server 5.7.9 [12] with the modified InnoDB storage

engine on top of the modified Linux 4.1.24, in which we implemented a prototype of `DBIO` whereas the client runs one of the two benchmarks at a time, remotely accessing the database running on the server.

MySQL/InnoDB Setup We set the InnoDB page size to 4KB, which was 16KB by default, due to our current lack of support. We place the MySQL data and MySQL log files on separate logical RAID-0 volumes to protect them from affecting each other. Also, we make InnoDB use two log files in a circular fashion, setting the size of each log file to 256MB, and keep the default setting of *innodb_flush_log_at_trx_commit* for full ACID compliance.

For each benchmark, we vary the Innodb flush method (`O_DBIO` for `DBIO`, `O_DIRECT` for direct I/O, fsync for buffered I/O) and the buffer pool size from 1GB to 8GB. Note that we limit the maximum buffer pool size to 8GB because we found that setting it to larger than 8GB is very likely to lead to out of memory errors. As inferred from Table 3.1, when we set the buffer pool size to larger than 8GB, the RSS of MySQL server exceeds the amount of the main memory physically available in the system. Also note that buffer pool size must be a multiple of 1GB because InnoDB automatically rounds up the buffer pool size to a value that is a multiple of *innodb_buffer_chunk_size* $\times$ *innodb_buffer_pool_instances*, which are 128MB and 8 by default, respectively. All the rest of the configurations are the same as defaults.

3.4.2 Methodology

We use MySQL server as a backend database system for all the benchmarks. Before evaluation, we insert the initial data needed for all benchmarks to the MySQL server and then do the physical backup of all databases. Before we ran each test, we restored the databases from the physical backup to guarantee that the experimental result is not affected by the previous test. For each test, we

run *mysqldump* [41] to warm-up the memory, then conduct the benchmark test for 3600 seconds to obtain the stable results. *mysqldump* loads the structures and contents of MySQL databases and tables into the user buffer to perform the logical backup. It is worth mentioning that for buffered I/O and `DBIO`, it might warm-up the kernel buffer along with the user buffer because they both utilize the kernel buffer. Before each test, we drop the page cache, dentries, and inodes so that each experiment is run under the same condition for fair comparison.

For evaluation, we compared `DBIO`'s throughput, which is basically provided by the benchmarks, to direct I/O and buffered I/O. We also choose to compare the page cache hit ratio rather than the combined hit ratio for the user buffer and the page cache. Because when it comes to the combined hit ratio it is hard to distinguish the difference among three I/O modes while even 1% of the difference can significantly impact the performance. For the same workload with the identical size of the buffer pool the user buffer hit ratios of three I/O modes are equal, therefore comparing only the page cache hit ratio can help explain more about the performance improvements by `DBIO`. However, page cache statistics cannot be obtained without some efforts.

Page Cache Profiler We wrote a simple kernel module to collect the page cache statistics only for the DB data file. For `DBIO`, we can easily achieve the goal by counting the page cache hits and misses only on the `DBIO` I/O path while for buffered I/O, it is hard to separate it from the general page cache statistics. To step aside this problem, we create another version of kernel that has a clone of buffered I/O path but it can be followed only when the target file is opened with `DBIO` mode. As a result, we can track the page cache statistics on the DB datafile for both buffered I/O and `DBIO`, making InnoDB use `O_DBIO` as the flush method. Note that we do not consider direct I/O for collecting page cache statistics because it does not utilize the page cache anyway.

Just before we start each workload, we load the kernel module, initializing the page cache statistics and enforcing the kernel to start counting the page cache events. When the workload ends, we unload the kernel module after printing out the summary of the page cache statistics including the page cache references, hits, misses, hit ratio, and miss ratio for read and write I/O requests.

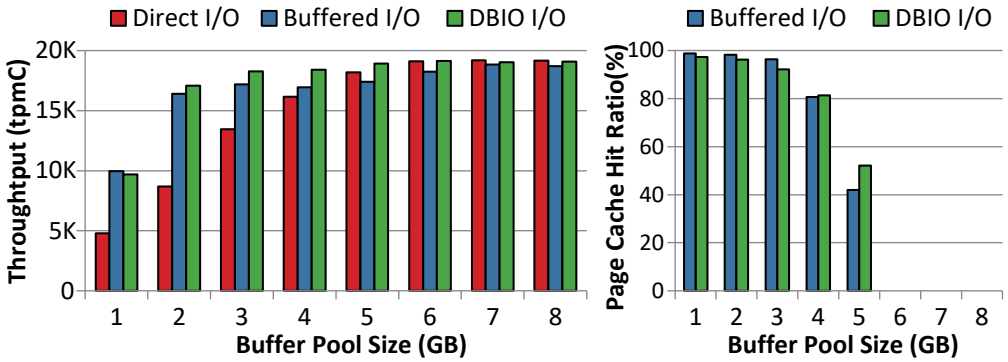
3.4.3 TPC-C Benchmarks

TPC-C benchmark [42] is an industry standard OLTP benchmark that simulates an order-processing environment where a population of users execute various transactions against a database. It uses nine different tables and five distinct transactions. The transactions are New-order, Payment, Order-Status, Delivery, and Stock-Level, each of which involves a combination of select, update, insert, and delete operations. The scaling factor of TPC-C benchmark is *warehouse*, which is directly related to the data size. TPC-C benchmark reports the performance in new-order transactions per minute.

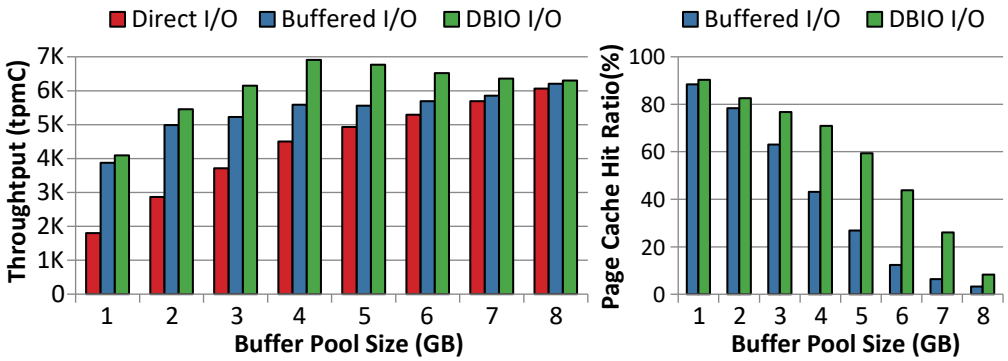
We use the tpcc-mysql [43], which is one of the TPC-C implementations provided by Percona. While running the transactions of 128 clients, we conduct the experiments for two scaling factors: 50 and 150 warehouses whose database sizes are approximately 5GB and 15GB, respectively.

Results Figure 3.3 shows the throughput and the page cache read hit ratio according to the varying size of InnoDB buffer pool.

When we use 50 warehouses for the dataset, it can be in-memory workload depending on the buffer pool size. In left of Figure 3.3a, we can see that the performance is saturated when the buffer pool size is equal or larger than the dataset size. However, when all data cannot reside in the buffer pool, `DBIO` performs slightly better than buffered I/O, given the same size of the buffer pool.



(a) 50 Warehouses, 5GB



(b) 150 Warehouses, 15GB

Figure 3.3: TPC-C Benchmark Results

Looking at the page cache read hit ratio as shown in right of Figure 3.3a, we found that the page cache read hit ratio of `DBIO` is actually slightly lower than that of buffered I/O despite higher throughput. After deep investigation, we found that higher read miss latency occurs for buffered I/O, which means that buffered I/O makes the disks too busy to handle the actual user requests, doing read-ahead. It might be due to lack of parallelism in HDDs even though we use RAID-0 to provide some sort of parallelism. However, we do not think that the result would change even when using SSDs because buffered I/O provides only slightly higher page cache hit ratio in spite of the aggressive read-ahead. Note that when the buffer pool size is large enough to hold all the dataset all data

is served in user buffer thus there is no page cache reference for such cases.

In left of Figure 3.3b, performance gap between buffered I/O and DBIO becomes the widest in the 4GB buffer pool. With buffered I/O, the effective memory size becomes minimum when using half of system memory as the buffer pool because all data are duplicated across the buffer pool and the kernel buffer.

The ideal buffer pool size for DBIO is hard to predict without testing because there is a trade-off between the size of buffer pool and the size of kernel buffer; the larger the buffer pool, the smaller the kernel buffer. Since each of DBIO two-tier cache has own strengths (one is fast but consumes more memory for the same data and another is relatively slow but consumes less memory), the best ratio between them can differ depending on the workload. In this case, using DBIO with the 4GB buffer pool shows the best performance among all results.

Compared to buffered I/O and direct I/O, DBIO improves the throughput of TPC-C, for the same buffer pool size by up to 24% and 128% respectively, for the best result regardless of the buffer pool size by up to 12% and 14% respectively. Right of Figure 3.3b demonstrates that DBIO achieves significantly higher page cache read hit ratio than buffered I/O, effectively maintaining exclusiveness of data.

3.4.4 YCSB Benchmarks

Yahoo! Cloud System Benchmark (YCSB) [44] includes a common set of workloads for evaluating the performance of different key-value cloud stores. We choose two representative workloads for our evaluation. Workload A is an update heavy workload consisting of 50% reads and 50% updates while workload B is a read heavy workload consisting of 95% reads and 5% updates. For both workloads, the popularity of data follows the zipfian distribution with zipfian constant 0.99, which generates an obviously skewed workload. Each workload is

tested with 128 client threads and unthrottled operations per second. For each workload, we have tested on three different datasets: 2000k, 5000k, and 10000k records (4GB, 10GB, and 20GB, respectively).

Results Figure 3.4 and Figure 3.5 show the throughput and the page cache read hit ratio varying the buffer pool size for YCSB workload A and B which are update-heavy and read-heavy, respectively. Not surprisingly, for all datasets `DBIO` is more effective for workload B because `DBIO` is designed for read optimization. In Figure 3.4a and 3.5a, two YCSB workloads with 2000K records both have similar trends to TPC-C benchmark with 50 warehouses in Section 3.4.3 although `DBIO` has more meaningful impact on performance until the workload becomes in-memory.

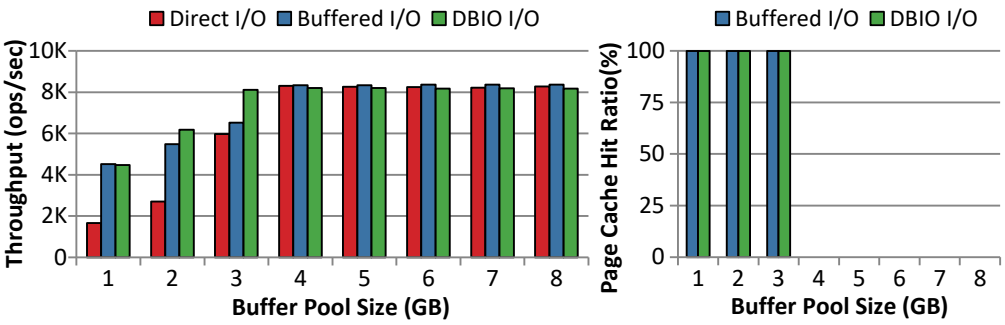
However, when the data size is slightly larger than system memory size we could see substantial performance improvements with `DBIO` in YCSB workloads with 5000K records, which are shown in in Figure 3.4b and 3.5b. In left of figure 3.4b and 3.5b, we find the interesting thing that when using buffered I/O, throughput drops as the buffer pool size grows and gets closer to 5GB. Throughput increases again after the 5GB buffer pool. The drop in throughput is due to the reduced effective memory size as we mentioned in Section 3.4.3.

In contrast, `DBIO` works incredibly well for all the size of the buffer pool because `DBIO` effectively utilizes the system memory, eliminating double caching problem. For the workload A with 5000K records, compared to buffered I/O and direct I/O, `DBIO` improves the throughput, for the same buffer pool size by up to 75% and 168% respectively, for the best result regardless of the buffer pool size by up to 17% and 7% respectively. For the workload B with 5000K records, compared to buffered I/O and direct I/O, `DBIO` improves the throughput, for the same buffer pool size by up to 335% and 1065% respectively, for the best result regardless of the buffer pool size by up to 33% and 44% respectively.

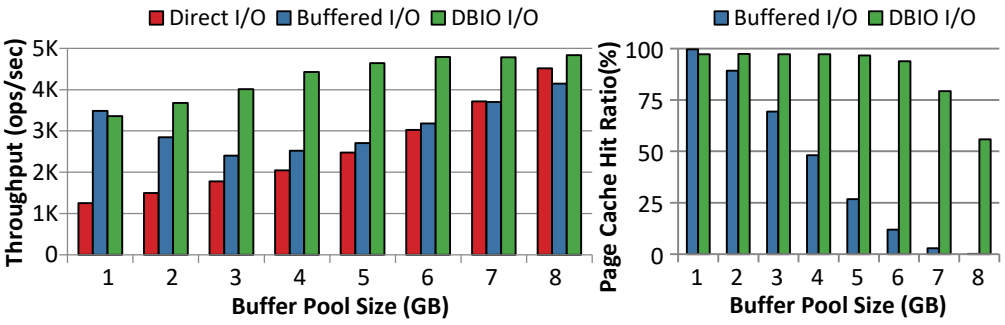
Right of Figure 3.4b and 3.5b indicate that the page cache read hit ratio is greatly improved on `DBIO` in comparison with buffered I/O.

For the workloads with 10000K records in Figure 3.4c and 3.5c, the amount of performance improvement with `DBIO` becomes smaller but still considerable. It also shows that the throughput of `DBIO` decreases monotonically with the increase of buffer pool size. The reason of this is because when the data size is much larger than the system memory, retaining more data in memory becomes more important than having slightly better latency for data access.

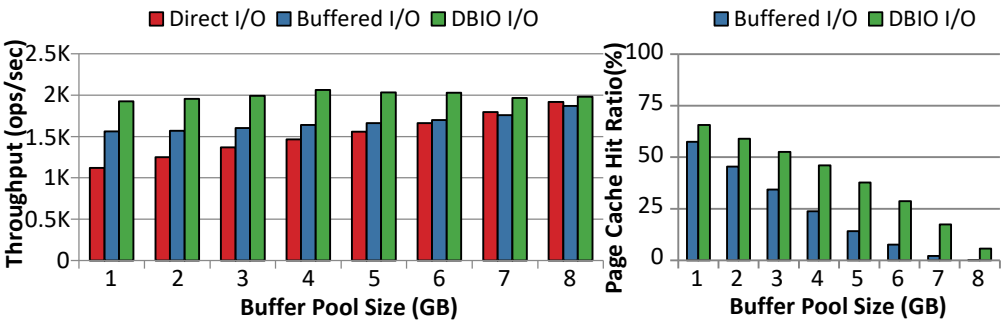
For the workload A with 10000K records, compared to buffered I/O and direct I/O, `DBIO` improves the throughput, for the same buffer pool size by up to 25% and 72% respectively, for the best result regardless of the buffer pool size by up to 10% and 8% respectively. For the workload B with 10000K records, compared to buffered I/O and direct I/O, `DBIO` improves the throughput, for the same buffer pool size by up to 64% and 171% respectively, for the best result regardless of the buffer pool size by up to 28% and 27% respectively.



