Adaptive Memory Reclamation in Multitenant Cloud Systems

A Thesis

submitted to the designated
by the General Assembly
of the Department of Computer Science and Engineering
Examination Committee

by

Rodopi Kosteli

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN DATA AND COMPUTER SYSTEMS ENGINEERING

WITH SPECIALIZATION
IN ADVANCED COMPUTER SYSTEMS

University of Ioannina
School of Engineering
Ioannina 2025

Examining Committee:

- **Stergios V. Anastasiadis**, Professor, Department of Computer Science and Engineering, University of Ioannina (Advisor)
- **Vassilios V. Dimakopoulos**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Panayiotis Tsaparas**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina

Dedication

To my family.

Acknowledgements

I would like to express my sincere gratitude and appreciation to all those who have contributed to the completion of this thesis.

First, I would like to express my gratitude to my supervisor, Professor Stergios Anastasiadis, who guided me throughout my research journey. His commitment to academic excellence, in combination with his advice and his insights in this field have made the whole experience even more inspiring for me.

I am deeply thankful to all my friends for their continuous support and encouragement throughout this journey.

Most importantly, none of these could have happened without the support of my family. My beloved grandmother and grandfather offered me their encouragement through phone calls and by sending packages full of food and fresh vegetables. I would also express my gratitude to my parents, my sisters, and my brother for providing me with unfailing support and profound belief in my abilities. Every time I was ready to quit, you did not let me, and I am forever grateful.

Last but not least, I could not be more indebted to Giorgos, for his endless patience and understanding, especially during the most demanding phases of this thesis. Thank you for always being there, cheering me up, and reminding me to take breaks when I needed them most.

This work was supported in part by the TOPPERS research project implemented in the framework of the Hellenic Foundation for Research & Innovation call "Basic research financing (Horizontal support of all sciences)" under the National Recovery and Resilience Plan "Greece 2.0" funded by the European Union – NextGenerationEU (H.F.R.I. Project Number: 15206).

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Rodopi Kosteli, M.Sc. in Computer Science, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, 2025.
Adaptive Memory Reclamation in Multitenant Cloud Systems.
Advisor: Stergios V. Anastasiadis, Professor.

The rapidly growing memory demands of modern data-intensive applications, such as data analytics and machine learning, coupled with the rising costs of dynamic random-access memory prices, have made physical memory a significant infrastructure expense in multitenant cloud environments. This observation highlights the need for efficient dynamic memory allocation to optimize resource utilization and decrease operational costs. However, this is a challenging problem due to the unpredictable variation of the memory requirements over time. Cloud tenants often resort to overprovisioning memory for accommodating peak application demands. Unfortunately, this approach leads to significant memory waste, as resources are allocated for peak demand but remain underutilized during normal operation.

In the present thesis, we introduce an automated approach to reclaim the underutilized memory of data-intensive applications by leveraging the page reclamation method of the Linux kernel. Our approach monitors several metrics offered by the system kernel to detect underutilized pages across the running applications. Subsequently, we implement a Contention Service to generate controlled memory pressure and enforce the return of the underutilized pages from the application back to the system.

We regard MapReduce as a representative framework commonly deployed on cloud environments and evaluate both synthetic and real-world MapReduce applications. Our results demonstrate that the Contention Service efficiently identifies and reclaims underutilized memory. In comparison to a state-of-the-art approach from

the industry, the Contention Service tracks the actual memory demands of the applications over time with higher accuracy and releases up to 36% more memory with negligible overhead on the containers running the applications.

Εκτεταμενη Περιληψη

Ροδόπη Κωστέλη, Δ.Μ.Σ. στη Μηχανική Δεδομένων και Υπολογιστικών Συστημάτων, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, 2025.
Προσαρμοστική Ανάκτηση Μνήμης σε Πολυμισθωτικά Συστήματα Νέφους.
Επιβλέπων: Στέργιος Β. Αναστασιάδης, Καθηγητής.

Οι ταχέως αυξανόμενες απαιτήσεις μνήμης των σύγχρονων εφαρμογών, όπως η ανάλυση δεδομένων και η μηχανική μάθηση, σε συνδυασμό με το αυξανόμενο κόστος των τιμών της μνήμης τυχαίας προσπέλασης, καθιστούν τη φυσική μνήμη σημαντικό κόστος υποδομής στα πολυμισθωτικά συστήματα υπολογιστικής νέφους. Η παραπάνω παρατήρηση αναδεικνύει την ανάγκη για αποτελεσματική δυναμική κατανομή μνήμης για τη βελτιστοποίηση της χρήσης των πόρων και τη μείωση του λειτουργικού κόστους. Ωστόσο, η σχεδίαση μεθόδων για δυναμική κατανομή μνήμης αποτελεί πρόκληση, καθώς οι απαιτήσεις των εφαρμογών σε μνήμη παρουσιάζουν ασταθή και απρόβλεπτη συμπεριφορά με την πάροδο του χρόνου. Συχνά, οι χρήστες υπερεκτιμούν την ποσότητα μνήμης που απαιτεί η ορθή εκτέλεση εφαρμογών με υψηλές απαιτήσεις σε μνήμη προκειμένου να καλύψουν τη μέγιστη ζήτηση. Αυτή η προσέγγιση έχει ως συνέπεια να παραμένει αδρανής μια ποσότητα δεσμευμένης μνήμης, καθώς οι πόροι διατίθενται για τη ζήτηση αιχμής, αλλά παραμένουν ανεκμετάλλευτοι κατά τη διάρκεια της κανονικής λειτουργίας.

Στην παρούσα εργασία στοχεύουμε στην επίλυση του παραπάνω προβλήματος και προτείνουμε μια αυτοματοποιημένη προσέγγιση για την ανάκτηση της αδρανούς μνήμης των εφαρμογών με υψηλές απαιτήσεις σε μνήμη. Η προσέγγισή μας αξιοποιεί την υπηρεσία ανάκτησης σελίδων του πυρήνα του λειτουργικού συστήματος Linux. Πιο συγκεκριμένα, παρακολουθεί μετρικές που παρέχει ο πυρήνας με στόχο να ανιχνεύσει σελίδες που παραμένουν αναξιοποίητες από τις εφαρμογές που εκτελούνται στο σύστημα.

Υλοποιούμε μια υπηρεσία, η οποία δημιουργεί ελεγχόμενη πίεση στη μνήμη, δεσμεύοντας την κατάλληλη ποσότητα μνήμης, με στόχο την αφύπνιση του μηχανισμού ανάκτησης αδρανών σελίδων του πυρήνα του Linux για την επιστροφή των σελίδων που δεν χρησιμοποιούνται ενεργά από την εφαρμογή πίσω στο σύστημα.

Θεωρούμε το MapReduce ως ένα αντιπροσωπευτικό μοντέλο εφαρμογών που εκτελούνται σε περιβάλλοντα υπολογιστικής νέφους και αξιολογούμε τόσο συνθετικές, όσο και πραγματικές εφαρμογές MapReduce. Τα αποτελέσματα δείχνουν ότι η προσέγγισή μας αναγνωρίζει αποτελεσματικά την αδρανή μνήμη και την επιστρέφει στο σύστημα. Σε σύγκριση με μια λύση αιχμής για το πρόβλημα της αναγνώρισης αδρανούς μνήμης, η προσέγγισή μας εντοπίζει με υψηλότερη ακρίβεια τις πραγματικές απαιτήσεις μνήμης των εφαρμογών κατά τη διάρκεια εκτέλεσής τους και επιστρέφει εώς και 36% περισσότερη αδρανή μνήμη στο σύστημα με ελάχιστη επιβάρυνση στη εκτέλεση των εφαρμογών.

Chapter 1

Introduction

The rapid development of data-intensive applications, such as big data analytics, machine learning, and databases in cloud computing, necessitates efficient memory management to handle large-scale data operations. Applications often require substantial memory resources, leading to frequent triggering of page reclamation algorithms. Both background and foreground reclamation can degrade performance, as the former consumes system resources and the latter causes application stalls [4, 5].

Resource allocation in cloud environments is typically static and overprovisioned due to difficulties in predicting memory needs. Applications may allocate large memory objects that are infrequently accessed, resulting in low average memory utilization (40%-60%) [6]. This necessitates a method to release idle memory without impacting application performance in order to optimize resource usage and support more tenants in a containerized cloud environment.

1.1 Problem Statement

The focus of this thesis is to study and improve the management of memory in datacenter servers that host applications from different tenants. Modern datacenters host diverse workloads, such as web services, data analytics, and artificial intelligence. These workloads often vary significantly in their memory usage patterns making the management of memory a complex task. Moreover, memory is inherently constrained by physical hardware limits, making its effective utilization a top priority for the datacenter [7, 8].

Application and DevOps engineers cannot estimate the actual resource requirements of the applications that they deploy in the datacenter. Applications often experience dynamic and unpredictable workload patterns. For instance, a web application might experience sudden surges in traffic during specific events, while a batch processing job might require additional memory only at certain computation stages. To account for such variability, engineers typically overprovision memory by reserving a larger amount than the application's average consumption. This approach is designed to mitigate the risk of memory exhaustion during peak loads, which could lead to performance degradation or out-of-memory killing. However, the memory allocation for the peak load leads to the underutilization of the memory resource, as most applications use only a fraction of the total reserved memory during their lifecycle, leaving large portions of memory idle and unavailable for other uses.

We define the portion of physical memory that the tenants reserve for their applications but remains unused for a long period of time as *underutilized memory*. The problem that we try to solve in this thesis is to automatically identify and reclaim the underutilized memory, in order to make it available for use by tenant applications that need it.

1.2 Motivation

In modern cloud environments, resource allocation and memory management are critical challenges, particularly when managing applications from multiple tenants with varying memory usage patterns. Applications often allocate memory in advance to ensure sufficient resources during peak loads, which leads to large portions of memory being reserved but not actively used. This idle memory, although reserved

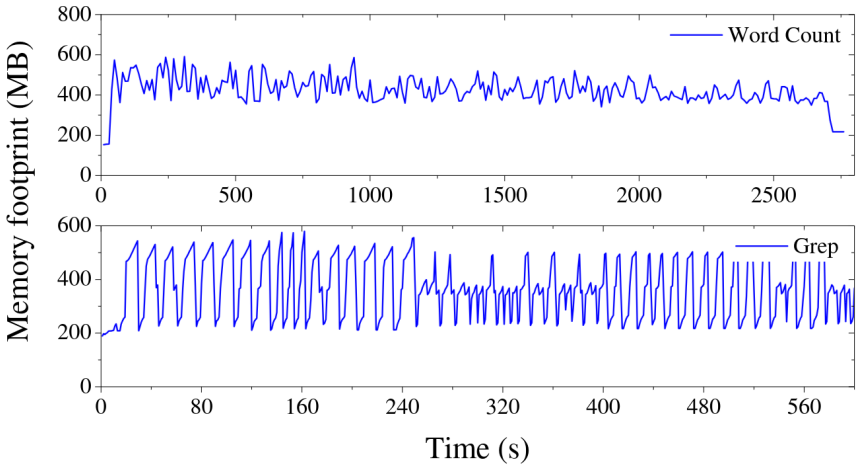

Figure 1.1: The memory footprint trace of Hadoop MapReduce applications [1].

for future use, represents an inefficiency in resource management, as it stays unused while other applications or tenants could benefit from it.

To emphasize the relevance of the above problem, we analyze the memory demands introduced by MapReduce application workloads [1], as illustrated in Figure 1.1. Observations from the CloudScale system show that these workloads exhibit alternating phases of high and low memory consumption, often leading to inefficient memory allocation. Both high and low memory demand last approximately 10-15s. To accommodate peak demand, memory is often over-allocated, even though the actual usage fluctuates significantly throughout the application's lifecycle. Such variations expose the limitations of static provisioning strategies, which fail to dynamically adapt to changing resource needs.

A small portion of memory, known as the working set, is accessed frequently by applications. However, there are also temporary bursts where applications utilize significantly larger amounts of memory for short durations. Since these bursts involve memory that is unlikely to be accessed again soon, static allocation strategies result in suboptimal resource utilization.

The above behavior underscores the need for an adaptive approach for reclaiming underutilized memory in cloud environments, in order to reduce the impact of overprovisioning and achieve more efficient utilization of resources.

As applications in cloud datacenters grow in size and complexity, the ability to dynamically manage memory resources becomes increasingly important. For instance, web applications, databases, and machine learning systems often make use of caching

mechanisms or store intermediate results to avoid redundant computations. These practices can be considered forms of ”soft state”, where memory is allocated for data that is not actively being used but has the potential to improve performance if accessed in the future.

Overprovisioning memory to hold the soft state can waste system resources, while underprovisioning can cause performance bottlenecks when applications experience sudden spikes in load or require more resources than anticipated. Additionally, different applications benefit from caching and memory allocation to varying degrees, making it difficult to find a one-size-fits-all solution.

This problem is compounded in multitenant cloud environments where multiple applications, often with diverse resource requirements, are hosted on a shared infrastructure. Without efficient memory management mechanisms, memory allocation becomes wasteful, limiting the efficiency of cloud systems. Therefore, there is a need for a dynamic and elastic memory management scheme that can effectively reclaim underutilized memory, optimize resource usage, and minimize performance degradation.

In this thesis, we aim to address this problem by proposing a system that automatically detects and reclaims underutilized memory in cloud environments. Our approach ensures that allocated memory that is not actively used by applications is adaptively reclaimed following their real-time needs. We focus on the efficient identification and reclamation of the underutilized memory without impacting the performance of applications.

1.3 Contributions

In this thesis, we examine approaches for efficiently identifying and reclaiming underutilized memory in containerized environments. We focus on leveraging existing kernel mechanisms and computing techniques to dynamically optimize memory usage while minimizing the impact on application performance.

We can summarize our contributions as follows:

- We provide a review of related research on memory management, idle-page tracking and memory reclamation techniques used in cloud environments, highlighting their strengths and limitations.

- We introduce the architecture of the Contention Service (CS), that identifies the underutilized memory allocated by applications based on system metrics. Through the Memory Stresser (MS) component, it generates controlled memory pressure, in order to enable the background reclamation process of the Linux kernel to release the underutilized memory.

- We implement a prototype of the Contention Service that monitors several metrics offered by the Linux kernel and applies the required pressure through standard memory allocations, in order to initiate background reclamation. Thus, our approach does not require modifications to the user applications or the underlying kernel.

- We perform an experimental evaluation of the Contention Service, demonstrating its effectiveness in reclaiming underutilized memory without excessive performance degradation. Using synthetic and real-world applications, we demonstrate the effectiveness of our system, highlighting its ability to identify underutilized memory with better accuracy and reclaim it faster than state-of-the-art approaches. Our results show that the Contention Service incurs minimal overhead and can effectively optimize memory utilization in a multitenant environment.

- Finally, we highlight future research opportunities and propose potential improvements to further enhance the system's utilization, efficiency, and elasticity in managing resources within virtualized environments.

1.4 Thesis Organization

The rest of this thesis is organized as follows.

Chapter 2 provides a detailed analysis on Linux memory management. It begins by explaining the fundamental aspects of memory management, including the organization of virtual and physical memory. It then delves into Linux memory reclamation techniques, focusing on mechanisms such as the page cache, LRU lists, and the Out of Memory (OOM) killer for efficient resource management.

Chapter 3 explores resource management in modern computing systems, focusing on elasticity and resource monitoring mechanisms. It begins by explaining the feature

of elasticity and its importance in cloud environments. The chapter then discusses resource monitoring through the Linux control groups (cgroups) and Pressure Stall Information (PSI) interfaces. Finally, it examines the cgroup-specific metrics, and the system-wide metrics, which provide valuable insights into the system state and help identify potentially underutilized memory pages.

Chapter 4 outlines the design of the proposed system, starting with the design goals and challenges in detecting and removing idle memory. Then, it describes key components such as the Metrics Collector, Resource Controller, and Memory Stresser, along with the synthetic workload generator used to evaluate the system.

Chapter 5 details the implementation of the proposed system, focusing on the Metrics Collector, Resource Controller, and Memory Stresser. It explains the prototype's functionality, including thread management, tenant container handling, and protecting the Memory Contention service from swapping, to ensure robust memory contention management.

Chapter 6 evaluates the proposed system through experiments conducted in various scenarios. It examines the Memory Stresser's performance with one and two tenants, its effectiveness with MapReduce applications, and its comparison with existing approaches. The evaluation emphasizes its efficiency in identifying and managing idle memory, as well as its impact on system performance.

Chapter 7 presents a review of the existing literature on memory management, idle-page detection, and memory reclamation techniques. It examines the current state-of-the-art approaches, highlighting their strengths and limitations. Additionally, the chapter compares these methods with the proposed approach, positioning its contributions within the broader context of the field to demonstrate its advantages.

Finally, Chapter 8 highlights opportunities for future research and summarizes the conclusions of our work.

Chapter 2

Memory Management in the Linux kernel

Memory is a critical resource for computing systems due to its limited amount and its importance for system performance and functionality. Hence, memory should be managed conservatively. In modern computing systems, the demand for memory is continuously growing, driven by factors such as the increasing complexity of software, the resource-intensive tasks, and the rise of big data applications.

Memory management is one of the most complex activities in an operating system. The efficient handling of the physical and virtual memory resources plays a fundamental role in ensuring the reliable and secure operation of software applications. This is especially crucial in multiprogramming systems, where memory must be optimally allocated among processes.

The goal of efficient memory management is to optimize the system performance, enhance stability, and mitigate risks such as crashes, data corruption, or security vulnerabilities. To achieve this, operating systems employ various techniques to manage memory properly, including allocation strategies, caching mechanisms [9, 10], swapping policies, and memory protection mechanisms [11].

In time-sharing systems, the available main memory is shared among various processes. Having multiple processes in memory at once, means that as long as some processes use the processor, there are other processes waiting for I/O to finish. Thus, there is an ever-increasing need for efficient resource allocation to individual processes to ensure the efficient utilization of system resources [11].

This section introduces the key concepts of memory management, laying the foundation for the subsequent discussions on specific techniques and strategies that the Linux kernel employs to manage memory effectively.

2.1 Virtual Memory

Modern systems provide a mechanism called *virtual memory*, that allows a process to address more memory than the amount of physical memory [12]. More specifically, virtual memory manages the content of the main memory when the combined size of the program, data, and stack of a process exceeds the available physical memory capacity.

Most systems implement virtual memory using paging, a memory management approach in which a process address space is organized into fixed-sized units which are called *pages*. Similarly, physical memory is divided into corresponding fixed-sized chunks called *page frames* [13]. Both pages and page frames have the same size in order to achieve optimum utilization of the main memory.

