Optimizing Memory and Storage Disaggregation for Data-intensive Systems

by

Qirui Yang

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved March 2024 by the
Graduate Supervisory Committee:

ARIZONA STATE UNIVERSITY

May 2024

ABSTRACT

Data-intensive systems such as big data and large machine learning (ML) systems experience serious scalability challenges due to the ever-increasing data demand from ML and analytics applications and the resource fragmentation caused by conventional monolithic server architecture. Memory and storage disaggregation emerges as a pivotal technology to address these challenges by decoupling memory and storage resources from individual servers and managing and provisioning them to applications as a shared resource pool. This dissertation investigates several important aspects of memory and storage disaggregation and proposes novel solutions to support data-intensive applications.

First, caching is a fundamental way to utilize disaggregated storage, but building a large disaggregated cache is challenging because the commonly-used fix-sized cache block allocation scheme is unable to provide good cache performance with low memory overhead for diverse cloud workloads with vastly different I/O patterns. The dissertation proposes a novel adaptive cache block allocation approach that dynamically adjusts cache block sizes based on changing I/O patterns. This approach significantly improves I/O performance while reducing memory usage, outperforming traditional fixed-size cache systems in diverse cloud workloads.

Second, large ML applications such as large language model (LLM) inference are memory demanding, but to support them using disaggregated memory brings challenges to memory management since disaggregated memory has higher memory access latency compared to local memory. The dissertation proposes latency-aware memory aggregation which cautiously distributes memory accesses to minimize the latency gap between local and disaggregated memory. It also proposes NUMA-aligned tensor parallelism to further improve the computing efficiency. With these optimizations, LLM inference achieves substantial speedups.

Finally, to address the cost, power consumption, and volatility of DRAM, the dissertation proposes to incorporate flash memory into memory pools within the disaggregation framework. By establishing a tiered memory architecture which combines fast-tier local DRAM with slow-tier DRAM and flash memory in the memory pool and effectively migrates data based on hotness across memory tiers, this approach not only reduces expenses but also maintains the overall performance and scalability of data-intensive systems.

ACKNOWLEDGMENTS

I wish to convey my deepest gratitude to Dr. Ming Zhao for his invaluable guidance and support over the past few years. I am immensely grateful for the opportunity to pursue a range of research interests, discovering my passion along the way. Among the many virtues I aspire to emulate from him, his integrity, ethics, profound insight, and unwavering support stand out the most.

My heartfelt thanks go to my family, particularly my wife, Runyu Jin, for their unwavering support throughout my PhD journey. Their encouragement and steadfast belief in me were crucial in overcoming the challenges I faced and instrumental in the successful completion of my dissertation.

I extend my appreciation to my committee members: Dr. Aviral Shrivastava, Dr. Fengbo Ren, and Dr. Jia Zou, for their openness, valuable advice, and constructive criticism, all of which were vital in finalizing this thesis. The quality of my research benefitted significantly from our meaningful discussions.

I also wish to acknowledge Wenji Li, Yitao Chen, Kaiqi Zhao, Zige Huang, Yiting Yao, Sungho Hong, Eugene Kuznetsov, Jorge Cabrera, and my colleagues from the VISA Lab for their contributions to my research development and idea formulation. I am eager to see where our paths might intersect in the future.

A special thanks to my close friends, whose presence made my PhD journey far more enjoyable. The friendships I've formed and the lessons learned from them have been among the highlights of my experience. I wish them all success in their future ventures and am thankful for their indirect yet meaningful support of my work.

Lastly, I owe a debt of gratitude to the administrative staff of ASU, notably Christina Sebring, Monica Dugan, Pamela Dunn, and Jaya Krishnamurthy, for their assistance.

Thank you, everyone, for your support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Chapter 1

INTRODUCTION

1.1 Problem Statement

Data-intensive systems, including those used for big data analytics and machine learning (ML), are currently facing significant scalability issues. These challenges stem from two main sources. First, as ML and data analytics are rapidly adopted by broader domains, so does the demand for substantial data processing capabilities. These applications are characterized by their need to process and analyze vast amounts of data quickly. The surge in data volume, coupled with the complexity of processing tasks, places enormous stress on today's computing infrastructures. The challenge is not just managing this vast data but doing so in a manner that is both time and resource-efficient.

The second major challenge stems from the inefficiencies of traditional monolithic server architecture which tightly couples different types of resources in homogeneous server enclosures and is increasingly becoming bottlenecks in the face of growing data demands. This architecture, designed with a one-size-fits-all mindset, often leads to resource fragmentation and wastage, as it is difficult to simultaneously fully utilize all the different types of resources including compute, memory, storage, and networking. As data volume and complexity grow, these monolithic systems struggle to adapt, unable to scale resources effectively to meet demand. This architectural limitation not only hurts application performance but also increases operational costs, making it imperative to explore and adopt more innovative, flexible architectures capable of scaling with the data-intensive computing.

One promising approach to overcome these hurdles is memory and storage disaggregation. This innovative strategy involves separating memory and storage resources from individual servers and pooling these resources across the network or interconnect. Such a method allows for a more dynamic and efficient allocation and management of these critical resources. By decoupling memory and storage from specific servers, resources can be made available to applications precisely when needed, facilitating a more responsive and adaptable infrastructure. This dynamic allocation is instrumental in optimizing resource utilization, ensuring that memory and storage resources are not left idle and are instead employed to their maximum potential.

Furthermore, this approach significantly enhances the flexibility and scalability of data-intensive systems. By making resources centrally available and dynamically allocated, systems can more effectively meet the demands of advanced ML and analytics workloads. The ability to scale resources on demand without the constraints of traditional server architectures allows improved efficiency. This optimization of resource utilization not only addresses current scalability challenges but also prepares data-intensive systems to handle future increases in data volume and processing requirements. Consequently, memory and storage disaggregation emerges as a key enabler for next-generation data processing, offering a promising solution to the scalability issues of modern data-intensive applications.

Considering these factors, this dissertation delves into the complexity of memory and storage disaggregation, exploring its potential to solve the challenges in today's infrastructure of data-intensive systems. Through comprehensive analysis, it introduces innovative solutions designed to leverage disaggregated storage and memory, thereby significantly enhancing the efficiency of data-intensive applications. By addressing the key challenges associated with traditional monolithic server architecture and proposing novel strategies for resource management, this dissertation aims to pave the way for more scalable, efficient, and flexible data processing systems capable of accommodating the evolving needs of ML and analytics applications.

1.1.1 Rack-scale Disaggregated Cache System

First, in public, private, and hybrid cloud environments, hard disk drive (HDD) based block storage is essential for its fast, scalable, and reliable data access, yet it doesn't match the performance of directly attached NVMe SSD storage, prompting the use of NVMe SSD caching to enhance access speeds for data-intensive applications. The prevalent host-side caching mechanism faces challenges in uneven cache utilization across servers due to the inflexibility of fixed-size cache blocks and the inability to share caches among servers, limiting efficiency. Rack-scale cache disaggregation offers a solution by enabling shared cache resources across servers, improving utilization and system scalability. However, the traditional fixed-size cache block management struggles to accommodate the dynamic nature of cloud workloads. Using smaller cache blocks like 32KiB can achieve better I/O performance as it incurs smaller cache miss penalty compared to larger cache block sizes. However, its metadata overhead for managing the cache resource is higher, which causes larger memory footprint as the metadata usually needs to be cached in memory for performance. Conversely, using larger cache blocks such as 512KiB can improve the cache hit ratio by exploiting the spatial locality within the requests and reduce the memory overhead associated with metadata. Nevertheless, this comes at the cost of larger cache miss penalty, which can significantly reduce I/O performance if the spatial locality is rare.

To solve the problems, we aim to design a rack-scale disaggregated cache solution that provides good cache performance with low metadata overhead, regardless of the cloud workloads. We propose AdaCache, a rack-scale disaggregated cache system that employs variable-sized cache blocks to adapt to various cloud workloads. Ada-

Cache allocates cache blocks of different sizes based on the I/O request size. For requests with large I/O sizes, large cache blocks are allocated to reduce the number of allocated cache blocks, thus improving I/O performance and reducing metadata memory overhead. For requests with smaller sizes, AdaCache assigns small cache blocks to avoid read/write amplification between the cache system and backend storage as well as cache pollution. In cases where requests are only partially cached, AdaCache dynamically determines the optimal cache block sizes. It utilizes a greedy allocation strategy, aiming to minimize both the number of cache blocks required and I/O amplification by tracking every portion of the request not currently in the cache.

According to the evaluation results, AdaCache has demonstrated significant improvements in I/O performance compared to traditional fix-sized cache. Specifically, it can improve read latency by 20% and write latency by 9% compared to 32KiB block-sized cache in trace replay. AdaCache is also capable of saving up to 74% I/O traffic to cloud block storage and up to 63% I/O traffic to the cache compared to 256KiB block-sized cache. Moreover, AdaCache has achieved up to 41% memory savings compared to 32KiB block-sized cache. All of these improvements are accomplished with merely 2 microseconds of computation overhead at the cache layer compared to a traditional fix-sized cache.

1.1.2 LLM Inference Framework for CPU

Second, existing Large Language Model (LLM) inference frameworks on CPUs adopt multithreading and data parallelism as acceleration methods. We observe that these two methods fail to improve inference performance. On one hand, we find that offering more computing resources with multithreading does not improve the LLM inference performance effectively. Based on our analysis, the performance improvement is constrained by the memory bottleneck. The elevated memory latency caused by memory throughput contention due to the increased computing cores has impeded performance improvement and even led to a plateau of performance gains. On the other hand, we argue that data parallelism is not suitable for large models, which have high demands for computing power; replicating inference instances in this context can rapidly use up available computing resources. Besides, the memory throughput contention problem, as discussed above, becomes more severe with data parallelism.

To address the above problems, we propose latency-aware memory aggregation and NUMA-aligned tensor parallelization. In the first method, we introduce additional memory resources, such as CXL memory expanders, to alleviate the stress on the local memory. The introduction of additional memory through CXL can potentially increase the system memory latency due to its high memory latency, which might adversely affect performance. To tackle this issue, we optimize the distribution of workloads across various memory types to achieve the lowest memory latency. In the second method, we employ a "divide and conquer" strategy to improve computing efficiency. In our second method, we adapt tensor parallelism, a technique that divides large models into smaller sub-models to be processed in parallel. Our innovative application of this technique on CPUs processes these sub-models with fewer CPU and memory resources, thereby improving the utilization efficiency of these resources. We design NUMA-aligned tensor parallelization which binds CPU cores and sub-NUMA nodes based on data locality to eliminate cross-NUMA memory accesses.

We implement a CPU-based inference framework incorporating the two optimization approaches. The evaluation results show that our proposed memory aggregation method can reduce the first token latency by 9%, the average token latency by 24%, and the end-to-end latency by 22% for a 6.7b model inference when batch size is 1. Our proposed tensor parallelism method can reduce the first token latency by

41%, the average token latency by 29%, and the end-to-end latency by 33% for a 66b model when the model is parallelized across four groups of CPUs and the batch size is 8. When we combine both methods, we can achieve 61% improvement in first token latency, 33% improvement in average token latency, and 43% improvement in end-to-end latency for a 66b model when the model is parallelized across eight groups of CPUs and the batch size is 8.

1.1.3 CXL-based Hybrid Memory Pool

Finally, the largest cost of the public cloud still comes from the memory itself. Public providers, such as Azure, spend half the server cost on DRAMs and the cost will continue to grow due to the growing demands of data-intensive applications. Also, DRAMs use two to three times more power and cooling than SSDs do. At the same time, NVMe SSDs are getting higher throughput while continuing to be more power-efficient and cost much less, nearly one fifth the cost of DRAM. To further reduce the cost from DRAM, SSDs are good candidates for disaggregated memory pools. They can be incorporated with DRAM to save even more cost. The addition of fast SSDs to the memory pool forms another tiered memory architecture in the existing two-tier memory pooling systems (local memory and remote memory) which presents new challenges to memory disaggregation in both performance and cost.

To validate the potential of incorporating SSDs into the memory pool. We first emulate the behavior of the CXL-based hybrid memory pool using software based solutions in the absence of real hardware through cache-coherent NUMA architecture. The tiered memory system uses local DRAM as the first tier with the lowest memory access latency. The hybrid memory pool is the second tier with higher memory access latency. Within the memory pool, there is a sub-tiered memory system with NUMA node 2 as the first tier and NVMe SSD as the second tier. We then evaluate how the hybrid memory pool of both DRAM and SSDs affects the applications performance compared to DRAM-only memory pool in the cloud environment by changing different compositions of DRAM and SSDs, or overcommit ratio, to quantize the performance degradation. The overcommit ratio is defined as the ratio of required memory and actually allocated memory. By doing the above steps, we shed some light in solving the previously mentioned challenges of tiered memory disaggregation.

The evaluation results show that applying SSDs to the memory pool does reduce cost while maintaining the same level of performance for these types of workloads. On one hand, database and analytic workloads, such as TPC-C and TPC-H, which have high requirements for memory latency and bandwidth, are severely affected by the hybrid memory pool. On the other hand, the performance of computing intensive applications FFmpeg degrades slightly by 0.3% when the overcommit ratio increases from 1 to 1.5. When we continue to increase the overcommit ratio, the performance degradation does not get much worse, dropping by 11.9% for an overcommit ratio of 4. By using NVMe SSD, even with a mere addition of NVMe SSD at an overcommit ratio 1.5, we can already save 32% of total cost of memory. What's more, we can save up to 72.5% of total cost of memory when the overcommit ratio is 4.

1.2 Contributions

The contributions detailed in this dissertation help enhance the application of memory and storage disaggregation techniques. By refining and applying these designs, the research significantly boosts the efficiency, scalability, and cost efficiency of computing systems. This is particularly relevant in today's surge in data volume and complexity, a trend that is expected to continue into the future. This dissertation not only addresses the immediate challenges posed by today's data-intensive applications but also lays a robust foundation for future innovations. The result is a set of optimized systems that are well-equipped to handle the increasing demands for data processing, thereby supporting the growth and evolution of next-generation computing technologies.

1.3 Outline

The rest of this dissertation is organized as follows: Chapter 2 presents the rack-scale disaggregated cache system with variable-sized cache blocks; Chapter 3 presents the CPU-based LLM inference frameworks with latency-aware memory aggregation and NUMA-aligned tensor parallelism; Chapter 4 presents the design of CXL-based hybrid memory pool; and Chapter 5 concludes the dissertation with insights and visions for resource disaggregation and data-intensive computing.

Chapter 2

IMPROVING CACHE EFFICIENCY THROUGH DISAGGREGATION

2.1 Disaggregated Cache

2.1.1 Rack-Scale Cache Disaggregation

Cloud block storage has been widely adopted by today's public, private, and hybrid cloud infrastructure for primary data storage ebs (2023); ibm (2023); goo (2023); cep (2023). With block storage, data is partitioned into fix-sized blocks and stored on the underlying storage medium. These blocks can be directly accessed by applications or through mounted file systems Liu *et al.* (2019a, 2020), allowing for quick modification of specific blocks to efficiently serve I/O requests.

NVMe SSDs are commonly used as a caching solution in large-scale cloud block storage systems to improve I/O performance Zhou *et al.* (2020). Typically, caches are deployed on computing hosts to mitigate the high network latency to the storage clusters. However, cloud providers often encounter the challenge of load imbalance where some cache devices are more heavily used than others, leading to overloaded, under-loaded, or well-loaded cache devices on computing hosts Afzal and Kavitha (2019). This results in unbalanced cache utilization and wasted cache resources.

Cache disaggregation presents a solution to the aforementioned issues by disaggregating all the cache resources, enabling cache to be shared and managed as a whole. It decouples SSD cache from the computing nodes and allows independent utilization of cache resources regardless of where an application is placed. In this sense, the cache resources are shared by all the applications and the cache load imbalance problem is addressed. In cloud environments, this can be achieved at either cluster scale or

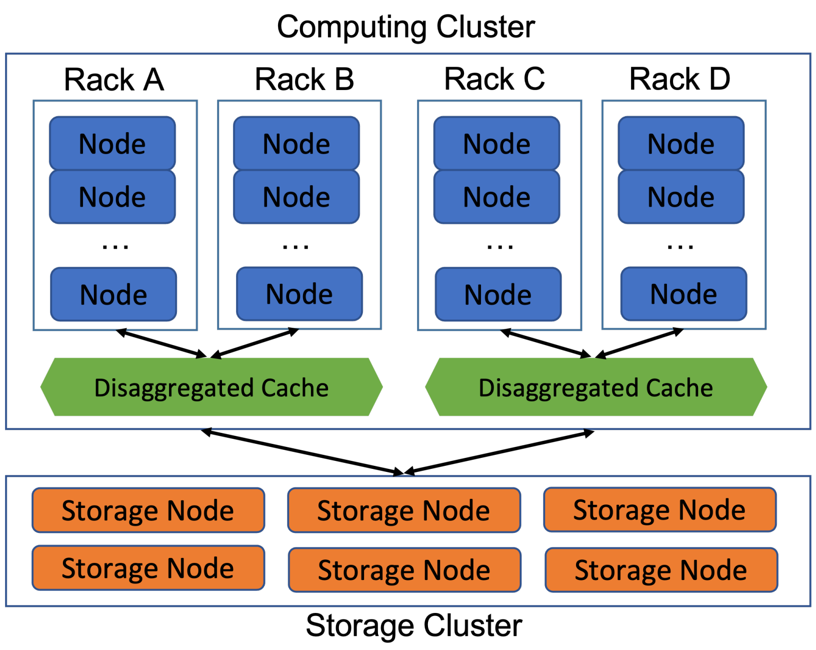


Figure 2.1: Rack-Scale Cache Disaggregation

rack scale. Cluster-scale cache disaggregation offers more pooled cache resources and consequently can result in better cache utilization compared to rack-scale. However, it suffers from higher network latency to access cache across the cluster which can negatively impact I/O performance. Additionally, it requires complicated software design and may inversely bring unacceptable software overhead and offset its benefit. Conversely, rack-scale cache disaggregation can provide superior cache resource utilization compared to the local cache and involve much lower network and software overhead compared to cluster-scale. As such, it provides an optimal trade-off between cache resource utilization and I/O performance. Figure 2.1 illustrates an example of rack-scale cache disaggregation.

Rack-scale cache disaggregation enables cache devices within the same group of racks to share a cache server, providing computing servers of the same rack group with a pool of shared cache resources. The fast data transfer between computing nodes and the cache server can be achieved with the adoption of NVMe over Fabrics (NVMeoF) nvm (2023a) technology, which is a protocol designed to provide storage to computing servers through the network using the NVMe protocol. It adds less than 10 microseconds of additional latency to locally attached NVMe devices nvm (2021), making it an ideal choice for connecting the cache pool to the computing nodes. According to a recent performance report nvm (2023b), NVMeoF using RDMA nvm (2023a) has demonstrated impressive speed, achieving more than 11M 4K IOPS with an average latency of 231 microseconds using 100 Gbps NICs. As network bandwidth continues to double every few years, this performance is expected to improve even further. With such high performance, a single cache server can effectively serve thousands of concurrent NVMeoF connections. Furthermore, a single cache server can provide large storage capacities. For example, Samsung's Poseidon reference system ins (2021) can support up to 24 Samsung PM1733 NVMe SSDs with a total capacity of up to 368TiB. This capacity is sufficient to support thousands of cache clients for cloud block storage.