(a) 2000k records(4GB)

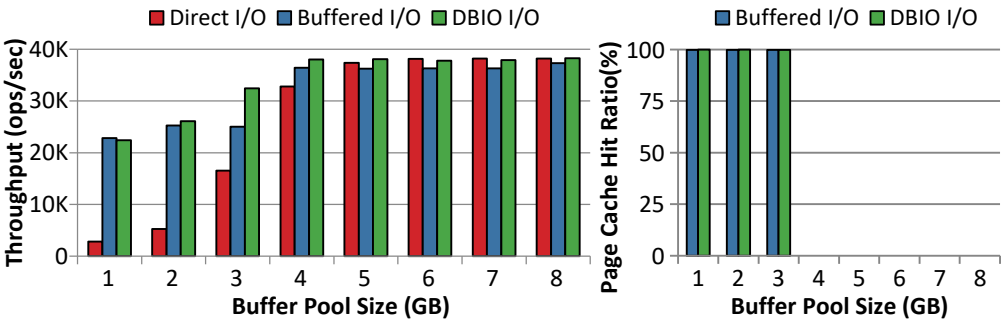


(b) 5000k records(10GB)

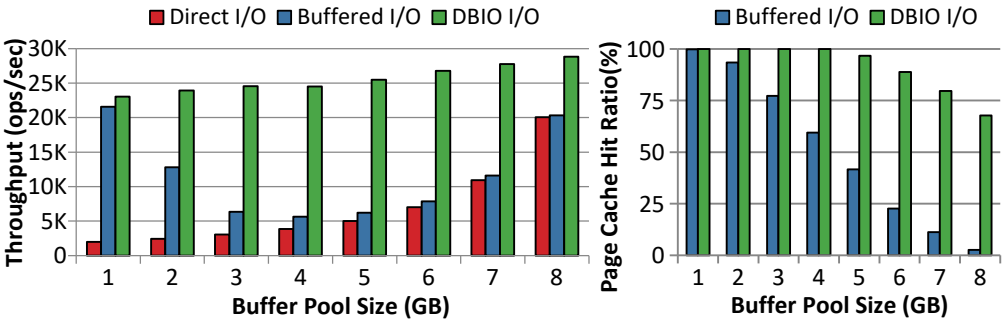


(c) 10000k records(20GB)

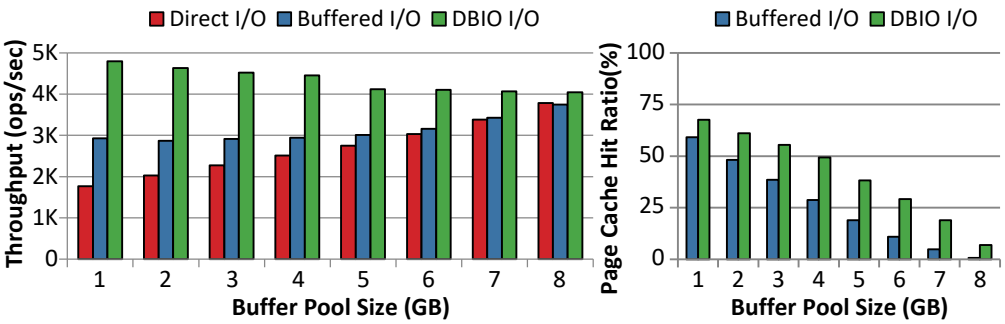
Figure 3.4: YCSB Benchmark Results for Update-intensive Workload A



(a) Workload B, Read-Intensive, 2000k records(4GB)



(b) Workload B, Read-Intensive, 5000k records(10GB)



(c) Workload B, Read-Intensive, 10000k records(20GB)

Figure 3.5: YCSB Benchmark Results for Read-intensive Workload B

3.5 Summary

In this chapter, we present DBIO, a new file I/O mode for efficient user-level caching with the help of kernel page cache. DBIO utilizes OS page cache as a victim cache for user-level file content cache, constructing a two-tier cache hierarchy in memory. Using various OLTP benchmarks with a diversity of workloads, we demonstrate that utilizing both user and kernel caching layer could achieve better performance and higher in-memory cache hit ratio for the given system memory. We believe DBIO can be applicable to various system layers suffering from double caching, which we leave for future work.

Chapter 4

Memory Efficient Zero-copy I/O

4.1 Motivation

4.1.1 The Problems of Copy-Based I/O

Traditionally, read/write system calls have been used to access files. By default, the Linux kernel uses buffered I/O (copy-based I/O) for read/write system calls [27]. For simplicity, the term "`READ`" will be used throughout this chapter to refer to a read system call with buffered I/O.

Figure 4.1(a) shows the caching behavior of `READ`. When handling read requests, the kernel first reads a section of a file in the page cache and then copies the data from the page cache to the application buffer. At the expense of a single memory copy, this caching behavior can realize substantial gains in the user-perceived I/O performance for subsequent access; it helps to hide the relative slowness of the underlying storage. For this reason, buffered I/O is usually the default I/O mode enabled by most operating systems, and it is great choice in most cases. However, there are some pitfalls.

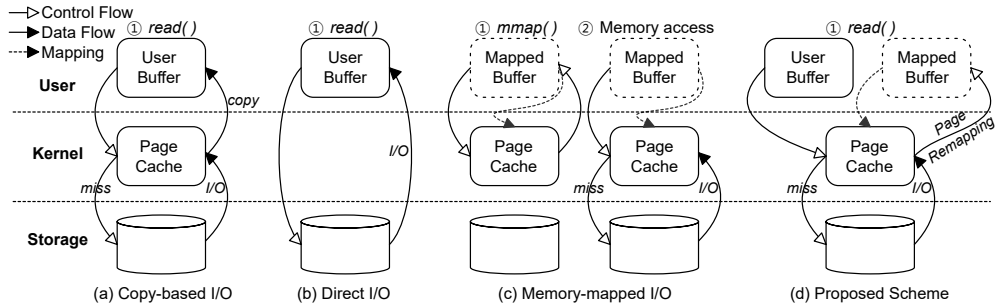


Figure 4.1: Existing file I/O interfaces and the proposed scheme

Table 4.1: Latency breakdown of a 4 KB `READ` for a cache hit

Type	Page Copy	Page Search	Etc.	Total
Latency (ns)	7815 (68%)	1322 (11%)	2422 (21%)	11559

First, a memory copy during buffered I/O adds latency overhead. In terms of the latency, the memory copy overhead may be negligible compared with the disk I/O time for a page cache miss. However, the situation is different for a page cache hit. We study the average latency of a read I/O operation for a page cache hit, measuring the time for each function in the kernel read path. Table 4.1 provides a detailed breakdown of the latency for a 4 KB read in our system (see §4.4 for detailed system configuration information). As shown in the table, the memory copy overhead is significant; it accounts for over two-thirds of the total read time.

Second, a memory copy between the user and kernel buffers consumes memory bandwidth. For modern systems that support high I/O bandwidth, copy-based I/O can consume significant fraction of total memory bandwidth [45]. For example, a four-lane interface speed of PCIe 4.0 NVMe SSD can reach up to 8 GB/s. By RAID-zeroing four SSDs, the aggregate bandwidth can be up to 32 GB/s, which is comparable to the bandwidth of a two-channel DDR4-2133 memory system (17 GB/s x 2 channels = 34 GB/s). Moreover, when all data

are cached in the kernel page cache, the I/O performance can be bottlenecked by the memory bandwidth. In our system, the measured peak I/O bandwidth is about 25 GB/s, which is in agreement with the sustainable memory bandwidth measured by using the STREAM benchmark [46]. In other words, in such a case, a memory copy for every I/O request leads to the saturation of the memory bandwidth.

4.2 Related Work

4.2.1 Zero Copy I/O

For decades, many studies related to zero copy [47–52] have been carried out to reduce the data transfer overhead in various areas in computer systems. Although each study targets different layers, they can fall into two categories.

Direct I/O Schemes. There are zero copy schemes that support direct user-level access to devices for user applications, removing the kernel from the critical I/O path. EMP [51] is a zero copy message passing layer, completely implemented on the programmable Alteon NIC. Thanks to their MPI support, a user application can directly access the NIC through the MPI. Kesavan et al. [52] explored the application of zero copy NIC DMA to in-memory databases. These works rely on the device's DMA capabilities, which allow the device to directly access the host memory. Our study is in line with these works in terms of eliminating memory copy overhead. However, in contrast to their focus on networking, where the processing time is dominated by the copying of packet data from the NIC to the host memory, we focus on storage I/O. In storage I/O, the total read time is dominated by the time required to read the data from storage device, not the memory copy overhead, which is negligible. In other words, in terms of latency, there is no real performance gain from removing memory copy if data is served from disk. In fact, it could worsen the user-

perceived I/O performance because every I/O request always goes to storage (no page cache hit).

Memory-Mapped I/O Schemes. There are other approaches that provide applications with the ability of accessing the shared buffer. Druschel et al. [47] presented *fbufs*, a fast buffer for a network subsystem based on immutable buffers that can be shared across a domain. Thadani and Khalidi's work [48] and IO-Lite [49] both extended *fbufs* from the network to the file system for general-purpose OSs (UNIX). While our work is also based on the memory-mapped I/O scheme, the main difference from above is the application transparency. Their non-transparent approaches may provide higher I/O performance than transparent approaches including ours, by skipping the tasks required for application transparency. However, a major disadvantage of the non-transparent approaches is that applications have to be modified to use new APIs and programming models, which may significantly burden developers.

Unlike the aforementioned studies, Chu [50] presented a transparent zero copy scheme based on page remapping and CoW techniques. It only required minimal changes (e.g., buffer alignment) to an application, keeping existing interfaces unchanged. However, it targeted network subsystem, which have several different characteristics from our target, storage I/O subsystem. For example, his zero copy read scheme could completely eliminate the CoW fault overhead by allowing an application to take the ownership of the kernel page, rather than to share them. By ensuring no sharing of pages, there is no write protection needed. This was possible because by semantics the same network data is not read again by applications. However, in contrast to network data, storage data can be read many times even by different applications. To benefit from kernel-level caching, the kernel page needs to be shared with applications but be write protected using the CoW technique. Therefore, for storage I/O, the CoW fault

overhead is inevitable. Moreover, his scheme was evaluated in the UltraSPARC system, where the benefits of zero copy, which were realized by preventing cache thrashing, could exceed the costs of page remapping. In the latest x86 server that has a few tens of cores, the cost of page remapping can be higher than that of page copy due to the increased TLB shootdown cost explained in §2.1.3.

4.2.2 TLB Shootdown

Many efforts have been devoted to TLB shootdown mitigation [28,53–55]. Linux kernel tracks the address space that is active on each core and only sends TLB shootdown IPIs to the cores that currently use the address space to which the target TLB mapping belongs. By reducing the number of the cores that are involved in a TLB shootdown operation, this optimization helps to avoid some unnecessary shootdowns. For instance, for single-threaded applications, no remote TLB shootdowns are required for TLB synchronization, only local TLB flushes. However, for multi-threaded applications, the kernel still must send IPIs to all of the cores that share the same address space.

Villavieja et al. [53] presented *DiDi*, a new hardware architecture with a shared second-level TLB. This second-level TLB contains all of the information about which TLBs hold which PTEs, thus allowing lightweight TLB invalidation. Awad et al. [54] proposed the concept of self-invalidating TLB entries that allow the OS to skip TLB shootdown operations for the *expired* TLB entries. However, both studies require hardware changes.

LATR [55] is a lazy TLB shootdown approach for virtual memory operations such as *free* and *page migration*. Its key idea is to defer TLB invalidation until a context switch and to prevent the reuse of virtual and physical memory in the meantime. However, unlike the case of *LATR*, TLB shootdown cannot be deferred in a transparent zero copy file I/O scheme. To keep the semantics of

`READ`, TLB synchronization has to be handled before `READ` returns to the user mode. Otherwise, other cores might use the stale mapping when accessing the shared user memory.

Amit [28] presented a TLB shootdown optimization that detects the private mappings by tracking an *accessed bit* in a PTE. It leveraged the fact that when the page is accessed, the access bit of its PTE is set by hardware. His work also introduced a software-based *direct TLB insertion* technique that allows the direct insertion of a mapping into a TLB of a core without the access bit set. With the access bit tracking and direct TLB insertion techniques, the kernel can determine if the mapping is private to the current core. However, his approach can introduce non-negligible overheads (up to 9%) for general applications that do not frequently change memory mappings because direct TLB insertion is performed for every demand paging request. Therefore, only few applications could take advantage of the optimizations. On the other hand, we create a new application area (storage I/O) for his optimization techniques. Our scheme is more effective and practical, because 1) storage I/O is more repeatedly used for many applications than memory mapping operations, and 2) in `z-READ`, direct TLB insertion is performed only for read I/O requests, rather than demand paging requests, thus no performance penalty for other normal applications.

4.2.3 Copy-on-Write

The CoW technique has been used in many studies to improve system performance and resource utilization [56–58]. However, it is reasonable to assume that the reason why the CoW optimization helps to improve performance in these studies is because CoW faults occur relatively rarely (compared to the number of copy operations avoided). If CoW faults frequently occur, eager copy is better choice than lazy copy (CoW). Our focus is to anticipate the likelihood

of occurrence of CoW faults and to dynamically determine whether or not to perform zero copy.

4.3 Design and Implementation

Our goal is to tackle two main challenges for a transparent zero copy scheme to be applied in storage system in modern x86 server: 1) page remapping and 2) CoW fault overheads. In this section, we describe the design and implementation of `z-READ` that provides transparent zero copy read file I/O while maintaining the benefit of kernel-level caching. Figure 4.1(d) shows an overview of our proposed scheme. Note that the term "*kernel page*" is used to refer to a page of the page cache, while "*user page*" is used to refer to an anonymous page allocated by the user.