Overall, virtual memory provides several advantages as it increases memory capacity and allows memory protection because each virtual address is translated to a physical address.

2.1.1 Demand Paging

Demand paging is a memory management strategy where pages are only loaded into physical memory when they are accessed for the first time. Unlike preloading all

pages into RAM, demand paging defers memory allocation, reducing memory usage and increasing efficiency.

When a program accesses a memory page that is not currently stored in RAM and needs to bring it from the secondary storage, the system responds to the unmapped reference with an exception referred as *page fault*. In the event of a page fault, the operating system retrieves the contents of the required page from the secondary memory, picks a less frequently used loaded page frame to evict and fetches the referenced page into the discarded page frame. This process is known as *page replacement*. Various algorithms like *Least Recently Used (LRU)*, which replaces the page that has been unused for the longest time, or *First-In-First-Out (FIFO)*, which replaces the oldest loaded page, are used to decide which page to replace [13].

Once the required page is brought into memory, the page table is updated to reflect the new mapping. Demand paging, while efficient, introduces overhead due to page faults.

2.1.2 Swapping

Swapping is a similar approach to demand paging, which is used to relieve the memory pressure. Swapping is an essential technique to manage memory when the main memory capacity is not enough to hold all the currently active processes, by swapping out to disk the inactive pages in order to free up space for active processes. Swapping involves anonymous pages, which require swap partitions for storage. The pages swapped out from memory are stored in the *swap area*, which may be implemented either as a separate disk partition or as a file.

The *swap cache* is a temporary storage area used to facilitate the swapping process. It acts as an intermediary between system memory and disk storage, temporarily backing up anonymous and shared memory pages during the swapping process in order to make it easier to swap them back into memory when needed. It helps coordinate the swap-in and swap-out operations and ensures synchronization, preventing conflicts. When pages are swapped out from memory, the swap cache stores them until they are swapped back into memory when needed. This improves the overall performance of memory management during swap operations.

The Linux kernel provides the *swappiness* parameter that determines how aggressively the kernel will swap out anonymous pages relative to file pages [14]. The

parameter takes values from 0 to 200. The default value is 60. Table 2.1 shows how the parameter of swappiness determines the LRU lists sizes. The higher the swappiness value, the more likely the kernel to move anonymous pages out of memory to the swap area.

Table 2.1: Control of swapping out anonymous memory pages in Linux kernel.

Swappiness parameter	**Effect at memory shortage**
swappiness=0	The kernel avoids swapping out anonymous memory pages and discards only file pages.
swappiness=60	The kernel reclaims both anonymous and file pages but prefers to discard file pages first (default).
swappiness=100	The kernel treats anonymous and file pages equally when reclaiming memory.
swappiness=200	The kernel avoids reclaiming file pages and swaps out only anonymous memory pages.

While swapping ensures memory availability, the excessive swapping between disk and memory can lead to severe performance degradation, a phenomenon known as *thrashing* [11]. Several sophisticated algorithms, such as the working set policy [15], are designed based on the notion of the principle of locality [16] to prevent thrashing.

2.2 Physical Memory

Physical memory refers to the actual hardware memory, typically in the form of Random Access Memory (RAM), installed in a computer system. The Linux kernel employs a page-based memory organization, wherein physical memory is divided into page frames. Each page frame, typically 4 KB on x86 architectures, serves as the fundamental unit for memory allocation and management. These pages are allocated either to user mode processes or kernel tasks.

Each page frame has a *page descriptor* that holds flags indicating whether the page is locked, dirty, active/inactive, or has been reclaimed, which facilitates the kernel to keep track of its status. It also includes a reference counter that counts how many

page tables belong to that page [17].

To address the growing complexity of modern systems, Linux organizes physical memory into multiple layers and components. For large-scale systems, the Non-Uniform Memory Access (NUMA) model is used to optimize the memory access across multiple processors and nodes. Memory is further divided into zones to address hardware limitations and enable architecture-independent memory management. Additionally, the kernel employs mechanisms like the *memory watermarks* (described in section 2.2.3) to monitor the free memory levels and trigger actions like the *page frame reclamation* (described in section 2.5) when necessary.

2.2.1 Memory Nodes

A memory node is a software abstraction in Linux that represents the physical memory attached either to all processors in a Symmetric Multi-Processing (SMP) system or to a group of processors in a Non-Uniform Memory Access (NUMA) system. In an SMP system, all processors are connected to a single, shared physical memory and can access any memory block in the same amount of time. This uniform memory access ensures consistent performance across all processors. On the other hand, in a NUMA system, physical memory is divided into multiple nodes, each connected to a specific group of processors. While each processor can access both its local memory and the memory associated with other processors groups, the access time varies depending on the distance between the processor and the memory location. As a result, memory access latency increases when accessing remote memory nodes [17]. This architecture reduces memory latency and improves scalability in multi-processor systems.

2.2.2 Zones

The Linux operating system organizes the memory of each node into *zones* in order to provide an architecture-independent memory description to make the memory accessible to hardware devices. The kernel uses zones to group pages of similar properties, for instance, their physical address into memory.

More specifically, some hardware devices are capable of performing direct data transfer between memory and peripheral devices, a feature called Direct Memory Access (DMA), to only certain memory addresses, while some architectures are capable of physically addressing larger amounts of memory than they can virtually address.

Consequently, some memory is not permanently mapped into the kernel address space. The hierarchical design of nodes and zones helps Linux to handle memory efficiently, regardless of the underlying hardware architecture.

Zones in the Linux kernel are categorized into three types [2], *DMA*, *Normal*, and *High Memory*. The DMA zone includes memory frames in the lower regions of physical memory, required by a number of legacy hardware devices. The memory frames within the Normal zone represent directly addressable memory to the kernel. This is where most kernel operations occur, for instance, the allocation of kernel data structures. Finally, the memory frames within the High Memory zone refer to memory which is not directly addressable by the kernel, but it is used for user-space allocations.

2.2.3 Memory Watermarks

Each zone has a *zone descriptor* that keeps track of the page usage statistics in the zone, including important information for page frame reclamation such as the determination of the memory *watermarks*. Each memory zone has defined three watermark levels, the *minimum*, *low* and *high*, which are thresholds used to manage memory pressure and availability.

The minimum watermark (*wm_min*) is a minimum number of free pages that the system must keep available in the zone (*min_free_kbytes*) in order to satisfy allocations of kernel data structures without significant performance degradation. The low watermark (*wm_low*) is slightly higher than the min watermark and defines an amount of memory to satisfy allocations while the *background* memory reclaim operations are triggered. The high watermark (*wm_high*) is an amount of memory above the low watermark, indicating a healthy level of free pages that allows the system to handle the peak allocations.

The watermarks are a proportion of the total memory pages managed by the zone, called *zone_managed_pages*. The distance between the low and high watermarks, called *wm_distance* changes based on the *watermark scale factor (wsf)* parameter.

The watermark scale factor defines the amount of memory remaining in the system before the kernel needs to start reclaiming memory and the amount of memory that should be reclaimed.

Table 2.2 provides more details about the formulas used for the calculation of

watermarks by the Linux kernel. We further describe how these watermarks trigger the Linux page frame reclamation in section 2.4.

Table 2.2: The formulas used for calculating the three watermarks in Linux kernel [2].

Watermark	**Formula**
wsf	10 (Default)
wm_distance	(zone_managed_pages * wsf) / 10000
wm_min	min_free_kbytes
wm_low	wm_min + wm_distance
wm_high	wm_min + 2*wm_distance

2.3 The Page Cache

The Linux kernel implements a *page cache* in main memory to store frequently used data and reduce disk I/O operations. The page cache consists of physical pages requested for reading or writing using the system *read()* and *write()* calls, or file system data mapped to virtual memory with the *mmap()* system call [18].

When performing an I/O operation, the kernel first checks the cache for the data. In the event that the requested data is available within the cache, the kernel promptly retrieves it from memory without necessitating a read from the disk. Conversely, in the absence of data within the cache, the kernel initiates a read from the disk and subsequently stores the data within the cache for expedited access in the future.

Regarding write operations, the kernel employs two distinct strategies: the *write-through* strategy and the *write-back* strategy. The write-through strategy updates both the page in the page cache and the file on the disk with each write operation. In contrast, the write-back strategy performs write operations directly in the page cache and incrementally writes changes to disk [19].

The Linux kernel maintains multiple flusher threads that are responsible for flushing the dirty pages of the disks assigned to them. This feature enables data to be flushed to multiple disks at varying rates. The kernel oversees the *per-backing device info (BDI)* structure, which includes disk-specific details such as the list of dirty pages,

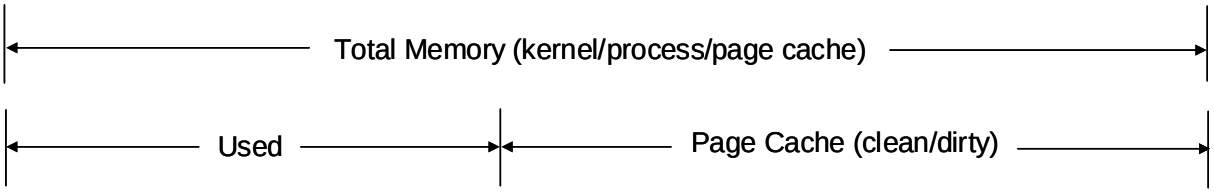

Figure 2.1: Linux Memory Management.

the amount of memory that can be prefetched into memory (read ahead), and flags. There are several parameter that can be used for cache tuning through userspace interfaces [18].

Although the utilization of page cache enhances the overall performance of the system, it is derived from the same memory pool that is utilized by the remainder of the system. The kernel allocates all currently free memory to the page cache as shown in Figure 2.1. As the kernel is required to allocate additional memory for other tasks, it is able to reclaim pages from the page cache, as the contents of the page cache can be restored from disk blocks when necessary.

2.4 Page Frame Reclamation

The Linux *Page Frame Reclamation Algorithm (PFRA)* is responsible for selecting old pages that can be freed and reclaiming them when almost all free memory is allocated to processes or caches. The objective of the PFRA algorithm is to evict from memory allocated page frames that are not frequently used, thus freeing memory for reuse by other processes. The pages selected by the PFRA algorithm are page frames that can be freed, so the selection of candidates for eviction considers the page type.

The Linux Page Frame Reclamation Algorithm (PFRA) is initiated when there is a shortage of free memory. The physical memory recovery process is performed by page *reclamation*, the process of evicting idle memory pages to disk.

2.4.1 Page Types

A physical page can be used for storing different types of data, such as kernel data structures, buffers for device drivers, data from a filesystem, or memory allocated by user space processes.

Memory pages are categorized into *unreclaimable* and *reclaimable*. Unreclaimable

pages are pages pinned in memory by a process, temporarily locked pages, or pages used by the kernel for specific purposes, such as kernel mode stacks. Reclaimable pages are pages that can be freed at any time, for instance, anonymous pages that can be moved between physical memory and disk, parts of files on disk that are mapped to memory using the *mmap* system call, or unused pages in the page allocator [19].

Two distinct types of userspace pages are the *anonymous* pages and the *file* pages. Anonymous pages are not associated with any files on disk and are used by user space programs. When an anonymous page is reclaimed, the kernel must store its contents in a disk partition or file called swap space. File pages, on the other hand, are brought into memory by buffered I/O operations or memory mapping of part of a file. These pages can be discarded immediately if they are considered clean, or after writing their contents to the appropriate disk file if they are considered dirty [19].

2.4.2 Page Frame Reclaim Initiation

The Linux Page Frame Reclaiming Algorithm (PFRA) is activated under specific circumstances to manage memory efficiently and prevent system performance degradation. These scenarios can be broadly categorized into low memory conditions, hibernation, and periodic maintenance [19].

Firstly, *background* or *periodic reclaiming* occurs as part of the regular maintenance to ensure the system remains healthy and responsive. The kernel employs dedicated threads for this purpose, called *kswapd*. The kswapd threads monitor the memory zones and initiate reclamation when the number of free pages falls below the *low* watermark.

Secondly, the *direct* or *low-on-memory reclaiming* is triggered when the kernel detects that memory resources are critically low. This occurs in situations where functions responsible for memory allocation fail to secure the necessary resources. These failures prompt the PFRA to reclaim memory urgently to meet the allocation demands.

Lastly, the *hibernation reclaiming* is activated when the system is preparing to enter a suspend-to-disk state, saving the contents of RAM to storage and restoring them when the system is resumed.

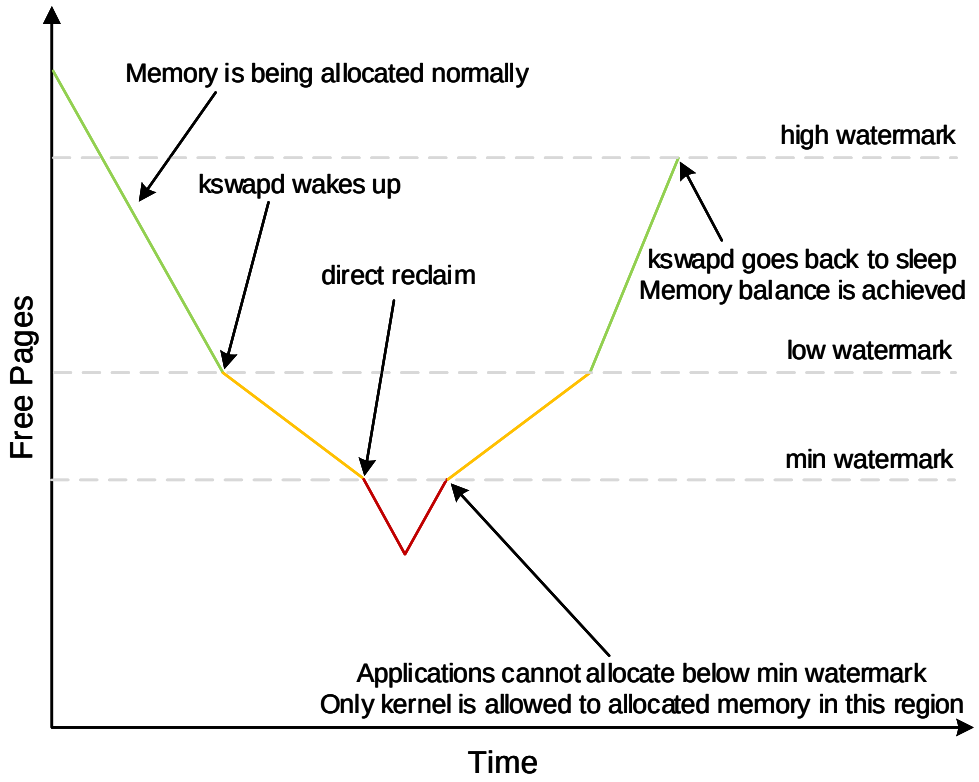

Figure 2.2: The Zone Watermarks [2].

2.5 Memory Reclamation

The Linux kernel relies on page frame reclaiming to address the memory pressure. The PFRA ensures that the Linux kernel maintains a balance between resource availability and system performance across diverse workloads.

When the amount of free memory falls below the low watermark, memory pages are asynchronously reclaimed in the background by the *kswapd* thread. The kswapd thread goes back to sleep when the number of free pages reaches the high watermark. The Linux kernel uses the watermark scale factor to control the execution of kswapd by defining the distance between the low and high watermark [14].

On loaded systems where the memory page allocations are performed faster than page reclamation by the kswapd threads, the amount of free memory continues to fall below the min watermark. In the above situation, each memory allocation request forces the kernel to perform a *synchronous* reclamation before the request can be satisfied. The synchronous method is considerably slower due to the blocking caused by waiting for memory to be released. Figure 2.2 illustrates the memory zone watermarks that trigger the actions described above for memory reclaiming.

The Linux kernel tries to load as much data as possible into memory for an appli-

cation. Consequently, the available memory is exhausted due to caching file contents (page cache) as well as kernel objects, for faster access. During page frame reclamation, the Linux kernel tries to evict a mix of userspace pages and kernel objects. Anonymous pages are *swapped out*, while file pages cached in page cache must be written back to disk if they are *dirty* or immediately discarded when they are clean. Historically, file pages have been prioritized for reclaiming over anonymous pages.

Eventually, reclaiming a page frame executes disk I/O operations resulting in the increase of latency and application performance degradation. Therefore, it is advantageous for the PFRA to steal pages that are unlikely to be accessed again in the near future. Predicting the probability that a page will be accessed again in the near future is a challenging operation. The Linux PFRA algorithm uses *temporal* locality, which assumes that recently accessed pages are more likely to be accessed again.

2.5.1 The Least Recently Used (LRU) Lists

The Linux kernel uses a set of Least-Recently-Used (LRU) lists as core data structures for implementing its page replacement policy and prioritizing pages for eviction during memory reclaim. These lists are organized within a data structure called *lruvec* [3].

The kernel maintains two linked LRU lists: the *inactive* list and the *active* list. Pages in the active list are recently accessed (”hot”), while those in the inactive list have not been accessed for a certain period of time (”cold”). The page frame reclaiming algorithm primarily targets pages in the inactive list for removal. Pages can transition between these lists based on access patterns, they may be moved, remain on the same list, or be evicted during reclamation.

Separate active and inactive LRU lists are maintained for file and anonymous memory pages, reflecting their distinct properties [3]. In addition, a separate list is maintained for unevictable pages, which cannot be reclaimed. Each NUMA node and memory *cgroup* maintains its own *lruvec*, along with a dedicated kswapd thread for memory reclamation.

The Linux kernel uses a number of page *flags*, as shown in Table 2.3, to determine whether a page frame should be activated or deactivated [19, 3]. Pages are dynamically moved between the active and inactive lists based on their access patterns. For instance, intermittently accessed pages may oscillate between the two lists, making it

Table 2.3: Page states relevant to reclaim [3].

Page Flag	**Page State**
LRU	Page is on any LRU list.
Active	Page is on active list.
Referenced	Inactive page has been accessed recently.
Workingset	Page is considered part of active userspace's workingset.

challenging to predict behavior. To address this, the kernel uses the *referenced* flag to track recent accesses. Multiple accesses within a defined interval are required to move a page from the inactive list to the active list. Similarly, the referenced flag increases the tolerance for missed accesses before moving a page from the active list to the inactive list.

Page frames are initially stored in the inactive list with their *active* and referenced flags set to 0. When a page is accessed, its referenced flag is set to 1, but the page remains in the inactive list. If the page is subsequently accessed again, it is moved to the active list with its active flag set to 1 and its referenced flag reset to 0. Conversely, if the page in the inactive list is not accessed within a time frame, its referenced flag is reset to 0. Similarly, a subsequent access of a page that is recently moved to the active list sets its referenced flag to 1. Thus, the active list ends up containing pages of the *working set*, and the inactive list contains pages that are candidates for eviction.