Figure 2.2 compares the I/O performance of different storage setup: local NVMe SSDs (local), remote NVMeoF SSDs (nvmeof), and remote all-flash Ceph Rados Block Devices (rbd) cep (2023). Local and nvmeof each consists of four Samsung PM9A3 NVMe SSDs that form a RAID0. Rbd consists of 12 Samsung PM9A3 NVMe SSDs from a 3-node Ceph cluster that form a RAID0. We use local to demonstrate the performance of the local cache, and nvmeof to demonstrate the performance of the disaggregated cache. Rbd is an open-sourced cloud block storage system used to demonstrate the performance of cloud block storage without NVMe SSD caching. We ran the FIO fio (2023) benchmark issuing 30 minutes of asynchronous random 4K reads and writes with the same I/O queue depth to different storage setups. We observe that local NVMe SSDs outperform cloud block storage by 60X. Remote SSDs

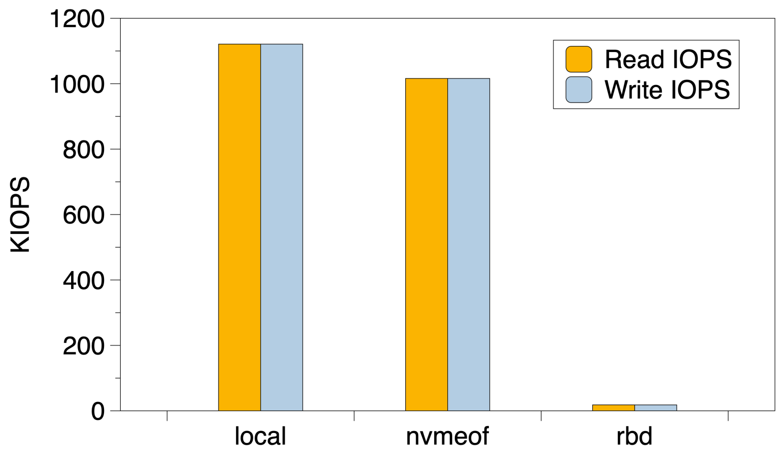


Figure 2.2: IOPS Comparison of Local SSDs, NVMeoF SSDs, and All-Flash Ceph RBD.

using NVMeoF have comparable performance to local NVMe SSDs with merely a 9% drop in IOPS.

2.1.2 *Rack-Scale Cache Management*

A cache block is the minimum unit of cache that can be read from or written to. The block size determines the size of an I/O operation that can be performed. Common cache block sizes range from 512B to 64KiB Waldspurger *et al.* (2015); Zhang *et al.* (2020); Li *et al.* (2016); Arteaga *et al.* (2016); Fu *et al.* (2018). The choice of cache block size can impact the performance, endurance, and cost of a storage solution by affecting cache hit ratio, I/O volume, and in-memory metadata overhead. Therefore, it's important to select a cache block size that fits the workload best. Smaller cache blocks often have better I/O performance due to the smaller I/O volume, which comes from the smaller cache block allocation and smaller cache miss penalty. However, they may have a lower cache hit ratio because they cannot fully leverage the spatial locality within the application requests Hennessy and Patterson

(2011).

For a rack-scale cache with hundreds of terabytes of cache space, the large memory footprint for the metadata is another concern for small block sizes. For example, assuming each cache block only requires 40 bytes of memory metadata to provide a source address to cache address mapping (including source address, cache address, a pointer for indexing, and two pointers for LRU) Arteaga *et al.* (2016); mem (2023), a 368 TiB cache with 16 KiB cache block size would require 920 GiB of memory footprint, which is difficult to fit in memory, considering memory density grows 10 times slower than SSD density mem (2021).

Large cache blocks, on the other hand, can potentially improve hit ratio Hennessy and Patterson (2011) due to better exploitation of spatial locality. Additionally, the memory footprint reduces linearly with the increased size of the cache block. Take the last example: a 368 TiB cache with 512 KiB cache block size would require merely 29 GiB of memory footprint. However, large cache blocks lead to large cache block allocation and large cache miss penalty which can significantly harm I/O performance. These reasons stop large cache blocks from being applied in reality. Section 4.2 presents a thorough comparison of I/O performance using cache of different cache block sizes.

The cloud environment is dynamic and changes rapidly over time with varying workloads. Some workloads involve small requests, such as those from transactional databases, while others have large requests, such as those from multimedia systems. We conducted an analysis of request size cumulative distribution functions (CDF) from three real-world traces: Alibaba block I/O Traces (*alibaba*) Li *et al.* (2020), MSR Cambridge Traces (*msr*) Narayanan *et al.* (2008), and Systor '17 Traces (*systor*) Lee *et al.* (2017) (detailed information about the traces is presented in Section 4.2). Figure 2.3 shows the results. We observe that the distribution of request sizes varies

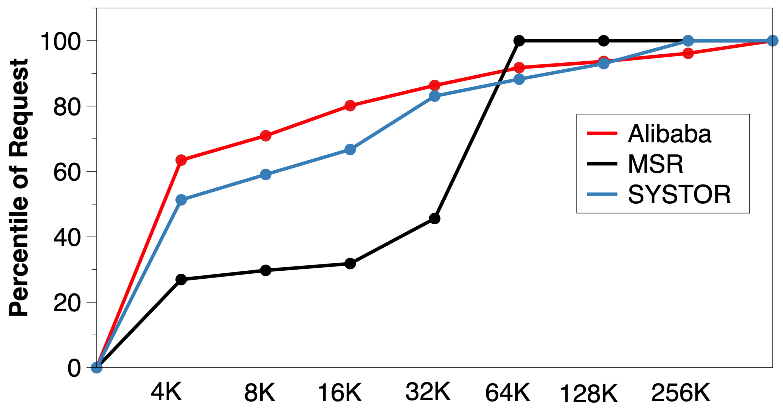


Figure 2.3: Request Size CDF of different traces

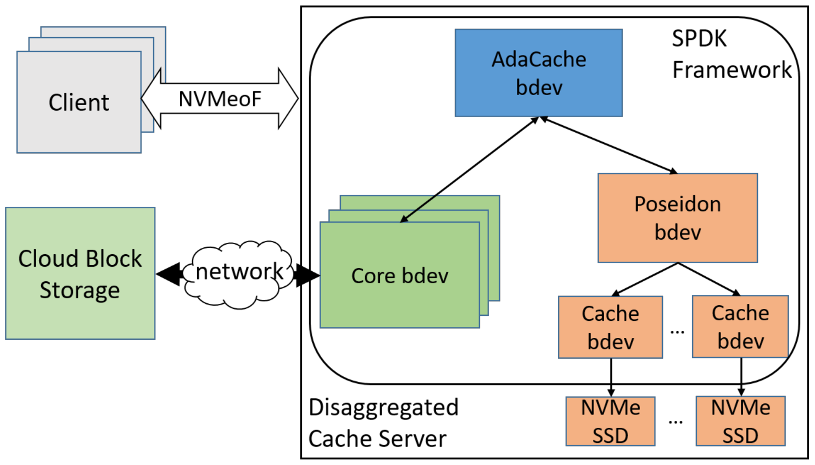


Figure 2.4: Disaggregated Cache Architecture

across the traces. For *alibaba* and *systor*, more than half of the requests are smaller than or equal to 4KiB. For *msr*, more than half of the requests are larger than 32KiB. Based on the above observations, a traditional fix-sized block cache is insufficient for today's complex cloud environment. Instead, we design an adaptive cache that can adapt the cache block size to different cloud workloads which is elaborated in Section 2.2.

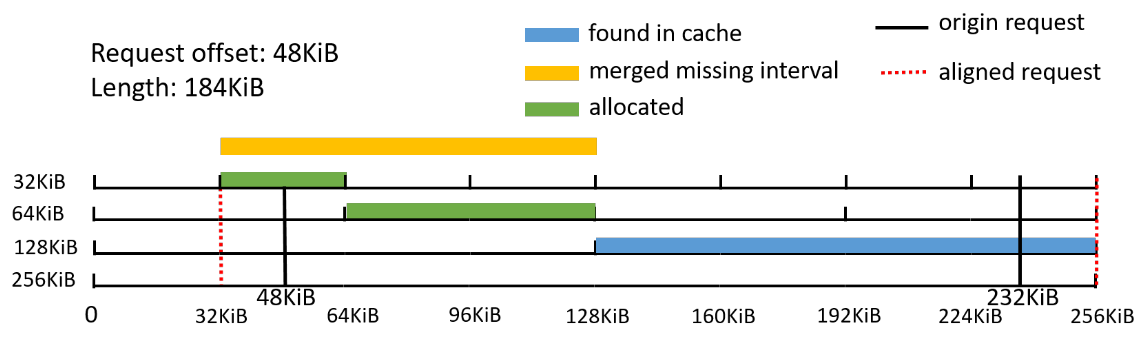


Figure 2.5: Adaptive Cache Block Allocation

2.1.3 Implementation

AdaCache extends PoseidonOS poe (2023), a userspace software-defined storage (SDS) solution providing high-throughput and low-latency flash storage virtualization with capacity elasticity and data protection (RAID), to offer rack-scale disaggregated cache service for cloud block storage. It is implemented as a virtual block device (bdev) module bde (2023) using the SPDK framework. By using a virtual bdev module, AdaCache can be seamlessly integrated with a wide range of cloud block storage bdevs, enabling compatibility with existing storage systems.

Figure 2.4 illustrates the architecture of AdaCache. Each local NVMe SSD is represented by a cache bdev in the SPDK framework. All the cache bdevs are managed by PoseidonOS to offer a large virtualized disaggregated cache space to AdaCache. Each virtual drive in the cloud block storage is represented by a core bdev. AdaCache claims the cache and core bdevs and redirects I/Os between them with no requirement for knowledge of the I/O and network protocol specifics of the underlying bdevs. AdaCache uses GLib's The GNOME Project (2023) hash table implementation for the in-memory key-value stores.

2.2 Adaptive Cache Block Size

2.2.1 Fix-sized Cache Allocation

Traditional fix-sized cache block allocation has three major steps: address alignment, address lookup, and cache block allocation. Address alignment aligns the offset of the original I/O requests to the aligned offset based on cache block size. Assume R_o is the request offset, B is the cache block size, and A_o is the aligned offset. A_o is computed using the following Equation 2.1.

$$A_o = floor(R_o/B) * B \tag{2.1}$$

For example, a read request with offset 33KiB using 32KiB as cache block size aligns to aligned offset 32KiB.

During address lookup, the aligned offset is used as the key to look up the cache address in an in-memory key-value store. In case of a read cache hit, data is read from the cache address directly. Otherwise, a new cache block is allocated and data is read from the backend storage and cached to the newly allocated cache block.

In case of a write cache miss, data is first read from the backend storage and cached to a newly allocated cache block. If the cache uses write-back policy, data is written to the cache block and dirty cache blocks are written back to the backend storage periodically or when they are replaced from the cache. If the cache uses write-through policy, data is written to the cache block and backend storage simultaneously to maintain data consistency. When the cache becomes full, a replacement algorithm such as Least Recently Used (LRU) or Least Frequently Used (LFU) is used to determine which data to replace before allocation happens.

2.2.2 Variable-Sized Cache Allocation

Cloud workloads are dynamic in nature, and therefore, the cache system should be able to adapt itself to different workloads that may have varying request sizes. For small requests, small cache blocks are deemed sufficient while large cache blocks may cache unnecessary data, resulting in cache pollution and increased I/O volumes. Conversely, for large requests, large cache blocks can reduce the number of I/Os between the cache and the cloud block storage, and can also reduce the metadata memory overhead. AdaCache uses adaptive cache block allocation which allocates different sizes of cache blocks based on the request size.

AdaCache first generates a list of missing intervals for all the parts of the request that are missing in the cache. As shown in Figure 2.3, a request can be larger than 256KiB and cover multiple cache blocks. AdaCache determines the aligned range of the request by aligning the request offset and end address (offset + length) to the smallest block size and iterates through the request to find out all the missing intervals.

Because the cache employs variable cache block sizes, it needs to check the in-memory key-value store of every block size to find out if any part of the request is cached under each block size. Figure 2.5 illustrates an example where a request at offset 48KiB with length 184KiB on a cache that employs cache block sizes of 32KiB, 64KiB, 128KiB, and 256KiB. In this example, the latter part of the request (from 128KiB to 232KiB) is cached under the 128KiB block size. The aligned request range is from 32KiB to 256KiB.

Within the request range, AdaCache starts the search from the smallest cache block size (32KiB in the example), and checks if the current address is cached under any of the cache block sizes. AdaCache first aligns the current address to different cache block sizes using Equation 2.1. For the example, the aligned offsets are 32KiB, 0, 0, and 0 for the cache block sizes of 32KiB, 64KiB, 128KiB, and 256KiB respectively. It then uses these aligned offsets to search the in-memory key-value store of each cache block size. If the result is all misses, then it knows that the current address with the smallest cache block size (the interval between 32KiB and 64KiB in the example) is not cached, and it adds the interval to the list of missing intervals. AdaCache merges missing intervals if they are contiguous to allocate the largest possible cache block for the intervals. AdaCache then moves on to the next address covered by the request (64KiB in the example) and repeats the above process. After checking the whole request, AdaCache gets a complete list of missing intervals. In the example, the interval from 32KiB to 128KiB is missing in the cache. Algorithm 3 presents the pseudo-code of the missing intervals generation.

For each missing interval in the list, AdaCache tries to allocate using the largest possible cache block size. This greedy allocation ensures that AdaCache reduces the number of allocated cache blocks and I/O counts. To determine if a block size is suitable for the missing interval, AdaCache makes sure the cache block is within the range of the missing intervals because the addresses that go beyond these intervals may have been cached.

In the example, AdaCache first checks how to allocate for the interval from 32KiB to 128KiB. The largest possible cache block for this interval is actually 32KiB, because all the larger cache blocks start beyond this interval. For the remaining missing interval from 64KiB to 128KiB, the largest possible cache block is 64KiB, because the interval from 64KiB to 128KiB is within the range of the missing interval (64KiB to 128KiB). Therefore, at the end of this greedy allocation process, AdaCache caches two blocks that include one 32KiB cache block from 32KiB to 64KiB and one 64KiB cache block from 64KiB to 128KiB. Algorithm 2 presents the pseudo-code of the

Algorithm 1 Missing Intervals Generation

```
1: Remarks:
       $B_n, \dots B_1$: block size from large to small, $H_B$: hash table for
       block size $B$, $A_B(O)$: align offset $O$ using block size $B$,
       $M_{AP}(B, E)$: merge offset interval $\{B, E\}$ to $MissingIntervals$
2: Inputs:
       $O$: request offset in bytes, $L$: request length in bytes
3: Output:
       $MissingIntervals$: a list of missing cache blocks
4: $MissingIntervals \leftarrow \{\}$, $begin \leftarrow A_{B_1}(O)$, $end \leftarrow A_{B_1}(O + L) + B_1$
5: while $begin \neq end$ do
6:     $hit \leftarrow$ false
7:     for $B \leftarrow B_1, \dots B_n$ do
8:         $begin\_aligned = A_B(begin)$
9:         if $begin\_aligned \in H_B$ then
10:            $begin \leftarrow begin\_aligned + B$
11:            $hit \leftarrow$ true
12:            break
13:        end if
14:    end for
15:    if $hit \neq$ true then
16:        $M_{AP}(begin, begin + B_1)$
17:        $begin \leftarrow begin + B_1$
18:    end if
19: end while
20: return $MissingIntervals$
```

Algorithm 2 Greedy Cache Block Allocation

```
1: Remarks:
       $B_1, \ldots B_n$: block size from small to large
       $H_B$: hash table for block size $B$
       $A_B(O)$: align offset $O$ on block size $B$
       $BA(I)$: the begin address of interval $I$
       $EA(I)$: the end address of interval $I$
2: Inputs:
       $MissingIntervals$: a list of cache blocks to allocate
3: for each $I \in MissingIntervals$ do
4:     $begin \leftarrow BA(I)$
5:     $end \leftarrow EA(I)$
6:     while $begin \neq end$ do
7:         for $B \leftarrow B_n, \ldots B_1$ do
8:             if $begin \neq A_B(begin)$ then
9:                 continue
10:            end if
11:            if $B > end - begin$ then
12:                continue
13:            end if
14:            $H_B \leftarrow begin \cup H_B$                ▷ allocate cache block
15:            $begin \leftarrow begin + B$
16:        end for
17:    end while
18: end for
```

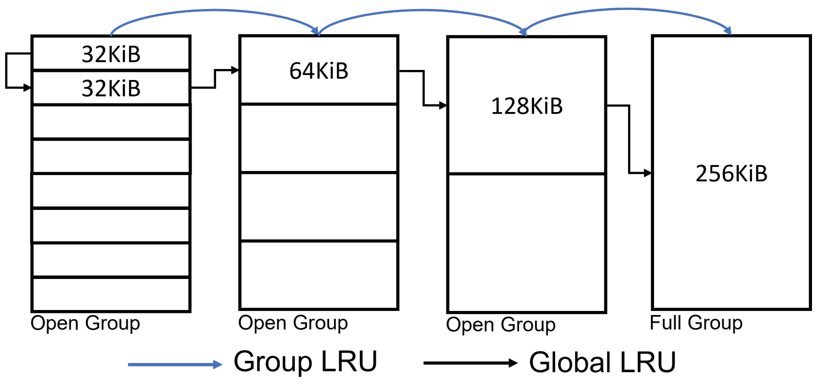


Figure 2.6: Group-Based Cache Organization

greedy cache block allocation.

Assuming N is the request length, M is the number of different cache block sizes, and K is the total number of cache blocks in the cache, the algorithm's time complexity of fix-sized and adaptive cache block allocation have upper bounds of $\mathrm{O}(K * N)$ and $\mathrm{O}(K * N * M)$, respectively. In practice, M is set to a constant value, such as 4 in Figure 2.5 where the time complexity can be approximated as $\mathrm{O}(K * N)$, which is equivalent to the fix-sized cache block allocation. The space complexity of the algorithm is identical to the fix-sized cache block allocation, which is $\mathrm{O}(K)$.

2.2.3 Group-Based Cache Organization

Adaptive cache block allocation is an effective technique that can leverage both small and large blocks, making it suitable for dynamic cloud workloads. However, it incurs fragmentation. When the cache becomes full and adaptive cache blocks get allocated, the cache space is divided into non-contiguous variable-sized pieces. When large requests come, the replacement of smaller blocks can generate many scattered small holes and it is hard to fit a large cache block in.

To address the issue of fragmentation, AdaCache utilizes the concept of slab allocator Bonwick *et al.* (1994); mem (2023), which involves grouping cache blocks of the same size together into identical-sized groups. Cache blocks belonging to the same group are stored physically adjacent to each other in the cache. Consequently, when the cache is full, a whole group is replaced, creating a contiguous piece of cache space for cache block allocation.

AdaCache chooses the largest cache block size as the group size. In this way, replacement of a whole group can free just enough cache space for the largest cache block allocation. In the case of small block allocation, the replacement of a whole group creates an open group that can be used to allocate many cache blocks of that block size. Figure 2.6 illustrates an example of the group-based cache organization. The cache block sizes are 32KiB, 64KiB, 128KiB, and 256KiB and the group size is 256KiB. There are three open groups storing 32KiB, 64KiB, and 128KiB cache blocks, respectively, and one full group storing a 256KiB cache block.

When allocating a cache block, AdaCache checks if the cache is full. If it is not, the allocator examines if there is an open group with the same block size. If such a group exists, the block is allocated from the open group. If there is no such open group, AdaCache creates a new one and allocates the cache block from there. If the cache is full, AdaCache replaces an entire group and follows the above procedure. Assume M is the number of different cache block sizes, there are a maximum of M open groups kept in the cache at any given time, and it does not waste significant cache space. For example, in Figure 2.6, at most 4 256KiB open groups are kept in the cache and used to allocate cache blocks for coming requests.

2.2.4 Two-Level Cache Replacement

Following group-based cache organization, AdaCache uses a group-based LRU replacement policy that links all the groups together for cache replacement. When a cache block is accessed, the group that contains the cache block is promoted to the head of the group-based LRU list. When the cache is full, AdaCache replaces the group that is at the tail of the LRU list. Although each cache miss may trigger a write-back I/O of the whole group to be evicted, the I/O volume is smaller than that of using large fix-sized cache blocks. Every time a whole group is evicted, all of its space is freed up at once in the cache and can be used to store a number of small cache blocks from future requests.

One potential drawback of the group-based replacement policy is that it may retain cold blocks that are in the same group as the frequently accessed hot blocks in the cache, leading to cache pollution. To alleviate the problem, AdaCache incorporates a global cache block LRU replacement policy in addition to the group-based replacement policy. Figure 2.6 illustrates the two-level LRU lists.

All the cache blocks are linked using a global LRU list. When AdaCache tries to allocate a new cache block in case of a full cache, it first checks the tail of the global LRU list. If the tail cache block has the same size as the new cache block, AdaCache replaces it and promotes both the cache block and its group to the head of the LRU lists. If the size mismatches, AdaCache uses group-based LRU replacement policy to replace a whole group. The use of two-level cache replacement does not incur high lock contention overhead when the cache is accessed in parallel as AdaCache leverages the lockless design of modern high performance storage framework Corporation (2023).

2.3 Evaluation

We evaluate the performance of AdaCache using both the simulation and prototype following the design and implementation described in Section 2.2.