4.3.1 Prerequisites for z-READ

Because page remapping is performed with page granularity, `z-READ` requires that both the user buffer and requested file offset are page-aligned. Note that even if they are not aligned, `z-READ` can fall back to `READ` without error.

However, the requested I/O size and file size do not need to be page-sized if and only if the user buffer and file offset are page-aligned. When the requested I/O size is not page-sized, the kernel performs `READ` for the last *user page* that will be partially filled. For example, if the requested I/O size is 9 KB and the file size is sufficiently large, the first two 4 KB *user pages* are remapped to *kernel pages* using `z-READ`, while the remaining 1 KB of data is copied from the associated page cache to the user buffer. This method is implemented in this way because page remapping for the last partial *user page* may result in the unintended overwriting of data next to the user buffer.

Unlike the requested I/O size, the file size does not affect the feasibility of

Algorithm 4.1: z-READ Main Procedure

```
   Input: file_offset, user_buf, buf_len
   Output: bytes_read
 1 if user_buf ≠ page aligned or file_offset ≠ page aligned then
 2     fall back to READ for all pages;
 3 else
 4     idx = 0;
 5     foreach offset in a page unit (start from file_offset) do
 6         page = find_page_cache(offset);
 7         if page == NULL then  perform disk I/O and goto 6;
 8         else
 9             target_pages[idx++] = page;
10             page→ref_count++;
11             if idx == BATCH_SIZE then
12                 idx = 0;
13                 Batching Procedure( );
14     if idx != 0                          /* Dealing with unhandled pages */
15      then
16         Batching Procedure( );
```

`z-READ`. If the end of the file is reached before the requested size of the I/O is handled, the kernel can still perform `z-READ` even for the partial *kernel page* at the end of the file. This makes sense because the rest of the last *kernel page* is filled with zeros, and the application issuing I/O already assumes that the data in the user buffer can be fully overwritten.

4.3.2 Overview of z-READ

Read. For `z-READ` requests, the kernel first checks the alignment of the user buffer and file offset and then continues to search the page cache for the requested blocks (Algorithm 4.1, lines 1–6). After finding the *kernel page* containing the requested data, the kernel stores their pointers in the page array to handle page remapping and TLB flushing in a batched manner (lines 8–9). It also increases the reference count of the target pages by one to prevent them from being evicted from the page cache while they are still mapped (line 10).

Algorithm 4.2: Batching Procedure

```
   Input: start_addr, end_addr, target_pages[ ], nr_target_pages
 1 preempt_disable( );
 2 Page Remapping Procedure( );
 3 if IPIALL_bitmap ≠ 0 then
 4 |   bitmap = IPIALL_bitmap;
 5 |   cpu = -1;
 6 |   Bitmap-based Flush(bitmap, IPIALL, cpu);
 7 foreach set bit i in zREAD_CPU_bitmap do
 8 |   if OTHER_bitmap[i] ≠ 0 then
 9 |   |   bitmap = OTHER_bitmap[i];
10 |   |   cpu = i;
11 |   |   if i == current_cpu then
12 |   |   |   Bitmap-based Flush(bitmap, LOCAL, cpu);
13 |   |   else
14 |   |   |   Bitmap-based Flush(bitmap, SINGLE, cpu);
15 Direct TLB Insertion( );                    /* Please refer to [28] */
16 preempt_enable( );
```

When the number of the stored page pointers reaches *BATCH_SIZE*, which will be detailed in §4.3.3, or the system call is about to return to the user with the unhandled pages, the kernel starts the batching operation (lines 11–16).

A single batch consists of two main jobs: page remapping and TLB flushing. For page remapping, the kernel first obtains the PTE of each *user page* by a page table walk and then checks if the *user page* is already mapped to the *kernel page* (Algorithm 4.3, lines 3–7). If so, the kernel just decreases the reference count of the *kernel page* pointed by the PTE to allow the page to be reclaimed in the case of memory pressure (line 11). Otherwise, to avoid unnecessary memory consumption, the *user page* that was originally pointed by the PTE is marked to be lazily freed by the kernel page reclamation mechanism later (line 9). Regardless of whether or not the page is mapped, the kernel then manipulates the PTE mapping to point to the *kernel page* and sets the write protection bit in the PTE to prevent the *kernel page* from unintended modification by the

Algorithm 4.3: Page Remapping Procedure

```
   Input: start_addr, end_addr, target_pages[ ], nr_target_pages
   Output: zREAD_CPU_bitmap, IPIALL_bitmap, OTHER_bitmap[], pte_ptrs[ ]
 1 foreach addr in a page unit from start_addr to end_addr do
 2     idx = calculate_from_address(addr);
 3     pte = page_table_walk(addr);
 4     pte_ptrs[idx] = pte;
 5     user_page = pte_to_page(*pte);
 6     zread_cpu = get_zread_cpu(*pte);
 7     if zread_cpu == -1                                  /* Not mapped yet */
 8      then
 9         mark the user page to be lazily freed;
10     else
11         user_page→ref_count--;
12     manipulate PTE to point the target_pages[idx];
13     set write protection bit in the PTE;
14     set_zread_cpu(pte, current_cpu);
15     if access bit == 1 then
16         set the idx-th bit in IPIALL_bitmap;
17     else
18         if zread_cpu != -1 then
19             set the idx-th bit in zREAD_CPU_bitmap;
20             set the idx-th bit in OTHER_bitmap[zread_cpu];
21         else
22             set the idx-th bit in IPIALL_bitmap;
```

user (lines 12–13). The kernel also updates *z-READ CPU*, which is the CPU core ID where the previous `z-READ` was performed and which is stored in the unused six bits (52–57) of the corresponding PTE (line 14).

After page remapping, the kernel performs TLB flushing for the manipulated PTEs. In our scheme, we minimize the number of remote TLB shootdown operations by several optimization techniques, which will be explained in §4.3.3. After synchronously waiting for the remote CPUs to invalidate the stale PTEs for the requested range, the kernel returns to the user space. Note that we disable preemption during the batching operation for accurate private PTE detection (Algorithm 4.2, line 1). Inaccurate detection results can allow cores

to access the incorrect page using stale PTEs cached in their TLBs.

Write. Since our zero copy I/O scheme focuses on read file I/O, the kernel behavior on write has not been altered. Note that even if the application issues a write request with the user buffer already mapped to *kernel pages*, the kernel can copy the data from the buffer to the target page cache; the copy is actually performed between *kernel pages*.

Modify. When an application attempts to modify the user buffer that is mapped to a *kernel page*, a CoW fault occurs owing to the protection bit in the PTE set by the previous `z-READ` operation. The kernel CoW fault handler then checks if *z-READ CPU* is set. If so, it allocates a new page and modifies the PTE mapping to point to the new page. This is followed by shooting down TLB entries, whose cost can be reduced by private PTE detection optimization. After TLB flushing, the kernel decreases the reference count of the old page that was pointed by the PTE and then returns to the user.

Free. When an application terminates or frees the user buffer, the remapped page must be properly handled. When this occurs, the kernel checks *z-READ CPU* for each PTE. If *z-READ CPU* is set, this means that the page to be freed is a *kernel page*. Since the reference count of the *kernel page* has been increased owing to page remapping in `z-READ`, the kernel decreases it by one for these pages in a batched manner. Only the pages whose reference counts reach zero are freed. Note that since the pages are about to be freed, the kernel does not have to roll back the pages to the exact state in which they were before page remapping.

4.3.3 TLB Shootdown Optimization

The TLB shootdown overhead occupies the majority of the page remapping time. To minimize the overhead of TLB shootdown, we present several tech-

niques including private PTE detection and batching TLB shootdown operations.

Private PTE Detection. As discussed in §4.2.2, the *direct TLB insertion* technique plays a significant role in the detection of private mappings. In this work, we fully implement the *direct TLB insertion* scheme in Amit's work [28], unlike in our previous study [59] where we simplified the implementation under the assumption that the user buffer is not accessed by other cores during I/O. In contrast to our prior work, we can directly insert a PTE into the local TLB without setting the *access bit* of the PTE. Thus, there is no need to manually clear the *access bits*. We refer the reader to [28] for the details of the implementation of this technique.

With help of this technique, we can utilize the PTE's *access bit* to determine if the PTE has been cached by any remote TLBs. After page remapping and TLB flushing, the kernel performs *direct TLB insertion* for private PTE detection (Algorithm 4.2, line 15). Until the PTE is evicted from the TLB, local access to the virtual address is possible without setting the *access bit* owing to the cached PTE in the local TLB. In contrast, access to the address by the remote cores will trigger the hardware *page table walk*, resulting in setting the corresponding PTE's *access bit*.

The kernel can utilize these facts to eliminate unnecessary remote TLB shootdowns. For example, on the next `z-READ` request with the same user buffer, the kernel checks the *access bit* of the corresponding PTE (Algorithm 4.3, line 15). If set, it means that some remote cores have accessed the page and may have the PTE cached in their TLBs. In this case, the kernel must send TLB flush IPIs to all of the remote cores sharing the same address space while performing a local TLB flush (line 16); this is the same behavior as default Linux. When referring to this type of TLB flush, `IPI ALL FLUSH` is used. If clear, the

kernel checks if the previous core that performs `z-READ` on the page is equivalent to the current core (Algorithm 4.2, line 11). If so, a local TLB flush (`LOCAL FLUSH`) is sufficient to guarantee that no TLBs contain stale mappings for the addresses corresponding to the user buffer (line 12). Otherwise, it performs a single remote TLB shootdown (`SINGLE FLUSH`) for the previous *z-READ CPU* without the need for a local TLB flush (line 14). Note that for the first `z-READ` request with the *user pages* that are not remapped, the kernel cannot exploit this optimization because the *access bits* are probably set when applications initialize the *user pages* prior to use (Algorithm 4.3, line 22). With this optimization, the kernel can replace some remote TLB shootdown operations (`IPI ALL FLUSH`) with either `LOCAL FLUSH` or `SINGLE FLUSH`. For single-threaded applications, we disable *direct TLB insertion* because even without it, the native kernel carries out only `LOCAL FLUSH` as we explained in §4.2.2. In this case, our private PTE detection only adds overhead.

Batching TLB Shootdowns. For a multiple page-sized `z-READ` request, we can reduce the number of TLB flushes by batching them rather than performing TLB flushes on every page remapping. `z-READ` applies batched page remapping and TLB flushing with a batch size of 32.

In terms of the latency, flushing the entire TLB (`GLOBAL FLUSH`) is preferred to performing too many individual flushes (`INDIV FLUSH`). `GLOBAL FLUSH` requires two instructions that read and write from/to the CR3 register, while every individual flush needs an *invlpg* instruction to invalidate a single page at a time. `GLOBAL FLUSH` is fast at the expense of more TLB misses that would not have been incurred if non-target TLB entries were not invalidated. After `GLOBAL FLUSH`, the TLB must be filled from the page table by hardware on every access to memory. To balance the TLB flush latency and the TLB miss penalty, Linux uses a tunable threshold called *tlb_single_page_flush_ceiling* whose default value

is 33. When the address range to be invalidated is larger than 33 pages, the native kernel performs `GLOBAL FLUSH`.

Similarly, we choose the default *BATCH_SIZE* to be 32 in order to limit the TLB flush latency. In `z-READ`, when the requested I/O size is larger than 128 KB (32 4 KB pages), the kernel performs either `GLOBAL IPI ALL FLUSH`, `GLOBAL LOCAL FLUSH`, or `GLOBAL SINGLE FLUSH`. Even in this case, we still exploit private PTE detection to minimize the number of remote cores that are involved in TLB shootdown operations. Note that the `GLOBAL` and `INDIV` prefixes are used to indicate how many TLB entries are invalidated at once in the local TLB, while the `IPI ALL`, `LOCAL`, and `SINGLE` prefixes are used to denote where TLB flushing has to be performed. After the first batch is done, there is no need for any other TLB flushes on the cores whose TLBs are entirely flushed in the first batch. The kernel tracks these cores to prevent unnecessary TLB flushes.

We implement a bitmap-based ranged TLB flush technique because the native Linux kernel only supports a ranged TLB flush with contiguous addresses from the start address to the end address. For batching, two types of bitmaps are used: an `IPIALL` bitmap for `IPI ALL FLUSH` and several `OTHER` bitmaps for `LOCAL FLUSH` and `SINGLE FLUSH`. The number of `OTHER` bitmaps matches the number of cores in the system. Each bit in the bitmap represents a page in the range starting from the base address. With the bitmaps and base address, the receiver of a TLB shootdown IPI can calculate the virtual addresses of the pages that need to be invalidated in the local TLB. In this way, the remote cores that are involved in a TLB flush receive an IPI only once per batch (once per I/O request for `GLOBAL FLUSH`).

During batched page remapping, the kernel checks the *access bit*, *z-READ CPU*, and the current core ID for each page to determine the type of TLB flush

to use. The result determined for each page is recorded in either the `IPIALL` bitmap or one of the `OTHER` bitmaps corresponding to the *z-READ CPU*. These bitmaps are used for TLB flushing in a batch.

4.3.4 Copy-on-Write Optimization

Our prior version of `z-READ` [59] does not provide the performance guarantee that `z-READ` achieves at least as much performance as `READ`; it actually performed worse under mixed read/write workloads that can cause numerous CoW faults. In this work, we resolve this problem by anticipating when the performance gain of `z-READ` is offset by the CoW overhead and by dynamically changing the I/O mode between `READ` and `z-READ`.

Performance Loss Model. There is a trade-off between `READ` and `z-READ`. With `READ`, there is no need to worry about the *CoW overhead* since the application has its own copy of the data. However, owing to the copy overhead, there might be a performance loss compared to that when using `z-READ` if no CoW faults occur.

If we can calculate the *expected performance losses* for `READ` and `z-READ`, we can choose the one whose performance loss is likely to be lower. The expected performance losses for `READ` ($\mathbb{E}(Loss_{read})$) and `z-READ` ($\mathbb{E}(Loss_{zread})$) are given by

$$\mathbb{E}(Loss_{read}) = (LAT_{read} - LAT_{zread}) * (1 - P(cow)), \tag{4.1}$$

$$\mathbb{E}(Loss_{zread}) = LAT_{cow} * P(cow), \tag{4.2}$$

where LAT_{read} and LAT_{zread} are the latencies of `READ` and `z-READ`, respectively; $P(cow)$ is the probability of the occurrence of a CoW fault; and LAT_{cow} is the

Table 4.2: The measured average latencies (ns) for our performance loss model

LAT	$read$	$read$ $+mod$	$zread$	$zread$ $+mod$ $+cow$	$read$ $-zread$	cow
4 KB	1781	1926	1547	3972	234	2279
8 KB	3255	3557	2387	7054	869	4366
...	...	...	...	...	...	...
16 KB	5601	6766	3946	12367	1655	7257
...	...	...	...	...	...	...
32 KB	12910	13670	6578	28059	6331	20720

CoW overhead, which can be calculated by

$$LAT_{cow} = (LAT_{zread+mod+cow} - LAT_{zread}) - (LAT_{read+mod} - LAT_{read}), \quad (4.3)$$

where $LAT_{zread+mod+cow}$ and $LAT_{read+mod}$ are the respective latencies of `z-READ` and `READ` that are followed by modification of the buffer. Note that CoW fault overhead is included in $LAT_{zread+mod+cow}$.