In more detail, the kernel provides a number of key functions involved in managing pages in the LRU lists. The functions *mark_page_accessed()*, *page_referenced()*, and *refill_inactive_zone()* handle transitions between the LRU lists by evaluating the active and referenced flags.

The *mark_page_accessed()* function is called whenever the kernel determines that a page has been accessed, such as demand paging, file reads, or buffer cache lookups. The function moves the page to the active LRU list by invoking *activate_page()* and clears its referenced flag if is already set. Otherwise, it marks the page as referenced.

The *refill_inactive_zone()* function is triggered when memory is under pressure and caches need to shrink. It moves pages from the active list to the inactive list to balance the length of both lists. Initially, the function scans a small number of pages in the

active list, but as memory pressure increases, it scans more pages. Pages are taken from the end of the active list. If the referenced flag is set, the flag is cleared, and the page is moved to the top of the active list as it is still ”hot”. If the flag is not set, the page is moved to the inactive list, and the *referenced* flag is set for quick promotion to the active list if the page is accessed again.

The *page_referenced()* function assesses whether a page has been recently accessed by checking the *referenced* flag. Pages with this flag set remain in the active list, while others are moved to the inactive list by the *refill_inactive_zone()* function.

2.5.2 The Out Of Memory Killer

The *Out of Memory (OOM) Killer* is a mechanism in the Linux kernel designed to handle extreme memory pressure situations where the system runs out of available memory. Despite the Page Frame Reclaiming Algorithm (PFRA) efforts to free up memory, there are cases where all swap space is full, and disk caches have been completely shrunk, leaving the system unable to allocate memory. In such scenarios, the system could freeze entirely as processes compete for memory without success.

To prevent a complete system halt, the OOM Killer intervenes by forcibly terminating one or more processes. It selects a “victim” process to kill based on specific criteria to minimize disruption to the system. The chosen process is usually one that uses a large amount of memory, has low priority or is less critical, to avoid leaving the system in an unstable state.

The OOM Killer uses an internal scoring system to evaluate processes and identify the best candidate for termination. Once a victim is selected, the kernel sends a termination signal (typically *SIGKILL*) to that process, freeing up its memory. This drastic action ensures that other processes and the system itself can continue to function, even in the face of severe memory shortages.

2.6 Summary

Memory management ensures efficient resource utilization and system stability. Techniques like virtual memory, demand paging, and swapping optimize memory usage, while strategies such as page replacement and swappiness help balance resources and prevent thrashing.

Linux organizes physical memory into page frames, memory nodes, and zones for efficient management. The page cache reduces disk I/O, and memory watermarks regulate memory reclamation to maintain system stability.

The Page Frame Reclamation Algorithm (PFRA) in Linux is responsible for freeing up memory during periods of memory pressure. Linux uses an LRU-like approach to manage physical memory by organizing pages into active and inactive lists for both anonymous and file cache pages. The active lists contain recently used pages, while the inactive lists store less frequently accessed pages.

When the system experiences memory pressure, the kernel scans these lists, updates the page flags, and moves pages between the lists to identify which pages to reclaim. Memory reclamation is triggered by three watermarks: high, low, and minimum. When memory falls below the low watermark, the kernel initiates a background reclamation process, where the kswapd thread wakes up to free pages until the memory reaches the high watermark. If memory drops below the minimum watermark, the system forces each memory allocation request through a synchronous reclamation process before proceeding with the allocation. If PFRA is unable to free enough memory, the Out of Memory (OOM) Killer is invoked to terminate processes and prevent a system crash.

Chapter 3

Resource Monitoring and Metrics

Resource management is a core function of computer systems. Its primary goal is to efficiently allocate resources such as CPU, memory, storage, and network bandwidth to maximize performance and stability.

Three objectives in resource management are the achievement of high resource utilization, system performance and fairness among competing users. Resource management dynamically observes and adjusts resource allocations to avoid bottlenecks, reduce contention, and handle workload variations. This is particularly important in modern computing paradigms like cloud platforms, where multiple tenants and applications share resources.

3.1 Elasticity

Elasticity is a core feature of cloud computing, which refers to the ability of cloud services to provision, deprovision and reconfigure computing resources on demand. Efficient resource handling is a key aspect to make providers improve their resource

utilization which allows the serving of more users on each machine, increase the return on investment using the same hardware, and reduction of the need to expand their infrastructure.

Cloud resource elasticity can be achieved either through *vertical* scaling (scaling up/down) or *horizontal* scaling (scaling out/in) [17]. Vertical scaling means scaling the capacity of a single server by changing the resources assigned to an already running instance, such as allocating processor cores or memory to a running virtual machine or container. Horizontal scaling means scaling by adding or removing instances based on load, either increasing or decreasing the number of virtual machines or containers as demand changes.

In cloud memory management, auto-scaling systems help optimize resource utilization by dynamically adjusting the allocated resources based on workload demands. These systems rely on predefined policies, adjusting resources through either horizontal or vertical scaling. However, auto-scaling faces challenges such as underprovisioning, overprovisioning, and resource waste.

Overprovisioning occurs when more resources are allocated than the actual workload demands, leading to applications consuming more memory than the number of pages actually needed resident for their normal operation.

Thus, there is a need to prevent issues such as overprovisioning, which can lead to inefficient resource utilization and unnecessary costs. To achieve this, it is essential to identify and reclaim the underutilized resources effectively. Accurate and appropriate resource monitoring is crucial for making informed decisions on when and how to reclaim resources, ensuring high system performance.

3.2 Resource Monitoring

In this section, we explore the Linux kernel interfaces, Control Groups and Pressure Stall Information, that provide various statistic information about the system state and the system resources utilization.

3.2.1 Control Groups (Cgroups)

Control groups (cgroups) [20] is a kernel mechanism that limits, accounts for, and isolates the resource usage of a collection of processes. A cgroup is a collection of

subsystems or *controllers*, that each one controls a single type of resource, such as CPU, memory, disk IO, and network devices.

The cgroups mechanism allows processes to be organized into hierarchical groups, where child cgroups inherit certain attributes from their parent cgroup. Cgroups can be used for a wide range of tasks, including isolating resources for specific applications, managing the resources consumed by users, and controlling the overall performance of the system.

The kernel's cgroup interface is provided through a pseudo-filesystem called *cgroupfs*. This filesystem is used to export the control group hierarchy and the associated settings and statistics as a set of virtual files and directories.

The version 1 of cgroups (cgroups v1) [20] organizes processes into multiple hierarchies and attaches controllers separately to each hierarchy. Consequently, one can create a hierarchy and attach the memory controller to it. Another hierarchy can be created for the CPU controller, which could contain the same processes as the first hierarchy. However, it is expensive for the kernel to track the controllers that apply to processes that are assigned to multiple hierarchies. Moreover, controllers cannot effectively cooperate with each other, since they typically operate on different hierarchies.

Version 2 of cgroups (cgroups v2) [21] introduces a unified hierarchy for all controllers. Controllers can be enabled or disabled separately for specific subtrees of the single hierarchy. Processes can be attached only as leaves to each subtree of the hierarchy. Below, we provide a brief description of the most important controllers available in cgroups v2.

Cpu. This controller manages how the scheduler shares CPU time among different process groups and can be used to limit or prioritize the CPU usage of specific processes or groups of processes.

Cpuset. This controller pins a subset of processors and memory nodes to the processes in a cgroup. This binds each process in a group to run and allocate memory on this subset of CPUs and memory nodes.

Memory. The memory controller is responsible for setting limits on memory usage by processes in a cgroup. Keeping track of which areas of memory are in use and which are not, the memory controller also provides automatic reports on memory resources used by those processes. Currently, the following types of memory usages are tracked: i) userspace memory - page cache and anonymous memory, ii) kernel

data structures such as dentries and inodes, and iii) TCP socket buffers.

The memory controller [22] relies on four interface files to control memory limits. The `memory.low` is the best-effort memory protection, a "soft guarantee" that if the memory usage of the cgroup is below this threshold, the cgroup's memory will not be reclaimed unless memory can't be reclaimed from any unprotected cgroups. The `memory.min` specifies hard memory protection, a minimum amount of memory the cgroup must always retain, i.e., memory that can never be reclaimed by the system. If the memory usage of a cgroup reaches this limit and can't be increased, the system OOM killer is invoked. The `memory.high` is the main mechanism to control the memory usage of a cgroup, a memory usage throttle limit that if a cgroup's memory use goes over this boundary, the cgroup's processes are throttled by the kernel and put under heavy reclaim pressure. The default value is max, which means there is no limit. The `memory.max` is the final protection mechanism, a memory usage hard limit that if a cgroup's memory usage reaches this limit and can't be reduced, the system OOM killer is invoked on the cgroup. Under certain circumstances, usage may go over the memory.high limit temporarily. Cgroup v2 also introduce two new parameters, `memory.swap.max` and `memory.swap.high`, to limit swap memory usage.

Io (Blkio in cgroups v1). The IO controller regulates the distribution of IO resources. It can set both weights and absolute bandwidth limits for each disk or block device. It also allows the rate of I/O operations for each process group, by limiting the number of read/write operations and bytes.

Pid. This controller limits the number of processes that can be created by a specific process group. This is useful in a situation where a group of tasks are potentially running infinite loop, causing a fork bomb attack.

3.2.2 Cgroup Metrics

Cgroup metrics provide a detailed view of memory pressure within a container, but they do not directly indicate memory that is allocated but underutilized. We analyze the metrics provided by the `memory.current` and `memory.stat` files of the Linux cgroup v2 interface [21] that help to identify the underutilized memory pages. Table 3.1 summarizes these metrics.

The `memory.current` file represents the total memory currently used by the cgroup and its descendants, including the page cache size. This metric accounts for the phys-

Table 3.1: Metrics provided by the cgroup v2 interface.

Metric	**Description**
memory.currect (RSS)	The total physical memory actively used by processes executing in the cgroup.
active LRU list size	The size of memory pages currenlty in use by processes within the cgroup.
inactive LRU list size	The size of memory pages accessed less frequently by the processes within the cgroup.
swapcached	The anonymous memory pages that have been swapped out to disk but also cached in memory.
workingset_refault_anon	The number of anonymous memory pages that were previously swapped out and then requested again.
workingset_refault_file	The number of file pages that were previously evicted to disk and then requested again.
pgscan	The number of pages scanned from the inactive LRU list.
pgsteal	The number of pages reclaimed from the inactive LRU list.
pgscan_kswapd	The number of pages scanned from the inactive LRU list by the kswapd thread.
pgsteal_kswapd	The number of pages reclaimed from the inactive LRU list by the kswapd thread.
pgscan_direct	The number of pages scanned from the inactive LRU list through direct reclamation.
pgsteal_direct	The number of pages reclaimed from the inactive LRU list through direct reclamation.
page faults	A page fault occurs when a process accesses memory that is not currenlty mapped to its address space in RAM.
major page faults	A major page fault occurs when the program attempts to access a page that is not in RAM, requiring the kernel to fetch it from disk.

ical memory actively used by processes, including their Resident Set Size (RSS), which refers to the portion of memory held in RAM. In addition to RSS, memory.current also includes in-kernel data structures, and network buffers. Changes in this statistic can indicate new activity within the cgroup, such as increased memory usage, or alterations in the application's working set.

The `memory.stat` file provides detailed memory-related statistics for the cgroup. Among these, specific metrics offer insights into underutilized memory within the cgroup and potential indications of memory pressure.

In particular, monitoring the size of the Least Recently Used (LRU) lists maintained by the operating system for each cgroup can help identify potentially idle memory pages. The *active* LRU list contains pages currently in use by tasks within the container, while the *inactive* list holds pages accessed less frequently and prioritizes them for eviction if the system runs low on memory. File-backed memory pages are moved to the active list as soon as they are accessed, even if the container is not under memory pressure. In contrast, anonymous pages remain on the inactive LRU list until the container's memory limit is reached. When this limit is reached, the page reclamation process triggers, and the kernel moves a portion of pages marked with the active flag to the active LRU list.

The *swapcached* metric refers to anonymous memory pages that have been swapped out to disk but are also cached in memory. The kernel keeps a number of swapped-out pages into memory to save I/O operations on subsequent access to those pages by applications. An increased swapcached value indicates that memory pressure occurred previously, but the system had already swapped out and cached these pages to avoid further I/O.

The *workingset_refault_anon* statistic represents the number of anonymous pages that were previously swapped out and then requested again. Similarly, the *workingset_refault_file* statistic represents the number of file pages that were previously evicted to disk and then requested again. Thus, refaulted back in pages are useful pages for the application that the kernel pushed out of memory and accessed again. These metrics can provide an indication that the system has already removed the idle memory pages. We use these metrics to measure the performance degradation occurred after the *kswapd* reclamation has started to reclaim memory pages.

Statistics such as *pgscan*, *pgsteal*, *pgscan_kswapd*, *pgsteal_kswapd*, *pgscan_direct*, and *pgsteal_direct* provide insights into into memory pressure within a container. These

metrics track the total number of pages scanned and reclaimed from the inactive LRU list, distinguishing between pages handled by the *kswapd* thread and those managed by direct reclamation. More specifically, *pgscan* and *pgsteal* measure the total number of scanned pages and reclaimed pages, respectively. The *pgscan_kswapd* and *pgsteal_kswapd* statistics track the amount of scanned and reclaimed pages by the *kswapd* thread when the background VM's memory reclaim logic is triggered due to a memory shortage in a zone. Conversely, *pgscan_direct* and *pgsteal_direct* track the amount of scanned and reclaimed pages through direct reclamation when the memory cgroup reaches its memory limit. Together, these counters signal the onset of memory pressure and reflect the additional memory the application may require. However, these metrics capture past memory events rather than predicting future needs.

Other event counters provided by Linux *cgroup v2* include counts for *page faults* and *major page faults*. A page fault occurs when a program accesses memory that is not currently mapped to its address space in RAM. Page faults can be further categorized into *minor* and *major* faults. A minor page fault can occur when a process accesses either a shared memory page or a newly allocated page using *malloc* or *mmap*. In the first case, the faulted page is already loaded into physical memory by another process but has not yet been mapped to the address space of the faulting process. In the second case, the page has not been allocated by the kernel yet, as the kernel defers the allocation of each page until it is first accessed (demand paging [19]). A major page fault occurs when the program attempts to access a page that is not in RAM, requiring the system to fetch it from secondary storage, such as a disk or swap space. Major page faults can significantly impact performance, as accessing data from disk is much slower than retrieving it from RAM.

The major page fault count, along with the workingset_refault count (which tracks pages that were swapped out and later accessed again) can indicate memory pages that the kernel fetches from disk and are parts of an application's working set. These metrics are useful for identifying performance degradation, especially if a significant number of major page faults occur, indicating that the system is struggling to maintain enough pages in physical memory for active processes.

3.2.3 System Metrics

The kernel maintains system-wide metrics in the *procfs* filesystem. These metrics are essential for monitoring memory usage and the overall state of a system. Key system-wide metrics from the `/proc/meminfo` file include, the *MemTotal*, the *MemFree*, and the *MemAvailable*.

MemTotal represents the total amount of physical memory (RAM) that is usable by the system.

MemFree indicates the amount of memory left unused by the system. However, this metric does not include memory allocated to caches and buffers, which can be reclaimed if necessary. This may cause *MemFree* to underestimate the actual available memory for new workloads.

MemAvailable provides an estimate of the total memory available for allocation to new processes, without invoking swapping. This metric accounts for free memory pages, reclaimable memory used by caches and buffers, kernel-managed memory pages, including slabs and memory watermarks.

3.2.4 Pressure Stall Information (PSI)

Pressure Stall Information (PSI) [23] is a feature of the Linux kernel's scheduler that provides pressures metrics that represent the amount of lost work due to the lack of a system resource (memory, CPU, and IO).

PSI in conjunction with cgroup v2 metrics allows the detection of resource shortages. When resources are depleted, processes start to compete for them and resource contention scenarios occur. PSI metrics can enable immediate reactions, including the pause or kill of non-essential processes, reallocation of memory among different tasks of the system, load scheduling, or other actions.

PSI can be calculated for a single process, a container, or the entire system. It introduces two pressure indicators. The "some" indicator tracks the percentage of time when at least one process is stalled for a resource, while the "full" indicator tracks the percentage of time when all processes are delayed simultaneously.

PSI tracks memory pressure by recording specific events that occur when there is a shortage of memory. These events include i) reclaiming pages when memory is full and attempting to allocate new pages, ii) reclaiming pages when a limit is reached on the memory controller of a cgroup, iii) waiting for IO after a major fault against

a page that was recently evicted from the page cache, and iv) blocking on reading a page from the swap device.

Calculating IO stalls on block devices accurately is challenging due to lack of device contention measurements from existing hardware. Therefore, any process waiting on block IO completion is considered stalled due to increased storage I/O response times.

CPU stalls are accounted for as the periods when a process is runnable but needs to wait for an idle CPU. CPU full pressure metric indicates that none of the processes can execute due to external competition or configured CPU cycle limits in the cgroup.

While PSI provides valuable insights into resource contention and the overall system pressure, a more granular view of resource utilization and management can be obtained through cgroup metrics provided through the `memory.stat` file described in section 3.2.2. By analyzing metrics specific to control groups, administrators can not only understand the broader resource pressure but also pinpoint the performance and resource consumption patterns of individual containers or processes.

3.2.5 The Senpai user-space agent of PSI

Senpai [23] is a user-space agent designed by Meta to manage memory pressure adaptively, aiming to maintain a low PSI (Pressure Stall Information) threshold. The design of Senpai focuses on dynamically determining how much memory to reclaim based on workload demands and hardware characteristics.

The first version of Senpai, available as open-source software [24], runs as a daemon at the host and continuously monitors the memory pressure of a target cgroup. Based on the memory pressure statistics it dynamically adjusts the cgroup high memory limit to control memory reclamation. Lowering the limit forces the kernel to reclaim memory, while increasing it relieves pressure and allows the workload to expand. However, this approach introduces issues with dynamic workloads, because the processes of a container may become blocked if their memory usage grows faster than the rate at which Senpai increases the limit.

To address this limitation, TMO [23] introduces the memory.reclaim cgroup control file to direct the kernel's reclamation algorithm to request specific amounts of memory to be reclaimed while leaving the kernel to decide which pages to release. This stateless mechanism enables precise memory reclamation without modifying the memory limit, ensuring that expanding workloads are not unintentionally stalled

while still allowing efficient memory management.