Testbed Setup. The testbed consists of three components which are the client, the disaggregated cache server, and the cloud block storage cluster. The client is-

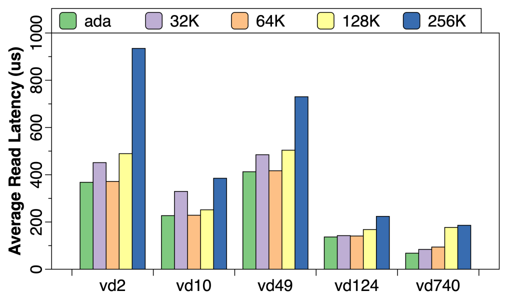
Table 2.1: Specifications Of The Testbed.

Server	**CPU**	**DRAM**	**SSD**	**OS**	**Software**
Client	2x Intel Platinum 8260 96 cores	384GB DDR4	/	Ubuntu 18.04	Replayer / Simulator
Disaggregated Cache Server	2x Intel Platinum 8260 96 cores	384GB DDR4	4x Samsung PM9A3 PCIe Gen3 3.84TB	Ubuntu 20.04	Poseidon OS v0.11
3-node Ceph RBD	2x AMD EPYC 7702	512GB DDR4	4x Samsung PM9A3 PCIe Gen4 3.84TB	Ubuntu 20.04	Ceph Quincy

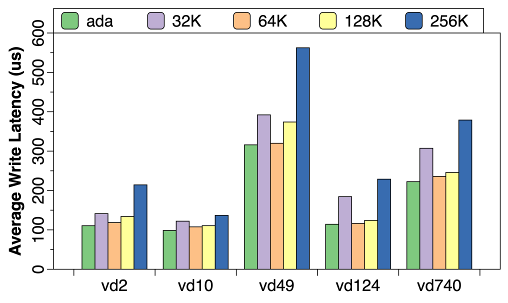
Table 2.2: Trace Segments Statistics.

	alibaba	***msr***	***systor***
#Reads	24.5M	61M	40.7M
#Writes	25.5M	9M	19.3M
Read Traffic GiB	607.3	2416.8	1109.2
Write Traffic GiB	375.9	207.2	271.9

sues the I/O workloads to the disaggregated cache server through NVMeoF RDMA using a 100Gbps NIC. The disaggregated cache server runs AdaCache and provides the cloud block storage with NVMe SSD caching through the network using another 100Gbps NIC. The disaggregated cache server is configured as RAID0 using PoseidonOS consisting of four NVMe SSDs. The cloud block storage is a three-node Ceph

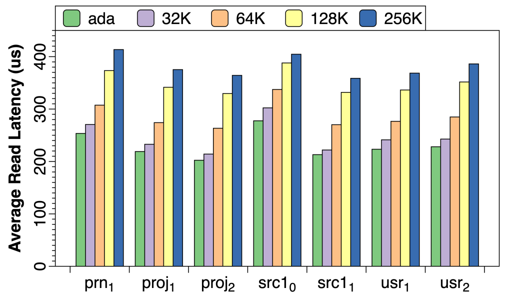


(a) Read Latency

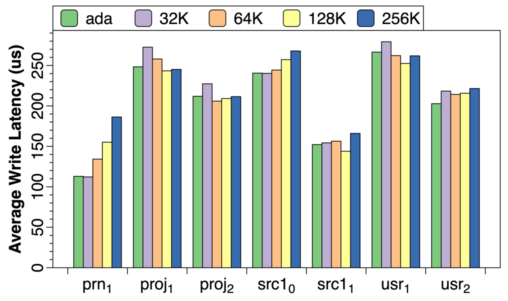


(b) Write Latency

Figure 2.7: I/O Latency for Alibaba Trace Replay



(a) Read Latency



(b) Write Latency

Figure 2.8: I/O Latency for Msr Trace Replay

cluster with Ceph Rados Block Devices (RBDs). The specs for each component are shown in Table 2.1.

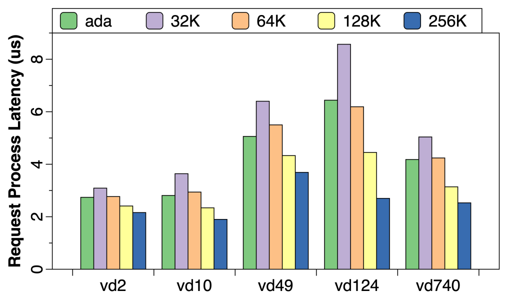
Workloads. We considered the following three real-world block I/O traces to provide a comprehensive evaluation:

- Alibaba block I/O Traces Li *et al.* (2020) (*alibaba*): *alibaba* is collected from an elastic block service cluster of Alibaba Cloud and it contains I/Os from 1000 virtual disks. Among them, we picked 5 virtual disks (vd2, vd10, vd49, vd124, and vd740) that have a large amount of I/O volumes for trace replay. We replayed the first 10 million I/O requests issued to the 5 virtual disks concurrently. Requests to vd2 and vd740 are write-dominant while I/Os to vd10 and vd124 are read-dominant. Vd49 has a similar amount of read and write I/Os.

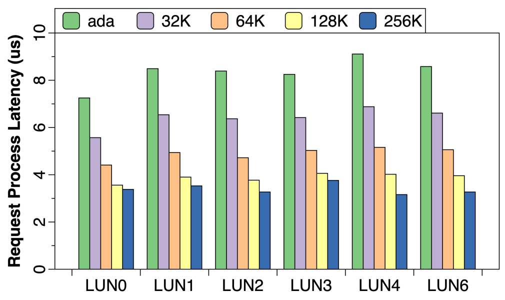
- MSR Cambridge Traces Narayanan *et al.* (2008) (*msr*): *msr* is block-level traces collected from Microsoft Research enterprise data centers and it contains I/Os from 13 servers. Among them, we picked traces from seven drives (prn_1, proj_1, proj_2, src1_0, src1_1, usr_1, and usr_2) that have more than 10 million I/Os. We replayed the first 10 million I/Os issued to the 7 servers concurrently. All the *msr* traces are read-dominant.

- Systor '17 Traces Lee *et al.* (2017) (*systor*): *systor* is collected from an enterprise Virtual Desktop Infrastructure (VDI) which contains I/Os from 300 VMs. All these VMs share 6 storage logical unit numbers (LUN). We replayed the first 10 million I/Os issued to the 6 LUNs concurrently. All the *systor* traces are read-dominant.

For trace segments replay, the cache employs a write-back policy and we can leverage related work Koller *et al.* (2013) to ensure cache consistency. Trace segments are replayed using pread() and pwrite() to issue direct I/Os to different target devices in parallel according to the trace. Each target device consists of 1 TiB Ceph RBD as the backend storage and 10% of each trace's total working set size (WSS) as the cache size. Table 2.2 shows the statistics of the trace segments that we use for replay. We also replay the entire traces using a simulator with the same implementation as the AdaCache prototype to show metrics from the whole trace simulation. In the evaluation, the cache block sizes used by AdaCache are 32KiB, 64KiB, 128KiB, and 256KiB. We compare AdaCache to fix-sized disaggregated caches with these four cache block sizes. Each experiment is repeated three times and we show the average results here. Due to the space limit, we only show evaluation results that are representative of all results.



(a) Alibaba Trace Replay



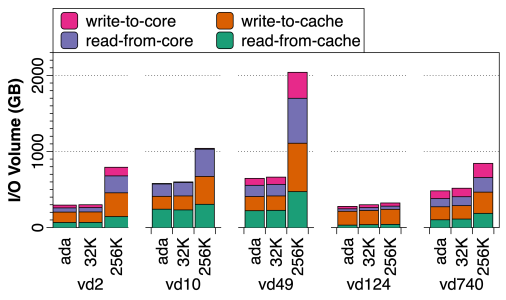
(b) Systor Trace Replay

Figure 2.9: Request Processing Latency

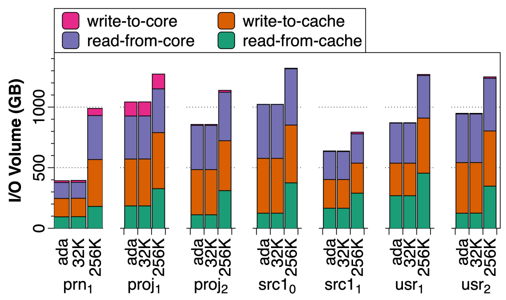
2.3.1 I/O performance

I/O Latency. Figure 2.7 shows the average read and write latency from *alibaba* trace replay. Results reveal that AdaCache has the best overall read and write latency compared to fix-sized caches with different trace segments. For read latency, AdaCache improves it by 19% for trace segment vd740 compared to 32KiB cache and 63% compared to 256KiB cache. For write, AdaCache has an improvement of 9% for trace segment vd10 compared to 64KiB cache and 50% for vd124 compared to 256KiB cache. Figure 2.8 shows the read and write latency from *msr* trace replay. AdaCache also improves the read latency by 7% compared to 32KiB cache for usr_1 and 44% compared to 256KiB cache for proj_2. For write latency, AdaCache can improve it by 9% compared to 32KiB cache for proj_1 and 39% compared to 256KiB cache for prn_1.

Comparing the two traces' latency results from fix-sized caches, *alibaba* mostly has the best read and write performance when using a 64KiB cache. *Msr* has the best read performance when using a 32KiB cache. For write, different cache block sizes perform differently for different trace segments. For example, trace segment prn_1 has the best write performance using 32KiB cache while trace segment proj_1 performs the best using 128KiB cache. This also proves that a fix-sized cache cannot provide



(a) Alibaba Trace Replay

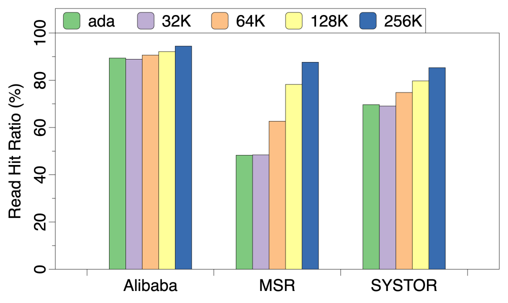


(b) Msr Trace Replay

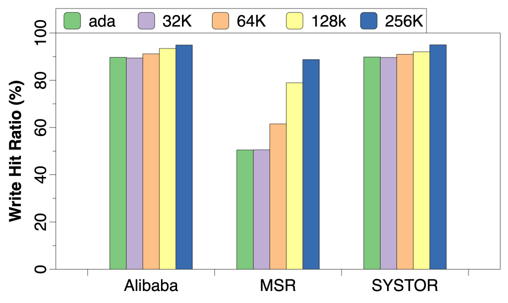
Figure 2.10: I/O volumes

optimal performance for different cloud workloads. Of the two traces, AdaCache outperforms all the fix-sized caches in both read and write. Although AdaCache has similar I/O volumes as 32KiB cache (discussed later in Section 2.3.2), it is achieving better performance because of the adaptiveness of AdaCache which allocates large cache blocks for large requests. These large cache blocks have reduced the number of I/Os and can therefore improve the performance.

Average Request Processing Latency. Figure 2.9 shows the average request processing latency from trace replay. This latency is captured from when an I/O request is received by the cache to when a processed I/O request is sent to the storage devices. It includes the latency for the cache block allocation as described in Section 2.2.1 and 2.2.2. This illustrates the cache block allocation overhead of AdaCache compared to fix-sized caches. Figure 2.9a shows the request processing latency from *alibaba* trace replay. For fix-sized caches, large cache blocks can reduce the number of cache block allocations and therefore reduce the request processing latency. We also observe that AdaCache outperforms 32KiB cache in request processing latency by 25% for vd124. There are two reasons behind this. First, AdaCache uses large cache blocks for large requests which can help reduce the average request processing latency. Second, the high hit ratio for *alibaba* trace segment (around 70% for read and



(a) Read Hit Ratio



(b) Write Hit Ratio

Figure 2.11: Whole Trace simulation results

90% for write) has amortized the extra overhead of adaptive cache block allocation.

Figure 2.9b shows the results from *systor* trace replay. We observe that AdaCache has larger average request process latency than fix-sized caches by 29% compared to 32KiB cache for LUN1. *Systor* trace segment has around 60% read hit ratio and because it is read dominant, the low hit ratio fails to amortize the overhead. Although AdaCache brings extra process overhead from adaptive cache block allocation, the overhead is merely a few microseconds and does not hurt the I/O performance as we have seen previously from the I/O latency results.

2.3.2 I/O Volumes

Figure 2.10 shows the total I/O volumes from *alibaba* trace replay and *msr* trace replay. The I/O volume consists of writes to the cloud block storage (write-to-core), reads from the cloud block storage (read-from-core), writes to the cache (write-to-cache), and reads from the cache (reads-from-cache). Due to the space limit, we only show 32KiB cache and 256KiB cache I/O volumes which have the smallest and the largest amount of I/O volumes, respectively. As discussed in Section 2.1, using large cache blocks may cache unnecessary data and lead to cache pollution and high cache miss penalty. We also observe that AdaCache has a similar amount of I/O volumes as

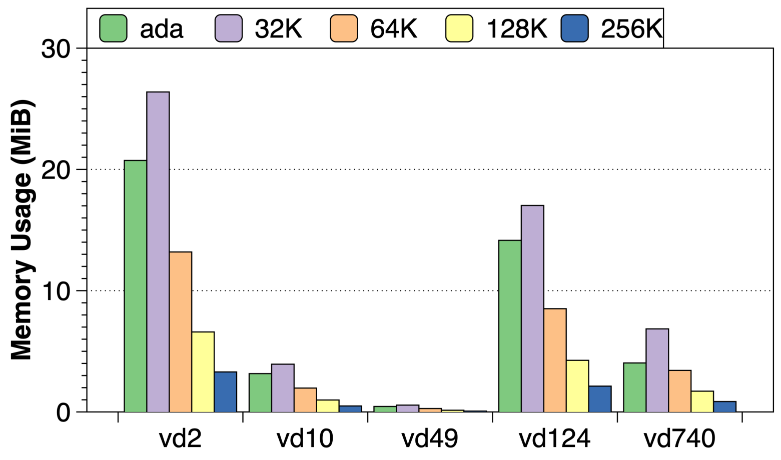
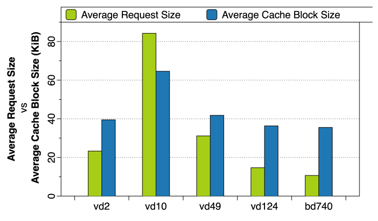
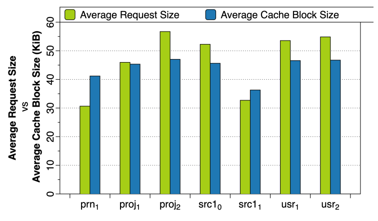


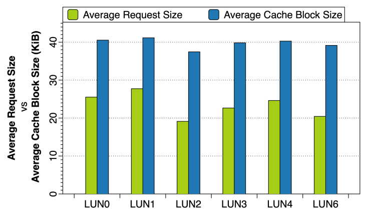
Figure 2.12: Memory Usage For Alibaba Trace Replay



(a) Alibaba Trace Replay



(b) Msr Trace Replay



(c) Systor Trace Replay

Figure 2.13: Average Request Size v.s Average Cache Block Size

the 32KiB cache. This is because although it uses large cache blocks, it caches only necessary data based on the request size. It does not suffer from the large cache miss penalty as the 256KiB cache does. Of the four types of I/O volumes, I/Os to cloud block storage has much larger overhead than I/Os to cache. Compared to 256KiB cache, AdaCache can save 74% I/Os to cloud block storage and 63% I/Os to cache for vd49 from *alibaba*.

2.3.3 Memory Usage

Figure 2.12 compares the average metadata memory usage of AdaCache to fix-sized caches during the trace replay of *alibaba*. For larger cache blocks, the number of cache blocks used is smaller which leads to smaller metadata memory usage. Ada-

Cache saves 41% memory usage compared to 32KiB cache for vd740. From the request size analysis in Section 2.1, *alibaba* mostly consists of small requests. For workloads that have larger requests, AdaCache tends to allocate larger cache blocks and can potentially save more memory.

2.3.4 Hit Ratio

Figure 2.11 shows the read and write hit ratio from the whole trace simulation of *alibaba*, *msr*, and *systor*. As discussed in Section 2.1, larger cache blocks can benefit from the potential spatial locality within the requests and can achieve better hit ratio compared to smaller cache blocks. We also observe similar behavior when replaying the trace segments. For the whole trace simulation, compared to 256KiB cache, AdaCache has up to 39% drop in read hit ratio and up to 38% drop in write hit ratio from *msr*. For trace replay, AdaCache has up to 60% drop in read hit ratio and up to 59% drop in write hit ratio from *msr* compared to 256KiB cache. Although the hit ratio is much lower for AdaCache, it has up to 39% improvement in write performance and 40% improvement in read performance in trace replay compared to 256KiB cache. This shows that compared to the hit ratio and memory usage, I/O volumes play a more significant role in affecting the cache performance.

2.3.5 Effectiveness of Adaptive Cache Block Allocation

Figure 2.13 validates the effectiveness of AdaCache block allocation algorithms. It shows two metrics: the average request size for all the missed requests v.s. the average cache block size that AdaCache allocates when a cache miss occurs during trace replay. The core design idea of AdaCache is to adaptively allocate variable-sized cache blocks based on the request size. The differences between these two metrics tell us how well AdaCache follows the design idea. We observe that AdaCache follows the trend of the request size to allocate cache blocks. With larger requests, the average cache block size also gets larger. For small requests which are mostly seen from *alibaba* and *systor*, the average cache block size of AdaCache is bounded by the smallest cache block size 32KiB. For the best case, AdaCache achieves merely a 1% difference in *msr* trace replay of trace segment proj_1.

2.4 Related Works

Flash Caching. Flash caching Luo *et al.* (2013); Koller *et al.* (2015); Fu *et al.* (2020b,a) has been extensively studied to improve the I/O performance for slow primary storage systems. Solutions have been proposed to solve the capacity and endurance Li *et al.* (2016, 2014a), multi-tenancy Arteaga *et al.* (2016); Waldspurger *et al.* (2015); Zhang *et al.* (2020); Meng *et al.* (2014); Fu *et al.* (2020a) and multi-tier Yadgar *et al.* (2007); Ou *et al.* (2005); Li *et al.* (2005) problems of flash caching. For example, CloudCache Arteaga *et al.* (2016) presents an on-demand cache management solution that meets the performance requirements of each tenant by introducing the Reuse Working Set (RWS) cache demand model. SHARDS Waldspurger *et al.* (2015) is an Miss Ratio Curve (MRC) approximation algorithm that focuses on improving MRC efficiency for online cache reassignment by employing uniform randomized spatial sampling. These orthogonal works can be integrated with AdaCache to improve the cache utilization in a disaggregated cloud environment. Nitro Li *et al.* (2014a) is a host-side flash cache solution that performs deduplication and compression on the data blocks, after which the compressed variable-sized data chunks are stored in the cache as fixed-size Write-Evict Units (WEUs). Nitro uses LRU at the granularity of WEU for cache replacement. Besides the coarse-grained cache replacement policy employed by both Nitro and AdaCache, AdaCache also uses the fine-grained cache block replacement policy to further improve the cache hit ratio by replacing cold cache blocks inside each group as discussed Section 2.2.

Flash Disaggregation. Storage disaggregation cep (2023); Amar *et al.* (2004); Callaghan (2002); ebs (2023); contributors (2021); Amazon Web Services (2021) is common practice in production environment. High-performance flash disaggregation is also an active research area Guz *et al.* (2017); Nanavati *et al.* (2017). Since modern NVMe SSDs are significantly faster than SATA SSDs and hard drives, the software overhead becomes nonnegligible. Guz et al. Guz *et al.* (2017) evaluated the overhead of NVMe SSD storage disaggregation through NVMeoF nvm (2023a) and concluded that the overhead of remote access is negligible compared to local NVMe SSDs. Decibel Nanavati *et al.* (2017) is a solution for flash storage disaggregation at the rack scale, which follows a design of sharing-nothing and provides virtualized storage with low latency by minimizing the software overhead through the integration of network and storage layers.

In-Memory Caching. In-memory caching systems mem (2023); Carlson (2013); Bulkowski and Srinivasan (2013) are widely used in modern software architecture to improve application performance and scalability. For example, Memcached mem (2023) is a lightweight DRAM key-value store that stores key-value pairs of the same value size in slabs of the same slab class. Unlike AdaCache which does global cache block groups replacement, Memcached does time-consuming slab reassignment Byrne *et al.* (2019); Berger *et al.* (2018); Hu *et al.* (2015) across slab classes due to the high concurrency. Data structure optimization Fan *et al.* (2013); Li *et al.* (2014b); Chen *et al.* (2017) to save the metadata memory overhead has also been studied. For example, MemC3 Fan *et al.* (2013) reduces the metadata memory footprint by up to 30% for Memcached by using concurrent Cuckoo hashing and CLOCK LRU-approximation cache replacement. These data structure optimization techniques are complementary to AdaCache and can be leveraged to further reduce the metadata memory overhead.