To build a performance loss model, we need LAT_{read}, LAT_{zread}, $LAT_{read+mod}$, and $LAT_{zread+mod+cow}$. We write a simple user application that measures the read I/O latencies of `z-READ` and `READ` for varying I/O sizes. The results in our system are listed in Table 4.2. Note that building the model is a one-time offline task and this model can be updated by our simple kernel module written for this purpose.

Historical Statistics. We can infer the probability of the occurrence of a CoW fault if we understand its causes and maintain a history of related kernel events. In `z-READ`, a CoW fault occurs when an application attempts to change the contents of the I/O buffer that is mapped to the *kernel pages*. This application's behavior is usually caused by either 1) data characteristics or 2)

application characteristics.

One data characteristic that is highly related to CoW faults is the frequency of data updates. For instance, some data in a file tend to be frequently updated. For such data, `READ` is preferred to avoid CoW fault overhead.

However, although tracking CoW faults is easy, it is difficult to relate this information to the data in a file. At the time of a CoW fault, the kernel only knows the *user page* to which the application wants to write the new data; the kernel does not know the data in a file that the application is about to update at that time.

The read/write ratio for each *file page* can be used instead to infer the probability of the occurrence of a CoW fault. After reading the data from a file to *user pages*, an application may want to update the data permanently. Note that the same data reside in *kernel pages* at this point. For permanent updates, the application first updates the data in *user pages*, which will cause CoW faults and then issues a write system call to effect the changes to *kernel pages*. From this, we can find the correlation between the write system call and the modification of the buffer.

Since `z-READ` is performed in units of pages, we can count the number of read and write I/Os for each *kernel page* while sacrificing little accuracy compared to object-level tracking. To record the I/O event history, we add two counters for the read and write I/Os in the page structure of the page cache. Whenever I/O occurs, the kernel increases the corresponding count for each I/O event (read/write) on the *kernel pages*.

Given the performance loss model and the read/write ratio of the target *kernel page*, the kernel can calculate $\mathbb{E}(Loss_{read})$ and $\mathbb{E}(Loss_{zread})$ online. If $\mathbb{E}(Loss_{zread})$ is higher than $\mathbb{E}(Loss_{zread})$, the kernel chooses `READ` over `z-READ`, and vice versa.

There can be some CoW faults whose cause is not related to data but the application itself. For instance, an application may initialize the I/O buffer for reuse. This behavior has no relationship with the data. We can estimate $P(cow)$ for each *user page* by simply tracking 1) how many CoW faults have been occurred on the page and 2) the number of read system calls with the page. However, this type of CoW fault can easily be eliminated by minor application changes (no initialization for reuse); we have chosen to leave this problem unsolved in this work.

4.3.5 Implementation

We implement a `z-READ` prototype in Linux kernel 4.12.9. We have modified or added less than 1000 lines of code across the read I/O path, CoW fault handler, and page unmapping handler in the kernel. For implementation and debugging convenience, we have also added a new file open flag, `O_ZREAD`. Similar to the `O_DIRECT` flag, `z-READ` is used only when a file is opened with the `O_ZREAD` flag.

4.4 Evaluation

This section presents the evaluation results of our prototype. The evaluation covers the individual effectiveness of each of the two major optimization techniques presented in this article and the overall effectiveness and performance interference intensity of `z-READ`. We compare several variants of our scheme to `READ`. The variants of our scheme are as follows:

- `z-READ(Earlier)`. This is the earlier version of `z-READ` from our previous work [59].
- `z-READ`. This is the enhanced version of `z-READ` that includes the full implementation of *direct TLB insertion* and the CoW optimization.

- `z-READ(NoTLB)`. This is a variant of `z-READ` for which private TLB detection is disabled.

- `z-READ(NoCoW)`. This differs from `z-READ` in that the CoW optimization is disabled.

- `z-READ(Optimal)`. This represents the optimal performance of `z-READ` if we know which core caches which PTEs without the private PTE detection overhead. To emulate it, we force the kernel to perform a local TLB flush only, having each job running on a dedicated core.

4.4.1 System Configurations

Experiments are performed on an NUMA system with two nodes, each of which is equipped with an Intel Xeon Processor E5-2650 v4 running at 2.2 GHz (12 cores/24 threads) supporting four memory channels and 128 GB (32 GB $\times$ 4) DDR4 memory running at 2133 MHz. Two 375 GB Intel Optane DC P4800X SSDs are configured in RAID-0 for the underlying storage device.

Since we do not have a sufficient number of storage devices to saturate the entire memory bandwidth, we choose to use only two memory channels to limit the maximum memory bandwidth, by installing memory into four DIMM slots associated with two memory channels in a node. However, the sustainable memory bandwidth of our system is not significantly different from when it uses four memory channels (25 GB/s vs. 28 GB/s).

All of the benchmarks used in the evaluations are either run until the stable experimental results are obtained or executed 10 times for the average results. We omit error bars from these results since the observed variances are negligible. In order to isolate our results from NUMA effects, we use only one node with help of the *numactl* command.

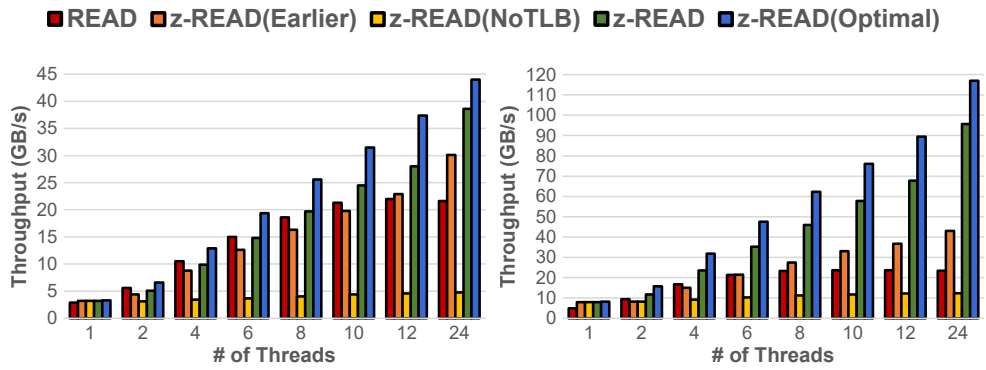
Our evaluations were conducted where swapping is disabled because our prototype currently does not support page reclamation for the remapped pages. We also disable transparent huge pages (THP) owing to the lack of support for our prototype. In practice, we can simply enforce the kernel to use `READ` for huge pages.

4.4.2 Effectiveness of the TLB Shootdown Optimization

We first evaluate the effectiveness of our TLB shootdown optimization using FIO, a Flexible I/O tester synthetic benchmark [60]. We run FIO with the 4 KB/16 KB random read workload, varying the number of jobs from 1 to 24. Each job involves reading from the 128-MB-sized file allocated for the job. We always perform a dummy run with the invalidate flag clear before every experiment, fully warming up the page cache. Note that our main memory capacity is sufficiently large to hold all of the file contents in memory so that all I/Os result in a page cache hit.

We also use the pthread/thread model instead of the default fork/process model. As we described in §4.2.2, in the fork/process model, where each job has its own address space, even the native kernel can minimize the number of remote TLB shootdowns, only performing a local TLB flush on the core running the job. In other words, the results of FIO with the default fork/process model cannot show how much our TLB shootdown optimization affects the I/O performance.

Results. We present the performance results in Figure 4.2. We evaluate `READ`, `z-READ(Earlier)`, `z-READ(NoTLB)`, `z-READ`, and `z-READ(Optimal)`. Compared to `READ`, `z-READ` improves the I/O performance by up to 1.79 times for the 4 KB random read (up to 4.07 times for the 16 KB random read). The performance of `READ` only scales until it saturates the memory bandwidth (about 25 GB/s), while both `z-READ(Earlier)` and `z-READ` scale linearly, even surpass-



(a) FIO 4 KB Random Read (b) FIO 16 KB Random Read

Figure 4.2: The effectiveness of the TLB shootdown optimization

ing the maximum theoretical memory bandwidth. This is possible because they do not consume memory bandwidth until the data is actually accessed.

The full implementation of the *direct TLB insertion* scheme in `z-READ` helps achieve a further improvement over `z-READ(Earlier)`. This is because *emulated direct TLB insertion* in our previous work [59] uses a PTE lock, which is shared by 512 pages in the native kernel, when clearing the *access bit* of each PTE. This limits the scalability when other cores access the pages among the 512 pages that share the same PTE lock. In contrast, `z-READ` can insert the PTE into the local TLB without setting the *access bit*; thus, no clearing process is needed.

A comparison of `z-READ(NoTLB)` with `z-READ` and `z-READ(Earlier)` shows the effectiveness of private PTE detection. Without it, the performance of `z-READ` severely decreases because every page remapping requires a remote TLB shootdown to all cores sharing the same address space. The performance of `z-READ(NoTLB)` does not scale at all. It is inferior to `READ` in every respect, except for single-threaded applications.

As shown in Figure 4.2a, even with our TLB shootdown optimization, the

I/O latency of `z-READ` can be slightly worse than that of `READ` for 4 KB read I/O when the memory bandwidth is not saturated. This means that the costs of page remapping and TLB flushing for each page is higher than the page copy overhead. The performance gap between `READ` and `z-READ` becomes wider when the I/O size is increased from 4 KB to 16 KB. This result demonstrates that the TLB shootdown batching technique enhances the I/O performance for multiple-page-sized I/O requests, reducing the number of remote TLB shootdowns. We can conclude that `z-READ` becomes effective in terms of the latency when the I/O size is larger than 8 KB.

There is still a room for improvement in reducing the private PTE detection overhead, which can be inferred from a comparison of the increased performance of `z-READ(Optimal)` relative to that of `z-READ`. Aligned with the conclusion of Amit's work [28], the most overhead can be avoided by simple hardware enhancements.

4.4.3 Effectiveness of CoW Optimization

In order to demonstrate the effectiveness of CoW optimization, we run FIO with almost the same configuration as the previous evaluation, but a mixed random read/write workload is used instead. We vary the read/write ratio of the workload while fixing the number of jobs to 24. Please be advised that for the write workload, FIO writes random data to the user buffer before issuing a write system call; thus, every write results in a CoW fault. We evaluate `READ`, `z-READ(NoCoW)`, and `z-READ`.

Results. Figure 4.3 shows the read and write I/O throughputs for various read/write ratios. For `z-READ(NoCoW)`, when the read/write ratio is 95:5, the read performance increases by 325% compared to that of `READ`. However, its performance becomes worse than that of `READ` as the ratio of writes increases

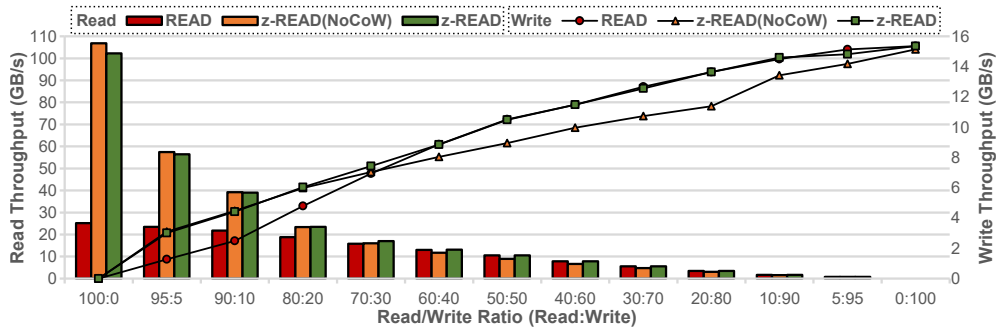
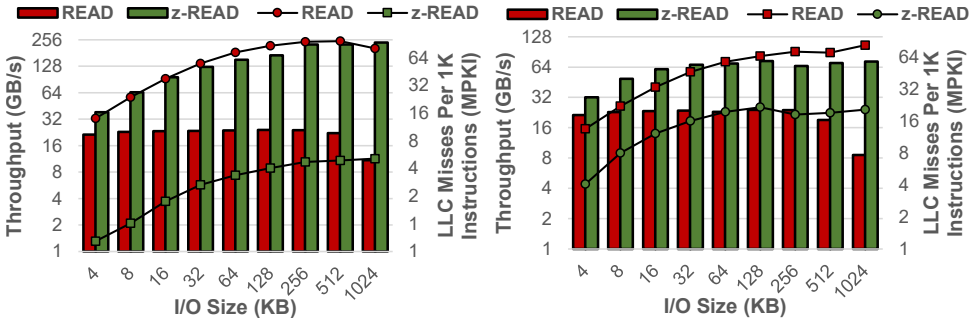


Figure 4.3: The effectiveness of CoW optimization

(a slowdown of 15% for a 50:50 ratio compared to `READ`). The reason for this is due to the CoW fault overhead, which is relatively large compared to the performance gain from `z-READ`.

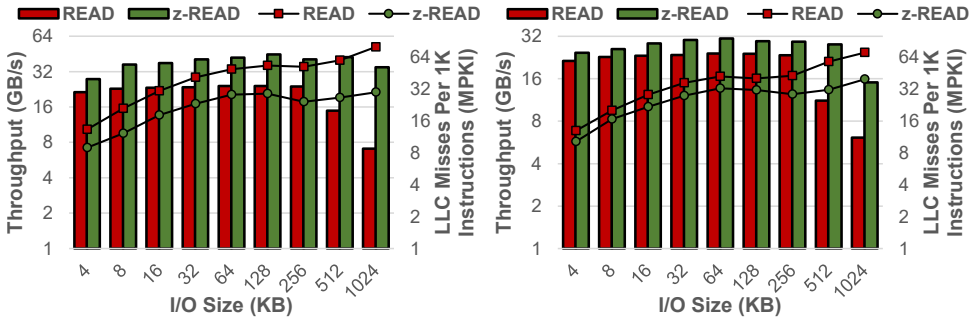
Our CoW optimization resolves this problem. For `z-READ`, the read performance and write performance are better than or equal to those of `READ` for all read/write ratios. Its performance is preserved even under dynamic workloads because `z-READ` changes the I/O mode between `z-READ` and `READ` for each page on the basis of the read/write history. Owing to the decision overhead, `z-READ` performs slightly worse than `z-READ(NoCoW)` for read-heavy workloads (100:0, 95:5).