The second version of Senpai monitors the memory pressure of a cgroup. When it reaches a threshold, it calculates the amount of memory that should be reclaimed and performs a reclamation request to the kernel. The kernel responds to the request by reclaming the requested amount of memory. Senpai determines the amount of memory to reclaim by using a formula that considers the current memory usage, the memory pressure, and a reclaim ratio [23]. It calculates the memory to reclaim based on the pressure stall information (PSI), which measures how much time processes are waiting for memory. The formula compares the current PSI value (psi_some) with a predefined threshold (psi_threshold) to assess memory pressure. If the pressure is high, Senpai reclaims less memory, while if the pressure is low, it reclaims more memory, adjusted by the reclaim_ratio. This approach allows Senpai to reclaim memory dynamically, scaling the amount of memory to be freed based on the system's memory pressure and the configured reclaim ratio.

3.3 Summary

Resource management strategies like task scheduling, load balancing, and dynamic scaling are essential for optimizing resource usage while preventing resource bottlenecks and contention. In cloud computing, elasticity, through both vertical and horizontal scaling, enables cloud providers to provision and adjust resources on demand.

Resource monitoring plays a key role in managing elasticity. The Control Groups (Cgroups) and Pressure Stall Information (PSI) kernel mechanisms help to track resource utilization and detect contention. Cgroups enable fine-grained control over CPU, memory, and IO resources by organizing processes into groups and applying limits, while PSI quantifies resource pressure by monitoring stalls in CPU, memory, and IO.

More specifically, cgroup metrics provide detailed insights into memory usage and resource pressure, such as memory reclaim events, page faults, and swap activity. System metrics, including total, free, and available memory, offer a comprehensive view of the system's memory state. Besides, both cgroup and system-wide metrics feature potentially underutilized memory resources that could be reclaimed. Overall, effi-

cient resource management and monitoring are fundamental for maintaining system performance, optimizing resource usage, and responding to changing workloads.

Chapter 4

Design

Efficient memory management is critical in containerized environments, especially in data centers where resources are shared among multiple applications. Traditional methods, such as direct memory reclamation, often lead to performance issues because they block application execution while freeing memory. To address this challenge, we propose the Contention Service, a tool that proactively reclaims unused memory without causing delays for applic«ations.

The Contention Service monitors the memory usage in the containers and triggers the kernel's background memory reclamation process, kswapd, to free up memory that is no longer actively used. By generating controlled memory contention, the system ensures that memory reclamation does not interrupt the running applications. It operates transparently, requiring no modifications to the applications themselves, and incurs minimal overhead during operation.

In this chapter, we describe the design of the Contention Service, detailing how it utilizes the memory statistics from the Linux kernel to identify and release idle memory. We also explain how its key components, the Metrics Collector and Resource Controller, work together to monitor memory usage. Furthermore, we outline the conditions under which the Resource Controller triggers the Memory Stresser component to generate the necessary contention into memory in order to activate the kernel's reclamation process and how it terminates the process to prevent adverse effects on application performance.

4.1 Design Goals

We set the following goals to the design of our system:

1. **Utilization.** Increase useful utilization and reduce overprovisioning to achieve lower cost of ownership of the infrastructure.

2. **Efficiency.** Identify opportunities for reducing the resident resources from running applications without excessive cost for statistics gathering and resource reclamation.

3. **Compatibility.** Reclaim resources without requiring modifications of the applications or the underlying system kernel. Additionally, there should be no requirement for explicit interaction with the applications regarding the monitoring of the utilized resources or their return back to the system.

4. **Elasticity.** Dynamically adjust the resident resources of each application according to the current real needs rather than those reserved.

5. **Performance.** Resource reclamation should not negatively affect the measured performance of an application in comparison to the default system operation.

4.2 Challenges in Detecting and Removing Idle Memory

Reviews of profiled memory usage in data center applications [25, 23, 6] reveal that the memory usage of a container process is typically below the assigned memory to the container. This indicates that the applications running in the tenant containers may allocate a large amount of memory that is potentially idle. The removal of idle memory can be a challenging task, as it may result in severe performance degradation.

Existing solutions identify and reclaim an application's idle memory pages using techniques that require kernel modifications [25, 26, 27, 28] to track the age of pages. A widely adopted approach among existing solutions is the idle page tracking technique [28, 29] which detects the set of pages that are actively used by the workload by frequently monitoring the accessed bit set in the page table entries. This technique results in significant CPU and memory overheads.

Other approaches [23, 30, 6] add or remove memory from containers by resizing their memory limits based on the memory needs. This technique triggers the kernel's direct reclamation process on each container upon memory limit shrinking to reclaim the difference. The direct reclamation process stalls the applications execution due to the lack of memory resources.

The key objective of our solution is to avoid direct memory reclamation, which subjects containers to considerable memory pressure and results in memory thrashing, thereby slowing down the applications response time. We propose the preemptive triggering of the global reclamation process by applying pressure on the system. The pressure that we apply causes the amount of free memory pages to drop below the low memory watermark. The global reclamation process wakes up a *kswapd* thread per container running on the system and scans pages in the inactive LRU lists to reclaim memory pages. Moreover, unlike direct reclaim, the asynchronous reclaim through *kswapd* allows the applications to continue running without being blocked by the memory management tasks and leads to more balanced and effective memory reclamation by taking into consideration the memory usage patterns of all processes.

4.3 Metrics

The identification of idle memory pages within a container is not a trivial task. Existing solutions rely on cgroup or system-wide metrics to detect and respond to memory

pressure on a container. Common approaches include usage of the Pressure Stall Information (PSI) feature [23], tracking the number of major page faults [25, 31] and refaults [25] occurring within specified time frames, as well as monitoring tasks' memory utilization statistics [32] and application-reported latency and throughput metrics [33]. While these methods provide valuable feedback, they are primarily reactive, focusing on addressing memory pressure after it occurs, rather than proactively identifying and reclaiming underutilized memory.

Cgroup metrics offer a granular view of memory pressure events within a container. Most memory statistics exposed through the cgroup interface begin counting when a container reaches its high memory limit and starts experiencing memory pressure. However, these metrics do not clearly indicate the amount of memory that is allocated but not actively used during runtime.

To address this, we leverage specific statistics provided by the Linux *cgroup v2* interface [21] to identify potentially "cold" memory pages and manage memory pressure effectively, aiming to prevent performance degradation in applications. By analyzing these metrics, we can identify underutilized memory within a container and proactively manage memory pressure. This approach optimizes the memory utilization while minimizing the risk of application performance degradation.

4.4 Contention Service Overview

The *Contention Service (CS)* runs within a Linux container, with the primary objective of releasing idle memory pages allocated by applications by proactively triggering the Linux kernel's memory reclaiming process. Figure 4.1 shows a high-level overview of the Contention Service system.

Built on top of Linux Control Groups, the contention service includes the *Metrics Collector (MC)* component responsible for retrieving per-container memory usage statistics from the Linux cgroup interface files and system-wide memory usage statistics through the proc filesystem. Measurements are taken on demand by the Contention Service, which currently supports the release of underutilized memory resources.

The *Resource Controller (RC)* module continuously monitors memory resource usage statistics provided by the MC component to activate the *Memory Stresser (MS)*

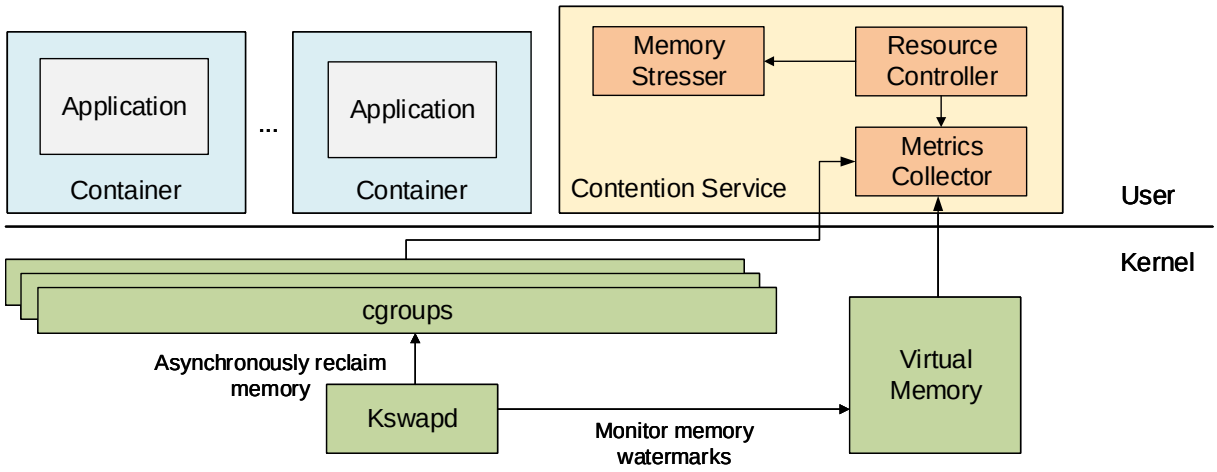

Figure 4.1: An architectural overview of the Contention Service.

process. The MS allocates the necessary amount of memory, in order to prompt the kernel to trigger the background memory reclamation process. The RC is also responsible for terminating the pressure activity if it begins to negatively impact the application's performance. The RC decides if the MS needs to be started by examining the Resident Set Size (RSS) and the size of the inactive Least Recently Used (LRU) lists of the tenant containers. More specifically, it tracks the sizes of the inactive LRU lists of both anonymous and file-backed memory and compares them with the RSS size. This helps the RC to identify changes in the working set of the tenant containers, in order to spawn the Memory Stresser process.

The *Memory Stresser (MS)* is a multi-threaded process that allocates an adaptive amount of memory, referred to as *contention memory*. We define the size of contention memory as *contention memory size*. By default, the contention memory size is dynamically updated based on the system memory usage and equals the difference between the free memory and the low memory watermark. The purpose of contention memory is to create sufficient pressure to force the activation of the Linux kernel's *kswapd* page reclamation process. The MS continuously accesses this memory until a configurable number of refaults occur (by default 4% of the RSS). The *memory contention size* refers to the total amount of allocated contention memory at any given time. It represents the minimum free memory required to avoid triggering synchronous direct page reclamation.

The *Memory Stresser Threads (MSTs)* are responsible for dynamically allocating and repeatedly accessing the contention memory. The contention memory is initially divided into equally sized portions, with each portion assigned to a separate thread.

Each thread allocates its assigned portion of contention memory and activates it by touching the first three bytes of each block. This action forces the kernel to mark the page as referenced, setting the accessed bit in the page table entry. As a result, the page remains in the active LRU list during the page frame reclamation process. After completing this activation, the thread waits until all other threads have finished their allocations and activations.

Once all threads have completed the initiation phase, each thread begins accessing its assigned portion of memory in chunks (the size of each chunk is equal to the page size of the system). This ensures that the memory remains active and is continually recognized by the system.

To prevent performance degradation, the threads periodically assess the system's memory pressure. At regular intervals, after accessing a configurable number of chunks (e.g., every 128 chunks), each thread checks the refaults counter of the tenant containers. Refaults occur when recently used pages are reloaded into memory after being evicted to disk. If the number of refaults exceeds a configurable threshold (default: 4% of the RSS), the thread halts memory access immediately to avoid overloading the system.

Once a thread completes accessing its assigned portion of memory, it evaluates the amount of free memory in the system. If sufficient memory has been reclaimed by the kernel, the thread expands its allocated portion of contention memory. The size of this new allocation equals the difference between the current free memory and the low memory watermark. If the system has not reclaimed enough memory, the thread refrains from expanding its allocation to avoid further pressure.

Finally, when the thread detects high refault count based on the current system conditions, indicating that the system is under significant memory pressure, it releases the allocated contention memory and terminates execution.

4.5 Metrics Collector

The Metrics Collector (MC) is a component of the Contention Service that is responsible for collecting system usage statistics. It gathers both system-wide and per-container statistics from the *proc* filesystem and the interface provided by the Linux *cgroup (v2)*. The MC currently aggregates statistics related to memory resource utiliza-

tion and it is called by the contention service whenever it is necessary to determine if memory contention should be applied. Data collection occurs in real time.

The MC collects both system-wide and per-container memory statistics, which are used by the RC to make decisions about when to invoke the MS. By monitoring both the system memory utilization and the memory consumption of the tenant containers, the MC provides a comprehensive view of the memory state, which helps the RC to determine the need for memory contention.

At the system level, the MC collects data about the amount of free physical memory in the system, which helps determine the intensity of the memory contention. It also gathers statistics from the memory zones, including thresholds and scaling factors, which are used to assess the system's memory availability. Additionally, the memory zone statistics, including memory thresholds and scaling factors, are collected to calculate the available memory for contention and estimate the amount of memory that should be reclaimed.

The MC also collects critical per-container metrics to monitor the status and the memory utilization of the tenant containers. These metrics include the total memory in use (RSS) by each container, the number of the processes running within the tenant containers, and key memory statistics such as the sizes of the active and inactive LRU lists, the page faults, and the working set refaults.

4.6 Resource Controller

The Resource Controller (RC) is responsible for proactively directing the Linux kernel LRU-based mechanism to release the underutilized memory pages allocated by the tenant applications. It achieves this by proactively generating controlled memory contention in order to trigger the wake up of the *kswapd* thread, which asynchronously reclaims memory. The asynchronous page reclamation minimizes the execution of costly direct reclaim, thus preventing the application performance degradation.

To initiate the Memory Stresser (MS), the RC first confirms that there is a quantity of memory allocated to the tenant containers that is currently not in active use by the applications. The RC aggregates information from the MC regarding (a) the amount of memory allocated to the applications running in the tenant containers and held in RAM, (b) the amount of memory actively used by the application during runtime, and

(c) the amount of memory likely to be transferred to the tail of the inactive LRU lists and ultimately reclaimed. The RC uses these statistics as indicators of underutilized memory pages allocated by the tenant containers in order to trigger the MS.

The RC monitors the node's memory usage through the MC component. After evaluating the statistics it receives, the RC decides whether memory contention should be applied and instructs the MS to start. The RC is implemented as an iterative loop that tracks the following key metrics for each cycle: the number of *inactive* pages, the number of *active* and *inactive file* pages, the current *Resident Set Size (RSS)*, and the count of currently *running processes* within the container. By comparing these statistics across cycles, the RC detects changes in the workload patterns and determines whether enough underutilized memory pages are present in RAM. This assessment enables the RC to decide if intervention is necessary.

In particular, the RC relies on several configurable parameters to make its decisions:

1. List Size Ratio (LSR): The proportion of inactive RSS pages to the total RSS pages in the tenant containers.

2. List Size Difference (LSD): The percentage growth in the inactive LRU list sizes between consecutive cycles.

3. File List Size (FLS): The minimum number of file memory pages that must be maintained in the active and inactive LRU lists.

Using the above parameters, the RC evaluates the current memory usage. It first verifies whether the RSS page count, along with the number of processes executing in the tenant containers, has changed since the last cycle, indicating the emergence of new activity within the container. If a change is detected, the RC examines whether the number of RSS pages in the inactive LRU lists exceed the *List Size Ratio (LSR)* threshold of the total RSS pages of the tenant containers. This would indicate that a significant portion of memory is occupied by inactive pages, potentially signaling a presence of underutilized memory. Additionally, an increase in the sizes of the inactive LRU lists for both anonymous and file pages, beyond the *List Size Difference (LSD)* of their sizes in the previous cycle, can indicate changes in workload patterns.

When handling file-backed pages, where the accessed pages are immediately moved to the active file LRU list, the RC checks if the active or inactive LRU list

size exceeds the *File List Size (FLS)* threshold. This criterion is essential to detect high file-related memory activity, as tenant containers may already have loaded application binaries and shared libraries in RAM.

If all the conditions to initiate the MS are satisfied, the RC creates a child process to run the MS in order to apply memory contention on the containers. The MS temporarily consumes memory to encourage the system to free up unused or less critical pages allocated by the tenant applications.

4.7 Memory Stresser

The Memory Stresser (MS) is a multi-threaded process executed in a dedicated cgroup, which is configured with unbounded memory limits. The MS is responsible for enabling the asynchronous background memory reclamation on the system by allocating a specific amount of memory, referred to as contention memory size. This amount is determined by calculating the difference between the node's free memory and its low memory watermark. MS continuously accesses the contention memory until a configurable number of refaulted pages is reached. During each memory contention cycle, the service checks if the total number of freed pages has reached the high memory watermark in order to adjust the memory contention size based on the current memory usage statistics.

Below we explain the characteristics of the Memory Stresser in terms of the memory contention size, the type of the contention memory pages it accesses, the number of threads used, the memory access pattern, and the termination conditions.

Memory Contention Size. The memory contention size is determined by leveraging the Linux memory watermarks presented in Figure 4.2, with the goal of proactively activating the *kswapd* thread to reclaim pages by swapping out anonymous memory or flushing the page cache. The memory contention size is calculated as the amount of free memory required to be allocated in order to bring the system to the low memory watermark. This represents a safe amount of free memory needed by the system to operate, before synchronous direct page reclamation is activated.

We anticipate *kswapd* will reclaim at least the watermark distance amount of memory at each step. If this amount of memory is freed in a cycle, the MS redefines the memory contention size to prompt *kswapd* to reclaim more pages.

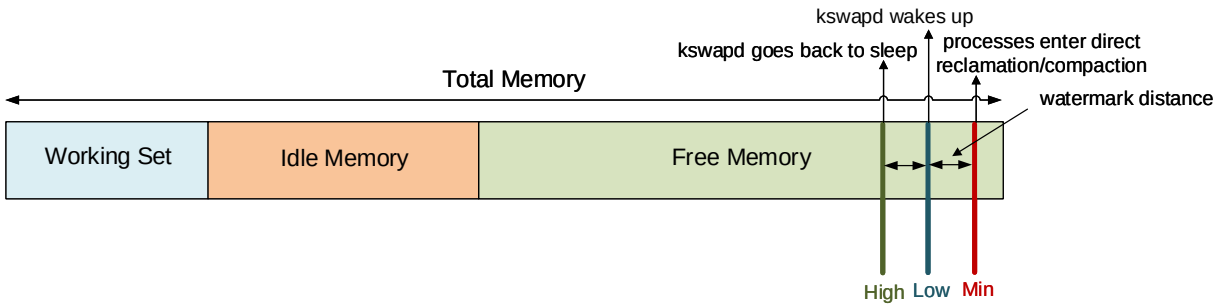

Figure 4.2: Linux Kernel Page Frame Reclamation (background/foreground) activation based on memory watermarks.