Adaptive Cache Block Sizes. The performance impact of varying cache block sizes for both memory and storage cache has been thoroughly studied in literature Dubnicki and LeBlanc (1992); Smith (1987); Przybylski (1990); Loh and Hill (2011); Prybylski *et al.* (1988); Agarwal *et al.* (1989). However, few have studied the benefits and drawbacks of a cache system with adaptive cache block sizes. Jeremic et al. Jeremic *et al.* (2021) proposed a two-size cache block allocation mechanism that employs a small-block and a large-block SSD cache. The source address space is divided into segments of contiguous source blocks where either the small or the large cache block size can be used. The assignment relationship between segments and cache block sizes is adjusted in the background based on the measurement of I/O latency. AdaCache differs from the related work including but not limited to 1) AdaCache supports different numbers of cache block sizes to cater to the workloads' characteristics without delay, 2) AdaCache adapts the cache block size based on the request size which is more efficient and effective than monitoring I/O latency of the system. To our best knowledge, AdaCache is the first practical storage cache solution using adaptive cache block sizes.

Chapter 3

IMPROVING LARGE LANGUAGE MODEL INFERENCE ON CPUS THROUGH CXL MEMORY EXPANSION

3.1 Background

In this section, we introduce transformer-based LLM and analyze the rationale of LLM inference on CPUs.

3.1.1 Transformer-based LLM

Transformer-based Vaswani *et al.* (2017) LLMs have revolutionized the field of natural language processing (NLP), due to their superior performance in a wide range of NLP tasks, including language translation, question answering, and text generation. Its ability to capture long-range dependencies in text allows it to understand and generate more coherent and contextually relevant text compared to previous architectures like Recurrent Neural Networks (RNN) and Long Short-Term Memory (LSTM) Vaswani *et al.* (2017). Transformer-based LLMs can be categorized into encoder-only models, decoder-only models, and encoder-decoder models. In this paper, we focus on the decoder-only models as they are the most frequently used models in today's generative artificial intelligence (AI) applications such as AI chatbots cha (2023); bar (2023); cla (2023).

LLM inference requires a lot of memory due to its large model size Chowdhery *et al.* (2023); Brown *et al.* (2020); Zhang *et al.* (2022) and Key-Value (KV) cache. KV-cache is a commonly used technique which stores the computed key and value vectors in memory during inference. These vectors are derived from the input tokens (the processing units in LLMs) to represent the relevance across tokens. KV-cache allows for the reuse of these vectors in subsequent token generations, significantly improving inference efficiency by reducing redundant computations. However, this efficiency comes with a trade-off in terms of memory usage. As the size and complexity of LLMs grow, so does the size of the cache needed to store these pairs. For example, for a 66b model Facebook and Huggingface (2023c) using BF16, the model weights require 126GB of memory and the KV-cache requires 4.5GB for one single inference request Kwon *et al.* (2023). GPUs, even the most advanced, equipped with as much as 80GB of memory, can quickly be overwhelmed by the memory demands.

The inference process of decoder-only LLM is auto-regressive, making it less compute-bound. During inference, LLM usually takes a sequence of input tokens and generates one output token at a time, and the output token is further fed into the model as the new input. This step repeats until an End-Of-Sentence (EOS) sign or the maximum output token length is met. Because the inference process generates only one output token at a time, it requires much fewer computational resources compared to the training process which processes the maximum sequence length in parallel, making GPUs less efficient for latency-sensitive inference tasks Nvidia (2023).

3.1.2 LLM Inference on CPUs

In practice, there is a significant number of deep-learning computations, especially model inference, taking place on the general-purpose CPUs due to their high availability and cost efficiency Liu *et al.* (2019b); Xiang and Kim (2019). For LLM training as well as offline inference with large batch sizes, GPUs are generally preferred since they are efficient for high throughput tasks. In comparison, online inference typically uses a small batch size due to its strict latency requirement in interactive and streaming applications like interactive chatbots, voice assistants, and online recommendation systems. As a result, the efficiency of GPUs, which are optimized for large batch processing is diminished for online inference Liu *et al.* (2019c). The substantial memory capacity of CPUs offers an additional advantage, particularly useful for storing large model weights and KV-cache associated with LLMs. This factor, combined with the low computing requirements for online inference, positions CPUs as a more suitable and cost-effective solution for LLM inference tasks.

Meanwhile, modern CPUs have evolved to improve the performance of tasks, such as LLM inference. These improvements include a substantial increase in computing cores and the development of AI-specific Instruction Set Architecture (ISA) extensions. Processors like the Intel Xeon Scalable series Chernykh *et al.* (2019) and AMD EPYC Velten *et al.* (2022) feature extremely high core counts, with some models having over 64 cores and premium models surpassing 100 cores. AI-specific ISA extensions enhance CPUs' computing efficiency, particularly for tasks like matrix operations. Notable examples include Intel's Advanced Matrix Extensions (AMX) Intel (2023a) for boosting matrix multiplication efficiency.

Existing LLM frameworks for CPU inference MegEngine (2023); OpenMNT (2023a) utilize two methods to improve inference performance: multithreading and data parallelism. Multithreading utilizes multiple CPU cores to do inference parallelly. It is usually implemented through parallel programming libraries like OpenMP Womack *et al.* (2023) to achieve parallelism. Data parallelism replicates inference instances or tasks Wu *et al.* (2015); Jin *et al.* (2018); Xiang and Kim (2019); Kogan (2023) to handle different data inputs simultaneously, which enhances the throughput of inference.

3.1.3 CXL Memory Expansion

Inference tasks, as previously introduced, tend to be less computing-intensive and are more likely to be memory-bound Kwon *et al.* (2023). This is particularly rel-

Table 3.1: Resource Comparison Between CPUs and GPUs.

	Computing Power (BF16)	**Memory Capacity**	**Memory Bandwidth**
NVIDIA A100	312 TFLOPS	80GB	2039GB/s
NVIDIA L40	181 TFLOPS	48GB	864GB/s
Intel SPR	218 TFLOPS	¿1TB	600GB/s

evant when considering the much lower CPU memory bandwidth compared to the GPU. To enhance CPU performance, especially in the context of LLM inference, one can utilize the Compute Express Link (CXL) CXL (2023), a cache-coherent interconnect based on PCIe, which can expand a system's memory capacity and bandwidth, providing expansion to memory devices such as DRAM and persistent memory with memory semantics Yang *et al.* (2022a). CXL-based memory expansion has the potential to address challenges like memory throughput contention for LLM inference, but adding CXL-based memory expansion presents several challenges: first, the added memory has higher memory latency than local memory which can affect the overall performance. Second, integrating CXL memory requires sophisticated memory management strategies to effectively utilize both the local and expanded memory.

Table 3.1 shows the resource comparison between state-of-the-art CPUs and GPUs. The computing power is computed using the BF16 input type. In theory, Intel SPR (dual sockets) with the latest AMX extension can achieve comparable computing performance to high-end GPUs such as NVIDIA A100 and L40. To understand the performance improvement of AMX, we ran matrix multiplication of different sizes on CPUs with and without AMX enabled. Figure **??** shows the results. In general, AMX can improve the matrix multiplication performance by 40%. When the matrix gets larger, the performance is even better with up to 60% improvement. This confirms

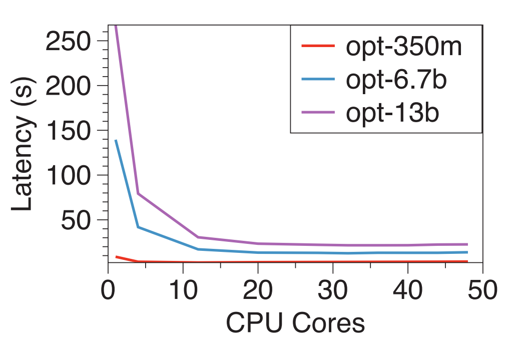


Figure 3.1: LLM Inference Latency Change With More CPU Cores.

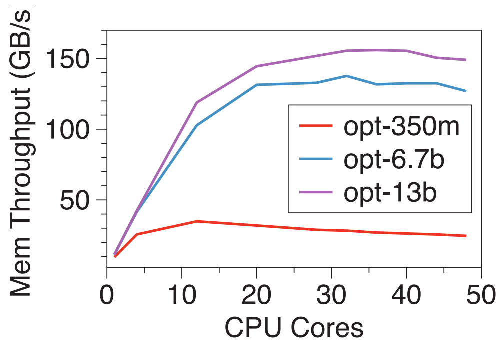


Figure 3.2: Memory Throughput Change With More CPU Cores.

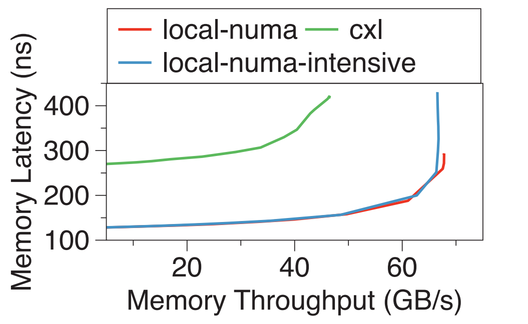


Figure 3.3: The Relationship Between Memory Throughput and Latency.

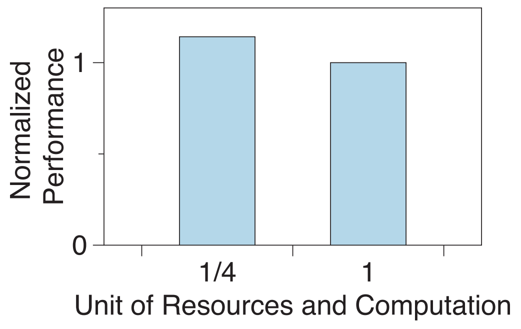


Figure 3.4: Normalized GEMM Performance with Varying Resources and Computation.

the potential of CPU in LLM inference with its much-improved computing power.

3.2 Analysis of LLM Inference on CPUs

In this section, we present the analysis of existing performance bottlenecks for LLM inference on CPUs.

3.2.1 More Computing Resources is not Better

Following the multithreading method used by existing CPU frameworks, we increase the number of CPU cores for LLM inference task. We use a single Intel Sapphire Rapids (SPR) processor with 48 cores and 8 channels of DDR5 4800MHz memory in this set of evaluation. The specifics of our test configuration and software used are elaborated in Sections 3.3.3 and 3.4. Figure 3.1 presents the single batch inference end-to-end latency of different LLMs (opt-350m Facebook and Huggingface (2023b), opt-6.7b Facebook and Huggingface (2023d) and opt-13b facebook (2023)) as we increase the number of CPU cores from 1 to 48, which reveals a sublinear improvement of inference speed across all models.

We observe three phases from Figure 3.1: at the beginning, we see that the end-to-end inference latency (measured in seconds per request) improves linearly as more cores are added. This upward trend quickly tapers off in the second phase. Although inference latency still improves, it does so at a much slower pace with the addition of more cores. Finally, the inference latency reaches a minimum, after which the performance levels off and does not further improve with additional cores. From the observations, we have the insight that the inference performance does not scale with more computing resources, indicating that existing solutions cannot achieve effective parallelism utilizing CPUs. The memory system emerges as the bottleneck that impedes the speedup, as elaborated in the subsequent section.

3.2.2 Memory Bottleneck

The auto-regressive inference process reduces the arithmetic intensity which defines the computation needed per memory access. Table 3.2 shows the arithmetic intensity of different models' Multilayer Perception operation (MLP) for training and inference. With low arithmetic intensity for inference, the CPU cores often complete computation quickly and wait for the next batch of data to be loaded from memory. This makes memory the bottleneck of overall performance. Low arithmetic intensity also makes techniques used by CPUs to hide memory latency like out-of-order execution Mutlu *et al.* (2003) and stream prefetching Jain *et al.* (2023a) less effective because of the frequent memory stalls.

In the initial phase of Figure 3.1, the inference process is compute-bound due to a shortage of CPU cores, making the processor's computational capability the primary bottleneck. In the later stages, where there are a sufficient number of CPU cores, the inference bottleneck transits to memory. To understand how memory latency and throughput change in the second and third phases, we conduct additional experiments focusing on system memory. For this purpose, we use Intel VTune Profiler Intel (2023e) to capture the memory throughput of LLM inference. The results, depicted in Figure 3.2, illustrate how memory throughput changes with an increasing number of computing cores. As shown in the figure, the memory throughput first increases substantially as more cores are added. This is because when computing cores keep increasing, the compute operations are accelerated, thereby reducing the compute time. Consequently, the processor's capability to process data surpasses its ability to access new data, leading to a demand for greater memory throughput.

In the second experiment, we employ the Intel Memory Latency Checker (MLC) Intel (2023c) to issue continuous memory requests to investigate the relationship between memory throughput and latency of local Non-Uniform Memory Access (NUMA) node and CXL memory expander. We also enable Intel's Sub-NUMA Cluster (SNC) Intel (2023f) feature to create four sub-NUMA nodes out of the original local NUMA node so that each sub-NUMA node has the same number of memory channels as the CXL memory expander so they are comparable. Figure 3.3 presents the result where $local - numa$ is the result of one sub-NUMA node and $local - numa - intensive$ is the result of doubling the number of cores used to issue memory requests.

Our observations indicate that the increase in memory throughput leads to an exponential rise in memory latency. This finding addresses a critical aspect of our analysis. In the second phase, as the computing becomes faster, the inference begins to shift from being compute-bound to memory-bound. At the same time, with the reduction in compute time, there's a continuous increase in memory throughput and, correspondingly, in memory latency. This memory throughput contention emerges as the primary bottleneck, impeding further speedup in inference performance. Then in the third phase, the acceleration brought by additional CPU cores is subtle enough to be hidden from the elevated memory latency. As a result, even though the number of computing cores continues to grow, both the memory throughput and latency remain relatively constant. This phenomenon accounts for the observed plateau in Figure 3.1 and 3.2.

This analysis shows that with the ever-increasing computing power, the performance bottleneck for LLM inference is indeed on the memory system. The memory system is struggling to match the pace of the computing performance, and optimizing the memory system is essential to prevent it from impeding the overall performance. Some advanced memory devices such as High Bandwidth Memory (HBM) and Multiplexer Combined Ranks (MCR) ServeTheHome (2023) provide significantly higher memory bandwidth compared to DRAM and can mitigate this problem. However, they are inadequate as a standalone solution for LLM workloads on CPUs, due to either their limited capacity or their lack of maturity.

3.2.3 Parallelism for LLM Inference

As introduced in Section 3.1.2, data parallelism is another common method to improve the throughput for inference tasks on CPUs. However, this approach is more

Table 3.2: Arithmetic Intensity for LLM Training and Inference When Batch Size is 1.

	Inference	Training
350m	0.99878	585.1
6.7b	0.99969	1260.3
13b	0.99975	1365.3

useful for small models compared to large models for a couple of reasons. First, LLMs demand extensive computing resources for a single instance, and replicating the instances could exhaust the computing resources easily. Second, LLM inference already has high memory throughput contention compared to small models, and further replication could potentially result in even higher memory overhead caused by throughput contention. Given the memory bottleneck, the performance improvement for LLMs will not be as significant as it is for smaller models, or it can even decay. However, considering the effectiveness of data parallelism in the inference of small models, dividing a large model into smaller sub-models and processing them in parallel with fewer computing and memory resources is still feasible.

To validate the effectiveness of this concept, we evaluate the performance of General Matrix Multiplication (GEMM) across different scales with varying CPU and memory resources. We designate 48 CPU cores and 4 sub-NUMA nodes as one unit of resource and run the MLP layer of opt-30b Facebook and Huggingface (2023a) as one unit of computation. Figure 3.4 presents the result. Ideally, if the floating point operations' performance was to scale linearly with the resources, scaling both would result in similar performance. However, the computing efficiency diminishes with an increase in scale which can be attributed to several factors: first, the sub-optimal parallelization of GEMM operations; second, the overhead resulting from synchronization in a multithreaded environment; and third, the inability to speed up the sequential part of the operation as dictated by Amdahl's Law. Such a trend, favoring better resource efficiency with a smaller amount of resources, inspires us to adopt tensor parallelism for large models to mitigate the above issues, as elaborated in Section 3.3.2.

3.2.4 Memory Allocation Pattern of LLM Inference

The memory allocation pattern of LLM inference workloads is deterministic. Figure 3.5 shows the number of memory pages allocated during model loading (0-70s) and two consecutive LLM inferences (70-160s). Figure 3.6 shows the memory throughput used at the same time. In the figures, we use opt-30b Facebook and Huggingface (2023a) model with batch size equal to 1, input token length equal to 64, and output token length equal to 128. We also vary input and output token lengths, as well as hyperparameters like models and batch sizes, and observe the same pattern which indicates that over 99% of anonymous pages are allocated during the model loading phase and the allocation pattern is deterministic. Also, L1 and LLC cache miss ratios, as reported by Linux perf Linux (2023), remain consistent across various following inference tasks.

The above observations suggest that LLM inferences with different hyperparameters have consistent memory allocation patterns. Although memory throughput contention happens during inference, it is the model loading phase that determines the memory allocation pattern. Given that memory pages remain static once allocated, a static memory allocation policy can identify the optimal configuration via offline profiling and configure the memory allocation during the model loading phase, eliminating the need for online monitoring and the associated consumption of sys-

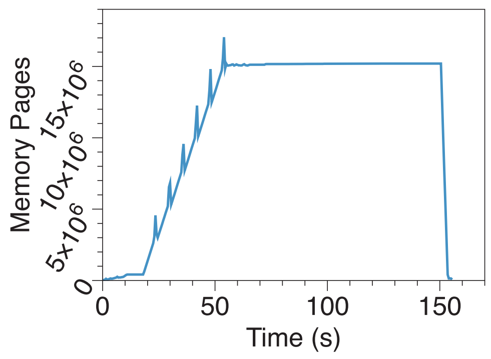


Figure 3.5: Memory Pages Allocated During Inference.

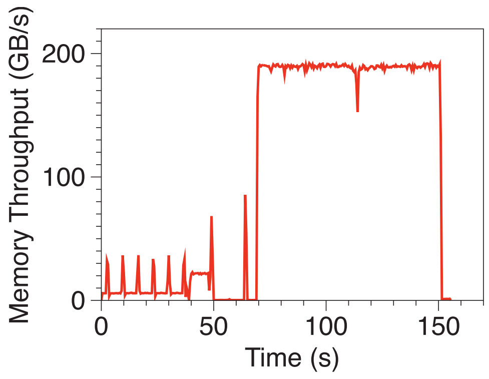


Figure 3.6: Memory throughput During Inference.

tem resources. This motivates our static memory allocation policy introduced in Section 3.3.1.

3.3 Optimization of LLM Inference on CPUs

Based on the analysis in Section 3.2, we can conclude that existing LLM inference solutions fail to achieve efficient parallelism from multithreading or data parallelism. We thus identify two key directions for optimization: reducing memory throughput contention to improve multithreading performance and utilizing tensor parallelism to improve the computing efficiency for large models. In the following sections, we will introduce each optimization method in detail.

3.3.1 Latency-aware Memory Aggregation

To mitigate memory throughput contention, our approach involves enhancing the memory resources by incorporating additional memory devices such as CXL memory expansion, persistent memory, and HBM. The aggregated memories are allocated and accessed as a whole and data does not migrate across memories. This strategy allows for a portion of the memory pages allocated to the added memory, reducing the amount of memory requests directed to the local memory. Consequently, the reduced memory throughput contention can result in a reduction in memory latency. In this paper, we use fast and slow memory to distinguish local memory and additional memory. With the help of the additional bandwidth from the slow memory, the latency of the fast memory can be improved, resulting an overall improvement in both latency and throughput. Note that the proposed aggregated memory system differs from a typical tiered memory system Maruf *et al.* (2023); num (2023); hot (2023). In a tiered memory system, data moves frequently between tiers based on the hotness of data. This behavior is harmful for latency-sensitive tasks as the data movement occupies additional memory bandwidth and increases memory latency Sun *et al.* (2023).