Since a CoW fault occurs only after performing `z-READ`, there is less chance for CoW faults for a write-intensive workload. Consequently, the performance gap between `READ` and `z-READ` becomes narrow after a ratio of 40:60.



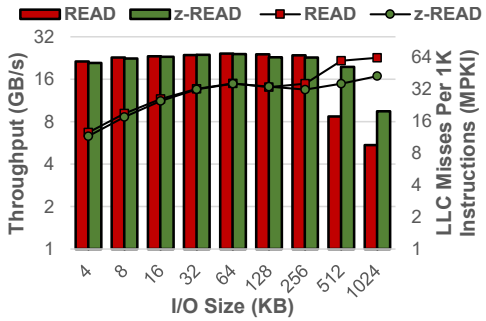
(a) 0% of data accessed

(b) 25% of data accessed



(c) 50% of data accessed

(d) 75% of data accessed



(e) 100% of data accessed

Figure 4.4: Relationship between the I/O throughput and the LLC MPKI

4.4.4 Analysis of the Performance Improvement

The key contributing factor to the improved performance of `z-READ` is the *lazy cache misses*. In `z-READ`, cache misses for the requested data can be deferred until the data are actually accessed. On the other hand, `READ` incurs cache misses during a page copy. This *eager cache miss* might be beneficial if and only if an application accesses all of the data right after a read. However, some applications may access only part of the data or access the data at a later time after it is evicted from the LLC.

To determine the relationship between the amount of data accessed after a read and the performance, we use a modified version of FIO that copies a certain amount of data from the I/O buffer to another buffer after reading data from a file. We vary the amount of copied data and the I/O size while keeping the number of jobs fixed to 24.

Results. The results are plotted on a logarithmic scale with base 2 in Figure 4.4, where the primary y axis is the I/O throughput and the secondary y axis is the number of LLC MPKI (misses per kilo instructions). As expected, the performance is inversely proportional to the number of LLC MPKI regardless of the amount of data copied. As the amount of copied data increases, the performance gap between `READ` and `z-READ` becomes narrow owing to the increased number of cache misses in `z-READ`. When all of the data is copied, the performance of `z-READ` is marginally lower than that of `READ`, except for large I/O sizes.

For large I/O sizes (512 KB and 1 MB), the performance of `READ` decreases because the 30 MB of LLC in our system cannot retain the data between the time of I/O and the time of data access; cores compete for the LLC. Therefore, copying more than 50% of the data with `READ` shows a similar tendency.

Table 4.3: SPEC CPU 2006 benchmarks

Memory Intensity	Benchmark
High	470.lbm, 462.libquantum
Medium	403.gcc, 401.bzip2
Low	481.wrf, 453.povray

Table 4.4: Mixed workloads

	Workloads
mix1	lbm + lbm + gcc + gcc + wrf + wrf
mix2	libquantum + libquantum + bzip2 + bzip2 + povray + povray
mix3	lbm + libquantum + gcc + bzip2 + wrf + povray

However, owing to the *lazy cache misses*, even with the full data copy, `z-READ` maintains better I/O performance than `READ`, avoiding some unnecessary cache misses. This characteristic is favorable for the other applications running on the server.

4.4.5 Performance Interference Intensity

To demonstrate that an I/O-intensive workload can cause harmful interference to co-located workloads, we run a mixture of several applications from SPEC CPU 2006 [61] in the foreground while running the 16 KB random read workload from the FIO benchmark on the other cores on the same node in the background indefinitely. We repeatedly run the foreground applications until all of them are finished at least once and then measure the execution time of each of them for the first run. To minimize the effects of other sources of interference caused by resource sharing, we allocate six physical cores to FIO while allocating the other six cores in a node to the foreground applications. For this evaluation, we set the number of jobs to 12 for FIO. Comparing the execution times of the co-located applications for each respective read I/O scheme, we indirectly

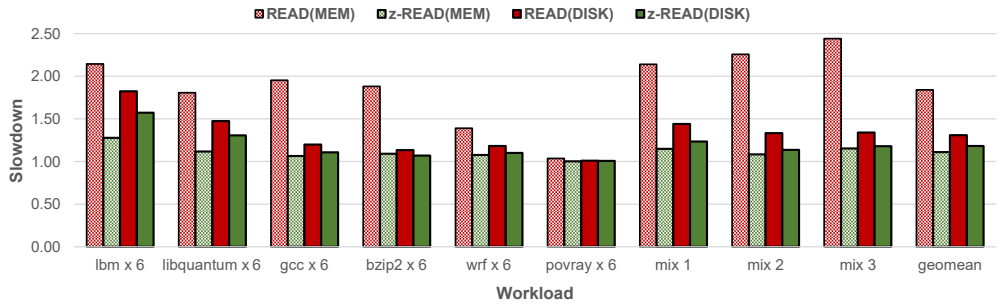


Figure 4.5: Slowdown of co-located workloads

evaluate the performance interference intensity of `READ` and `z-READ` under two circumstances: in-memory and on-disk.

For the on-disk configuration, we use *cgroup* to limit the memory available to FIO to 1 GB while forcing each job read from its own 10-GB-sized file (total 120 GB for 12 threads). Note that the configured memory limit also applies to the amount of page cache used by FIO; thus, most read requests have to go to the disk.

According to the memory intensity, we choose six benchmarks from SPEC CPU 2006 on the basis of the benchmark classification results from [62,63]. The benchmarks used in this evaluation are listed in Table 4.3. For the foreground workloads, we run either six identical benchmarks or a mixed workload that consists of several benchmarks. Three mixed workloads are summarized in Table 4.4. We use *ref* as the input for SPEC CPU 2006.

Results. The slowdown of the foreground workloads when running with the I/O-intensive background workloads are shown in Figure 4.5. Note that we use the geometric mean to compute the average slowdown of the foreground applications. In the in-memory I/O configuration in Figure 4.5, `z-READ` scarcely affects the execution time of the co-located workloads (only a slowdown of 11%) while the workloads with `READ` suffer up to a slowdown of 144%. Figure 4.5 demon-

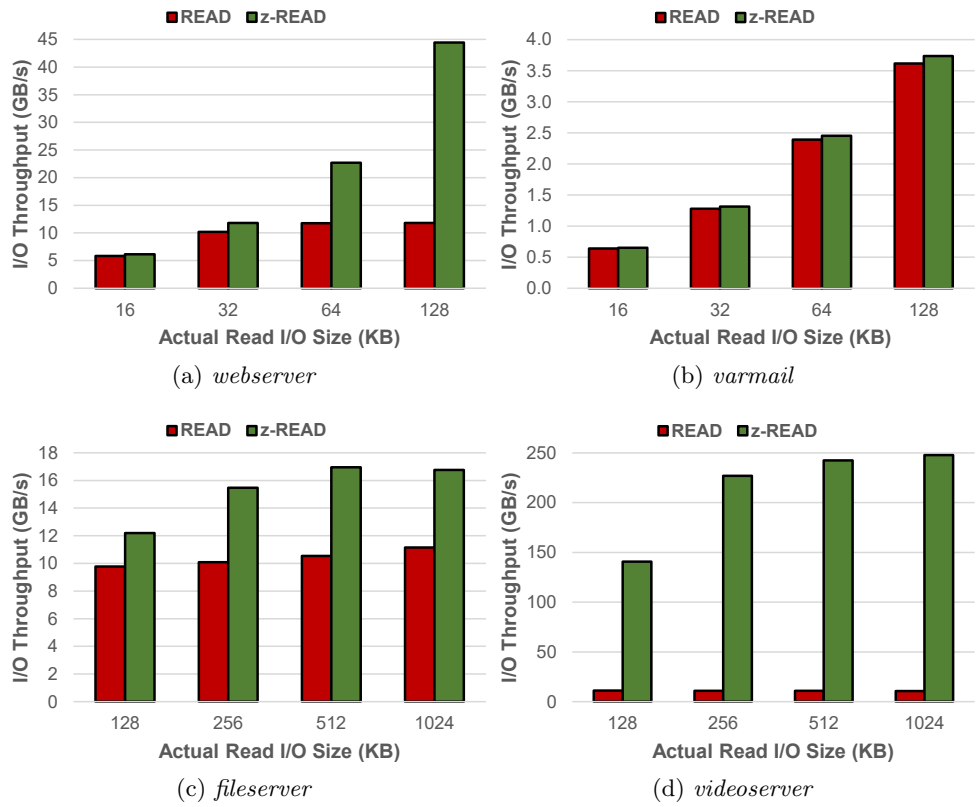
Table 4.5: Parameters for the Filebench workloads

Workload	Configuration
webserver	filesize=16k, 32k, 64k, 128k iosize=1m
varmail	meanappendsize=16k, 32k, 64k, 128k filesize=16k, 32k, 64k, 128k iosize=1m
fileserver	filesize=128k, 256k, 512k, 1m iosize=1m meanappendsize=16k
videoserver	readiosize=128k, 256k, 512k, 1m writeiosize=1m filesize=500m

strates that even in the on-disk I/O configuration, copy-based I/O (`READ`) can severely interference with the performance of co-running applications, incurring a slowdown of 31% on average. For the same case, the co-located workloads are 18% slower on average with `z-READ`. Interestingly, with `z-READ`, the on-disk I/O configuration causes more slowdown than the in-memory configuration. This originates from the zeroing out of the new page cache before reading the data from disk, which costs memory bandwidth. By avoiding memory copy and saving memory bandwidth, `z-READ` can outperform `READ` by up to 2.1×.

4.4.6 Effectiveness of z-READ in Macrobenchmarks

We use four macroworkloads from the Filebench benchmark [64]: *webserver*, *varmail*, *fileserver*, and *videoserver*. We run them with the parameter settings listed in Table 4.5. For the *videoserver* workload, we eliminate the *bwlimit* workflow to measure the peak I/O performance. For each workload, we vary different parameters that affect the actual I/O size for a single read system call. For example, for *webserver* and *fileserver*, we change the file size while keeping the I/O size fixed to 1 MB because the actual I/O size is the same as



(a) *webserver* (b) *varmail*

(c) *fileserver* (d) *videoserver*

Figure 4.6: Performance on macrobenchmarks

the file size when the file size is shorter than the requested I/O size. On the other hand, *varmail* uses *meanappendsize* when writing data to a newly created file. Since it reads the entire file after that, *meanappendsize* is directly related to the actual I/O size and the file size.

Results. We evaluate `READ` and `z-READ` in the in-memory configuration where all data are in memory. We present the performance results from Filebench in Figure 4.6.

For the *webserver* workload in Figure 4.6a, which emulates a web server that performs open-read-close operations, the improvement in performance obtained

by `z-READ` is not noticeable (about 5%) when the file size is small (16 KB). However, it improves the throughput by up to 3.8 times as the file size increases. Because of the in-memory configuration, the overheads for opening and closing a file are considerable, while these overheads are consistent regardless of the file size. Therefore, for a larger file size, the read performance more greatly affects the overall performance.

For the *varmail* workload in Figure 4.6b, there is only small performance gain with `z-READ` (less than 3%). The workload emulates a mail server that performs open-create-append-sync-close, open-read-append-sync-close, open-read-close, and delete operations. As a result, the read I/O contributes only a small fraction of the application runtime; thus, the read performance has little impact on the overall performance.

In Figure 4.6c, although the *fileserver* workload performs create-delete-append-read-write-stat operations, `z-READ` improves the throughput by up to 68% compared to `READ`. This is due to the relatively large file size. A performance gain is realized when an entire file is read.

For the *videoserver* workload in Figure 4.6d, which performs delete-create-write-close and read operations, `z-READ` outperforms `READ` by 12.7–23.3 times. As expected, since *videoserver* is the most read-intensive workload, it can considerably benefit from the use of `z-READ`.

4.5 Summary

In this chapter, we have presented `z-READ`, an efficient, transparent, and practical zero copy read file I/O scheme, and several optimization techniques that reduces two major sources of overhead: page remapping and CoW faults. `z-READ` provides better I/O performance, even for mixed read/write workloads that incur CoW faults, while significantly reducing the interference with the per-

formance of co-located workloads in data-center-like environments. In future work, we will apply `z-READ` to existing persistent datastores and explore the feasibility of transparent zero copy write file I/O.

Chapter 5

Memory Efficient Fork-based Checkpointing

5.1 Motivation

5.1.1 Fork-based Checkpointing

Fork-based checkpointing is simple yet effective scheme that is used for many popular IMDBs [23, 24]. Algorithm 5.1 shows the simple fork-based checkpointing model. It works as follows. When an IMDB wants to create a backup of consistent data in background, `fork()` is used to create a *child* process (*checkpointer*) that shares the same physical pages with the *parent* process (*servicer*). After `fork()`, *servicer* can continue to handle the requests from clients while *checkpointer* traverses and writes all the point-in-time data to a file. For update requests from clients during checkpointing, *servicer* may try to overwrite the contents of the shared pages. Owing to the CoW technique on which `fork()` relies, this overwriting does not affect the point-in-time data on *checkpointer*.

However, this can lead to the increase in the memory footprint during check-

Algorithm 5.1: Simple Fork-based Checkpointing Model

```
1 if (checkpointer = fork()) == 0 then
2     foreach objects in IMDB do
3         fwrite(obj_type, size, count, fp);
4         fwrite(obj_len, size, count, fp);
5         fwrite(obj_data, size, count, fp);
6 else
7     servicer continues to service clients' requests
```

pointing [26, 65–67]. At worst, updates (even single bit modifications) on every page can result in using up to two times the memory needed for the data set. The problem is that this increased memory cannot be reclaimed until *checkpointer* terminates; this will be discussed in more detail in the following section. In fact, the time it takes before the *checkpointer*'s memory is released is dependent on the checkpointing speed, which turns on the speed of storage device. That is, the slower the storage device, the longer the checkpointing time, thus the higher the chance of the memory being doubled for update-intensive workloads.

For this reason, many IMDB vendors [26] recommend users provision memory based on their peak memory usage, leaving a sufficient amount of memory available for the OS to be used for page duplication on CoW faults; because, in case of memory pressure, IMDBs will either be killed by the OS's out-of-memory killer or be severely slowed down due to *swap*. However, this conservative approach results in wastage of memory that can be utilized for storing more data in memory, especially where the peak memory usage is rarely reached.