Type of Contention Memory. The MS prefers the allocation of anonymous memory pages over file-backed pages during memory contention, based on the following considerations.

Using file-backed memory to induce contention would result in the eviction of page cache to disk, rather than anonymous memory, when available memory is low. Moreover, workloads relying on file-backed memory require continuous read and write operations to disk, leading to higher disk utilization and potential performance degradation, especially during prolonged or intense memory contention. Additionally, excessive memory contention would cause file-backed pages to be evicted to disk, further increasing disk I/O activity as these pages must be reloaded from storage when accessed again.

Furthermore, the pages of the contention memory should remain in RAM rather than being swapped out to disk. Using anonymous memory instead of file-backed memory simplifies their management through the Linux *cgroup* interface.

Parallelism in the Memory Stresser. While adding threads usually speeds up processing, increasing the number of threads that allocate and access the contention memory can cause severe delays to the MS service. Therefore, we configure the number of MS threads to match the number of cores assigned to the cgroup running the Memory Stresser process. In section 4.8, we further describe how MS threads allocate and access the contention memory.

4.8 Memory Stresser Threads

The Memory Stresser Threads (MSTs) are responsible for introducing short periods of memory contention by dynamically allocating and accessing memory. Each thread begins by allocating and initializing its assigned portion. To ensure that the memory pages remain active and are not paged out to disk, the thread touches a configurable number of bytes of each memory page within its allocated range. This configurable number of accesses on each page guarantees that the pages stay in memory.

Once all threads have initialized their portions, they enter a continuous loop, periodically accessing their assigned memory chunks. This repeated access helps simulate sustained memory usage and creates controlled memory contention.

As each thread accesses its allocated memory, it periodically checks if the system's memory usage exceeds certain thresholds. To do this, the thread interacts with the MC component, which tracks key memory metrics, such as the working set refaults (*ws_refaults*) and the Resident Set Size (*RSS*) of the tenant containers. The number of working set refaults measures the recently accessed pages that are paged out and then reloaded, which indicates potential memory inefficiency. By comparing the number of refaults to the RSS, each thread assesses if the system is under significant memory pressure. If the refaults exceed a threshold relative to the RSS of the container, referred to as the *refault ratio threshold (RRT)*, the thread concludes that further memory access is inefficient and initiates the termination process. This termination condition helps prevent excessive memory usage and protects the system performance.

4.9 Memory Stresser Termination

The termination process is initiated when a thread detects that the performance degradation exceeds the defined threshold. At this point, all threads proceed with their own process termination. This ensures that no thread continues its memory accesses once the system has reached a state of high memory contention.

If the termination condition is not met, each thread reviews the system's memory availability by requesting the current memory usage from the MC component. If enough memory has been released and the current free memory in the system reaches the high memory watermark, each thread attempts to expand its allocated contention memory by acquiring additional memory chunks. The memory contention size is

recalculated based on the updated memory usage statistics. One of the threads will successfully increase its portion of allocated contention memory making the total allocated contention memory to reach the updated memory contention size. As a result, the other threads will not increase their contention memory at this point.

4.10 Tenant Containers

Tenants request certain memory resources to run their applications in common Linux containers. Estimating the amount of memory resources the workloads will need is a difficult task. Therefore, we provide the tenants with the resources they request even if this decision leads in resource overprovisioning.

We rely on Linux cgroups to manage the resources of tenant containers. Each container is assigned to its own cgroup. To collect and aggregate statistics from individual cgroups, we use an *aggregation cgroup*, which resides directly under the root cgroup. The tenant containers are organized as sub-cgroups under the aggregation cgroup. The hierarchical setup simplifies the collection of statistics, as the aggregation cgroup automatically combines metrics from the sub-cgroups into cumulative counters.

We leverage the cgroup memory controller interface files to control the memory limits, setting the *memory.high* limit equal to the requested memory resources. The high limit is a soft limit of memory usage. If a cgroup's memory usage goes over this boundary, the kernel throttles the processes of the cgroup and puts them under heavy reclaim pressure.

4.11 Synthetic Workload Generator

This section outlines the development of a synthetic workload generator, designed to emulate the memory usage patterns commonly observed in MapReduce applications [1], as illustrated in Figure 1.1 of Chapter 1. The generator emulates the fluctuating memory demands of MapReduce applications by creating periodic spikes in the amount of accessed memory across multiple tenants and distributing memory access across multiple threads.

Memory allocation. We have implemented the workload generator to perform

anonymous or file memory accesses.

In the case of anonymous memory accesses the workload generator preallocates a large contiguous memory pool of parameterized size (1 GB in our experiments), designated as *allocated memory* (in Bytes), which serves as the workspace for memory operations performed by the application. Throughout the execution, the allocated memory pool remains constant, and all memory accesses are confined to this region.

In contrast, in the case of file memory accesses the workload generator does not allocate anonymous memory but allocates a small memory buffer of configurable size (4KB default) per thread to read data from a file of predefined size (1GB).

Access patterns. In CloudScale [1], the memory usage patterns of MapReduce applications exhibit two primary behaviors: steady memory consumption with minor variations (e.g., Word Count) and periodic bursts alternating between high and low memory demands (e.g., Grep). These fluctuations are influenced by workload phases, such as data processing and computation. For instance, in Grep, the high memory demand lasts approximately 10–15 seconds, followed by low memory demand for a similar duration, reflecting its alternating workload phases.

The synthetic workload generator emulates these patterns by dynamically alternating the memory demand between configurable high and low memory access sizes over adjustable demand durations, effectively capturing the key characteristics of real MapReduce applications observed in CloudScale. The *high memory demand (hmd)* parameter corresponds to phases of increased amount of accessed memory, wherein a considerable proportion of the allocated memory is accessed. The *low memory demand (lmd)* parameter is used to describe periods of reduced amount of accessed memory, whereby only a limited portion of the memory is accessed.

Timing intervals and spike generation. The workload generator introduces spikes in memory usage by controlling the duration of each memory access phase through intervals of configurable memory demand. The *high demand duration (hdd)* defines the duration for which the program accesses the high memory demand, which corresponds to periods of higher memory usage. The *low demand duration (ldd)* defines the duration for which the program accesses the low memory demand, which corresponds to periods of low memory usage.

Thread-based execution. The workload generator supports multi-threaded execution to emulate the concurrent nature of distributed MapReduce applications. In the anonymous memory test, each thread is responsible for accessing a portion of the

allocated memory pool in fixed-size blocks (4 KB) using a memory read operation. In the file-based memory test, we split the file into fixed-size chunks of contiguous blocks, and we assign each chunk to a thread. Each thread allocates a small memory buffer of configurable size and continuously reads its assigned blocks.

4.12 Summary

In this chapter, we introduced the Contention Service, a solution that enhances memory management in systems with containerized applications by automatically reclaiming underutilized memory. Unlike traditional memory reclamation methods, this mechanism works in the background, freeing memory without blocking or slowing down applications.

The Metrics Collector gathers memory usage data, which is then analyzed by the Resource Controller to determine when to trigger the Memory Stresser to provoke the underutilized memory reclamation. The Memory Stresser generates contention into memory in order to activate the kernel's background reclamation process, ensuring that idle memory is freed efficiently without impacting system performance.

The Contention Service is lightweight, application-agnostic, and optimizes the system memory usage. It offers an effective solution for containerized environments, where efficient resource allocation is essential, and applications must run without disruption.

Chapter 5

Implementation

This chapter provides a detailed explanation of the implementation of the Contention Service (CS) prototype. The CS releases underutilized memory pages allocated by tenant applications in a containerized, multi-tenant system.

5.1 Prototype Implementation

We implement a prototype of the CS in 963 lines of C code, excluding commented or black lines. We use the Linux *cgroup (v2)* interface to configure the memory limits through the memory controller and the CPU cores through the cpuset controller for the tenant containers and the Memory Stresser container. Additionally, we use the *cgroup (v2)* interface files to gather detailed memory usage metrics from the containers. Finally, we rely on the kernel proc filesystem to read system-wide statistics.

5.2 Metrics Collector

The Metrics Collector (MC) retrieves both system-wide and per-container metrics. For system-wide memory metrics, the MC collects statistics from the `/proc` interface. The MC reads the `/proc/meminfo` file to retrieve the *MemFree* statistic, which represents the amount of physical memory not used by the system. This statistic is used to determine the intensity of the memory contention. Another `/proc` file used is the `/proc/zoneinfo`, which aggregates information about the memory shared among different memory zones and calculates the *high*, *low*, and *minimum* memory watermarks. These statistics, along with the *watermark_scale_factor* received from the `/proc/sys/vm/watermark_scale_factor` file, are used to calculate the *wm_distance*. This value, together with the memory watermarks, define the amount of memory released by the contention service.

For container-specific metrics, the MC uses the cgroup v2 interface. The MC also collects statistics from the cgroups to monitor the tenant containers' status. It tracks the number of processes currently running by the tenants, the total amount of memory in use by the cgroups (from the `memory.current` file), and several useful statistics, such as the sizes of the active and inactive LRU lists, the number of page major faults, and the number of working set refaults (from the `memory/stat` file). These statistics are all used to determine the conditions under which the contention service should be applied.

5.3 Resource Controller

The Resource Controller (RC) ensures efficient memory reclamation by forcing the LRU-based kernel page management system to reclaim underutilized memory pages allocated by the tenant applications. The primary objective of the RC is to trigger the kernel's *kswapd* reclaimer by inducing controlled memory contention, thus preventing performance degradation caused by excessive memory usage.

Initiation. The RC uses a list of variables to monitor the memory state of both the containers and the system. These include variables for tracking the Resident Set Size (RSS) and the number of the active and inactive memory pages. These variables serve as baseline, for comparing changes in the memory usage across monitoring cycles to detect memory usage patterns and deciding whether to initiate memory contention

(lines 1-5 of Algorithm 5.1).

Algorithm 5.1: Resource Controller

Input: the aggregation cgroup name in the hierarchy of the tenant containers $cgrp$, the number of threads $thrnum$ to generate the memory contention

```
// LSR: The proportion of inactive RSS pages to the total RSS pages in
   the tenant containers.
// LSD: The percentage growth in the inactive LRU list sizes between
   consecutive cycles.
// FLS: The minimum number of file memory pages that must be maintained
   in the active and inactive LRU lists.
```

1 $inactive\_pages \leftarrow$ mcget($cgrp$, $inactive\_pages$)
2 $active\_file\_pages \leftarrow$ mcget($cgrp$, $active\_file\_pages$)
3 $inactive\_file\_pages \leftarrow$ mcget($cgrp$, $inactive\_file\_pages$)
4 $rss \leftarrow$ mcget($cgrp$, rss)
5 $running\_procs \leftarrow$ mcget($cgrp$, $running\_procs$)
6 **while** *(True)* **do**
7 **if** *(rss* $\neq$ *prev_rss* ***and*** *running_procs* $>$ *prev_running_procs)* **then**
8 **if** *((inactive_pages* $>$ *LSR* $\times$ *rss* ***or***
9 *active_file_pages* $>$ *FLS* ***or***
10 *inactive_file_pages* $>$ *FLS)* ***and***
11 *inactive_pages* $>$ *LSD* $\times$ *prev_inactive_pages)* **then**
12 $ms \leftarrow$ fork($MS(cgrp, thrnum)$)
13 wait(ms)
14 sleep(SEC)
15 **end**
16 **end**
17 update_statistics($prev\_rss, prev\_running\_procs, prev\_inactive\_pages$)
18 sleep(SEC)
19 **end**

Monitoring Loop. After the initiation step, the RC enters a continuous monitoring loop (line 6 of Algorithm 5.1) to evaluate the memory usage. Each cycle involves calls

to the MC component in order to gather the latest memory statistics for the containers. These metrics are aggregated at the parent cgroup level, encompassing the memory usage of all child containers.

Before evaluating the conditions to initiate the memory contention, the controller inspects the tenant containers for new activity by comparing the current RSS (*rss*) metric and the current number of running processes (*running_procs*) of the tenant containers with the respective metrics from the previous cycle of the monitoring loop (line 7 of Algorithm 5.1). By doing this, the controller delays the inspection of the conditions to generate memory contention until a tenant container executes a new workload. This is necessary, as the kernel does not zero the page activity statistics when a workload terminates, and as a result the controller may evaluate these conditions as true, even though no workload is running.

Memory Contention Trigger Conditions. The RC evaluates a number of conditions to determine whether to initiate memory contention (lines 8-11 of Algorithm 5.1). First, it verifies if the number of inactive pages exceeds a configurable proportion (*LSR*) of the RSS, initially set to 40% ($inactive\_pages > LSR \times rss$). This factor helps to determine if the system has accumulated a significant number of potentially underutilized pages.

For the file-backed pages, the system monitors changes in the size of the active file LRU list. It also checks if there are enough pages in the active or inactive file LRU lists that exceed a minimum size, set by default to 100 MB ($active\_file\_pages > FLS$ ***or*** $inactive\_file\_pages > FLS$). We consider a memory amount below 100 MB for the FLS parameter to be too small to justify enabling the Contention Service. Additionally, the RC evaluates whether the ratio of inactive pages between two cycles surpasses the LSD threshold ($inactive\_pages > LSD \times prev\_inactive\_pages$), which is set by default to 105%.

If these conditions are met, the system creates a child process (line 12 of Algorithm 5.1) to apply memory contention through the Memory Stresser (MS). When the MS returns, the RC pauses briefly (2 seconds) before resuming its monitoring loop (line 14 of Algorithm 5.1). This pause allows the kernel sufficient time to perform the memory reclamation before the next monitoring cycle.

State Update. After the MS terminates, the system updates the memory statistics (*prev_rss* and *prev_inactive_pages*) to reflect the current system state (line 17 of Algorithm 5.1).

5.4 Memory Stresser

The *MS* function (Algorithm 5.2) creates controlled memory contention. The MS is a multi-threaded process executed in a dedicated container whose cgroup is set to have unlimited memory. The MS cooperates with the MC component to receive memory usage statistics from the system and the tenant containers. Several parameters, such as the memory free space, the memory watermarks, and the working set refaults count are needed to apply controlled memory contention to the system.

Initialization. The MS starts by calling the *wait_pageload()* routine (Algorithm 5.3) in order to delay the execution of the MS until the working set pages are loaded into RAM. The loop continues as long as the number of occurring refaults or major page faults between two consecutive cycles drops below a number of pages (*REFLT_MINPG*), configured to 1000 pages (lines 3-6 of Algorithm 5.3).

Memory Stresser Loop. The MS then enters a loop where it continuously applies memory contention until at least one MST detects performance degradation (line 3 of Algorithm 5.2). First, it calculates the *memory contention size*, which is determined as the difference between the free memory (*mem_free*) and the low watermark (*wm_low*). The memory contention size (line 9 of Algorithm 5.2) represents the minimum amount of memory needed to be allocated, in order to reduce the system's free memory pages to the low memory watermark, which triggers the *kswapd* thread to begin reclaiming less recently used pages. Then, the MS creates a number of MSTs (line 10-13 of Algorithm 5.2) to gradually create memory contention in the system.

Memory Contention Service Loop Termination. The MS loop continues to run, as long as the sum of the working set refaults (*ws_refault_sum*) from each cycle is less than a configurable proportion of the RSS (*refault_ratio_threshold (RRT)*) (line 18 of Algorithm 5.2). We use the `pids.current` file of the parent cgroup in the hierarchy of the tenant cgroups to get the currently running processes. In each cycle, the MS waits for all threads to finish (lines 14-16 of Algorithm 5.2) their task and updates the memory usage statistics.

Algorithm 5.2: Memory Stresser

Input: the aggregation cgroup name in the hierarchy of the tenant containers $cgrp$, the number of threads $thrnum$ to generate the memory contention

```
// RRT: a configurable threshold representing the ratio of the sum of
   working set refaults to the RSS, by default set to 0.04
```

```
1  pthread_t thread[thrnum] // Wait workload to be loaded
2  wait_pageload(cgrp)
   // Main loop for the Memory Stresser
3  do
       // Gather current memory statistics
4      ws_refaults ← mcget(cgrp, ws_refaults)
5      mem_free ← mcget(cgrp, mem_free)
6      rss ← mcget(cgrp, rss)
7      wm_low ← mcget(cgrp, wm_low)
8      wm_dist ← mcget(cgrp, wm_dist)
       // Calculate the memory contention size
9      mc_size ← mem_free - wm_low
       // Create threads to apply memory contention
10     for (i = 0 to thrnum − 1) do
11         thread_args[i] = set(mc_size, ws_refaults, wm_low, wm_dist, cgrp)
12         new_thread(thread[i], MST, thread_args[i])
13     end
       // Wait for all threads to complete
14     for (i = 0 to thrnum − 1) do
15         wait_thread(thread[i])
16     end
       // Update memory statistics after memory contention
       // (ws_refault_sum, rss)
17     update_statistics(ws_refaults_sum, rss)
18 while (ws_refaults_sum < RRT * rss);
```

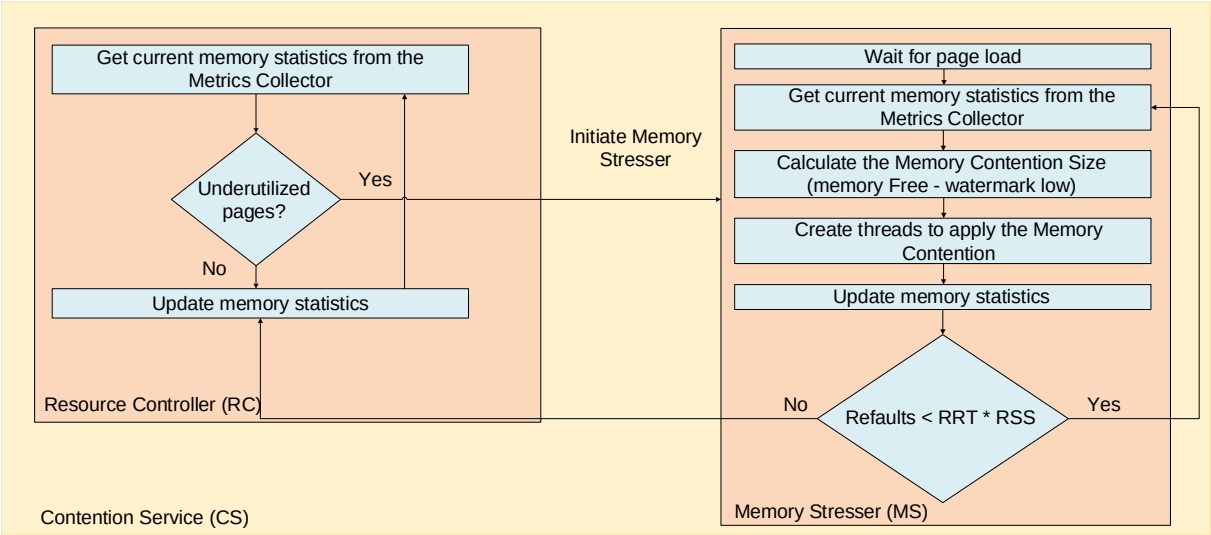

Figure 5.1: The interaction between the Resource Controller (RC) and the Memory Stresser (MS) components.