A critical challenge in this method is ensuring that the memory latencies of both the fast and slow memory are close to identical. High memory latency in either type of memory device can significantly hurt the performance, as computing may stall waiting on the slowest memory access. This necessitates that the latency gap between fast and slow memory must be small. However, as shown in Figure 3.3, CXL-expanded memory exhibits significantly higher memory latency compared to the local NUMA for the same amount of memory requests. Additionally, local NUMA experiences higher memory latency with more intensive workloads. This implies that we can tune the workload intensity which can be controlled by the percentage of memory pages allocated on the memory devices to adjust the memory latency. Simple memory allocation policies that are based on memory bandwidth or capacity of fast and slow memories, i.e., allocating memory pages based on the memory bandwidth or capacity ratio of fast and slow memories, fail to bring optimal performance as they neglect the impact of memory latency and different LLM inference hyperparameters. This is further confirmed by the evaluation results in Section 4.2.

To address the challenge, We propose a two-stage inference framework that is inspired by the observed static memory allocation pattern (Section 3.2.4) of LLM inference workloads: a warm-up stage followed by a serving stage. During the warm-up stage, the framework profiles the entire system using inference workloads to quickly determine the optimal percentages of memory pages to be distributed across the aggregated memories. Once the optimal memory configuration is established, it remains static and the serving stage starts, where actual inference tasks are performed with memory pages allocated according to the percentages determined in the warm-up stage. Note that different LLM hyperparameters and system configurations can affect the memory configuration while different input and output token length do not affect the configuration as shown later in Section 4.2.

A dynamic page allocation policy Sun *et al.* (2023) which monitors fast memory performance counters to make page allocation decisions does not work well for LLM inference workloads. First, this policy predicts application performance using a linear model trained on the performance counters from both the model-loading and inference phases which is inaccurate in predicting the model-loading phase (Figure 3.3). We simulate the policy with a 30b model and it only allocates 31% of memory pages locally which is sub-optimal as shown in Section 4.2. On the other hand, training the linear model with performance counters from the model-loading phase leads to allocation dependent on models only, disregarding other hyperparameters like batch size and system configuration which is also sub-optimal as shown in Section 4.2.

Moreover, the policy that utilizes only fast memory performance counters fails to monitor slow memory's performance. Given the performance difference between local and added memory, distributing memory pages to added memory can enhance local memory performance but deteriorate the performance of slow memory, ultimately leading to a decline in the system's overall performance. Finally, continuous moni-

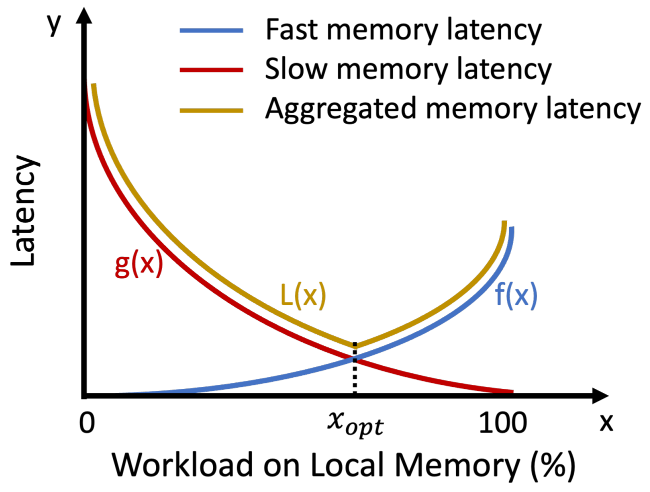


Figure 3.7: Fast and Slow Memory Latency Based on Different Workloads on Fast Memory

toring of memory performance requires extra system resources, which can hinder the performance of LLM inference. Consequently, this method fails to optimally utilize additional memory resources to alleviate memory throughput contention.

During the warm-up stage of our proposed framework, we need to find out the percentages of memory pages to allocate to the fast and slow memories to achieve the best performance. We formulate the issue into an optimization problem. Considering all the memory pages required by the LLM inference workload, we distribute $x\%$ to the fast memory and the remaining $(100 - x)\%$ to the slow memory. Based on Figure 3.3, we can construct Figure 3.7, where the x–axis represents the percentage of memory pages (x) allocated to the fast memory and y–axis represents the memory latency of the fast and slow memory when x percentage of memory pages are allocated to the fast memory. As x increases, fast memory latency rises, while that of the slow memory falls.

By interleaving the memory pages across the fast and slow memories, the aggregated memory latency ($L(x)$) is modeled by the maximum memory latency of fast and slow memory, which can be expressed as:

$$L(x) = max(f(x), g(x))$$

Here, $f(x)$ and $g(x)$ are the memory latency functions for fast and slow memory, respectively as shown in Figure 3.7. The objective is to identify the value of x that minimizes $L(x)$. Based on Figure 3.7, $L(x)$ can be visualized as the top portion of $f(x)$ and $g(x)$ in yellow. As per this graphical representation, $L(x)$ first decreases monotonically, after reaching a minimum point, it increases monotonically. It exhibits a single minimum value x_{opt} at the intersection point of $f(x)$ and $g(x)$, indicating the optimal memory configuration for the inference. This further substantiates the necessity that the latency gap between fast and slow memory needs to be small so that they have an intersection point in terms of memory latency.

Constructing an exact function for $L(x)$ is challenging, so a straightforward method to identify the best configuration would be to employ a linear search, iterating through all possible allocations to determine which one offers the best inference performance. Nevertheless, the theoretical shape of $L(x)$ allows for the use of binary search because, at any point in the sequence, we can determine which direction to move to find the minimum, which brings down the search time compared to linear search.

We thus propose the binary search algorithm as follows: we first define the initial low and high bounds for x and then calculate the midpoint m of the current range (low, high). By using $m : 100 - m$ memory configuration, we measure the inference performance $p(m)$. We then slightly move to the right of m at a *step* size and compare $p(m)$ and $p(m + step)$. If $p(m)$ is less than $p(m + step)$, then the high bound is updated to m. Otherwise, the low bound is updated to $m + step$. Repeating the iterative process until the low and high bounds meet, which indicates that the optimal point is the current low bound x. We can then distribute the percentage of memory pages as $x : 100 - x$ between fast and slow memories. If we have multiple fast memory devices and/or multiple slow memory devices, the workload is evenly distributed across the fast/slow memory devices. Algorithm 3 presents the pseudo-code of the whole process.

Algorithm 3 Search Optimal Memory Allocation

```
1:  Remarks:
        $p(x)$: get inference performance with x% workload on fast
        memory
        $l$: the low boundary
        $h$: the high boundary
        $step$: the granularity of workload adjustment
2:  Output:
        the percentage of workloads on fast memory
3:  $l \leftarrow 0$
4:  $h \leftarrow 100$
5:  while $l \neq h$ do
6:      $m \leftarrow (l + h)/2$
7:      if $p(m) > p(m + step)$ then:
8:          $l \leftarrow m + step$
9:      else:
10:         $h \leftarrow m$
11:     end if
12:     return $l$
13: end while
```

3.3.2 NUMA-aligned Tensor Parallelism

As discussed in Section 3.2, tensor parallelism is preferred for large models. Tensor parallelism has been commonly used in GPU-based LLM frameworks vll (2023); OpenMNT (2023b); microsoft (2023); HuggingFace (2023) to distribute model weights, activations, and KV-cache that are too large for the memory capacity of a single GPU across multiple GPUs Huggingface (2023); AWS (2023). This technique itself is not new; however, it has been overlooked by existing LLM inference frameworks tailored for CPU backends OpenMNT (2023a); MegEngine (2023), partly due to the ample memory capacity of CPUs. Contrary to GPUs, the advantage of applying tensor parallelism to CPUs is not derived from employing extra resources. Rather, it primarily originates from segmenting large models to leverage a smaller number of computing cores and memory resources for each sub-model, thereby enhancing computing efficiency.

Tensor parallelism on CPUs involves two stages: first, distributing different components of the model such as weights and activations across CPU cores. Each group of CPU cores is responsible for a sub-model's operations. For example, a large matrix multiplication within a layer is divided into smaller matrices, with each segment being processed by a subset of CPU cores. Second, performing an all-reduce operation to enable the exchange and combination of results from all the CPU cores. The tradeoff between the improved computing time and the additional communication overhead is critical to the overall performance improvement.

During computing, a significant overhead comes from the slow cross-socket memory access caused by the inter-processor interconnect Intel (2023d); AMD (2023). Moreover, modern CPUs divide NUMA nodes into smaller, more efficient clusters to enhance memory performance, causing the memory access latency across the sub-

NUMA nodes to be increased. To reduce this overhead, we minimize cross-NUMA memory access by aligning CPU cores with the closest sub-NUMA node(s) to form a group. By forcing each sub-model to only use the resources from this group, it eliminates all the cross-NUMA and cross-socket memory accesses during computing.

Following the above optimization, tensor parallelization is performed in four steps: first, the CPU cores and the sub-NUMA nodes are divided based on the *parallel degree*, i.e., the number of sub-models into which the model is split for processing. Parallel degree is determined by the available CPU cores and the model size. Given sufficient CPU cores, larger models can benefit more from a higher parallel degree. Second, we bind CPU cores and sub-NUMA nodes into groups based on data locality. Third, we split the large model into sub-models and each sub-model is allocated to a group. Fourth, activations from sub-models are combined and aggregated through all-reduce operation.

3.3.3 Implementation

We developed a CPU-based inference framework, incorporating all elements described in Section 3.3. To implement memory aggregation, we modified the page allocation mechanism within the Linux kernel (v6.3.2) to get more precise control over resource allocation. This includes three main modifications. First, we support the *m*:*n*:*k*... interleave policy to distribute memory pages across different NUMA nodes and memory devices following the *m*:*n*:*k*... ratio. Second, we support CPU cores and sub-NUMA nodes binding where the specified CPU cores can only allocate memory pages to the bound sub-NUMA nodes for tensor parallelism. Third, we introduce a /proc entry to the system to simplify the configuration of these settings. In the userspace, we implement binary search following Algorithm 3. To expedite the search, we assume local memory is fast memory and initiate the lower bound at

50%. We also set the step size to 2% to converge faster to the minimum latency. Our framework is integrated with LLM inference APIs (v4.31.0) from the transformers package Face (2023) and employs the Intel Extension for PyTorch (v2.1.0) Intel (2023b) to accelerate LLM inference on CPUs. NUMA-aligned tensor parallelism is implemented using the inference APIs from the DeepSpeed library (v0.12.1) Microsoft (2023) where we add sub-NUMA alignment support to each sub-model.

is that both the local memory tier and remote memory tier's memory latency should be at a similar minimum level. A slow memory latency, no matter in local or remote memory tiers, can largely hurt the performance as computing would stall waiting for the slowest memory request.

Reducing the memory latency can bring two benefits: first, the speedup will approach that of the theoretical speedup. Second, since the amount of memory requests for an inference workload is constant, reducing the memory latency can inversely improve the memory bandwidth to improve inference performance [1].

Improving computing efficiency also has two benefits: first, using the same amount of computing cores but can get better speedup. Second, LLM inference can use fewer cores to achieve the same computing efficiency and this can in turn reduce memory latency and further improve speedup.

From the analysis, two key conclusions emerge regarding the inference of LLMs. Firstly, when adding additional memory resources, it becomes crucial to pinpoint an optimal allocation point that harmoniously balances both computing and memory resources for efficiency. Secondly, LLM inference tends to achieve better computational speedup when employing a smaller amount of computing resources. This suggests that a smaller allocation of resources can be more efficient.

[1]Note that in Figure 3.3, MLC generates continuous memory requests, so the memory latency and memory bandwidth is positively correlated.

To address the challenge of memory latency, which is a significant bottleneck, the addition of extra memory is recommended. Alongside this, both computing and memory resources should be allocated in a manner that minimizes memory latency and maximizes memory bandwidth. In aligning with Amdahl's Law, a common strategy to maximize computing speedup is the "divide and conquer" approach. This involves dividing large models into smaller, more manageable units, allowing them to leverage the fullest extent of available computational speedup. The following section will delve into these two improvement strategies in greater detail.

as piecewise functions each consisting of two linear functions (Figure 3.7):

$$f(x) = \begin{cases} a_1 x + b_1, & \text{if } x \leq x_1 \\ a_2 x + b_2, & \text{if } x > x_1 \end{cases} \tag{3.1}$$

$$g(x) = \begin{cases} -c_1 x + d_1, & \text{if } x \leq x_2 \\ -c_2 x + d_2, & \text{if } x > x_2 \end{cases} \tag{3.2}$$

where a_1, a_2, c_1, c_2 are positive, with $c_1 > c_2$, $a_1 < a_2$ and $x_2 < x_1$.

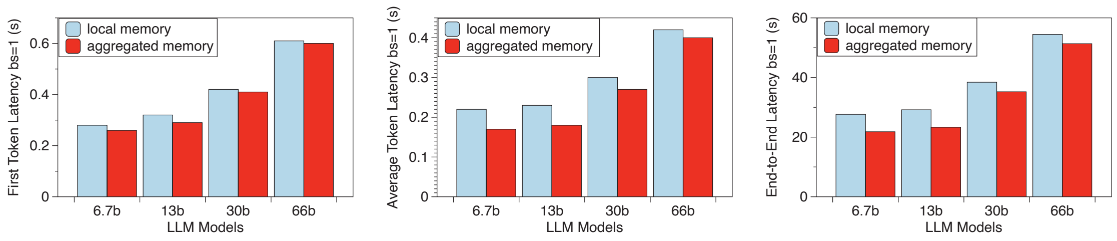
3.4 Evaluation

3.4.1 Methodology

Our experiment server, as shown in Table 3.3, is equipped with dual Intel Xeon 8456C CPUs (SPR) with AMX, 2TB of 4800MHz DDR5 memory, 3.84 TB of SSDs, and a pair of A1000 CXL memory expanders from Astera Labs with a total of 512GB memory. During the evaluation, hyperthreading is disabled to avoid the GEMM operations' contention among logical cores. To achieve more precise control over resource allocation, we enable the sub-NUMA cluster (SNC) feature, which results in the creation of four sub-NUMA nodes within each NUMA node.

Table 3.3: Hardware Specifics of the Experimenting Server.

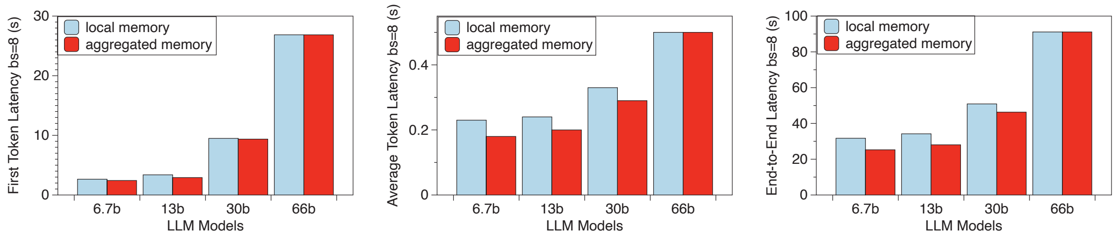
CPU	Dual socket Intel Xeon 8456C CPU Each socket has 48 Computing cores
Memory	2 * 8 channels Each channel has 128GB 4800MHz DDR5
Storage	2 * 1.92TB SSDs
CXL	2 * A1000 Gen5 x16 ASIC memory expanders Each CXL has 2 channels Each channel has 128GB 4800MHz DDR5



(a) First Token Latency (b) Average Token Latency (c) End-to-end Latency

Figure 3.8: LLM Inference Performance Improvement With Aggregated Memories When Batch Size is 1.

Since transformer-based LLMs follow similar architecture OpenAI (2023); Touvron *et al.* (2023); Chowdhery *et al.* (2023), we choose to use the open-sourced models from the Meta Open Pretrained Transformers (opt) suite Zhang *et al.* (2022). We evaluate models of different sizes including opt-6.7b Facebook and Huggingface (2023d), opt-13b facebook (2023), opt-30b Facebook and Huggingface (2023a), and opt-66b Facebook and Huggingface (2023c). During the inference, the input token length is set to 64 and the output token length is set to 128 with multinomial sampling. The data format is set to BF16. We use two batch sizes 1 and 8 to study



(a) First Token Latency (b) Average Token Latency (c) End-to-end Latency

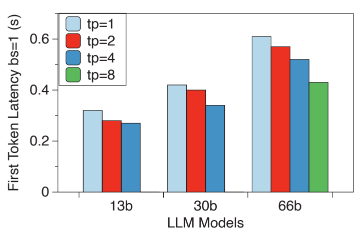
Figure 3.9: LLM Inference Performance Improvement With Aggregated Memories When Batch Size is 8.

the impact of different batch sizes. Our initial analysis indicates that for both batch sizes, the performance bottleneck identified in Section 3.2 persists, and neither batch size shifts the inference workload to being compute-bound. Each inference test is conducted 40 times, with the first 20 iterations serving as warm-up.

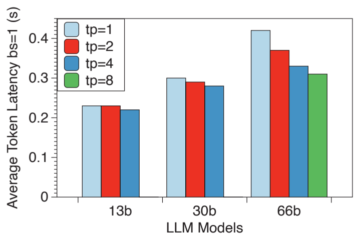
To illustrate inference latency, we use the below three metrics to show the responsiveness and efficiency of an application: *first token latency*, *average token latency*, and *end-to-end latency*. First token latency dictates how quickly the users begin to observe the generated output. In real-time interactive applications, it is crucial to have a low latency for the initial token. Average token latency is the average time taken to generate each subsequent token after the first one. It indicates the overall speed and fluidity of the inference task. End-to-end latency is the total time taken from the moment an input is given to the model until the completion of the entire output sequence. It is a comprehensive measure of the model's overall performance and efficiency.

3.4.2 Memory Aggregation

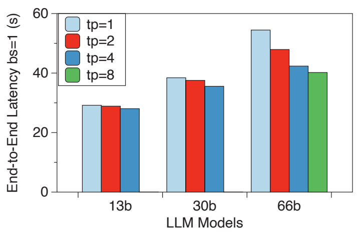
We tailor computing and memory resources to each model's size. For the 6.7b model, we use 12 CPU cores, one SNC sub-NUMA node for fast memory, and one



(a) First Token Latency

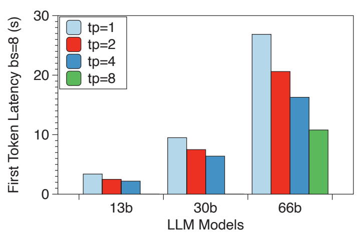


(b) Average Token Latency

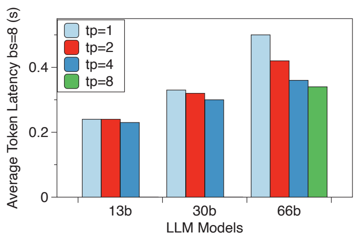


(c) End-to-end Latency

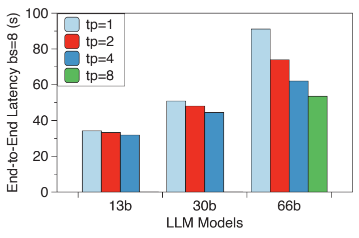
Figure 3.10: LLM Inference Performance Improvement With Tensor Parallelism With Batch Size 1. (Tp means the parallel degree.)



(a) First Token Latency



(b) Average Token Latency



(c) End-to-end Latency

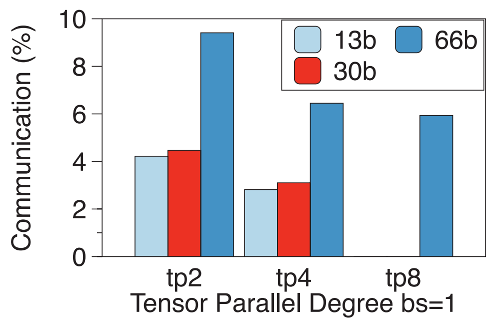
Figure 3.11: LLM Inference Performance Improvement With Tensor Parallelism With Batch Size 8. (Tp means the parallel degree.)

CXL memory expander for slow memory. The 13b model gets doubled resources: 24 cores, two SNC sub-NUMA nodes, and two CXL memory expanders. The 30b model scales up to 48 cores and four sub-NUMA nodes but keeps two CXL memory expanders. All these models operate within a single socket. The 66b model, spanning two sockets, employs 96 CPU cores, eight sub-NUMA nodes, and two CXL memory expanders, with one CXL expander per socket.