On the other hand, fork-based checkpointing may incur high latency spike on *servicer*, owing to `fork()` [66–68]. Since `fork()` is a blocking operation, the calling process (*servicer*) must wait for it to finish. Meanwhile, *servicer* cannot respond to the request from clients. Although copying the page table entries takes shorter time than copying the pages, it may cause a noticeable

latency spike for client requests during `fork()` when data set size is huge, due to `fork()`'s $O(n)$ time complexity. However, because this latency, in practice, can be alleviated by restraining the data set size of a single IMDB instance and running multiple instances within a single machine, we focus solely on the memory footprint issue in this work.

5.1.2 Approach

Our basic idea is simple but powerful: returning to the OS the pages that has been checkpointed by *checkpointer* as soon as possible, so that updates on *servicer* do not lead to memory duplication. One may think that freeing logical objects already checkpointed is enough to reduce the memory footprint during checkpointing. However, this is not the case for the fork-based one. After `fork()`, initially, all the pages are shared between *servicer* and *checkpointer*. As explained in §2.1.4, in order to prevent a CoW page from being duplicated, the page need to be private. The way to make these pages private without incurring a CoW fault is to let either *servicer* or *checkpointer* voluntarily return the memory of these page to the OS; then the OS can unmap these page from their address space.

However, this is not easily achievable due to the semantic gap between application (IMDB) and OS in the context of memory allocation/deallocation. The existence of user-level memory allocators (UMAs) [69], such as GNU libc allocator and jemalloc [70], is the cause of the gap. UMA helps applications to efficiently allocate and deallocate memory at object granularity while minimizing the OS's involvement. For instance, the memory of the freed objects is not immediately returned to the OS but is recycled for the application's other memory allocation requests to avoid excessive system call overhead. In fact, UMA keeps the memory once allocated from the OS until the associated pro-

cess terminates. Although OS can reclaim this allocated memory using *swap* in case of memory pressure, it is done transparently to applications. Therefore, there needs to be an explicit way for applications to return the pages to the OS. As explained in §2.2.2, the `madvise()` system call with the proper hint can do the job.

On the other hand, a new challenge arises from the fact that `madvise()` works only at page-level granularity while IMDB performs checkpointing at object-level (record or key-value pair). Since multiple objects can be stored in a single page, it is not allowed for the page to be returned to the OS until all the objects within the page are checkpointed. A naive approach to address this is to track the checkpointed objects within a page using a bitmap data structure. However, this will require the bitmap management costs, in terms of memory usage and CPU cycles. Moreover, because the objects are allocated and deallocated across address space, the order in which the objects are checkpointed is not necessarily the logical address order; two consecutive objects within a page can be checkpointed at two different time. Furthermore, some of the objects may be not meant to be checkpointed (e.g., temporary data). Only one tiny object left *uncheckpointed* can result in the failure of returning the page to the OS.

Our approach is to exploit physical memory dump, which is generally used for forensic analysis [71, 72]. By sequentially writing the contents of the pages in logical address order, it is possible to allow the pages to be returned to the OS once they are written to disk. However, the physical memory dump alone is not enough for a consistent snapshot of a IMDB to be used for restoring data because the mapping information between the virtual addresses, which can be converted to the dump file offset, and the object is not included in the dump file. For example, the IMDB cannot know for what object the first eight

bytes of the dump file represent. However, this mapping information can be reconstructed later if *checkpointer* logs the virtual addresses of the objects to be checkpointed. Therefore, for each write request for the objects, our scheme writes the virtual address of the given buffer (object), instead of the contents of the buffer. Given the virtual address of an object, our scheme can restore the object from the dump file.

5.2 Related Work

The creation of consistent snapshots has been thoroughly researched in IMDBs. Towards a fast checkpointing with low memory footprint and low latency for clients, many checkpointing scheme has been proposed. Naive Snapshot is the simple checkpointing technique that quiesces the system during taking a consistent snapshot of data. Some studies [73,74] applied this to scientific applications in high performance computing. However, the blocking approach is not suitable for IMDBs that needs to continue servicing the requests while checkpointing is done.

For non-blocking checkpointing, Copy-on-Update (COU) algorithm has been proposed by several studies [65,75]. These studies presented the memory-efficient consistent checkpointing scheme that supports the row-level COU rather than the page-level COU. In theory, the smaller granularity of duplication could lead to a smaller effect on update latency while providing slower increase in memory usage, compared with the page-level one. However, due to the presence of row locking, their average latency may be higher than the page-level COU [66, 67]. Our scheme is also a variant of page-level COU that relies on the OS's CoW support. Our study is in line with their work in terms of addressing the memory footprint problem. In contrast to the existing COU schemes, we focus on reducing the memory footprint while using page-level COU that provides superior

average latency.

Cao et al. [76] proposed two checkpointing algorithms that trade extra memory usage for lower overhead and latency. Whereas our work trades restoring speed for lower memory footprint and latency. Li et al. [66, 67] evaluated and compared the existing consistent checkpointing algorithms. Extensive evaluations revealed that the simple fork-based checkpointing could outperform the state-of-the-arts in terms of both average latency and maximum latency. Based on the result, they also proposed two improved algorithms that address the latency spike problem of the fork-based checkpointing scheme while achieving the comparable average latency. However, their average latency may be comparable but still lower, to that of the simple fork-based checkpointing while our scheme provides even better average latency (higher throughput). Moreover, in contrast to our scheme that takes full advantage of OS-level functionality, all the above works require high effort to be implemented in IMDBs.

Sharma et al. [68] introduced a new system call `vm_snapshot()`, which is a fine-grained version of `fork()`. Owing to the smaller number of page table entries to be copied, it allows faster snapshot creation if applications do not need to checkpoint all the data across the whole address space. Their work and our work have similarities in that both exploit OS supports to address the problem of the fork-based checkpointing. However, our work focuses on the memory footprint issue while their focus is to reduce the snapshot time.

5.3 Design and Implementation

5.3.1 Overview

On the basis of the idea discussed in §5.1.2, we present a new fork-based checkpointing scheme, called MDC, which effectively mitigates the memory footprint issue occurred from the existing fork-based checkpointing scheme.

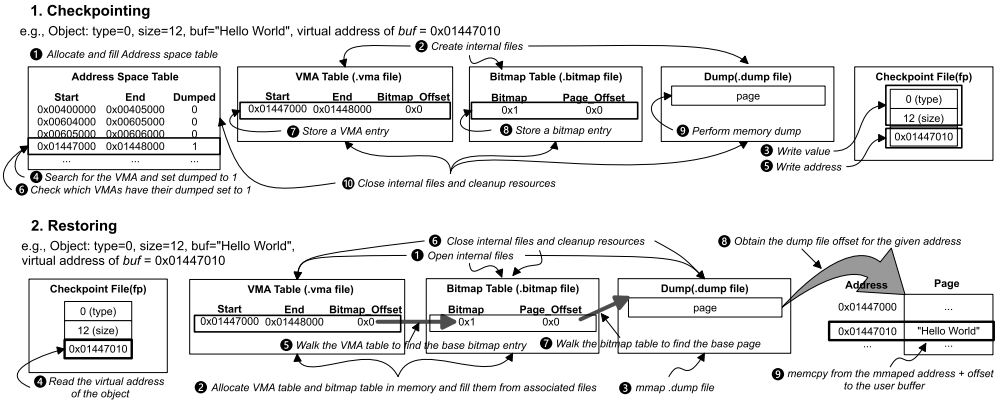


Figure 5.1: The Overview of MDC

Checkpointing. For checkpointing, MDC involves three steps as follows: (1) preparing, (2) logging, and (3) memory dump. In the preparation step (1), MDC first allocates an *address space table* in memory and loads the process's VMA information into it[1] (Figure 5.1-1 ❶); each row of the *address space table* is associated with a VMA, containing the start address and end address of the VMA, and a *dumped* flag, which indicates whether or not this VMA needs to be memory-dumped at the end of checkpointing. Then, MDC creates and opens three files that are intended to be used internally, transparently to applications (Figure 5.1-1 ❷); a `.dump` file is used for physical memory dump and a `.vma` file contains the information about virtual memory areas (VMAs) while a `.bitmap` file stores the bitmaps that indicate which pages in the VMA are written to the `.dump` file.

After the step (1), the logging step (2) is started. For a write request of the object during checkpointing, MDC searches for the VMA of the object's virtual address and marks it to be memory-dumped later by setting its *dumped* flag to 1 (Figure 5.1-1 ❹). Then, MDC writes the address of the object to the file

[1]In Linux, the process's VMA information can be obtained from /proc/self/maps.

that is explicitly given by the application (Figure 5.1-1 ❺). Exceptionally, for the objects allocated in stack, MDC writes the objects themselves to the file (Figure 5.1-1 ❸), which is the same behavior as the normal write operation. This is because the address of the object allocated in stack can be recycled in different function calls; due the time difference between (2) logging and (3) memory dump, our scheme requires all the addresses of the target objects to be unchanged until (3) memory dump is done. MDC also performs a normal write operation for the object whose size is less than eight bytes because in such a case, the size of the address (8 bytes in 64-bit system) becomes larger than the object. Note that the step (2) ends very quickly compared to writing all the objects to the file, owing to the reduced amount of writes.

At the end of checkpointing (3), MDC checks which VMAs have their *dumped* set to 1 (Figure 5.1-1 ❻). For the target VMA, MDC first stores a VMA entry into the `.vma` file (Figure 5.1-1 ❼). The VMA entry contains the start and end addresses of the VMA and the current location of the `.bitmap` file offset; they are used to find the base bitmap entry associated with the given virtual address for restoring process.

After storing the VMA entry, MDC starts to perform memory dump at multiple page granularity. We empirically select the default dump unit to be 64 pages (256KB) so that it enables gradual freeing of pages during memory dump while not incurring too much system call overhead. MDC also uses direct I/O [29] for writing memory dump to allow the I/Os to bypass page cache, preventing *servicer* from being interfered with memory copy between user and kernel. Since direct I/O requires I/O to be 512-byte aligned, taking advantage of direct I/O for normal applications is only possible with implementing their own buffer management. In contrast, MDC can utilize direct I/O without any additional efforts because it already writes memory dump at page granularity.

To avoid unnecessary writes, MDC performs memory dump only for the pages that page frame is assigned[2]. Therefore, for 64 pages, MDC checks which pages do not have page frames assigned, with help from our OS support `mincore2()`, which will be detailed soon. MDC then creates a bitmap entry that contains the current location of the `.dump` file offset and a bitmap whose bit corresponds to the pages to be dumped; they are used to find the `.dump` file offset associated with the given virtual address. After storing a bitmap entry into the `.bitmap` file (Figure 5.1-1 ❽), MDC writes only the pages that have page frame assigned into the `.dump` file (Figure 5.1-1 ❾). MDC finally closes the opened files and frees the *address space table* (Figure 5.1-1 ❿).

Restoring. Similar to the checkpointing process, the restoring process requires the preparation step. MDC first opens the three files (Figure 5.1-2 ❶), allocates a *VMA table* and a *bitmap table* in memory, and fills them from `.vma` and `.bitmap` files, respectively (Figure 5.1-2 ❷). Note that they do not consume much memory (less than 1MB for 10GB data set). MDC performs `mmap` to map the `.dump` file into the process's address space (Figure 5.1-2 ❸) to avoid the excessive overhead of page copy between user and kernel memory when reading an object from the file.

For restoring an object, our scheme performs two read operations: one for the virtual address from the file explicitly given by applications (Figure 5.1-2 ❹), the other for the content of the object from the `.dump` file. Given the virtual address of the object, MDC can calculate the location of the object within the `.dump` file (Figure 5.1-2 ❼) by searching for the corresponding VMA entry and bitmap entry while walking the *VMA table* and the *bitmap table* (Figure 5.1-2 ❺∼❻). MDC then can copy the content of the object from `.dump` file, using the obtained offset and the *mmapped* base address (Figure 5.1-2 ❽). After all

[2]Modern OSs use demand paging for virtual memory support [29].

Algorithm 5.2: MDC Checkpointing Model

```
1 if (childpid = fork()) == 0 then
2     mdc_checkpoint_start( );
3     foreach objects stored in IMDB do
4         mdc_fwrite(object_type, size, count, fp, MDC_VAL);
5         mdc_fwrite(object_len, size, count, fp, MDC_VAL);
6         mdc_fwrite(object_data, size, count, fp, MDC_REF);
7     mdc_checkpoint_end( );
8 else
9     servicer continues to service clients' requests
```

restoring process is done, MDC unmap the `.dump` file, closes the three files, and frees the *VMA table* and *bitmap table* (Figure 5.1-2 ❾).

Checkpointing Model. The checkpointing model of MDC (Algorithm 5.2) is very similar to the existing simple fork-based checkpointing model (Algorithm 5.1). Therefore, it is easy to port our scheme to the existing applications that use the fork-based checkpointing model; replacing `fwrite()` functions with our `mdc_fwrite()` and using `mdc_checkpoint_start()` and `mdc_checkpoint_end()` to wrap around the checkpointing codes is enough.

5.3.2 OS Support

Our scheme requires several minor OS supports to allow applications efficiently but safely returning pages at user-level.

mincore2. In Linux, `mincore()` is used to know if pages of the calling process's virtual memory reside in memory [29]. Our `mincore2()` is a slightly modified version of `mincore()` to return a vector that indicates whether pages of the calling process's virtual memory have ever had page frames; in other words, it includes the pages swapped out. This allows our scheme to avoid unnecessary writes of only the pages that contains nothing meaningful.

DONTNEED2. As explained in §2.2.2, `madvise()` with the `DONTNEED` hint allows applications to return the pages to the OS. However, in the fork-based checkpointing, returning all the pages of the *checkpointer* results in freeing of its private data as well. The `DONTNEED2` hint prevents this by returning only the pages that are shared with other processes or have became private owing to other process's CoW faults.

PrivateCoWed. To track whether the page has become private as a result of other process's CoW fault, we add a `PrivateCoWed` flag in the *flags* field in the page structure. When a CoW fault occurs, the kernel sets the `PrivateCoWed` flag of the old page if the old page is no longer shared with anyone. For `madvise()` with the `DONTNEED2` hint, this flag can be used as a hint to the kernel that those private pages are free to be returned to the OS.

5.3.3 Implementation

We implement MDC as an user-level library that provides simple APIs to applications. Our APIs corresponding to `fread()` and `fwrite()` are `mdc_fread()` and `mdc_fwrite()`, respectively. They take one more argument, which is `mdc_type`, compared with the existing ones. When the value of `mdc_type` is `MDC_VAL`, they behave exactly the same as the existing ones; applications can use this to checkpoint and restore the objects allocated in stack while `MDC_REF` can be used for the objects allocated in heap. We choose this non-transparent way because applications know better than the OS where the object is allocated. Due to the similarity of the checkpointing model and the APIs, the existing applications can be easily modified to use our scheme. When we apply our scheme into Redis, it only requires few tens of lines of code changes.