Algorithm 5.3: Wait for Page Load

Input: the aggregation cgroup name in the hierarchy of the tenant containers $cgrp$, the number of threads $thrnum$ to generate the memory contention

```
// REFLT_MINPG: a minimum number of refaults between consecutive cycles,
   by default set to 1000
```

1 $ws\_refaults \leftarrow$ mcget($cgrp$, $ws\_refaults$)
2 $pg\_majfaults \leftarrow$ mcget($cgrp$, $pg\_majfaults$)
3 **while** *(calculate_ws_refault_diff*($cgrp$, $ws\_refaults$) $>$ *REFLT_MINPG* ***or*** *calculate_pgmajfault_diff*($cgrp$, $pg\_majfaults$) $>$ *REFLT_MINPG)* **do**
4 $ws\_refaults \leftarrow$ mcget($cgrp$, $ws\_refaults$)
5 $pg\_majfaults \leftarrow$ mcget($cgrp$, $pg\_majfaults$)
6 sleep(SEC)
7 **end**

Figure 5.1 illustrates the interaction between the Resource Controller (RC) and the Memory Stresser (MS) components. The Resource Controller gathers the memory statistics from the system and the tenant containers, checks for underutilized memory, and determines when to activate the Memory Stresser. Once activated, the Memory Stresser applies memory contention by creating threads based on the available memory. The process terminates after a configurable number of refaults is reached. In

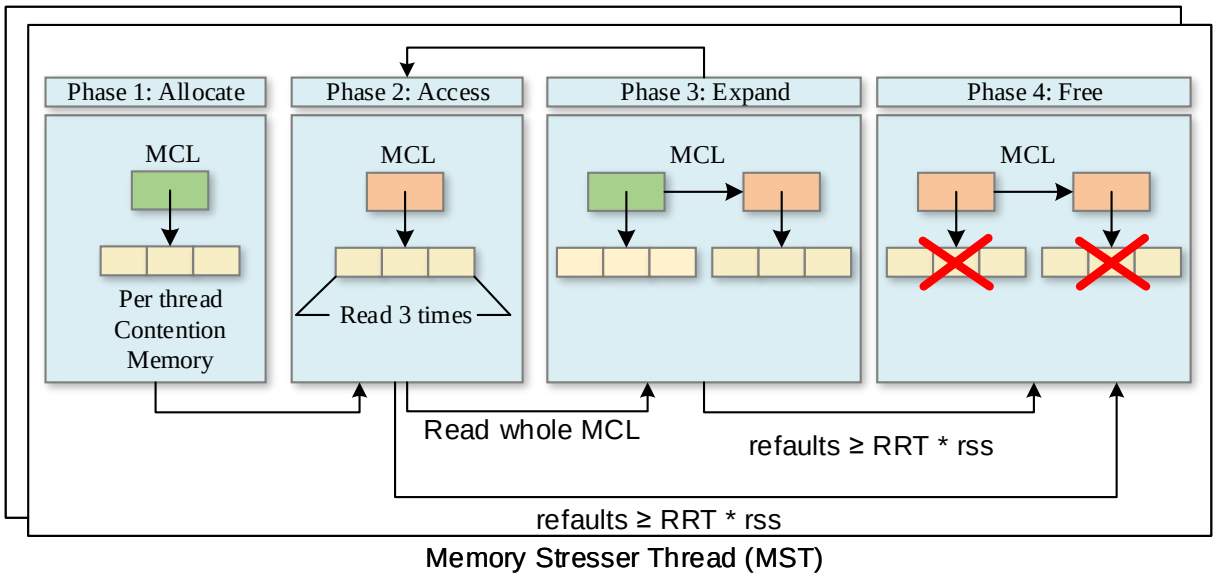

Figure 5.2: Operation stages of each Memory Stresser Thread and management of contention memory using a per-thread Memory Contention List (MCL).

section 5.4.1, we delve deeper into the implementation of memory contention by the stresser threads and describe the core operations performed by each thread to create controlled memory contention.

5.4.1 Memory Stresser Threads

The primary goal of the MSTs is to generate contention into memory, in order to allow the system to release underutilized memory pages. To achieve this, each MST first allocates an equal portion of the contention memory and then continuously accesses its portion until it observes a number of refaults. If the required amount of memory is reclaimed by the kernel, then each thread can try to dynamically expand the contention memory in order to force the kernel to reclaim more memory. We use a linked list data structure to manage the allocated contention memory area of each thread by keeping track of the memory allocated and accessed by each thread. This structure facilitates the deallocation of memory upon thread termination. Figure 5.2 describes the key operation stages of a MST.

Algorithm 5.4: Allocate Contention Memory

Input : a per-thread memory contention list MCL, the size mc_size of per-thread memory contention portion

Output: the initialized per-thread memory contention list

```
// PG_ACC_TIMES: The number of accesses needed to set a page active (by
   default set to 3)
```

1 $buffer \leftarrow$ alloc_aligned(mc_size, $BLOCK\_SIZE$)

2 $node \leftarrow$ create_node($buffer$)

```
// Initialize each memory chunk of the MCL node PG_ACC_TIMES times to
   keep it active
```

3 **for** ($i \leftarrow 0$ **to** $node.size()/BLOCK\_SIZE$) **do**

4 $\quad chunk \leftarrow i \times BLOCK\_SIZE$

```
   // Write each memory chunk of the MCL node PG_ACC_TIMES times to keep
      it active
```

5 $\quad$ **for** ($i \leftarrow 1$ **to** *PG_ACC_TIMES*) **do**

6 $\quad\quad$ write_access_chunk($node$, $chunk$)

7 $\quad$ **end**

8 **end**

9 MCL.add($node$)

10 **return** MCL

Memory Allocation. Each MST initially invokes the *memory_contention_alloc()* function (Algorithm 5.4) to allocate and initialize the memory chunks assigned to it aligned to BLOCK_SIZE (by default 4 KB). Upon initialization, the entire contention memory is represented as a single node in the linked list.

Each thread allocates a portion of the total contention memory as a memory node in a per-thread memory contention list (*MCL*). The allocation is performed using a buffer, which is then initialized by accessing the first three bytes of each memory chunk (lines 3-7 of Algorithm 5.4) in order to ensure that the pages are marked as active preventing them from being reclaimed by the kswapd page reclamation process. The newly created MCL node is subsequently added to the head of the list (line 9 of Algorithm 5.4). Finally, the function returns the updated MCL (line 10 of Algorithm 5.4), which includes the new node.

Algorithm 5.5: Access Contention Memory

Input: a per-thread memory contention list MCL, a pointer $thrargs$ to the struct thread arguments

```
// MIN_CHUNKS: the minimum number of accessed chunks required for a
   subsequent refault check, by default set to 128
```

```
1  while (node ← MCL.get_next()) do
2      for (i ← 0 to node.size()/BLOCK_SIZE) do
3          chunk ← i × BLOCK_SIZE
           // Access each memory chunk of the MCL node PG_ACC_TIMES times to
              keep it active
4          for (i ← 1 to PG_ACC_TIMES) do
5              read_access_chunk(node, chunk)
6          end
           // Check refaults every MIN_CHUNKS chunks
7          if (i mod MIN_CHUNKS == 0) then
               // Request metrics from the metrics collector
8              ws_refaults ← mcget(thrargs.cgrp,
                thrargs.ws_refaults-thrargs.rss)
9              if (ws_refaults ≥ RRT × thrargs.rss) then
10                 break // Exit the loop when condition is met
11             end
12         end
13     end
14 end
```

Memory Access. Each thread invokes the *memory_contention_access()* function to access its MCL. This function iterates over the memory nodes of the MCL, accessing the first three bytes (PG_ACC_TIMES) of each chunk in the buffer of the node (lines 2-6 of Algorithm 5.5). During this process, the thread monitors key metrics, such as the number of working set refaults and the RSS, to ensure that memory contention does not negatively impact performance. Every MIN_CHUNKS chunks, the thread checks the number of working set refaults (*ws_refaults*) and compares it with the current RSS. If the ratio of ws_refaults to RSS exceeds a threshold (*refault ratio threshold (RRT)*), which is set to 4% by default, the function breaks out of the loop (lines 9-11 of

Algorithm 5.5), effectively stopping memory accesses for that thread.

Algorithm 5.6: Memory Stresser Thread

Input: a pointer to the struct $thrargs$ thread arguments, the size mc_size of per-thread memory contention portion

```
// RRT: a configurable threshold representing the ratio of the sum of
   working set refaults to the RSS, by default set to 0.04
```

1 $stop\_memory\_contention \leftarrow 0$
2 $MCL \leftarrow$ memory_contention_alloc($NULL$, $thrargs.mc\_size/thrnum$)
3 **while** *(stop_memory_contention* $\neq$ *1)* **do**
4 memory_contention_access(MCL, $thrargs$)
5 $ws\_refaults$ = mcget($thrargs.cgrp$, $ws\_refaults$ - $thrargs.ws\_refaults\_before$)
6 $rss \leftarrow$ mcget($thrargs.cgrp$, rss)
7 **if** *(*$ws\_refaults \geq RRT \times rss$*)* **then**
8 stop_memory_contention $\leftarrow 1$
9 break
10 **end**
11 $mem\_free \leftarrow$ mcget($thrargs.cgrp$, mem_free)
12 **if** *(mem_free > (thrargs.wm_dist + thrargs.wm_low))* **then**
13 $mc\_size \leftarrow mem\_free - thrargs.wm\_low$
14 $MCL \leftarrow$ memory_contention_alloc(MCL, mc_size)
15 **end**
16 **end**
17 memory_contention_free(MCL)

Memory Expansion. Before the next cycle, each thread recalculates the memory contention size if an appropriate amount of memory is freed during page reclamation and additional nodes are added to the list (line 12-15 of Algorithm 5.6). In particular, if at least *wm_dist* pages are freed on a thread cycle, which means that the free memory in the system has reached the high memory watermark, the memory contention size is updated based on the current system state.

Thread Termination. However, when memory contention starts to negatively impacting the application's performance (*ws_refaults >= RTT* $\times$ *RSS*), the thread signals its intent to stop by setting a shared flag, referred to as *stop_memory_contention*, frees the allocated memory chunks, and terminates. Each thread continuously monitors the

shared *stop_memory_contention* flag. Once one thread sets the flag, all the other threads detect the signal and proceed with their own process termination. If at any point, the number of refaults exceeds the *RTT* of the RSS within the containers, the thread signals for termination by setting the *stop_memory_contention* flag (lines 7-10 of Algorithm 5.6). Once the thread determines that it should stop, either due to excessive refaults or a manual stop request, it frees the memory pool using the *memory_contention_free()* function.

Memory Deallocation. The *contention_memory_free()* function is invoked after all threads terminate in order to free all memory associated with the memory contention linked list. The function iterates through the MCL nodes, starting from the given contention memory pointer. For each node, it frees the memory buffer and then the MCL node itself.

Algorithm 5.7: Free Memory Contention

Input: a per-thread memory contention list MCL

```
// MCL: per-thread memory contention list
// Free each node in the memory contention list
1 for (node ∈ MCL) do
2     free(node) // Release memory of the node
3 end
```

5.4.2 Protect Memory Contention Service From Swapping

Under conditions of excessive memory contention, Linux by default allocates equal swapping opportunities to each container. In Linux cgroup (v1), the swappiness parameter is used to regulate the extent to which the system prioritizes reclaiming anonymous memory pages over page cache. However, this swappiness parameter has been removed from the memory controller in *cgroup (v2)*. Instead, cgroup (v2) introduces alternative mechanisms for controlling memory usage through the configuration of memory limits within the cgroup. These controls include parameters such as `memory.min`, `memory.low`, `memory.high`, and `memory.max`, which set thresholds for memory utilization, as well as `memory.swap.max` and `memory.swap.high`, which limit the usage of swap memory.

Our objective is to activate the memory contention service to remove the underutilized memory pages from the applications running within tenant containers. To achieve this, it is necessary to guide the Linux kernel to retain the memory allocated

by the contention activity service in memory. One approach involves configuring the memory.swap.max parameter within the cgroup to a minimal value (e.g., 100 MB), effectively limiting the amount of anonymous memory that can be swapped out. However, this method leads to the activation of the Out-Of-Memory (OOM) killer, resulting in the near-immediate termination of the contention service process.

To address this issue, we configure the `memory.low` limit for the container hosting the contention service to match the total available system memory. This configuration helps prioritize memory residence for the cgroup, making it less likely to be reclaimed unless other cgroups have no reclaimable memory. The memory.low limit ensures that the memory allocated to the contention service remains in memory during periods of high memory contention.

5.5 Summary

This chapter delves into the implementation details of the Contention Service (CS) components and their interactions.

The Resource Controller (RC) monitors the memory usage statistics of the containers and applies the Memory Stresser (MS) when workloads are running in the containers or there are changes in the working sets.

The MS generates contention memory to trigger the *kswapd* thread to initiate page reclamation. The MS determines the memory contention intensity based on the minimum amount of memory required to reach the low memory watermark in the system, which activates the asynchronous Linux page reclamation process.

We use a per-thread linked list structure to manage the contention memory. The MSTs handle portions of the total memory contention size and adjust their allocations when the page reclamation process successfully frees enough pages to reach the high memory watermark. The MSTs also monitor the memory usage statistics in the containers in order to terminate when their accesses negatively impact the performance of the applications running in the containers. Similarly, the MS stops the memory contention when it detects performance degradation.

Chapter 6

Experimental Evaluation

In this chapter, we first assess the efficacy of the Memory Contention mechanism using synthetic workloads generated by a custom workload generator. This generator emulates memory access patterns of MapReduce applications as observed in [1], which are widely used in cloud environments. The evaluation is conducted on a single machine under two scenarios: single-tenant and multi-tenant configurations.

Next, we validate the effectiveness of the CS by executing real-world MapReduce applications, such as Grep and KMeans. These benchmarks provide practical insights into the CS mechanism's performance in realistic scenarios. Finally, we compare our solution against a similar approach, Facebook's Senpai [24]. This comparative analysis highlights the strengths of our mechanism related to state-of-the-art solutions.

Our evaluation focuses on answering the following questions: (1) what percentage of unused memory is successfully released, (2) to what extent does the CS affect

application performance, (3) how much system memory remains available after memory release, (4) how our solution compares to alternative approaches, such as those modifying cgroup limits or those involving direct memory reclaim.

6.1 Methodology

This section describes the methodology and setup used for evaluating the Contention Service.

Servers. Our experimental environment consists of two rack-mounted nodes. Both nodes are equipped with two 4-core x86-64 processors and 11 GB of RAM. The first node contains one 2 TB Western Digital WD2001FASS SATA hard disk and one 500 GB Crucial MX500 SATA SSD. The second node contains two 2 TB Western Digital WD2002FAEX-0 SATA hard disks. Both nodes run Debian 11 with Linux Kernel version 6.6.52.

Configuration. We assign 2 CPU cores on the task that we use to collect resource usage statistics. This task runs for the whole duration of each experiment and collects resource usage statistics every second. We configure two containers using cgroup v2. In the first container we execute the experimental applications. We configure this container with 4 CPU cores by default using the cpuset controller, but we also test different numbers of CPU cores in some of the experiments. We also set the following memory limits using the memory controller: a) 50 MB minimum memory, b) 100 MB low memory, c) 5.1 GB high memory and d) 5.3 GB maximum memory. In the second container we run the CS. We configure the CS container with 2 CPU cores by default (we also tested different numbers of CPU cores in some of the experiments). At the memory side, we set the low, high and maximum limits equal to the total memory of the system. By setting the low memory limit to match the total memory of the system we advise the kernel to try not to reclaim memory from this container, unless it cannot reclaim memory from other containers. We also tested alternative methods as we explain below. Table 6.1 summarizes our setup. We perform each experiment three times and report the average values obtained from these runs.

Table 6.1: Container configuration.

Container	Assigned Cores	Minimum memory	Low memory	High memory	Maximum memory
Application	4	50 MB	100 MB	5.1 GB	5.3 GB
CS	2	50 MB	11 GB	11 GB	11 GB

6.2 Memory Stresser with a Single Tenant

We evaluate the MS on a single node with one tenant container to assess its effectiveness in reclaiming underutilized memory pages allocated by the applications running within the tenant containers. We use synthetic workloads generated by the workload generator described in section 4.11, testing both anonymous memory and file-backed memory working sets. In these tests, we fix the allocated memory at 1024 MB, while the high and low memory demand are generally set to 500 MB and 100 MB, respectively, or ranging from 400-600 MB and 200-400 MB. Below, we examine the impact of high and low memory demand duration and concurrency on the efficacy of the MS.

6.2.1 Memory demand durations

We examine various combinations of the high memory and low memory demand duration, as shown in Table 6.2. For each combination, we present the memory access patterns of the workload and the RSS of the tenant container resulting from the activation of the MS. The experiments include tests for both anonymous memory and file-backed memory workloads, each with specific low and high demand duration configurations.

In all cases of high and low memory demand ratios examined for the anonymous memory workload, applying memory contention consistently reclaimed 500MB of memory that was initially allocated but never accessed again by the application throughout the experiment.

We choose to present the case where the HDD is set to 10 seconds and the LDD is set to 40 seconds because it provides a representative example of how the memory usage adapts to the applications demands. Figure 6.1 illustrates the Resident Set Size (RSS) of the tenant container during the activation of the MS. A consistent pattern of

Table 6.2: Combinations of high and low demand duration values that were tested.

LMD: Low Memory Demand (MB)	**HMD: High Memory Demand (MB)**	**LDD: Low Demand Duration (s)**	**HDD: High Demand Duration (s)**
100	500	[1, 2, 4, 8]×HDD	[1, 2, 4, 8]
100	500	[1, 2, 4, 8]	[1, 2, 4, 8]×LDD
100	500	[1, 2, 4, 8]×HDD	10
100	500	10	[1, 2, 4, 8]×LDD
200-400	400-600	60	10
200-400	400-600	10-60	10-60

memory usage is observed across all tested combinations of low and high demand durations with the LDD equal to 10 seconds (shown in Table 6.2). The MS is triggered upon detecting a change in RSS, aiming to align the RSS of the tenant container with the application's actual memory access patterns.