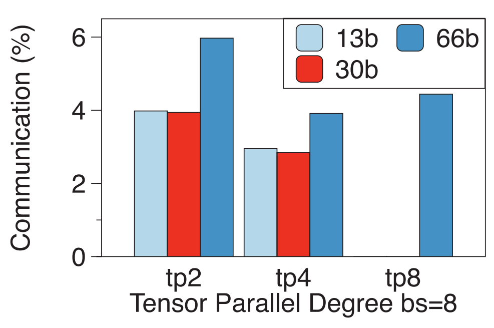
In this set of experiments, we compare LLM inference performance using aggregated memories to that using local memory only. Figures 3.8 and 3.9 illustrate the improvement in inference performance using aggregated memories for batch sizes of 1 and 8, respectively. The result indicates a notable performance boost with aggregated memories. For example, the 6.7b model, with a batch size of 1, achieves a 9% improvement in first token latency, 24% in average token latency, and 22% in end-to-end latency; the 13b model shows improvements of 16%, 19%, and 18% in the first token, average token, and end-to-end latency, respectively, with a batch size of 8.

Due to the limited quantity of CXL memory expanders in our testbed, for larger models, the performance improvement is less significant. This is because large models have more memory resources in the fast memory, necessitating more memory resources in the slow memory to achieve comparable memory latency. With a lack of memory resources in the slow memory, most memory pages are allocated locally, leading to poorer performance improvement than for smaller models. For example, for the 30b model at batch size 8, although it uses the same CXL memory resources as the 13b model, the performance gains are less pronounced: first token latency improves by 3%, average token latency improves by 12%, and end-to-end latency improves by 9%. For the 66b model, the improvement is even less significant, with 4%, 7%, and 7% respectively for first token latency, average token latency, and end-to-end latency at batch size 1. Given more memory resources in the slow memory, we expect to see a better performance improvement.

Of all the performance metrics, first token latency has the least significant performance improvement. For example, for the 6.7b model at batch size 1, the first token latency only improves by 8% while the other two metrics improve by more than 22%. This is because generating the first token involves processing the entire input prompt which has higher arithmetic intensity than generating subsequent tokens, resulting in the first token generation being more computing-intensive. When we add memory resources, it helps more in memory-bound workloads, such as generating subsequent tokens. The end-to-end latency is a combination of both first token

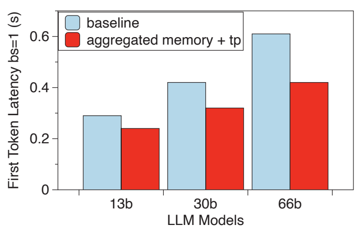


(a) Batch Size is 1.

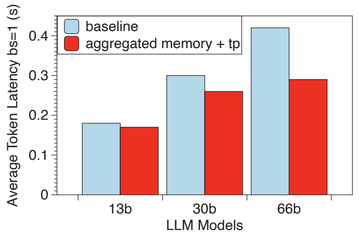


(b) Batch Size is 8.

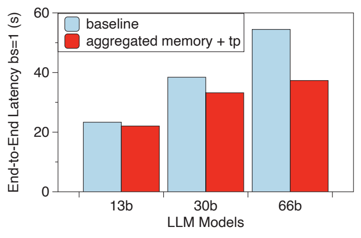
Figure 3.12: The Percentage of Communication Latency Relative to End-to-end Latency During Inference.



(a) First Token Latency



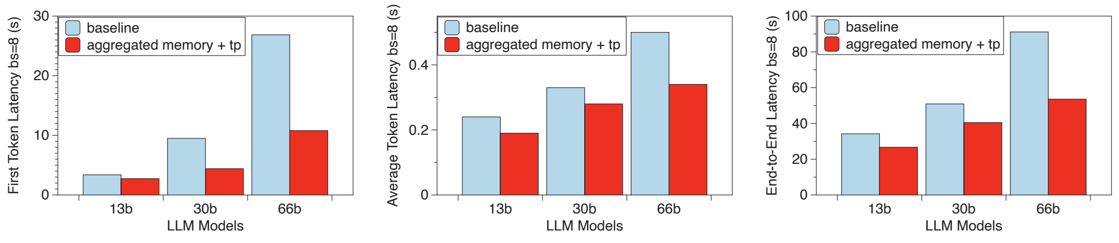
(b) Average Token Latency



(c) End-to-end Latency

Figure 3.13: LLM Inference Performance Improvement With Both Optimizations When Batch Size is 1. (Tp is tensor parallelism.)

latency and average token latency, so it has an improvement in the middle. Similarly, when comparing different batch sizes' performance results, a larger batch size is more computing-intensive and affects first token latency more. For example, when batch size is increased from 1 to 8, the average token latency only increases slightly within 5%. However, the first token latency increases substantially by 44X for the 66b model.



(a) First Token Latency (b) Average Token Latency (c) End-to-end Latency

Figure 3.14: LLM Inference Performance Improvement With Both Optimizations When Batch Size is 8. (Tp is tensor parallelism.)

3.4.3 Tensor Parallelization

Next, we evaluate the NUMA-aligned tensor parallelization method for CPUs. We use various parallel degrees to evaluate the performance improvement compared to no model splitting where all resources are used for the monolithic model (parallel degree is 1), representing existing LLM CPU-based inference frameworks OpenMNT (2023a); MegEngine (2023). For all the models, we still follow the same CPU and memory configurations described in Section 3.4.2, but we use only the fast memory. Additionally, we evaluate tensor parallelism only on the three larger models since 6.7b model uses only 12 cores and already shows good computing efficiency. For 13b, 30b, and 66b models, we experiment with parallel degrees of 1, 2, and 4. For the 66b model, we also experiment with the parallel degree of 8. Figure 3.10 and 3.11 depict the performance improvement with tensor parallelism when batch size is 1 and 8.

Based on the results, we can see that tensor parallelism can improve inference performance. For the 66b model, first token latency improves by 31%, average token latency improves by 27%, and end-to-end latency improves by 27% when the parallel degree is 8 and batch size is 1. The improvement is achieved by the improved computing efficiency and reduced cross-NUMA memory accesses. When the batch size is

8, the performance improvement is better, with first token latency improving by 61%, average token latency improving by 33%, and end-to-end latency improving by 43%. This is because when batch size increases, the inference gets more computing-intensive and can benefit more from tensor parallelism.

For smaller models, the improvement is less significant. For example, for the 13b model, first token latency improves by 17%, average token latency improves by 6%, and end-to-end latency improves by 5% when the parallel degree is 4 and batch size is 1. In contrast to larger models, smaller models utilize fewer CPU cores and have better computing efficiency. Consequently, they do not see as significant a performance enhancement from tensor parallelism as larger models do. Of all the performance metrics, first token latency has the best improvement. For example, for the 30b model, first token latency can be improved by 21% when the parallel degree is 4 and batch size is 1, while average token latency and end-to-end latency improvements are both 8%. As discussed in Section 3.4.2, generating the first token is more computing-intensive, and by using tensor parallelism, it can achieve better speedup.

Across all the models, the inference performance improves as the parallel degree increases. For example, when the parallel degree increases from 2 to 8 for the 66b model when batch size is 8, first token latency improves by 21%, while average token latency and end-to-end latency improve by 17%. Increasing the parallel degree splits the model into smaller sub-models and each sub-model can benefit more from the improved computing efficiency thus achieving a better performance boost. Based on the above observations, we can expect a better performance improvement with higher degrees of tensor parallelism for larger models.

We also compare the computing part (without considering all-reduce) of NUMA-aligned tensor parallelism with simple tensor parallelism that depends on the Linux kernel's memory allocation strategy to allocate memory pages based on data locality. For the 30b model, adopting NUMA-aligned tensor parallelism results in a 4% improvement when the parallelism degree is set to 4 and the batch size is 1. This modest increase is due to a couple of factors: first, LLM inference is the only running workload, thus ensuring effective Linux memory page allocation based on data locality. Second, the 30b model involves cross-sub-NUMA memory accesses with only small overhead, as they are located within the same socket. Nonetheless, with our NUMA-aligned tensor parallelism, we successfully eliminate all cross-NUMA overhead during the computing stage.

In the next evaluation, we show the communication overhead during the all-reduce operation using the percentage of communication latency relative to end-to-end latency. From Figure 3.12, we can see that the communication overhead remains low. For 13b and 30b, the communication overhead is up to 4.2%. For 66b, the overhead is a little higher, capped at 9.4% when the batch size is 1. This is because 66b uses NUMA nodes from both sockets. Since all-reduce requires the exchange and combination of results from all the CPU cores, resulting in cross-socket memory accesses, which have higher memory latency than that within the same socket. We also observe that with a larger batch size, the communication overhead is reduced. For example, the communication overhead reduces to 5.9% for 66b when the batch size is 8. This is due to the increased computation time caused by the large batch size, coupled with a modest increase in the amount of data communicated, which reduces the percentage of communication overhead.

3.4.4 Putting Everything Together

Based on our previous evaluation results, we show that memory aggregation and NUMA-aligned tensor parallelization can help reduce memory contention and improve

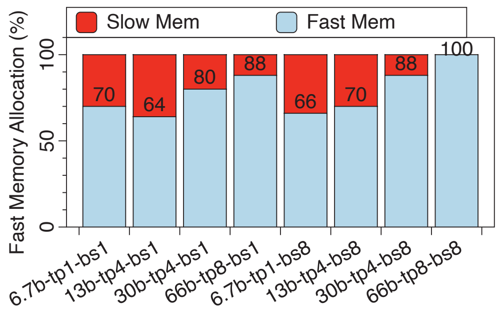


Figure 3.15: The Percentage of Workload Allocated to Fast Memory. (The x-axis is labeled by model-parallel degree-batch size, e.g., *6.7b-tp1-bs1* is 6.7b model inference with tensor parallel degree 1 and batch size 1.)

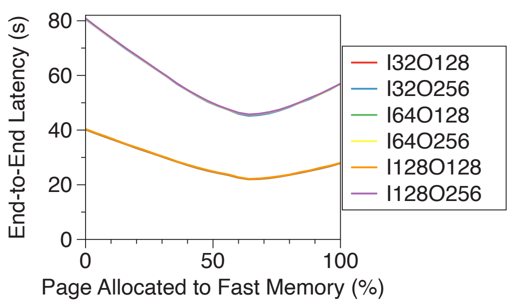


Figure 3.16: Impact of Different Inputs and Outputs. (Data series are labeled by input token length and output token length, e.g., *I32O128* means input token length 32 and output token length 128.)

computing efficiency. We now combine these two optimization methods and see how the performance gets improved. Since the 6.7b model does not use tensor parallelism, we show 13b, 30b, and 66b model results here. Figure 3.13 and 3.14 illustrate the performance results. Of different parallel degrees, we only show the one that achieves the best performance improvement in the figures.

When batch size is 1, for the 13b model, first token latency improves by 26%, average token latency improves by 27% and end-to-end latency improves by 25%. For the 66b model, all three latency metrics improve by 32%. When the batch size is 8, 30b model has 55% improvement in first token latency, 17% improvement in average token latency, and 21% improvement in end-to-end latency. For 66b model, the optimal memory configuration uses only the fast memory (explained in Section 3.4.5) thus the performance improvement is the same as that with tensor parallelization only, with first token latency improving by 61%, average token latency improving by 33%, and end-to-end latency improving by 43%.

3.4.5 Static Memory Allocation

Figure 3.15 shows the optimal memory configuration for different inferences using the percentage of workload allocated to the fast memory. On average, it takes four search cycles and tens of minutes to determine the best configuration. However, this warm-up stage time becomes less significant considering the system's serving time, which spans weeks or even months. From the figure, we can see that the memory allocation is different across various models, batch sizes, and parallel degrees, illustrating that a different inference hyperparameter requires separate profiling. Note that for a batch size of 8, the 66b model achieves optimal performance when most or all of the workloads are allocated locally. This is because the 66b model utilizes eight sub-NUMA nodes for fast memory, which significantly outperforms the two CXL memory expanders. We expect the system to allocate more pages to slow memory with more CXL memory expanders.

Other allocation strategies, such as those based on bandwidth or capacity ratio of fast and slow memories, i.e., allocating memory pages based on the memory bandwidth or capacity ratio of fast and slow memories, fail to deliver peak performance. In our test setup, the memory bandwidth ratio between a sub-NUMA node and a CXL memory expander is measured to be 53:47, while the memory capacity ratio is 50:50. As depicted in Figure 3.15, the ideal allocation percentage deviates from these ratios. For instance, for the 13b model inference under a bandwidth-oriented allocation, 53% of memory is allocated to fast memory, leading to a performance reduction of 10%, and under a capacity-oriented approach, it is 50%, the performance decrease can be as much as 14%. This is because basing the memory configuration solely on memory hardware specifications, without considering the memory latency and any LLM hyperparameters, does not yield the optimal allocation.

Additionally, we show inference results with various combinations of input and output tokens using 13b model as an example with batch size equal to 1. The results are found to overlap, and as depicted in Figure 3.16, there are merely two distinguishable lines. We confirm that the optimal configuration remains constant across different inputs and outputs. Also, as the percentage of memory pages allocated to fast memory varies, we notice that the performance changes adhere to the pattern as we proposed in Figure 3.7 of Section 3.3. This consistency reinforces the efficacy of our solution in determining the optimal memory allocation.

3.5 Related Works

Optimizing deep learning inference on CPUs. Improving the deep learning inference performance on CPUs can be done through operator fusion Ning *et al.* (2020), model quantization Shen *et al.* (2023a), operation optimization Dice and Kogan (2021), self-attention mechanism optimization Wang *et al.* (2020); Dao *et al.* (2022), sparsity Shen *et al.* (2023b); SJTU-IPADS (2023) and a mixture of the above Jiang *et al.* (2023); Liu *et al.* (2019d). Shen et al. Shen *et al.* (2023a) used INT4 quantization and highly-optimized kernel to boost LLM inference performance on modern Intel CPUs. Jiang et al. Jiang *et al.* (2023) leveraged data parallelism to improve the usage of CPU cores. Similarly, Kogan Kogan (2023) used data parallelism with fine-grained resource control for each ML inference instance to improve performance. As we discussed in Section 3.2, data parallelism is not practical for LLMs due to its extensive demand for computing resources and its overhead in memory latency.

Jain et al. Jain *et al.* (2023b) utilized software prefetching to reduce cache misses caused by irregular accesses to the embedding table and hyperthreading to reduce CPU stalls caused by memory access for CPU-based recommendation model inference. Given that LLM inference also uses an embedding table, software prefetching is complementary to our solution. However, the impact is limited as the embedding table constitutes only a small fraction of the overall model size, e.g., 3% for GPT3-6.7B Brown *et al.* (2020). Moreover, hyperthreading is believed to be harmful to GEMM-dominated LLM workloads as logical threads compete for computing units that are already heavily utilized PyTorch (2023).

NeoCPU Liu *et al.* (2019d) improves inference performance for CNN models through tensor-level and graph-level joint optimization which reduces memory access overhead without relying on single operator optimization. These techniques are complementary to our solution which focuses on solving the memory throughput contention and computing efficiency problem specific to CPUs, where neither area has been well explored in prior research.

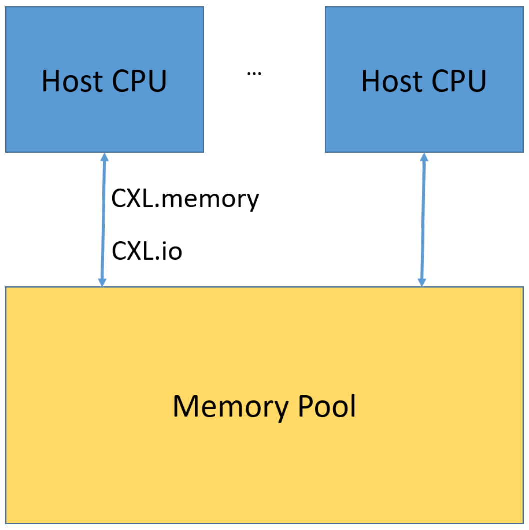
Breaking memory bottleneck for LLM. Many studies focus on solving the memory capacity bottleneck for GPU-based training and inference systems. PagedAttention Kwon *et al.* (2023) solves the GPU memory fragmentation problem caused by fixed-length KV-cache through paging, which shares the same idea as in today's operating systems. FlexGen Sheng *et al.* (2023) enables running LLM inference on a single GPU by aggregating memory from GPU, CPU, and storage. ZeRO Rajbhandari *et al.* (2020, 2021) reduces the GPU memory usage by efficiently distributing the model's parameters, gradients, and optimizer states across multiple GPUs, enabling the training of much larger LLMs.

Leveraging CXL memory expansion. CXL has been proposed for memory expansion in a single-machine Sun *et al.* (2023) or disaggregated memory pools Li *et al.* (2023); Yang *et al.* (2022b). Mempolicy $M : N$ interleave Maruf (2023) is a memory allocation policy for memory systems, allowing the allocation of $M : N$ memory pages to fast memory and slow memory, enabling fine-grained control over memory traffic distribution. Our solution automatically searches and applies the optimal distribution of memory pages across different memory devices to mitigate the memory contention problem of LLM inference on CPUs. Caption Sun *et al.* (2023) dynamically adjusts the page allocation percentage between main memory and CXL memory expanders by monitoring memory-relevant performance counters. As discussed in Section 3.3, dynamic page allocation policies such as Caption are not suitable for LLM inference workloads. Hotness-based page migration Maruf *et al.* (2023); num (2023); hot (2023) is commonly used for tiered memory systems. As discussed in Section 3.3, data migration introduces extra memory traffic and can worsen memory throughput contention which further degrades the inference performance.

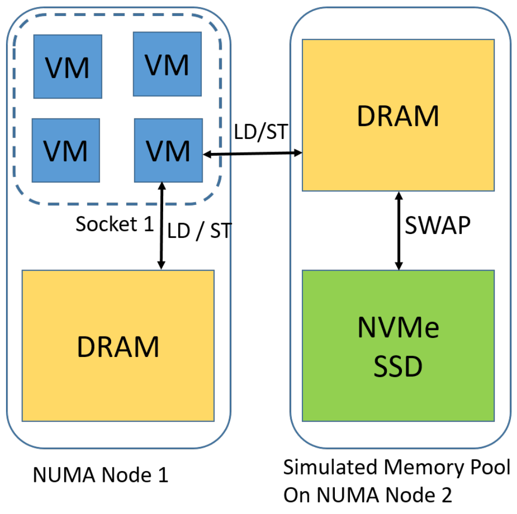
Chapter 4

IMPROVING MEMORY COST THROUGH CXL-ENABLED HYBRID MEMORY POOL

4.1 Methodology



(a) CXL-enabled Memory Pool



(b) Simulated testbed

Figure 4.1: Architecture Overview

4.1.1 Testbed Architecture

Figure 4.1a shows a common way to implement CXL-enabled memory pool hardware https://www.businesswire.com/news/home/20220303006046/en/Tanzanite-Silicon-Solutions-Demonstrates-IndustryNext-Generation-Composable-Data-Centers (2022); Li *et al.* (2022). CXL supports a variety of use cases via three protocols: CXL.io, CXL.cache, and CXL.memory. Among them, CXL.memory allows the host to access attached memory using load/store commands. The memory pool has multiple

CXL.memory endpoints that can be directly connected to the host CPUs, through which the host CPUs can access the memory pool using load/store commands. However, the hardware and software management of the memory pool is done by the memory pool itself and is transparent to the host CPUs.

As there is no real CXL-enabled memory pool at this time, we simulate the memory pool by leveraging the cache coherent NUMA architecture that is widely available on today's multi-socket systems. There are two reasons to choose NUMA architecture. First, NUMA architecture is cache coherent and uses load/store commands as CXL. Second, from the host CPUs' point of view, the memory provided by the attached memory pool has similar memory access latency compared to memory from the remote NUMA nodes CXL (2023). Even with SSDs added to the memory pool, an intelligent memory migration algorithm can help amortize the memory access latency. Figure 4.1b shows an example of the architecture of the simulated testbed. We are using only 2 NUMA nodes here for simplicity but in reality, the architecture can be expanded to a group of NUMA nodes.

NUMA node 1 serves as the local memory for the VMs running on socket 1's CPUs. NUMA node 2 and NVMe SSD form the tiered hybrid memory pool. In case of page allocation, pages are preferably allocated on NUMA node 1. When NUMA node 1 is under pressure, pages are then allocated in the hybrid memory pool. Inside the memory pool, most frequently used pages are cached in NUMA node 2. A page will be evicted from the NUMA node 2 to the NMVe SSD when NUMA node 2 is under pressure. Pages can migrate between NUMA node 1 and memory pool with the support of the hypervisor aut (2012). In this way, a tiered memory system is formed. NUMA node 1 is the first tier with the lowest memory access latency. The hybrid memory pool is the second tier with higher memory access latency. Within the memory pool, there is a sub-tiered memory system with NUMA node 2 as the first tier and NVMe SSD as the second tier.