5.4 Evaluation

5.4.1 Experimental Setup

System. All the experiments are conducted on two machines, each of which is equipped with a Intel Core i7-4790 CPU (3.6 GHz, 4 physical cores, 8 logical cores with hyperthreading), and 32 GiB of DRAM memory: one for a server machine and the other for a client machine. They are connected via a 10GbE network. We use two Samsung 850 Pro SATA SSD for the server machine: one for swap device and the other for checkpointing. In the Linux kernel configuration, we set the `vm.overcommit_memory` to 1 as guided in [26] while setting `vm.swappiness` to 1 in order to prevent swap unless absolutely necessary (e.g., run out of memory). We also disable *transparent huge page* (THP), which could degrade the performance of IMDBs [77]. Our evaluation is conducted on the Linux kernel 4.19.61 modified for MDC.

Redis. To demonstrate the effectiveness of our scheme in real applications, we apply MDC into Redis 5.0.5. We use the Redis's RDB mode where Redis generates consistent snapshots of the data set at specified intervals. We disable RDB compression in Redis because our prototype implementation does not currently support the feature. We change the *hash-max-ziplist-value* to 1024 from 64 so that `HASH` [78] data can be inserted to Redis more memory efficiently[3].

Workloads. To evaluate the performance of Redis (`Redis-FORK`) and MDC-based Redis (`Redis-MDC`), we choose the Yahoo! Cloud Serving Benchmark (YCSB) [79] as our target workload. Table 5.1 shows the detailed settings of YCSB used throughout the evaluation. We use the default record size.

Methodology. After loading the YCSB data into Redis, we run the YCSB workload while enforcing the Redis to start checkpointing in background. For

[3]YCSB data is stored as `HASH` data in Redis.

Table 5.1: Parameters of YCSB workloads

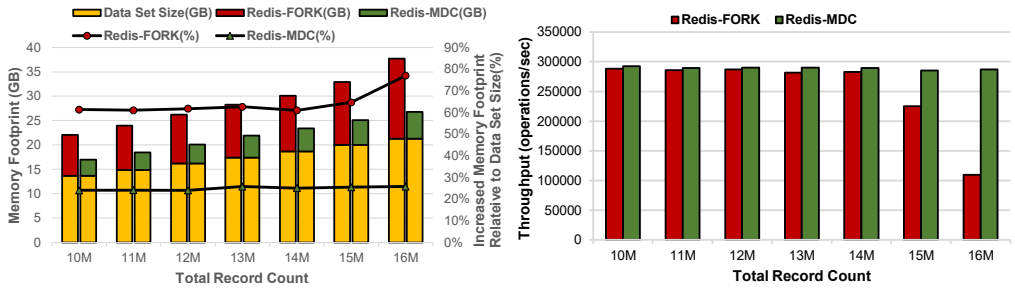
Parameters	Setting
Record count	10M,11M,12M,13M,14M,15M,16M
Update proportion	0.25, 0.50, 0.75, 1.00
Distribution	zipfian
Number of threads	128

fair comparison, we first measure the performance of MDC-based Redis (`Redis-MDC`) during the checkpointing while measuring its checkpointing time. Note that we stop the YCSB workload when the checkpointing is done. This measured checkpointing time is used when we evaluate the performance of the `Redis-FORK`; we measure the performance of `Redis-FORK` for this measured time so that the YCSB workload runs longer even after the checkpointing is done. For this extended period of time, the YCSB workload is not interfered by the checkpointing process, thus `Redis-FORK` can enjoy the benefit from slightly faster checkpointing. When measuring restoring time, we drop the page cache beforehand so that we can measure the time it takes to restore the data from the storage.

We use multiple Redis instances to utilize the CPU resources available in the system. Unless specified, we use two instances for evaluation. We run two YCSB clients for this case. Note that the total record count and the total number of threads are preserved regardless of the number of the YCSB clients. For example, for two YCSB clients, 64 threads are used for each client.

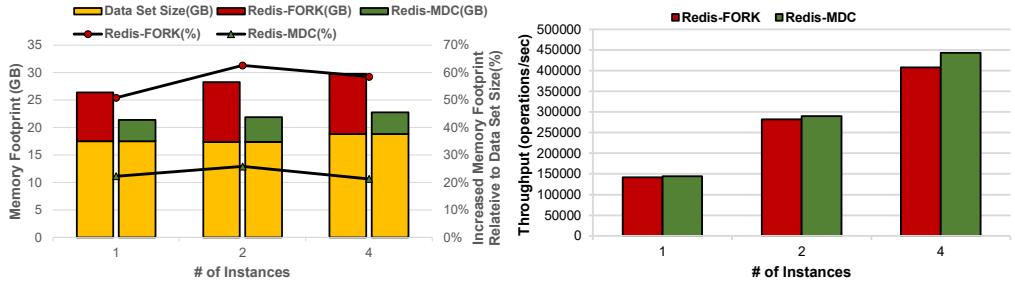
5.4.2 Performance

Memory footprint. Since Redis uses the simple fork-based checkpointing, its memory footprint can be increased by up to two times during checkpointing. Figure 5.2a shows the memory footprint with varying data set sizes for 0.5 update proportion. `Redis-FORK`'s memory footprint can be increased by up to 77%



(a) Memory Footprint (b) Throughput

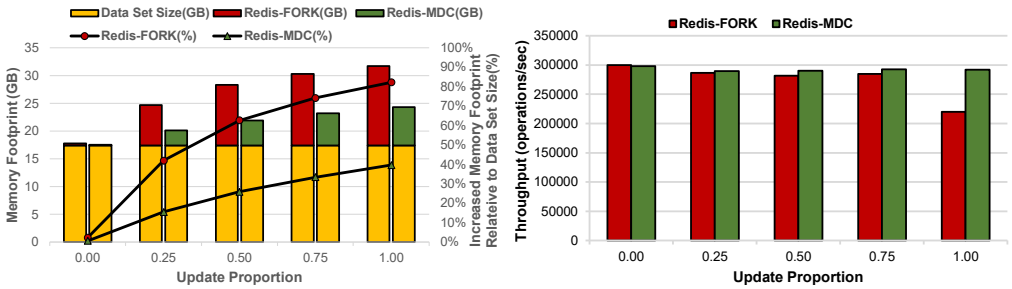
Figure 5.2: YCSB results with varying data size (2 instances, 0.5 update proportion)



(a) Memory Footprint (b) Throughput

Figure 5.3: YCSB results with varying the number of Redis instances (13M records, 0.5 update proportion)

while `Redis-MDC`'s memory footprint is increased by only up to 26% (relative to the data set size). This gap can be wider as the number of Redis instances increases (Figure 5.3a) and the update proportion is increased (Figure 5.4a). This is because of the higher update rate; more Redis instances have more powerful processing capability to handle updates in parallel while higher update proportion increases the number of incoming update requests for each Redis instance. To analyze how the memory footprint changes, we plot the memory footprints of `Redis-FORK` and `Redis-MDC` over time for the data set of 13M records, which is shown in Figure 5.5. The `Redis-FORK`'s memory footprint is incrementally



(a) Memory Footprint (b) Throughput

Figure 5.4: YCSB results for varying update proportion (13M records, 2 instances)

increased over time until checkpointing finishes. On the other hand, `Redis-MDC` finishes the logging step within 11 seconds after checkpointing process starts. It then starts to perform the physical memory dump while returning the pages to the OS. As a result, the increase in memory footprint starts to slow down after that. The memory footprint starts to drop at 24 seconds because as the number of pages returned to the OS increases, the chance that updates do not result in page duplication will increase. In summary, our scheme can halve the amount of memory increased during checkpointing.

Throughput. Figure 5.2b shows the throughput of the YCSB workloads with varying data set size for 0.5 update proportion, while checkpointing is being performed in background. We find that when system memory is enough, `Redis-MDC` offers only a marginal performance improvement (by 1.5% on average) over `Redis-FORK`, avoiding some page duplication on CoW faults. However, when the data size is larger than two-thirds of the system memory (from 15M records), the throughput of `Redis-FORK` starts to drop severely due to *swap*. We omit the results for the extreme case scenarios because YCSB clients stop running while complaining slow response time. In contrary, under the same

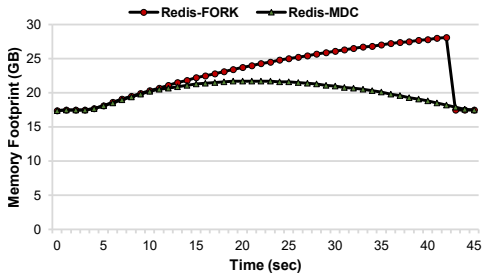


Figure 5.5: Memory footprint over time (13M records, 0.5 update proportion)

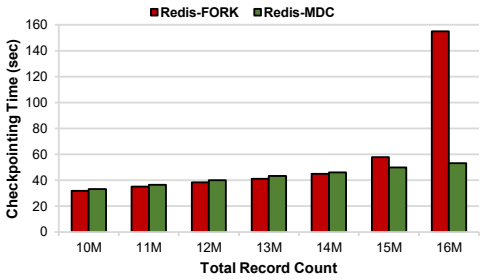


Figure 5.6: Checkpointing Time

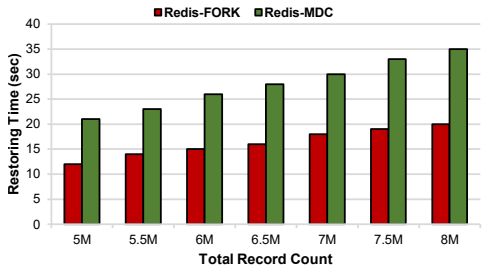


Figure 5.7: Restoring Time

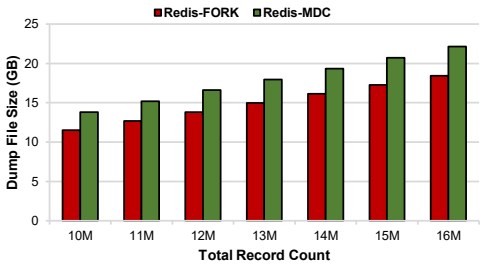


Figure 5.8: Checkpointing File Size

condition, `Redis-MDC` maintains the throughput thanks to the decreased maximum memory footprint. As shown in Figure 5.3b, the aggregated throughput of YCSB clients scales with the number of Redis instances. In the four Redis instances scenario, `Redis-MDC` improves the throughput by 9%, compared to `Redis-FORK`, even when swap does not occur. In contrast, the update proportion has relatively lower impact on the throughput, which can be inferred from Figure 5.4b. When memory is enough, the relative performance improvement of our scheme is marginal (up to 3%). However, in the 100% updates scenario, checkpointing of only 17.4 GB of data set (13M records) can incur swap (note that the total system memory is 32 GB).

Checkpointing Time. We find that the checkpointing time is not affected by other conditions such as the number of Redis instances and the update

proportion, thus here we show the checkpointing time with varying data set size in Figure 5.6. `Redis-MDC`'s checkpointing takes up to 5% longer time to finish than `Redis-FORK`'s when memory is sufficient. This slightly longer time comes from the increased amount of writes. We will explain this later when discussing about the file size.

Restoring Time. Figure 5.7 shows the restoring time with varying data set size. `Redis-MDC` takes longer time (up to 75%) for restoring, relative to `Redis-FORK`. This is because, although we use mmap to mitigate the memory copy overhead, MDC still requires two read operations for each object. Moreover, in contrast to `Redis-FORK` that sequentially reads the contents of the file, `Redis-MDC` rather randomly accesses the `.dump` file for the target objects. Therefore, the buffering behavior of `fread()` greatly improves the I/O performance of `Redis-FORK` but not that of `Redis-MDC`. However, because restoring data may not occurs frequently, we believe that reducing memory footprint is more preferable for IMDBs.

File Size. Figure 5.8 shows the aggregate size of the files related with checkpointing. The aggregate file size of `Redis-MDC` is 20% larger relative to that of `Redis-FORK`, regardless of the data set size. This is mainly due to the increased amount of writes. Although some data in memory does not have to be checkpointed, our scheme writes almost all pages[4] of the address space to the file, having no knowledge of the data characteristics at the memory dump time. We find that when an application has only objects in memory to be checkpointed, MDC increases the checkpointing file size by only 2% [5]. Therefore, the file size do not have to be sacrificed for performance if our scheme can write

[4]MDC excludes the pages that the physical page frame has not been assigned.

[5]We write a simple application that simulates *servicer* and *checkpointer* to demonstrate this. We omit the result due to the limitation of space.

only the pages that need to be checkpointed. Note that even with 20% larger file size, `Redis-MDC`'s checkpointing takes only 5% longer time to finish; MDC has room to improve. Our idea to address this issue is to create two separated memory pool in the user-level memory allocator; one for checkpointing and the other for all other purposes. The allocator can give the OS the hint about the memory regions so that the OS can perform memory dump only for the checkpointing memory regions. Our current prototype does not implement this. We leave it for future work.

5.5 Summary

The existing fork-based checkpointing scheme has been used for many popular IMDBs due to its simplicity yet state-of-the-art performance. In this chapter, we address the unsolved problem of the existing fork-based checkpointing scheme: increase in memory footprint during checkpointing. Our novel approach of using physical memory dump and cooperating with the OS, which we call MDC, can effectively mitigate the memory footprint problem. This is desirable for IMDBs because the memory that has to be reserved for the worst case now can be used for storing more data. As a future work, we will extends our scheme to an user-level memory allocator so that applications, the memory allocator, and the OS all coordinate for more efficient checkpointing scheme.

Chapter 6

Conclusion

Efficient utilization of memory resource is becoming ever significant. However, existing OS memory subsystem provides only limited support for applications to control the memory behavior, which result in inefficient memory utilization. In this dissertation, we have explored three memory-related problems with data-intensive applications in current OS memory subsystem and have addressed them by extending OS memory subsystem for better supporting applications.

In Chapter 3, we first explore the problems of existing datastore architecture for sidestepping double caching and conclude that its inefficiency comes from the non-cooperation between datastore and OS in terms of caching. Therefore, we propose cooperative caching approach that utilizes OS page cache as a victim cache for user-level file content cache, constructing a two-tier cache hierarchy in memory. Experimental results demonstrate that utilizing both user and kernel caching can achieve better performance and higher in-memory cache hit ratio for the given system memory.