In particular, we observe that the execution of the MS allows the RSS of the tenant container to closely match the application's memory access size. The RSS first decreases to the low memory demand, and then, it increases back to the high memory demand when the application requires it.

While the MS attempts to identify and respond to changes in the application's workload, it reduces the RSS below the low memory demand before it stabilizes near this value. This temporary reduction results in the eviction of anonymous memory pages, which are then immediately accessed by the application, and they are brought back into memory. A configurable number of refaults (by default 4% of RSS) terminates the MS.

When the low demand duration ends, the workload shifts from low to high memory demand. The MS identifies the increase in memory usage and is activate in order to force the kernel to reclaim newly underutilized memory pages. When the MS is activated, it waits for the application to load the necessary data into memory before it starts the application of memory contention. As long as the amount of high memory demand pages are actively in use by the application, the MS stops the application of memory contention, allowing the RSS to stabilize at the high memory demand level until the workload changes again.

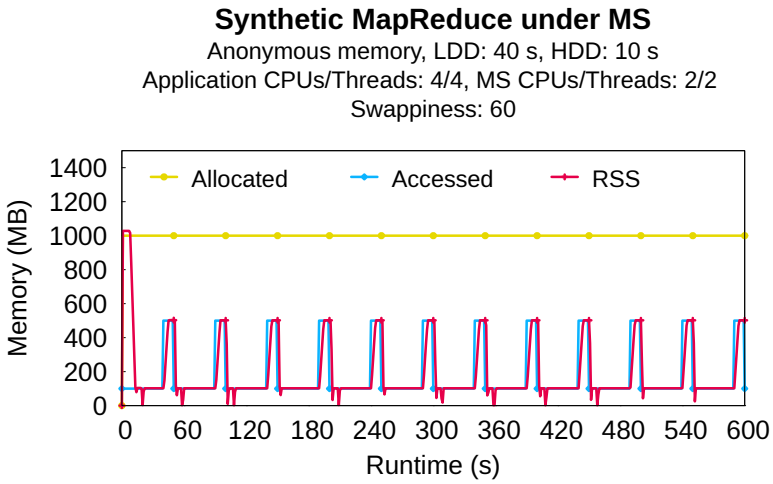

Figure 6.1: Emulation of the memory usage for a MapReduce-like application using the synthetic workload generator with the following parameter settings: allocated memory=1GB, LMD=100MB, HMD=500MB, LDD=40s, HDD=10s, and runtime=600s.

The RSS of the tenant container keeps switching between the high and low memory demand due to the repeated use of the MS. The same pattern of memory adjustment is observed across all combinations of high demand duration (HDD) with low demand duration (LDD) set to 10 seconds. With the MS active, the average memory utilization closely aligns with the actual memory accessed for each workload of the application.

Table 6.3 provides an overview of the average total allocated memory by the application (in MB), the memory actually accessed by the application (in MB), and the Resident Set Size (RSS) (in MB) of the tenant container throughout the experimental runtime of the anonymous memory workloads. When the high demand duration (HDD) increases, the MS observes more frequent periods of high memory demand (HMD). The MS treats this memory as part of the working set, which leads to higher average memory utilization. We notice that as the low demand duration (LDD) increases, the accessed memory amount decreases because the application spends longer periods of low memory demand (LMD). This also causes the RSS of the tenant container to decrease, indicating that the MS can adjust the RSS in line with the application's memory access patterns.

We conduct another experiment with random values for the low and high memory demand sizes, while keeping the low and high duration constant. In this setup, the high MS size ranges from 400 MB to 600 MB, and the low memory demand ranges

Table 6.3: Average memory accessed by the workload, Resident Set Size (RSS) of the tenant container, and total allocated memory by the anonymous memory workload in MB across different high and low demand durations tested.

LMD (MB)	**HMD (MB)**	**LDD (s)**	**HDD (s)**	**Accessed (MB)**	**RSS (MB)**	**Allocated (MB)**
100	500	10	10	299.67	293.40	1024
100	500	10	20	366.22	379.95	1024
100	500	10	40	419.46	523.39	1024
100	500	10	80	453.41	364.21	1024
100	500	20	10	233.10	234.40	1024
100	500	40	10	179.87	186.20	1024
100	500	80	10	139.94	169.51	1024
200-400	400-600	60	10	322.06	315.26	1024
200-400	400-600	1-10	20-100	319.16	319.13	1024

from 200 MB to 400 MB, with the low and high demand duration set to 60 seconds and 10 seconds, respectively.

Results in Table 6.3 show that the MS dynamically responds to fluctuations in the application's memory needs, quickly reclaiming temporary underutilized memory when it is no longer required. The MS adjusts the Resident Set Size (RSS) to match high memory access phases and scales it down as the application shifts to lower memory demands. This approach minimizes memory waste while ensuring sufficient memory availability during active periods, demonstrating flexibility of the MS in adapting to changing memory requirements.

The MS demonstrates the same effectiveness in adjusting RSS as in the scenario where only the low and high memory demand durations (LDD and HDD) vary. The RSS aligns closely with the actual memory usage, scaling up during high memory demand phases and reducing during low-demand periods.

We further observe the adaptability of the MS when we conduct an experiment where both the memory demand sizes, and the memory demand durations vary. As shown in Table 6.3, the high memory demand fluctuates between 400 MB and 600 MB, while the low memory demand ranges from 200 MB to 400 MB. Additionally, the high and low demand durations range from 20 to 100 seconds and 1 to 10

seconds, respectively. The average RSS (319.13 MB) of the tenant container is almost equal to the average accessed memory (319.16 MB).

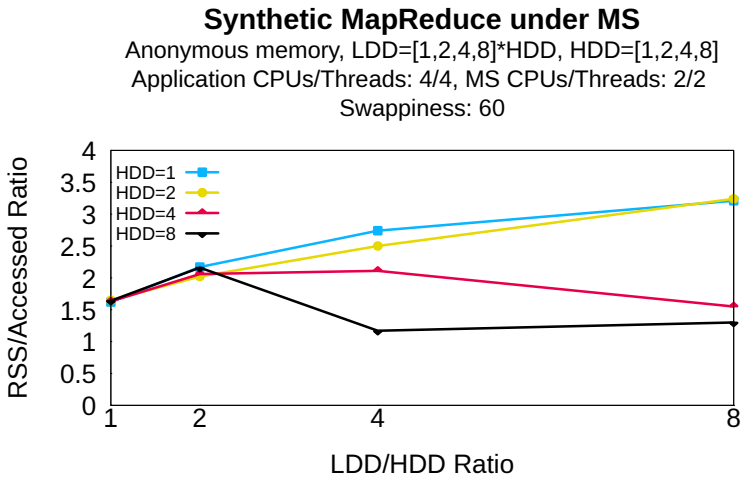

Figure 6.2: The RSS/Accessed ratio for anonymous memory workloads under different low and high demand durations. The experiments were conducted using the following parameter settings: LMD=100MB, HMD=500MB, HDD=[1,2,4,8]s, LDD=[1,2,4,8]×HDDs, and runtime=600s.

We evaluate scenarios where the high demand duration (HDD) takes values of 1, 2, 4, and 8 seconds, while the low demand duration (LDD) is defined as a multiple of the HDD by factors of 1, 2, 4, and 8. Additionally, we examine cases where the LDD takes values of 1, 2, 4, and 8, with the HDD scaling proportionally to the LDD. Figures 6.2 and 6.3 illustrate the RSS/Accessed ratio for anonymous memory workloads managed by the MS under these scenarios.

In Figure 6.2, the results indicate that when the low demand duration (LDD) is significantly longer than the high demand duration (HDD), the RSS of the tenant container closely aligns with the actual memory access patterns of the application. More specifically, the RSS/Accessed ratio approaches the optimal value of 1 in scenarios where (HDD:8, LDD:HDD×4), and (HDD:8, LDD:HDD×8). This shows that prolonged periods of low memory demand allow the MS to efficiently reclaim underutilized memory pages and track the memory usage patterns of the workload.

Conversely, more frequent fluctuations between the high memory demand (HMD) and low memory demand (LMD) reduce the ability of the MS to adapt the RSS of the container to the actual accessed memory, as indicated by the increasing RSS/Accessed ratios.

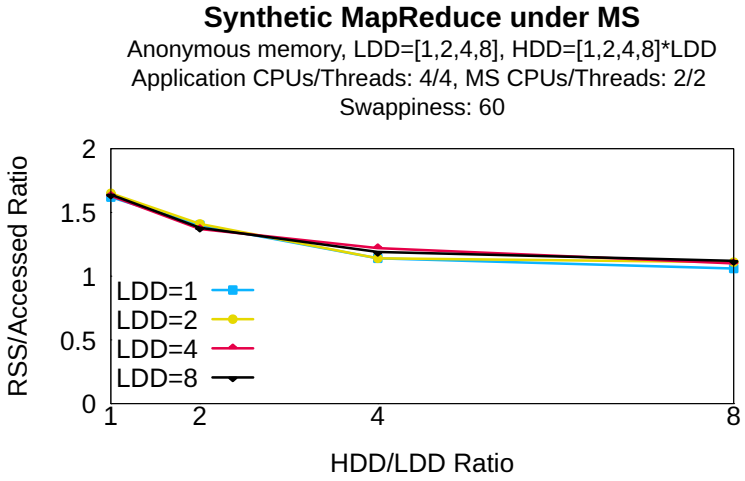

Figure 6.3: The RSS/Accessed ratio for anonymous memory workloads under different low and high demand durations. The experiments were conducted using the following parameter settings: LMD=100MB, HMD=500MB, LDD=[1,2,4,8]s, HDD=[1,2,4,8]×LDDs, and runtime=600s.

Similarly, in Figure 6.3 when the HDD increases proportionally to the LDD, the RSS/Accessed ratio decreases. Scenarios such as (LDD:1, HDD:LDD×4), (LDD:1, HDD:LDD×8), (LDD:2, HDD:×4), (LDD:2, HDD:LDD×8), (LDD:8, HDD:LDD×4), and (LDD:8, HDD:LDD×8) exhibit RSS/Accessed ratios that approximate the optimal value of 1. This behavior occurs because the memory accessed during high demand periods becomes the working set, reducing the MS's effectiveness in reclaiming memory as underutilized. Prolonged HMD relative to LMD ensures that the memory associated with high demand is retained in memory, as it is frequently accessed and classified by the system as critical to the workload.

To maintain consistency with the case used for the anonymous memory workload, we selected the same high and low demand duration values for the file-backed memory workloads. Figure 6.4 illustrates a representative case under these conditions. We observe that the RSS of the tenant container when the MS is activated can track the actual application memory accesses even for quick changes from high to low memory access sizes. The MS achieves the same adaptive behavior in the RSS of the container across all tested combinations of low and high demand duration.

More specifically, the application of memory contention results in reductions in the RSS when there is low memory demand and increases in the RSS when there is high memory demand. Overall, memory usage follows the access patterns observed throughout the experimental execution. However, for small low and high demand

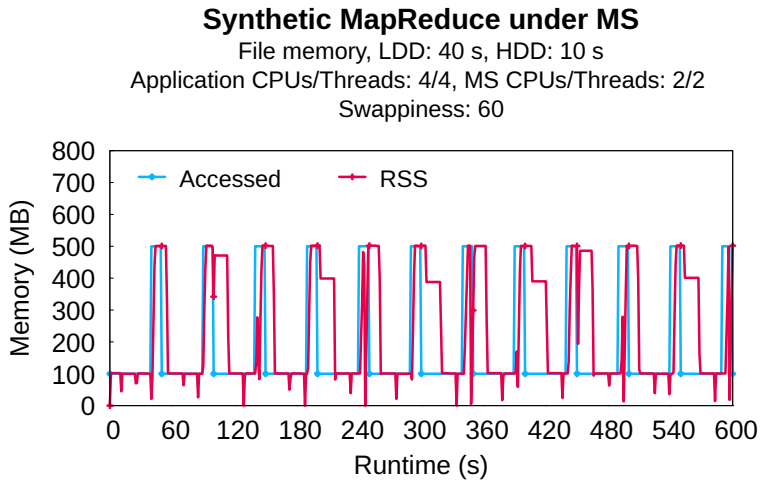

Figure 6.4: Spiky file accesses using the synthetic workload generator with the following parameter settings: LMD=100MB, HMD=500MB, LDD=40s, HDD=10s, and runtime=600s.

duration ratios, memory contention seems to decrease the application's RSS in an attempt to catch up with the low memory demand, but it cannot actually reach or exceed it immediately. This behavior is attributed to the Linux kernel's handling of file pages, which moves accessed file pages to the active LRU list and keeps them active as long as they are accessed. Meanwhile, the MS primarily tracks the inactive LRU list size as an indication of sufficient idle memory pages in the container.

Moving to larger ratios between the low and high demand duration, the application's RSS increases from low memory demand to high memory demand when the access pattern of the workload changes. After the RSS increases to the high memory access size and a subsequent change in the working set is observed by the MS, the RSS may decrease in a lower value and then increase again to a value near the high memory demand before decreasing to reach the low memory demand of the workload. When the workload accesses the low memory amount, due to contention in memory, the RSS decreases to reach the low memory demand and may even drop instantly below this value during the adaptation process.

In the experiments, where we randomly vary either the memory demand alone or both the memory demand and the memory demand duration, as results in Table 6.4 indicate, the MS responds to the application's changing memory needs, but the frequency of these changes influences its effectiveness.

When the time between low and high memory phases is short, the RSS may slightly lag behind or undershoot the actual memory needs, as it struggles to keep

Table 6.4: Average memory accessed by the workload, Resident Set Size (RSS) of the tenant container, and total allocated memory by the file-backed memory workload in MB across different high and low demand durations.

LMD (MB)	**HMD (MB)**	**LDD (s)**	**HDD (s)**	**Accessed (MB)**	**RSS (MB)**
100	500	10	10	299.67	446.84
100	500	10	20	366.23	485.16
100	500	10	40	419.46	464.93
100	500	10	80	453.41	478.52
100	500	20	10	233.11	427.30
100	500	40	10	179.86	265.73
100	500	80	10	133.27	190.40
200-400	400-600	60	10	322.07	358.54
200-400	400-600	20-100	1-10	319.16	347.73

up with rapid changes in demand. This delay likely occurs because the MS requires time to detect shifts in memory usage and adjust accordingly by releasing memory.

In the file-backed memory workload experiments, memory reads are performed in chunks of a configurable size. While we tested the buffer size with configurations larger (1 MB) than the default 4 KB, we primarily use the default value in our experiments. However, the larger sizes showed no difference in terms of memory reclaiming efficiency.

In Table 6.4, we observe that the RSS of the tenant container closely aligns with the application's actual memory access patterns as we move from smaller to larger ratios between the low and high demand duration. Regarding smaller LDD/HDD ratios, the RSS/Accessed ratio increases because the Linux kernel requires time to move pages from the active to the inactive LRU list. This delay in moving pages between the two lists affects the activation of MS, which relies on the size of the inactive LRU lists.

In scenarios where the high demand duration (HDD) increases compared to the low demand duration (LDD), the RSS and the accessed memory tend to be close to the high memory demand. This occurs because the memory accessed during the high demand periods is frequently used, making it part of the working set rather than underutilized memory. As a result, the MS is unable to reclaim these memory

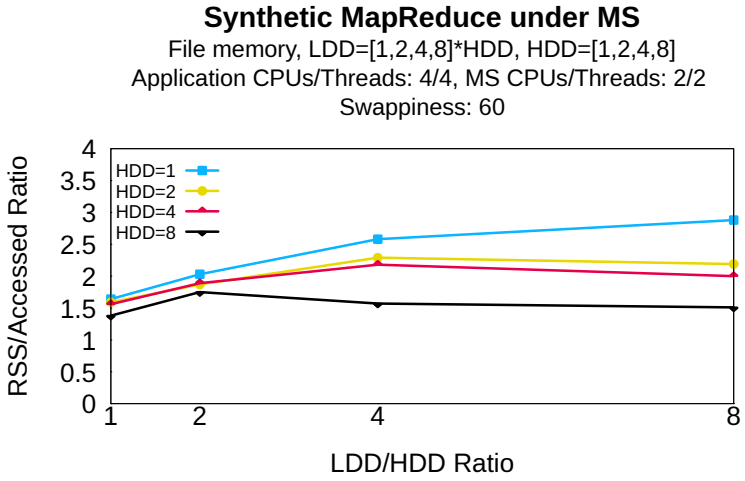

Figure 6.5: The RSS/Accessed ratio for file-backed memory workloads under different low and high demand durations. The experiments were conducted using the following parameter settings: LMD=100MB, HMD=500MB, HDD=[1,2,4,8]s, LDD=[1,2,4,8]×HDDs, and runtime=600s.

pages effectively, as they are actively accessed by the application. This behavior reflects the MS's design to avoid reclaiming frequently accessed memory to ensure sufficient memory availability during high demand phases.

The RSS of the tenant container closely approximates the memory accessed by the application. Additionally, the difference between the container's RSS and the application's accessed memory diminishes as the demand durations ratios increase. More specifically, the difference between the RSS and the accessed memory in the container ranges from 147 MB to 30 MB as the ratio between the low and high memory demand increases.

Similarly to the evaluation of anonymous memory workloads, we examine scenarios where the high demand duration (HDD) takes values of 1, 2, 4, and 8, while the low demand duration (LDD) is defined as a multiple of the HDD by factors of 1, 2, 4, and 8. Additionally, we explore cases where the LDD takes values of 1, 2, 4, and 8, with the HDD scaling proportionally to the LDD. Figures 6.5 and 6.6 summarize the RSS/Accessed ratio for file-backed memory workloads managed by the MS under these scenarios.