4.1.2 Software Simulation

To control how much memory should be allocated from the memory pool to guest VMs which run the applications, we modified the Linux cgroups cgr (2023) to allow separate memory control policy for each memory tier. In the simulation, the application has different cgroups memory limits for NUMA node 1 (first tier) and hybrid memory pool (second tier). Within the hybrid memory pool, another set of cgroups memory limits is set for NUMA node 2 (first tier) and NVMe SSD (second tier) respectively. In this way, we are able to simulate the tiered memory system.

Considering the large latency gap between DRAM and SSD, we use Linux swap to simulate page migration between NUMA node 2 and the NVMe SSD as the data access latency overwhelms the page fault handling overhead. Swap happens when NUMA node 2 runs out of memory. If a guest VM accesses the pages that have been swapped out by the hypervisor, page faults caused by Extended Page Table (EPT) violation are generated. Note that using real CXL hardware can potentially achieve better application performance since in the simulation, page fault handling is removed from the guest VM but still happens on the host side.

NUMA node 1 only migrates memory pages with NUMA node 2. To ensure that no pages on NUMA node 1 are swapped out to NVMe SSD, we pin the memory pages of a VM that are allocated to NUMA node 1 in DRAM. This action prevents the pages allocated to NUMA node 1 from getting evicted. Our simulator also extends swap to provide an interface so that different prefetching and cache replacement algorithms can be easily supported in the hybrid memory pool as a separate kernel module.

4.2 Evaluation

We choose four workloads that cover a large portion of today's cloud application categories including video processing, database, data analytic and deep learning training. Each workload runs inside a VM provisioned by QEMU with KVM acceleration enabled on a dual socket Linux server (Table 4.1). We run the workloads on physical machine to get the memory usage, and then set the VM memory size accordingly.

Overcommit ratio is set to help us understand the performance and cost impact by adopting NVMe SSDs and altering the composition of DRAM and SSDs in the hybrid memory pool. Equation 4.1 shows how the DRAM memory usage is computed with overcommit ratio. In the equation, we use R to represent the overcommit ratio, $actual\_m$ to represent the actual memory usage from DRAM, req_m to represent the required memory usage of the application.

$$actual\_m = \frac{req\_m}{R} \tag{4.1}$$

With overcommit ratio set as 1, no NVMe SSD is used and all the memory of a VM will be allocated on DRAM. When overcommit ratio is increased, the percentage of NVMe SSD used is increased. We allocate 50% of the DRAM memory space on the local NUMA node (first tier), and the rest on the remote NUMA nodes (second tier). For example, with a 32GB VM memory size and the overcommit ratio set to 2, at most 16GB of memory space will be allocated on DRAM and the rest 16GB of memory space is allocated using NVMe SSD. Of the 16GB of DRAM memory space, 8GB is allocated on local NUMA node and the rest is on remote NUMA nodes. In our evaluation, we use one local NUMA node and one remote NUMA node.

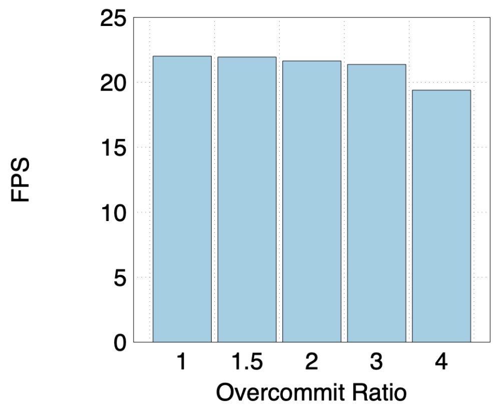
For each workload, we set the overcommit ratio to be 1, 1.5, 2, 3 and 4. In the evaluation, the video encoding VM is configured to use 4GB of memory and 2 virtual

Table 4.1: Specifications of the testbed.

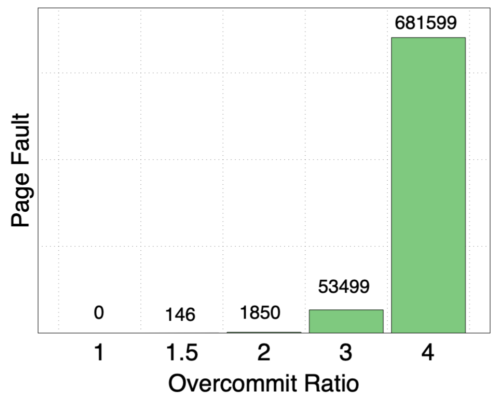
CPU	DRAM	SSD	GPU	QEMU	OS	Kernel
2x Intel Platinum 8268	2x 384GB DDR4	Samsung PM983 3.5TB	8x NVIDIA V100	4.2.1	Ubuntu 20.04 focal	5.15.37

cores, while the other three workloads VMs are configured to use 64GB of memory and 32 virtual cores. To understand the baseline performance of the hybrid memory pool, we use the Linux default prefetcher for the evaluation.

In the following evaluation, we will evaluate the performance and cost tradeoff of the four workloads using hybrid memory pool (overcommit ratio bigger than 1) compared to using DRAM-only memory pool (overcommit ratio equal to 1). Each workload runs three times and the mean values are presented.



(a) Frame Per Second (FPS)

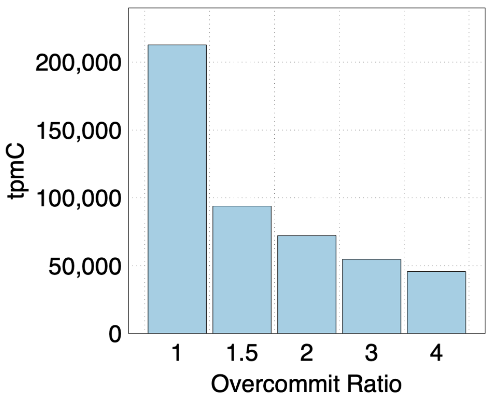


(b) Page Fault

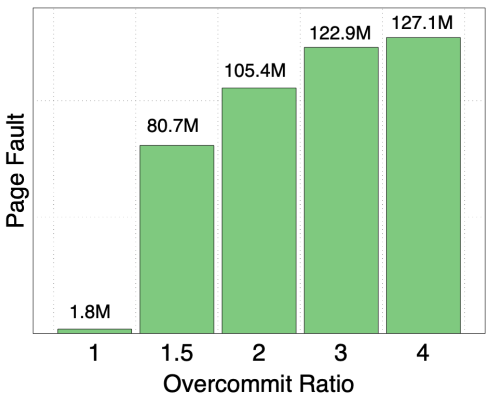
Figure 4.2: FFmpeg Evaluation Results

4.2.1 H.264 Video Encoding

H.264 is a video coding format for full High Definition (FHD) video and audio. FFmpeg is a suite of programs and libraries for video encoding, decoding and transcoding. We used FFmpeg 4.2.1 to encode a 1080P 10-minute long video to H.264 format using libx264. We use Frames Per Second (FPS) to measure FFmpeg's performance. Figure 4.2a and Figure 4.2b show the FFmpeg evaluation results. From the figures, when overcommit ratio increases from 1 to 1.5, the performance degrades slightly by 0.3%. When we continue to increase the overcommit ratio, the performance degradation does not get much worse, drops by 11.9% for overcommit ratio of 4. The performance degradation is unified with the increase of the page fault. Note that although the number of page faults increases a lot for overcommit ratio of 4, the performance degradation is not much. This is because FFmpeg is computation-bounded workloads and the increase in I/Os does not become the bottleneck of the overall performance.



(a) Transaction Per Min (tpmC)

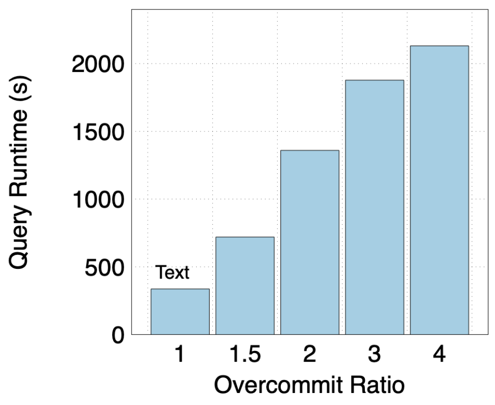


(b) Page Fault

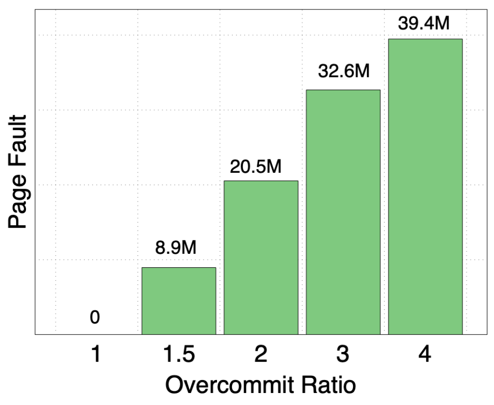
Figure 4.3: TPC-C Evaluation Results

4.2.2 TPC-C

TPC Benchmark C (TPC-C) is an on-line transaction processing (OLTP) benchmark. It is measured in transactions per minute (tpmC). We used MySQL 8.0.29 as the database. Figure 4.3a and Figure 4.3b show the TPC-C evaluation results of 1000 warehouses. When overcommit ratio increases from 1 to 1.5, the performance degrades abruptly by 55.8%. This degradation is in accordance with the abrupt increase in the number of page faults. For overcommit ratio 4, the performance degradation is the worst, drops by 78.5%.



(a) Query Runtime

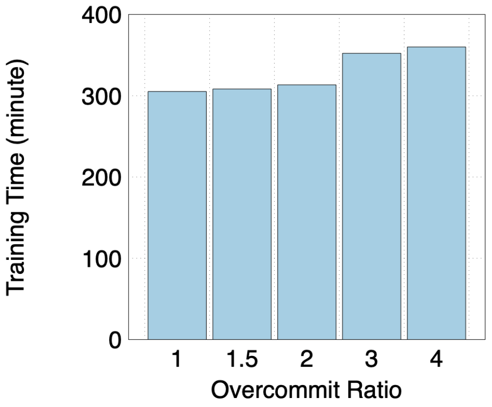


(b) Page Fault

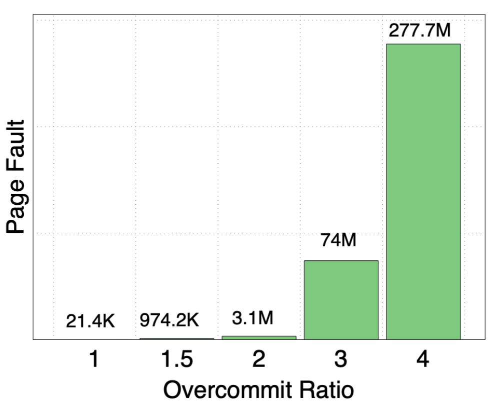
Figure 4.4: TPC-H Evaluation Results

4.2.3 TPC-H

TPC Benchmark H (TPC-H) is a decision support benchmark. It includes a suite of 22 business ad-hoc queries and concurrent data modifications. We used Greenplum 6.20.3 gre (2023), a Postgresql compatible Database Management System (DBMS) for data analytics, as the database deployed on a single VM. Figure 4.4a and Figure 4.4b show the TPC-H evaluation results of scale factor 30. We use the accumulated runtime of the 22 queries to measure its performance. When overcommit ratio increases from 1 to 1.5, the performance degrades a lot by 1.13X. With the increase of the overcommit ratio, the performance continues to degrade by 5.32X at last. The number of page faults increases following similar pattern.



(a) Training Time



(b) Page Fault

Figure 4.5: ResNet50 Evaluation Results

4.2.4 ResNet50

ImageNet is an dataset organized according to the WordNet hierarchy for image classification. We used ImageNet https://www.image-net.org/ (2022) to train ResNet50 He *et al.* (2016) using Pytorch 1.11.0 with 8 GPUs. We used training time of 15 epochs to measure the performance. All runs use the same training seed to provide consistent results. Figure 4.5a and Figure 4.5b show the ResNet50 evaluation results. When overcommit ratio increases from 1 to 1.5, the performance degrades only by 1%. With the increase of the overcommit ratio, the performance continues to degrade by 17.9% with overcommit ratio of 4. This is related to the large increase in the number of page fault.

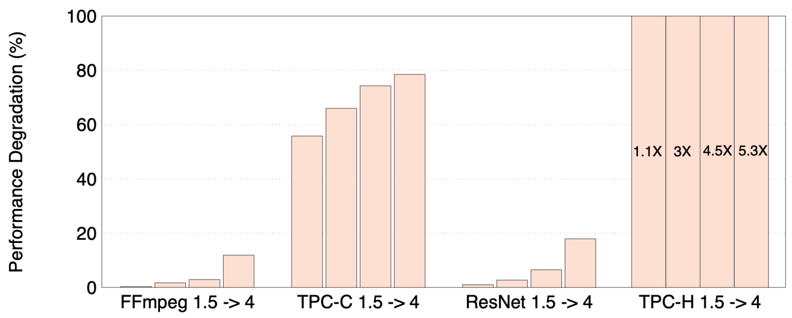


Figure 4.6: Normalized Performance Degradation of Four Workloads with Varying Overcommit Ratio

4.2.5 Analysis

Figure 4.6 shows the performance degradation for all four workloads. From the above evaluation, we find that the hybrid memory pool involving SSDs does affect the applications performance. However, the performance degradation is largely dependent on the application. For computation-intensive workloads, like video processing and deep learning training (FFmpeg has almost 100% CPU utilization and ResNet50 has 74% GPU utilization on average), the degradation is marginal. Since the performance bottleneck depends mostly on the computation instead of the addition of memory access latency introduced by slower media. Applying SSDs to the memory pool does reduce cost while maintaining the same level of performance for these type of workloads. On the other hand, for database and analytic workloads, such as TPC-C and TPC-H, which have high requirement to memory latency and bandwidth, are severely affected by the hybrid memory pool.

4.2.6 Total Cost of Memory (TCM)

We compute the TCM using Equation 4.2. Based on Equation 4.1, we use *mem_p* to represent the unit price of DRAM and *SSD_p* to represent the unit price of NVMe

SSD.

$$TCM = mem\_p * \frac{req\_m}{R} + SSD\_p * (req\_m - \frac{req\_m}{R}) \tag{4.2}$$

We computed the unit price of the DRAM and NVMe SSD in Table 4.1 as \$4.875 mem (2022) and \$0.16 ssd (2022) per GB, respectively. Based on our configuration, Figure 4.7 shows the saved TCM percentage at different overcommit ratio compared to the original TCM with overcommit ratio 1. By using NVMe SSD, even with a mere addition of NVMe SSD at an overcommit ratio 1.5, we can already save 32% of TCM. What's more, we can save up to 72.5% of TCM when overcommit ratio is 4.

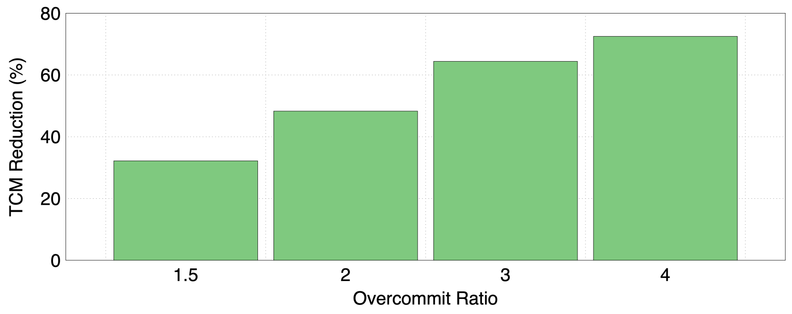


Figure 4.7: Total Cost of Memory (TCM) Reduction Percentage with Varying Overcommit Ratio

Chapter 5

CONCLUSIONS

This comprehensive study has successfully navigated the complexities of data-intensive systems and proposed solutions through memory and storage disaggregation. Through the presentation of three interconnected works, we have addressed some of the most pressing challenges in modern computing environments.

In the realm of cloud storage, “AdaCache” marks a shift by introducing an adaptive, disaggregated caching system. Its ability to dynamically allocate cache blocks based on request size not only optimizes I/O performance but also minimizes memory overhead. This approach represents a significant advancement over traditional caching methods, particularly in its adaptability to diverse cloud workloads. With AdaCache, storage resources are utilized more effectively, reducing wasted space and lowering costs for cloud service providers and their customers.

The optimization of CPU-based systems for LLM inference tackles the challenges of processing large language models efficiently. By introducing innovative methods such as latency-aware memory aggregation and NUMA-aligned tensor parallelism, this contribution has not only improved latency and resource utilization but also positioned CPUs as viable platforms for complex LLM inference tasks.

Finally, the exploration of memory disaggregation through the integration of NVMe SSDs into memory pools addresses critical issues like the high cost and power consumption of DRAM in cloud environments. This work’s insight into creating a more cost-effective and power-efficient tiered memory architecture highlights the potential of hybrid memory systems in enhancing cloud infrastructure performance.

In conclusion, each of these works contributes significantly to advancing the field of data storage and processing, addressing key challenges with inventive and effective solutions. Their collective insights and advancements not only enhance current practices but also pave the way for future innovations in edge computing, cloud storage, and large-scale data processing. This dissertation stands as a testament to the potential of focused, innovative efforts in overcoming the hurdles of modern-day computing and setting new benchmarks for efficiency, scalability, and cost-effectiveness in the digital age.

REFERENCES

"Autonuma: the other approach to numa scheduling", https://lwn.net/Articles/488709/ (2012).

"The density, cost, and marketing of semiconductor memory", https://news.skhynix.com/the-density-cost-and-marketing-of-semiconductor-memory/ (2021).

"Nvm express moves into the future", https://nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf (2021).

"Samsung's poseidon v2 e3.x reference system", https://www.inspursystems.com/product/open-storage/ (2021).

"Memory prices", https://memory.net/memory-prices/ (2022).

"Nvme ssd prices", https://diskprices.com/?locale=uscondition=new,used&disk_types=u2 (2022).

"Amazon elastic block store (ebs)", https://aws.amazon.com/ebs/ (2023).

"Bard", https://en.wikipedia.org/wiki/Bard_(chatbot) (2023).

"Block device user guide", https://spdk.io/doc/bdev.html (2023).

"Ceph block device", https://docs.ceph.com/en/quincy/rbd/index.html (2023).

"Chatgpt", https://en.wikipedia.org/wiki/ChatGPT (2023).

"Claude", https://en.wikipedia.org/wiki/Anthropic#Claude (2023).

"Control groups", https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html (2023).

"Greenplum: Massively parallel postgres for analytics", http://genzconsortium.org (2023).

"Ibm cloud block storage", https://www.ibm.com/cloud/block-storage (2023).

"Memcached", https://memcached.org (2023).

"NUMA balancing: optimize memory placement for memory tiering system", https://lore.kernel.org/linux-mm/20220221084529.1052339-1-ying.huang@intel.com/ (2023).

"Nvm express moves into the future", https://nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf (2023a).

"Persistent disk", https://cloud.google.com/persistent-disk (2023).

"Poseidonos", https://github.com/poseidonos/poseidonos (2023).

"Spdk nvme-of rdma (target & initiator) performance report release 22.09", https://ci.spdk.io/download/performancereports/SPDK_rdma_mlx_perf_report_2209.pdf (2023b).

"Tiered memory: Hot page selection", https://lore.kernel.org/lkml/20220622083519.708236-2-ying.huang@intel.com/T/ (2023).

"vllm", https://github.com/vllm-project/vllm (2023).

Afzal, S. and G. Kavitha, "Load balancing in cloud computing–a hierarchical taxonomical classification", Journal of Cloud Computing **8**, 1, 22 (2019).

Agarwal, A., J. Hennessy and M. Horowitz, "An analytical cache model", ACM Trans. Comput. Syst. **7**, 2, 184–215, URL https://doi.org/10.1145/63404.63407 (1989).

Amar, A., A. Raja and V. Sundararajan, "Glusterfs: a scalable network filesystem", in "Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation", (USENIX Association, 2004).

Amazon Web Services, "Amazon S3 - simple storage service", https://aws.amazon.com/s3/, accessed: March 3, 2023 (2021).