In Chapter 4, we examine the memory copy overhead of copy-based I/O in

terms of latency and performance interference. Zero-copy is a known solution to this problem but there is no zero-copy I/O scheme that simultaneously provides 1) transparent copy avoidance via read/write system calls and 2) the benefits of kernel-level caching. To this end, we present an efficient, transparent, and practical zero-copy read file I/O scheme, and several optimization techniques that reduces two major sources of overhead: page remapping and CoW faults. Experimental results show that our scheme provides better I/O performance, even for mixed read/write workloads that incur CoW faults, while significantly reducing the interference with the performance of co-located workloads in data-center-like environments.

In Chapter 5, we investigate the problem of the existing fork-based checkpointing scheme: increase in memory footprint during checkpointing for update-intensive workloads. Our novel approach of using physical memory dump and cooperating with OS, which we call MDC, can effectively mitigate the memory footprint problem. This is desirable for IMDBs because the memory that has to be reserved for the worst case now can be used for storing more data.

In the future work, we will apply `z-READ` to existing persistent datastores and explore the feasibility of transparent zero-copy write file I/O. We will also extend `DBIO` to utilize zero-copy read/write I/O to eliminate memory copy overhead when page is moved between user and kernel buffers. Finally, we will extends `MDC` to an user-level memory allocator so that applications, memory allocator, and OS all coordinate for more efficient checkpointing scheme.

Bibliography

[1] E. Lee and H. Bahn, "Caching strategies for high-performance storage media," *ACM Transactions on Storage (TOS)*, vol. 10, no. 3, p. 11, 2014.

[2] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, *et al.*, "Software-defined far memory in warehouse-scale computers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 317–330, ACM, 2019.

[3] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 267–278, ACM, 2009.

[4] V. Nitu, B. Teabe, A. Tchana, C. Isci, and D. Hagimont, "Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter," in *Proceedings of the Thirteenth EuroSys Conference*, p. 16, ACM, 2018.

[5] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1,

pp. 20–24, 1995.

[6] S. Akram, M. Marazakis, and A. Bilas, "Numa implications for storage i/o throughput in modern servers," in *3rd Workshop on Computer Architecture and Operating System co-design (CAOS'12)*, 2012.

[7] M. Bauer, "Big data, technology, and the changing future of medicine," *Medicographia*, vol. 38, no. 4, pp. 401–410, 2016.

[8] U. Kang, H.-S. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. S. Choi, "Co-architecting controllers and dram to enhance dram process scaling," in *The memory forum*, vol. 14, 2014.

[9] S.-H. Lee, "Technology scaling challenges and opportunities of memory devices," in *2016 IEEE International Electron Devices Meeting (IEDM)*, pp. 1–1, IEEE, 2016.

[10] F. Al-Turjman, "5g-enabled devices and smart-spaces in social-iot: an overview," *Future Generation Computer Systems*, vol. 92, pp. 732–744, 2019.

[11] "MongoDB.." `https://www.mongodb.com/`.

[12] "MySQL 5.7 Reference Manual.." `https://dev.mysql.com/doc/refman/5.7/en/`.

[13] "InnoDB Storage Engine.." `https://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html`.

[14] "WiredTiger Storage Engine.." `http://www.wiredtiger.com/`.

[15] "MMAPv1 Storage Engine.." `https://docs.mongodb.com/v3.2/core/mmapv1/`.

[16] G. Somani and S. Chaudhary, "Application performance isolation in virtualization," in *2009 IEEE International Conference on Cloud Computing*, pp. 41–48, IEEE, 2009.

[17] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for qos-aware clouds," in *Proceedings of the 5th European conference on Computer systems*, pp. 237–250, ACM, 2010.

[18] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi 2: Cpu performance isolation for shared compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 379–391, ACM, 2013.

[19] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 233–240, IEEE, 2013.

[20] H. Liu and B. He, "F2c: Enabling fair and fine-grained resource sharing in multi-tenant iaas clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2589–2602, 2015.

[21] L. Tomás, C. Vázquez, J. Tordsson, and G. Moreno, "Reducing noisy-neighbor impact with a fuzzy affinity-aware scheduler," in *2015 International Conference on Cloud and Autonomic Computing*, pp. 33–44, IEEE, 2015.

[22] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: Sql server's memory-optimized

oltp engine," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 1243–1254, ACM, 2013.

[23] antirez, "Redis," 2017. [Online].

[24] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *2011 IEEE 27th International Conference on Data Engineering*, pp. 195–206, IEEE, 2011.

[25] H. Mühe, A. Kemper, and T. Neumann, "How to efficiently snapshot transactional data: Hardware or software controlled?," in *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, pp. 17–26, ACM, 2011.

[26] antirez, "Redis administration," 2017. [Online].

[27] D. P. Bovet and M. Cesati, *Understanding the Linux kernel.* " O'Reilly Media, Inc.", 2005.

[28] N. Amit, "Optimizing the tlb shootdown algorithm with page access tracking," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, pp. 27–39, 2017.

[29] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.", 2005.

[30] J. Park, C. Min, and H. Yeom, "A new file system i/o mode for efficient user-level caching," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 649–658, IEEE, 2017.

[31] N. Y. Song, Y. Son, H. Han, and H. Y. Yeom, "Efficient memory-mapped i/o on fast storage device," *ACM Transactions on Storage (TOS)*, vol. 12, no. 4, p. 19, 2016.

[32] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel, "Transcendent memory and linux," in *Proceedings of the Linux Symposium*, pp. 191–200, Citeseer, 2009.

[33] A. Badam and V. S. Pai, "Ssdalloc: hybrid ssd/ram memory management made easy," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pp. 211–224, 2011.

[34] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang, "Ssd bufferpool extensions for database systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1435–1446, 2010.

[35] J. C. Mogul, E. Argollo, M. A. Shah, and P. Faraboschi, "Operating system support for nvm+ dram hybrid main memory.," in *HotOS*, 2009.

[36] M. Saxena and M. M. Swift, "Flashvm: Virtual memory management on flash.," in *USENIX Annual Technical Conference*, 2010.

[37] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann, "Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 957–968, IEEE, 2012.

[38] X. Li, A. Aboulnaga, K. Salem, A. Sachedina, and S. Gao, "Second-tier cache management using write hints.," in *FAST*, vol. 5, pp. 9–9, 2005.

[39] L. Wang, X. Liao, J. Xue, S. Weil, Y. Wen, and X. Yang, "Enhancement of cooperation between file systems and applications—on vfs extensions for optimized performance," *Science China Information Sciences*, vol. 58, no. 9, pp. 1–10, 2015.

[40] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 307–320, USENIX Association, 2006.

[41] "mysqldump.." `http://dev.mysql.com/doc/refman/5.7/en/mysqldump.html`.

[42] "TPC-C Benchmark.." `http://www.tpc.org/tpcc/`.

[43] "tpcc-mysql.." `https://github.com/Percona-Lab/tpcc-mysql`.

[44] "Yahoo! Cloud Serving Benchmark.." `https://github.com/brianfrankcooper/YCSB`.

[45] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled direct memory access: Isolating cpu and io traffic by leveraging a dual-data-port dram," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 174–187, IEEE, 2015.

[46] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers." `https://www.cs.virginia.edu/stream`.

[47] P. Druschel and L. L. Peterson, "Fbufs: A high-bandwidth cross-domain transfer facility," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5, pp. 189–202, 1993.

[48] M. N. Thadani and Y. A. Khalidi, *An efficient zero-copy I/O framework for UNIX*. Sun Microsystems Laboratories, 1995.

[49] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Io-lite: a unified i/o buffering and caching system," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 1, pp. 37–66, 2000.

[50] H.-k. J. Chu, "Zero-copy tcp in solaris," in *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pp. 21–21, Usenix Association, 1996.

[51] P. Shivam, P. Wyckoff, and D. Panda, "Emp: zero-copy os-bypass nic-driven gigabit ethernet message passing," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pp. 57–57, ACM, 2001.

[52] A. Kesavan, R. Ricci, and R. Stutsman, "To copy or not to copy: Making in-memory databases fast on modern nics," in *Data Management on New Hardware*, pp. 79–94, Springer, 2016.

[53] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 340–349, IEEE, 2011.

[54] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. H. Loh, "Avoiding tlb shootdowns through self-invalidating tlb entries," in *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*, pp. 273–287, IEEE, 2017.

[55] M. K. Kumar, S. Maass, S. Kashyap, J. Veselỳ, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "Latr: Lazy translation coherence," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 651–664, ACM, 2018.

[56] L. Shrira and H. Xu, "Thresher: An efficient storage manager for copy-on-write snapshots.," in *USENIX Annual Technical Conference, General Track*, pp. 57–70, 2006.

[57] X. Wu, W. Wang, and S. Jiang, "Totalcow: Unleash the power of copy-on-write for thin-provisioned containers," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, p. 15, ACM, 2015.

[58] M. Chowdhury and R. Rangaswami, "Native os support for persistent memory with regions," in *Proceedings of 33rd International Conference on Massive Storage Systems and Technology (MSST'17)*, 2017.

[59] J. Park, C. Min, H. Yeom, and Y. Son, "z-read: Towards efficient and transparent zero-copy read," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, IEEE, 2019.

[60] J. Axboe, "Fio-flexible io tester.{Online}." `https://github.com/axboe/fio`.

[61] C. D. Spradling, "Spec cpu2006 benchmark tools," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 130–134, 2007.

[62] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 55–64, IEEE, 2013.

[63] D. H. Yoon, M. K. Jeong, and M. Erez, "Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput," *SIGARCH Comput. Archit. News*, vol. 39, p. 295–306, June 2011.

[64] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *login: The USENIX Magazine*, vol. 41, no. 1, 2016.

[65] A.-P. Liedes and A. Wolski, "Siren: A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases," in *22nd International Conference on Data Engineering (ICDE'06)*, pp. 99–99, IEEE, 2006.

[66] L. Li, G. Wang, G. Wu, and Y. Yuan, "Consistent snapshot algorithms for in-memory database systems: experiments and analysis," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 1284–1287, IEEE, 2018.

[67] L. Li, G. Wang, G. Wu, Y. Yuan, L. Chen, and X. Lian, "A comparative study of consistent snapshot algorithms for main-memory database systems," *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[68] A. Sharma, F. M. Schuhknecht, and J. Dittrich, "Accelerating analytical processing in mvcc using fine-granular high-frequency virtual snapshotting," in *Proceedings of the 2018 International Conference on Management of Data*, pp. 245–258, ACM, 2018.

[69] U. Vahalia, *UNIX internals: the new frontiers*. Pearson Education India, 1996.

[70] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[71] S. Thomas, K. Sherly, and S. Dija, "Extraction of memory forensic artifacts from windows 7 ram image," in *2013 IEEE Conference on Information & Communication Technologies*, pp. 937–942, IEEE, 2013.

[72] B. Dolan-Gavitt, "The vad tree: A process-eye view of physical memory," *digital investigation*, vol. 4, pp. 62–64, 2007.

[73] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, "Recent advances in checkpoint/recovery systems," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pp. 8–pp, IEEE, 2006.

[74] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012022, 2007.

[75] M. Vaz Salles, T. Cao, B. Sowell, A. Demers, J. Gehrke, C. Koch, and W. White, "An evaluation of checkpoint recovery for massively multiplayer online games," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 1258–1269, 2009.

[76] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White, "Fast checkpoint recovery algorithms for frequently consistent applications," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 265–276, ACM, 2011.

[77] MongoDB, "Disable transparent huge pages (thp)," 2019. [Online].

[78] antirez, "Data types," 2017. [Online].

[79] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, ACM, 2010.

요약

최근 폭발적인 데이터 성장과 더불어 데이터베이스, 키-밸류 스토리지 등의 데이터 집약적인 응용들이 다양한 도메인에서 인기를 얻고 있다. 데이터 집약적인 응용의 높은 성능 요구를 충족하기 위해서는 주어진 메모리 자원을 효율적이고 완벽하게 활용하는 것이 중요하다. 그러나, 범용 운영체제(OS)는 시스템에서 수행 중인 모든 응용들에 대해 시스템 차원에서 공평하게 자원을 제공하는 것을 우선하도록 설계되어있다. 즉, 시스템 차원의 공평성 유지를 위한 운영체제 지원의 한계로 인해 단일 응용은 시스템의 최고 성능을 완전히 활용하기 어렵다. 이러한 이유로, 많은 데이터 집약적 응용은 운영체제에서 제공하는 기능에 의지하지 않고 비슷한 기능을 응용 레벨에 구현하곤 한다. 이러한 접근 방법은 탐욕적인 최적화가 가능하다는 점에서 성능 상 이득이 있을 수 있지만, 시스템 자원의 비효율적인 사용을 초래할 수 있다.

본 논문에서는 운영체제의 지원과 약간의 응용 수정만으로도 비효율적인 시스템 자원 사용 없이 보다 높은 응용 성능을 보일 수 있음을 증명하고자 한다. 그러기 위해 운영체제의 메모리 서브시스템을 최적화 및 확장하여 데이터 집약적인 응용에서 발생하는 세 가지 메모리 관련 문제를 해결하였다. 첫째, 동일한 데이터가 여러 계층에 존재하는 중복 캐싱 문제를 해결하기 위해 응용과 커널 버퍼 간에 메모리 효율적인 협력 캐싱 방식을 제시하였다. 둘째, 입출력시 발생하는 메모리 복사로 인한 성능 간섭 문제를 피하기 위해 메모리 효율적인 무복사 읽기 입출력 방식을 제시하였다. 셋째, 인-메모리 데이터베이스 시스템을 위한 메모리 효율적인 fork 기반 체크포인트 기법을 제안하여 기존 포크 기반 체크포인트 기법에서 발생하는 메모리 사용량 증가 문제를 완화하였다; 기존 방식은 업데이트 집약적 워크로드에 대해 체크포인팅을 수행하는 동안 메모리 사용량이 최대 2배까지 점진적으로 증가할 수 있었다.

본 논문에서는 제안한 방법들의 효과를 증명하기 위해 실제 멀티 코어 시스템에 구현하고 그 성능을 평가하였다. 실험결과를 통해 제안한 협력적 접근방식이 기존의 비협력적 접근방식보다 데이터 집약적 응용에게 효율적인 메모리 자원 활용을 가능하게 함으로써 더 높은 성능을 제공할 수 있음을 확인할 수 있었다.

주요어: 운영체제, 데이터베이스, 메모리 관리, 더블 캐싱, 무복사, TLB 격추, 카피 온 라이트, 체크포인팅

학번: 2013-20794