AMD, "Amd infinity architecture", https://www.amd.com/en/technologies/infinity-architecture (2023).

Arteaga, D., J. Cabrera, S. Sundararaman, J. Xu and M. Zhao, "CloudCache: On-demand flash cache management for cloud computing systems", in "Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)", (2016).

AWS, A., "Tensor parallelism", https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-extended-features-pytorch\-tensor-parallelism.html (2023).

Berger, D. S., B. Berg, T. Zhu, S. Sen and M. Harchol-Balter, "RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor", in "13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)", pp. 195–212 (USENIX Association, Carlsbad, CA, 2018), URL https://www.usenix.org/conference/osdi18/presentation/berger.

Bonwick, J. *et al.*, "The slab allocator: An object-caching kernel memory allocator.", in "USENIX summer", vol. 16 (Boston, MA, USA, 1994).

Brown, T., B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners", Advances in neural information processing systems **33**, 1877–1901 (2020).

Bulkowski, B. and S. Srinivasan, “Aerospike: Architecture of a real-time operational dbms”, IEEE Data Eng. Bull. **36**, 1, 3–9 (2013).

Byrne, D., N. Onder and Z. Wang, “Faster slab reassignment in memcached”, in “Proceedings of the International Symposium on Memory Systems”, pp. 353–362 (2019).

Callaghan, B., *NFS Illustrated* (Addison-Wesley, 2002).

Carlson, J., *Redis in action* (Simon and Schuster, 2013).

Chen, H., H. Zhang, M. Dong, Z. Wang, Y. Xia, H. Guan and B. Zang, “Efficient and available in-memory kv-store with hybrid erasure coding and replication”, ACM Trans. Storage **13**, 3, URL `https://doi.org/10.1145/3129900` (2017).

Chernykh, I., I. Kulikov, B. Glinsky, V. Vshivkov, L. Vshivkova and V. Prigarin, “Advanced vectorization of ppml method for intel® xeon® scalable processors”, in “Supercomputing: 4th Russian Supercomputing Days, RuSCDays 2018, Moscow, Russia, September 24–25, 2018, Revised Selected Papers 4”, pp. 465–471 (Springer, 2019).

Chowdhery, A., S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, “Palm: Scaling language modeling with pathways”, Journal of Machine Learning Research **24**, 240, 1–113 (2023).

contributors, M., “MinIO - object storage for the next generation”, `https://min.io/`, accessed: March 3, 2023 (2021).

Corporation, I., “SPDK: Storage Performance Development Kit”, `https://spdk.io/` (Accessed 2023).

CXL, “Compute express link: The breakthrough cpu-to-device interconnect cxl”, `https://www.computeexpresslink.org/` (2023).

Dao, T., D. Fu, S. Ermon, A. Rudra and C. Ré, “Flashattention: Fast and memory-efficient exact attention with io-awareness”, Advances in Neural Information Processing Systems **35**, 16344–16359 (2022).

Dice, D. and A. Kogan, “Optimizing inference performance of transformers on cpus”, arXiv preprint arXiv:2102.06621 (2021).

Dubnicki, C. and T. J. LeBlanc, “Adjustable block size coherent caches”, SIGARCH Comput. Archit. News **20**, 2, 170–180, URL `https://doi.org/10.1145/146628.139725` (1992).

Face, H., “transformers”, `https://github.com/huggingface/transformers` (2023).

facebook, “facebook/opt-13b model”, `https://huggingface.co/facebook/opt-13b` (2023).

Facebook and Huggingface, "facebook/opt-30b", https://huggingface.co/facebook/opt-30b (2023a).

Facebook and Huggingface, "facebook/opt-350m", https://huggingface.co/facebook/opt-350m (2023b).

Facebook and Huggingface, "facebook/opt-66b", https://huggingface.co/facebook/opt-66b (2023c).

Facebook and Huggingface, "facebook/opt-6.7b", https://huggingface.co/facebook/opt-6.7b (2023d).

Fan, B., D. G. Andersen and M. Kaminsky, "MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing", in "10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)", pp. 371–384 (USENIX Association, Lombard, IL, 2013), URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan.

fio, "FIO — Flexible I/O tester synthetic benchmark", http://git.kernel.dk/?p=fio.git (2023).

Fu, J., D. Arteaga and M. Zhao, "Locality-driven mrc construction and cache allocation", in "Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing", pp. 19–20 (2018).

Fu, J., Y. Liu and G. Liu, "Jcache: Journaling-aware flash caching", IEEE Access **8**, 61289–61298 (2020a).

Fu, J., Y. Lu, J. Shu, G. Liu and M. Zhao, "Cowcache: effective flash caching for copy-on-write virtual disks", Cluster Computing **23**, 623–639 (2020b).

Guz, Z., H. Li, A. Shayesteh and V. Balakrishnan, "Nvme-over-fabrics performance characterization and the path to low-overhead flash disaggregation", in "Proceedings of the 10th ACM International Systems and Storage Conference", pp. 1–9 (2017).

He, K., X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition", in "Proceedings of the IEEE conference on computer vision and pattern recognition", pp. 770–778 (2016).

Hennessy, J. L. and D. A. Patterson, *Computer architecture: a quantitative approach* (Elsevier, 2011).

https://www.businesswire.com/news/home/20220303006046/en/Tanzanite-Silicon-Solutions-Demonstrates-IndustryNext-Generation-Composable-Data-Centers, "Gen-z specification", (2022).

https://www.image-net.org/, "Imagenet", (2022).

Hu, X., X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang and Z. Wang, "LAMA: Optimized locality-aware memory allocation for key-value cache", in "2015 USENIX Annual Technical Conference (USENIX ATC 15)", pp. 57–69 (USENIX Association, Santa Clara, CA, 2015), URL `https://www.usenix.org/conference/atc15/technical-session/presentation/hu`.

Huggingface, "Tensor parallelism", `https://huggingface.co/docs/text-generation-inference/conceptual/tensor_parallelism` (2023).

HuggingFace, "text-generation-inference", `https://github.com/huggingface/text-generation-inference` (2023).

Intel, "Intel advanced matrix extensions", `https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/advanced-matrix-extensions/overview.html` (2023a).

Intel, "Intel extension for pytorch", `https://github.com/intel/intel-extension-for-pytorch` (2023b).

Intel, "Intel memory latency checker (intel mlc)", `https://www.intel.com/content/www/us/en/download/736633/intel-memory-latency-checker-intel-mlc.html` (2023c).

Intel, "Intel ultra path interconnect", `https://en.wikipedia.org/wiki/Intel_Ultra_Path_Interconnect#cite_note-Intel_QPI-1` (2023d).

Intel, "Intel vtune profiler", `https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.2iv2n8` (2023e).

Intel, "Intel xeon processor scalable family technical overview", `https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html/` (2023f).

Jain, R., S. Cheng, V. Kalagi, V. Sanghavi, S. Kaul, M. Arunachalam, K. Maeng, A. Jog, A. Sivasubramaniam, M. T. Kandemir *et al.*, "Optimizing cpu performance for recommendation systems at-scale", in "Proceedings of the 50th Annual International Symposium on Computer Architecture", pp. 1–15 (2023a).

Jain, R., S. Cheng, V. Kalagi, V. Sanghavi, S. Kaul, M. Arunachalam, K. Maeng, A. Jog, A. Sivasubramaniam, M. T. Kandemir *et al.*, "Optimizing cpu performance for recommendation systems at-scale", in "Proceedings of the 50th Annual International Symposium on Computer Architecture", pp. 1–15 (2023b).

Jeremic, N., H. Parzyjegla and G. Muhl, "On adapting the cache block size in ssd caches", in "2021 IEEE International Conference on Networking, Architecture and Storage (NAS)", pp. 1–8 (2021).

Jiang, J., J. Du, D. Huang, Z. Chen, Y. Lu and X. Liao, "Full-stack optimizing transformer inference on arm many-core cpu", IEEE Transactions on Parallel and Distributed Systems (2023).

Jin, X., Y. Wang and X. Tan, “Pornographic image recognition via weighted multiple instance learning”, IEEE transactions on cybernetics **49**, 12, 4412–4420 (2018).

Kogan, A., “Improving inference performance of machine learning with the divide-and-conquer principle”, arXiv preprint arXiv:2301.05099 (2023).

Koller, R., L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala and M. Zhao, “Write policies for host-side flash caches”, in “Presented as part of the 11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)”, pp. 45–58 (2013).

Koller, R., A. Mashtizadeh and R. Rangaswami, “Centaur: Host-side ssd caching for storage performance control”, The 12th IEEE International Conference on Autonomic Computing (2015).

Kwon, W., Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang and I. Stoica, “Efficient memory management for large language model serving with pagedattention”, in “Proceedings of the 29th Symposium on Operating Systems Principles”, pp. 611–626 (2023).

Lee, C., T. Kumano, T. Matsuki, H. Endo, N. Fukumoto and M. Sugawara, “Understanding storage traffic characteristics on enterprise virtual desktop infrastructure”, in “Proceedings of the 10th ACM International Systems and Storage Conference”, pp. 1–11 (2017).

Li, C., P. Shilane, F. Douglis, H. Shim, S. Smaldone and G. Wallace, “Nitro: A capacity-optimized SSD cache for primary storage”, in “Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference”, pp. 501–512 (USENIX Association, 2014a).

Li, H., D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal *et al.*, “Pond: Cxl-based memory pooling systems for cloud platforms”, in “Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2”, pp. 574–587 (2023).

Li, H., D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, I. Agarwal, M. Hill, M. Fontoura *et al.*, “First-generation memory disaggregation for cloud platforms”, arXiv preprint arXiv:2203.00241 (2022).

Li, J., Q. Wang, P. P. Lee and C. Shi, “An in-depth analysis of cloud block storage workloads in large-scale production”, in “2020 IEEE International Symposium on Workload Characterization (IISWC)”, pp. 37–47 (IEEE, 2020).

Li, W., G. Jean-Baptise, J. Riveros, G. Narasimhan and M. Zhao, “CacheDedup: Inline deduplication for flash caching”, in “Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST’16)”, (2016).

Li, X., A. Aboulnaga, K. Salem, A. Sachedina and S. Gao, "Second-Tier cache management using write hints", in "4th USENIX Conference on File and Storage Technologies (FAST 05)", (USENIX Association, San Francisco, CA, 2005), URL `https://www.usenix.org/conference/fast-05/second-tier-cache-management-using-write-hints`.

Li, X., D. G. Andersen, M. Kaminsky and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing", in "Proceedings of the Ninth European Conference on Computer Systems", EuroSys '14 (Association for Computing Machinery, New York, NY, USA, 2014b), URL `https://doi.org/10.1145/2592798.2592820`.

Linux, "perf: Linux profiling with performance counters", `https://perf.wiki.kernel.org/index.php/Main_Page` (2023).

Liu, Y., H. Li, Y. Lu, Z. Chen, N. Xiao and M. Zhao, "Hasfs: optimizing file system consistency mechanism on nvm-based hybrid storage architecture", Cluster Computing **23**, 2501–2515 (2020).

Liu, Y., H. Li, Y. Lu, Z. Chen and M. Zhao, "An efficient and flexible metadata management layer for local file systems", in "2019 IEEE 37th International Conference on Computer Design (ICCD)", pp. 208–216 (2019a).

Liu, Y., Y. Wang, R. Yu, M. Li, V. Sharma and Y. Wang, "Optimizing CNN model inference on CPUs", in "2019 USENIX Annual Technical Conference (USENIX ATC 19)", pp. 1025–1040 (USENIX Association, Renton, WA, 2019b), URL `https://www.usenix.org/conference/atc19/presentation/liu-yizhi`.

Liu, Y., Y. Wang, R. Yu, M. Li, V. Sharma and Y. Wang, "Optimizing {CNN} model inference on {CPUs}", in "2019 USENIX Annual Technical Conference (USENIX ATC 19)", pp. 1025–1040 (2019c).

Liu, Y., Y. Wang, R. Yu, M. Li, V. Sharma and Y. Wang, "Optimizing {CNN} model inference on {CPUs}", in "2019 USENIX Annual Technical Conference (USENIX ATC 19)", pp. 1025–1040 (2019d).

Loh, G. H. and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches", in "Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture", MICRO-44, p. 454–464 (Association for Computing Machinery, New York, NY, USA, 2011), URL `https://doi.org/10.1145/2155620.2155673`.

Luo, T., S. Ma, R. Lee, X. Zhang, D. Liu and L. Zhou, "S-cave: Effective ssd caching to improve virtual machine storage performance", in "Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques", pp. 103–112 (2013).

Maruf, H. A., "mempolicy: N:m interleave policy for tiered memory nodes", `https://lore.kernel.org/linux-mm/YqD0%2FtzFwXvJ1gK6@cmpxchg.org/T/` (2023).

Maruf, H. A., H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia and P. Chauhan, "Tpp: Transparent page placement for cxl-enabled tiered-memory", in "Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3", pp. 742–755 (2023).

MegEngine, "Inferllm", `https://github.com/MegEngine/InferLLM` (2023).

Meng, F., L. Zhou, X. Ma, S. Uttamchandani and D. Liu, "vCacheShare: Automated server flash cache space management in a virtualization environment", in "2014 USENIX Annual Technical Conference (USENIX ATC 14)", pp. 133–144 (USENIX Association, Philadelphia, PA, 2014), URL `https://www.usenix.org/conference/atc14/technical-sessions/presentation/meng`.

Microsoft, "Deepspeed", `https://github.com/microsoft/DeepSpeed` (2023).

microsoft, "Deepspeed-mii", `https://github.com/microsoft/DeepSpeed-MII` (2023).

Mutlu, O., J. Stark, C. Wilkerson and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors", in "The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.", pp. 129–140 (IEEE, 2003).

Nanavati, M., J. Wires and A. Warfield, "Decibel: Isolation and sharing in disaggregated rack-scale storage.", in "NSDI", vol. 17, pp. 17–33 (2017).

Narayanan, D., A. Donnelly and A. Rowstron, "Write off-loading: Practical power management for enterprise storage", ACM Transactions on Storage (TOS) **4**, 3, 1–23 (2008).

Ning, E., N. Yan, J. Zhu and J. Li, "Microsoft open sources breakthrough optimizations for transformer inference on gpu and cpu", tinyurl. com/y26jdsn9 (2020).

Nvidia, "Performance of tensorrt-llm", `https://nvidia.github.io/TensorRT-LLM/performance.html` (2023).

OpenAI, "Openai gpt-3", `https://openai.com/blog/gpt-3-apps` (2023).

OpenMNT, "Ctranslate2", `https://github.com/OpenNMT/CTranslate2` (2023a).

OpenMNT, "Lightllm", `https://github.com/OpenNMT/CTranslate2` (2023b).

Ou, L., X. He, M. Kosa and S. Scott, "A unified multiple-level cache for high performance storage systems", in "13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems", pp. 143–150 (2005).

Prybylski, S., M. Horowitz and J. Hennessy, "Performance tradeoffs in cache design", in "Proceedings of the 15th Annual International Symposium on Computer Architecture", ISCA '88, p. 290–298 (IEEE Computer Society Press, Washington, DC, USA, 1988).

Przybylski, S., "The performance impact of block sizes and fetch strategies", SIGARCH Comput. Archit. News **18**, 2SI, 160–169, URL `https://doi.org/10.1145/325096.325135` (1990).

PyTorch, "Grokking pytorch intel cpu performance from first principles", `https://pytorch.org/tutorials/intermediate/torchserve_with_ipex.html` (2023).

Rajbhandari, S., J. Rasley, O. Ruwase and Y. He, "Zero: Memory optimizations toward training trillion parameter models", in "SC20: International Conference for High Performance Computing, Networking, Storage and Analysis", pp. 1–16 (IEEE, 2020).

Rajbhandari, S., O. Ruwase, J. Rasley, S. Smith and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning", in "Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis", pp. 1–14 (2021).

ServeTheHome, "What is a mcr dimm or multiplexer combined ranks dimm", `https://www.servethehome.com/what-is-a-mcr-dimm-or-multiplexer-combined-ranks-dimm\` `\-sk-hynix-micron-samsung-intel-amd-nvidia/` (2023).

Shen, H., H. Chang, B. Dong, Y. Luo and H. Meng, "Efficient llm inference on cpus", arXiv preprint arXiv:2311.00502 (2023a).

Shen, H., H. Meng, B. Dong, Z. Wang, O. Zafrir, Y. Ding, Y. Luo, H. Chang, Q. Gao, Z. Wang *et al.*, "An efficient sparse inference software accelerator for transformer-based language models on cpus", arXiv preprint arXiv:2306.16601 (2023b).

Sheng, Y., L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica and C. Zhang, "Flexgen: High-throughput generative inference of large language models with a single gpu", in "International Conference on Machine Learning", pp. 31094–31116 (PMLR, 2023).

SJTU-IPADS, "Powerinfer", `https://github.com/SJTU-IPADS/PowerInfer` (2023).

Smith, A. J., "Line (block) size choice for cpu cache memories", IEEE Transactions on Computers **C-36**, 9, 1063–1075 (1987).

Sun, Y., Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong *et al.*, "Demystifying cxl memory with genuine cxl-ready systems and devices", in "Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture", pp. 105–121 (2023).

The GNOME Project, "GLib – C Utility Library", `https://developer.gnome.org/glib/` (2023).

Touvron, H., L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models", arXiv preprint arXiv:2307.09288 (2023).

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser and I. Polosukhin, "Attention is all you need", Advances in neural information processing systems **30** (2017).

Velten, M., R. Schöne, T. Ilsche and D. Hackenberg, "Memory performance of amd epyc rome and intel cascade lake sp server processors", in "Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering", pp. 165–175 (2022).

Waldspurger, C. A., N. Park, A. T. Garthwaite and I. Ahmad, "Efficient mrc construction with shards.", in "FAST", vol. 15, pp. 95–110 (2015).

Wang, S., B. Z. Li, M. Khabsa, H. Fang and H. Ma, "Linformer: Self-attention with linear complexity", arXiv preprint arXiv:2006.04768 (2020).

Womack, M., A. Wang, P. Flynn, X. Yi and Y. Yonghong, "Openmp programming book", (2023).

Wu, J., Y. Yu, C. Huang and K. Yu, "Deep multiple instance learning for image classification and auto-annotation", in "Proceedings of the IEEE conference on computer vision and pattern recognition", pp. 3460–3469 (2015).

Xiang, Y. and H. Kim, "Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference", in "2019 IEEE Real-Time Systems Symposium (RTSS)", pp. 392–405 (IEEE, 2019).

Yadgar, G., M. Factor and A. Schuster, "Karma: Know-it-All replacement for a multilevel cache", in "5th USENIX Conference on File and Storage Technologies (FAST 07)", (USENIX Association, San Jose, CA, 2007), URL `https://www.usenix.org/conference/fast-07/karma-know-it-all-replacement-multilevel-cache`.

Yang, Q., R. Jin, B. Davis, D. Inupakutika and M. Zhao, "Performance evaluation on cxl-enabled hybrid memory pool", in "2022 IEEE International Conference on Networking, Architecture and Storage (NAS)", pp. 1–5 (2022a).

Yang, Q., R. Jin, B. Davis, D. Inupakutika and M. Zhao, "Performance evaluation on cxl-enabled hybrid memory pool", in "2022 IEEE International Conference on Networking, Architecture and Storage (NAS)", pp. 1–5 (IEEE, 2022b).

Zhang, S., S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models", arXiv preprint arXiv:2205.01068 (2022).

Zhang, Y., P. Huang, K. Zhou, H. Wang, J. Hu, Y. Ji and B. Cheng, "Osca: An online-model based cache allocation scheme in cloud block storage systems", in "Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference", pp. 785–798 (2020).

Zhou, K., Y. Zhang, P. Huang, H. Wang, Y. Ji, B. Cheng and Y. Liu, "Efficient ssd cache for cloud block storage via leveraging block reuse distances", IEEE Transactions on Parallel and Distributed Systems **31**, 11, 2496–2509 (2020).

APPENDIX A

PREVIOUSLY PUBLISHED SECTIONS

The content presented in Chapter 2 has been disseminated through a publication in the proceedings of the IEEE International Conference on Cloud Computing in 2023, under the title "AdaCache: A Disaggregated Cache System with Adaptive Block Size for Cloud Block Storage."

The content of Chapter 4 was published in the proceedings of the IEEE International Conference on Networking, Architecture, and Storage in 2022, with the paper entitled "Performance Evaluation on CXL-enabled Hybrid Memory Pool."