The results show that when the LDD increases, the greater the difference between the LDD and HDD, the RSS/Accessed ratio is lower to the optimal value. For example, in cases where (HDD:8, LDD:HDD×4), and (HDD:8, LDD:HDD×8) the RSS/Accessed ratio is equal to 1.57 and 1.51, respectively. In contrast, when the LDD is closer to

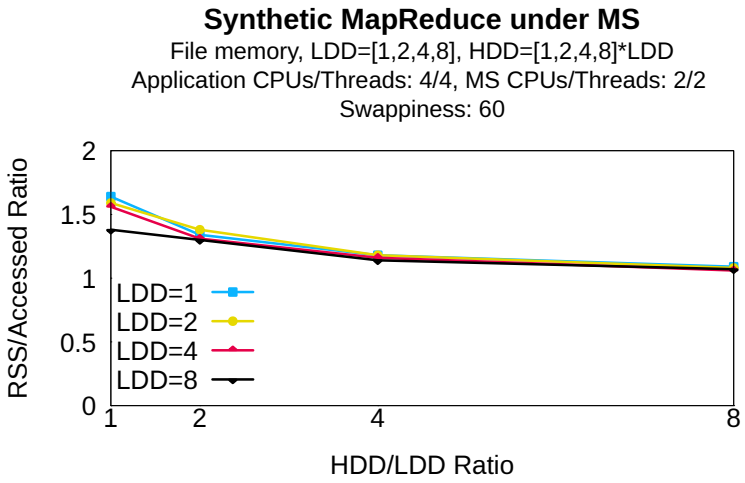

Figure 6.6: The RSS/Accessed ratio for file-backed memory workloads under different low and high demand durations. The experiments were conducted using the following parameter settings: LMD=100MB, HMD=500MB, LDD=[1,2,4,8]s, HDD=[1,2,4,8]×LDDs, and runtime=600s.

the HDD, the system struggles to identify which memory demand size is part of the working set due to the frequency with which the memory is accessed. In these cases, we observe that the RSS/Accessed ratio ranges from 1.64 to 2.88.

In scenarios where the HDD increases relative to the LDD, the average accessed memory stabilizes near the high memory demand (HMD). This indicates that the MS mechanism successfully retains the HMD in memory, identifying it as critical due to its frequent access. The system correctly classifies the HMD as active, avoiding its reclamation as underutilized.

6.2.2 Concurrency

To examine the impact of concurrency on the reclamation of idle memory pages, we conduct a series of experiments with varying levels of container concurrency between the container running the MS process and the container running the tenant application. Table 6.5 outlines the different cases that we tested.

Testing different degrees of container concurrency, as shown in Figure 6.7, we observe that changing the number of cores (#threads = #cores) used by the MS does not lead to more aggressive memory reclamation on the anonymous memory workloads. Despite the high initial memory allocations, only a small fraction is actively accessed, indicating potential over-allocation. This over-allocation appears to

Table 6.5: Different degrees of concurrency for the application and MS tested, where #threads = #cores.

Cases	Application Cores	MS Cores
Case 1 (default)	4	2
Case 2	2	4
Case 3	6	1
Case 4	2	2

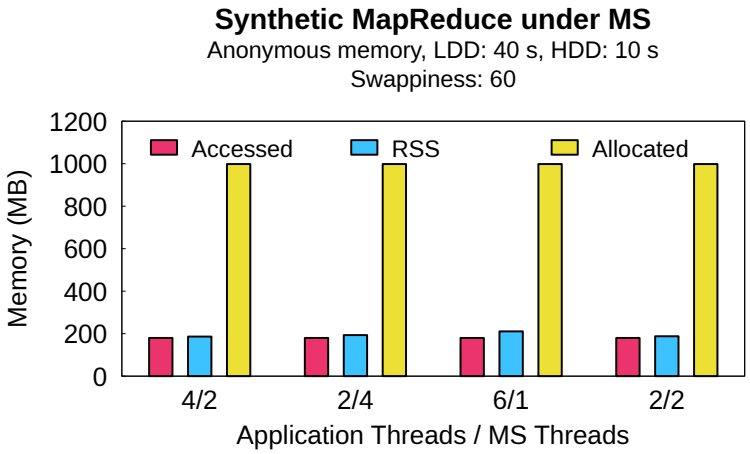

Figure 6.7: MapReduce-like anonymous memory workload with different degrees of container concurrency for both the application and the MS, where #threads = #cores.

be efficiently managed by the MS. The combination of high RSS and low accessed memory suggests that the MS optimizes the memory usage by dynamically adjusting the memory allocations based on the actual access patterns.

For file-backed workloads, the system maintains efficient memory usage without excessively retaining memory, as shown by the close alignment of RSS and accessed memory values. As shown in Figure 6.8, increasing concurrency does not affect these metrics significantly, suggesting that file-backed memory is efficiently managed, allowing the system to dynamically adjust and reclaim memory as needed based on the actual access patterns.

In a sensitivity analysis focusing on data access speed, we evaluate the Memory Stresser (MS) effectiveness under varying physical core allocations and threading levels, as shown in Table 6.6. The experiments are conducted with fixed low and high memory demands of 100 MB and 500 MB, respectively, each lasting for 40 and

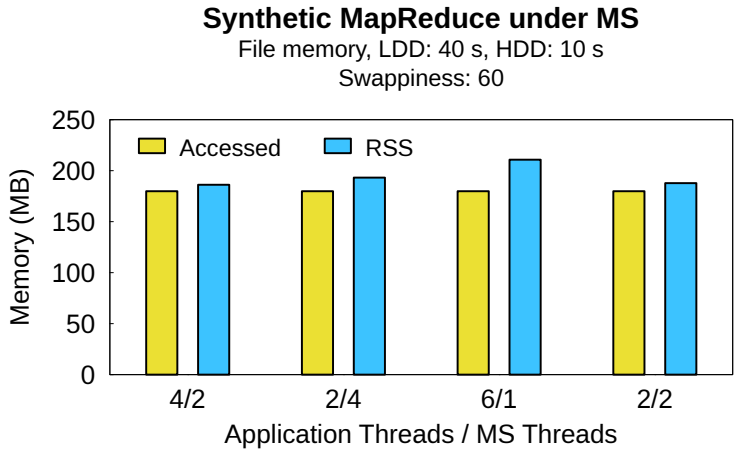

Figure 6.8: Spiky file accesses with different degrees of container concurrency for both the application and the MS, where #threads = #cores.

Table 6.6: Thread and core allocation configurations for application and Memory Stresser, where the #application cores = #MS cores. The experiments were conducted using the following parameter settings: LMD=100MB, HMD=500MB, LDD=10s, HDD=10s, and runtime=600s.

#Application Cores = #MS Cores	**Application Threads**	**MS Threads**
1	1	MS cores×[1,2,4]
2	2	MS cores×[1,2,4]
3	3	MS cores×[1,2,4]

10 seconds. The number of physical cores allocated to both the application container and the MS container increases from 1 to 3 in the tested scenarios. The number of application threads is equal to the number of application cores (1thread/core). For the MS threads, multiple configurations are tested, with the number of threads per MS core scaling as MS cores multiplied by factors 1, 2, and 4. This setup enables an analysis of how thread concurrency affects the effectiveness of the MS in memory reclamation.

Figure 6.9 represents the effectiveness of the Memory Stresser (MS) under different configurations of physical cores and threads for both the application (App) and Memory Stresser (MS). The y-axis represents the ratio of the RSS (Resident Set Size) to the accessed memory, which indicates the efficiency of the MS.

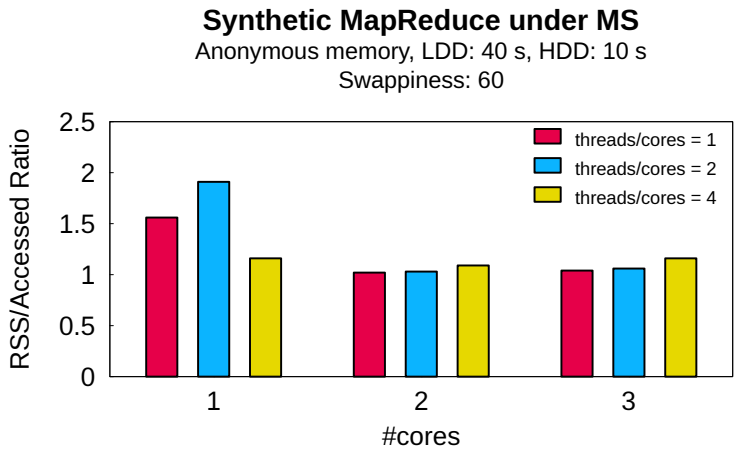

Figure 6.9: Impact of thread and core configuration on the MS effectiveness in a MapReduce-like application, where #application cores = #MS cores.

The results show that the RSS/Accessed ratio tends to decrease as the number of cores and threads increases in a balanced manner (e.g., matching cores and threads). The lowest ratio is observed for the configuration where 2 cores and 2 threads are used, indicating the highest memory efficiency. As the number of threads increases beyond the number of cores (e.g., 1 core with 2 or 4 threads), we observe higher RSS/Accessed ratios. Similarly, for higher core counts (3 cores), increasing the threads gradually impacts efficiency, with notable inefficiency occurring when threads significantly exceed cores (e.g., 12 threads). These results underscore the importance of balancing the number of cores and threads to optimize memory access and usage.

6.3 Memory Stresser with Multiple Tenants

We evaluate the MS using the synthetic workloads on two tenant containers. Each tenant container and the MS container is assigned 2 CPU cores, while 2 additional CPU cores are dedicated to the task that collects the resource usage statistics. The experimental applications running within the containers are configured with the following memory limits: 50 MB minimum memory, 100 MB low memory, 5.1 GB high memory limit, and 5.3 GB maximum memory limit for one tenant container. The second tenant container is set with a 2.1 GB high memory limit and 2.3 GB maximum memory limit. The third container, running the MS, is configured with the low, high, and maximum memory limits set equal to the total memory of the system.

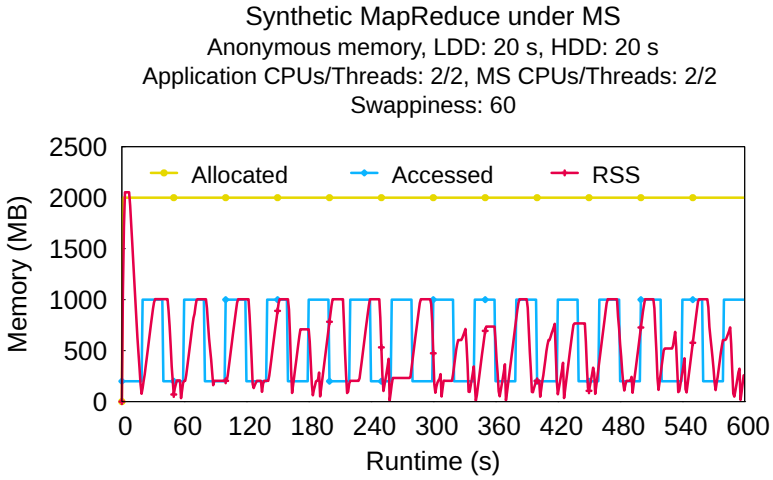

Figure 6.10: Performance of MS with two tenants running anonymous MapReduce-like workloads with the following parameter settings: HMD=500MB, LMD=100MB, HDD=20s, LDD=20s, and runtime=600s.

Initially, we examine the efficiency of the MS for both anonymous and file-backed workloads, where the high and low memory demands, as well as the demand durations, remain constant. Figures 6.10 and 6.11 demonstrate that the RSS (Resident Set Size) of the tenant containers closely follows the accessed memory, indicating that the MS efficiently releases idle memory pages. During transitions between low and high memory demand periods, the MS successfully adapts by aligning the RSS with the accessed memory, highlighting its ability to respond promptly to workload changes and ensure minimal memory waste.

The ratio of the average RSS to the average accessed memory provides further insights into the MS's efficiency in managing idle memory. For anonymous workloads, this ratio is approximately 0.92, indicating that the RSS is slightly below the accessed memory. This reflects the MS effectiveness in minimizing memory overhead while ensuring adequate memory allocation for application demands. In contrast, for file-backed workloads, the ratio is approximately 1.31, indicating that file-backed workloads tend to keep more idle pages in memory. However, the excess does not exceed 200 MB, demonstrating the capability of the MS to efficiently reclaim idle memory while maintaining performance stability.

We tested the MS with two tenant containers running file-backed workloads configured as follows: high memory demand ranging from 400–600 MB, low memory demand ranging from 200–400 MB, high demand duration ranging from 20–100 seconds, and low demand duration ranging from 1–10 seconds. As shown in Fig-

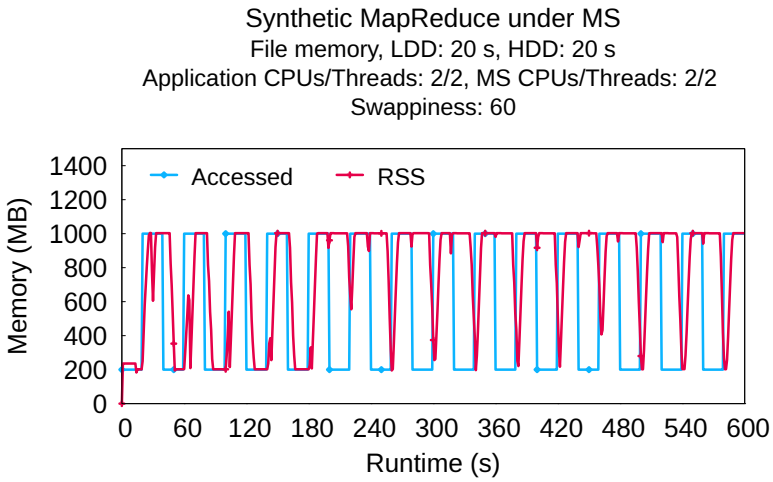

Figure 6.11: Performance of MS with two tenants running file-backed MapReduce-like workloads with the following parameter settings: HMD=500MB, LMD=100MB, HDD=20s, LDD=20s, and runtime=600s.

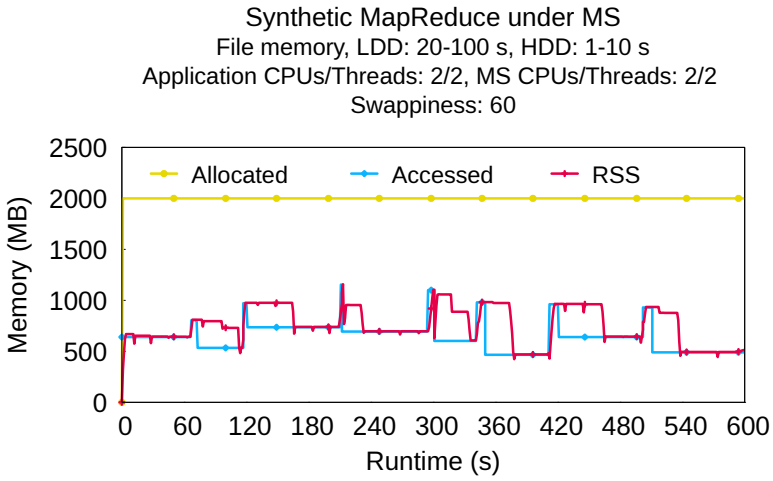

Figure 6.12: Performance of MS with two tenants running file-backed MapReduce-like workloads with the following parameter settings: HMD=400-600MB, LMD=200-400MB, HDD=20-100s, LDD=1-10s, and runtime=600s.

ure 6.12, the MS effectively adapts to the dynamic memory demand changes of the workload. Despite the fluctuating nature of the memory demands, the RSS of the containers closely tracks the accessed memory, demonstrating that the MS efficiently releases idle memory pages. The average ratio of RSS to accessed memory for this workload is approximately 1.19, indicating that the RSS is slightly higher than the accessed memory. This ratio is close to the ideal value of 1. The MS efficiently balances the aggressiveness of releasing unused memory pages to ensure optimal performance and utilization throughout the workload runtime.

6.4 Memory Stresser with MapReduce Applications

We evaluate the effectiveness of the Memory Stresser (MS) using real-world MapReduce applications that process large datasets and are commonly executed in multi-tenant cloud environments [1]. We focus on two representative MapReduce applications: Grep and KMeans. Each application is executed within a container with a memory limit of 5GB. For each application, we test two scenarios: (a) the execution of the MapReduce application without MS enabled, and (b) the execution of the MapReduce application with the MS enabled.

6.4.1 Setup and Datasets

We utilize two machines: one is configured with the HDFS file system, while the other hosts Hadoop [34], enabling the execution of MapReduce applications. We run Hadoop and HDFS 3.4 version. The HDFS file system stores data in a 2 TB Western Digital WD2002FAEX-0 SATA hard disk.

For the Grep application, we use the Wikipedia dataset (50GB) [35]. For the KMeans application, we employ the 20_Newsgroups dataset [36]. In all scenarios, we monitor the memory usage of each application and the execution time of the map-reduce operations. Table 6.7 summarizes the datasets used.

Table 6.7: Summary of the datasets used as input to the MapReduce applications.

Dataset	**Structure**	**Record Size**	**Total Size**
Wikipedia	XML format with records for articles, revisions, and users	Variable (256 KB to 1 GB)	50 GB (8 GB subset used)
20 Newsgroups	Subdirectories with text files, one per document	Variable (425 B to 158 KB)	91 MB

The Wikipedia dataset consists of data files extracted from Wikipedia articles. The data is structured in XML format, with individual records for articles, revisions, and

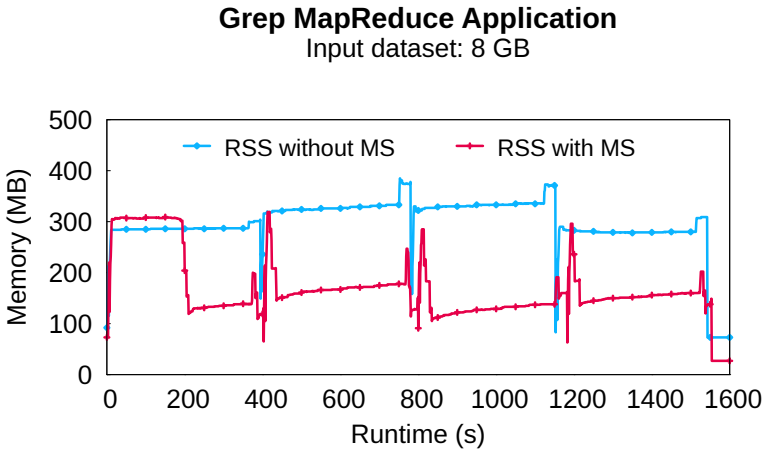

Figure 6.13: Grep MapReduce application memory usage with and without the use of the MS.

users. The size of each file ranges from 256 KB to 1 GB. For our experiments, we extract a random subset of files from the 50 GB dataset, which is equal to 8 GB. This dataset is typically used for tasks like text mining, topic modeling, and clustering.

The 20 Newsgroups dataset consists of approximately 20,000 newsgroup documents, distributed across 20 different topics, such as comp.graphics, rec.autos, and soc.religion.christian. The data is organized into subdirectories, each representing a newsgroup, with each file containing a single document. The documents vary in length, ranging from 425 B to 158 KB. We use the original unmodified dataset, whose total size is 91 MB, and it is widely used for text classification and clustering experiments.

6.4.2 Experiments Results

Figure 6.13 illustrates the memory usage of the Grep application. We run the Grep application with four different patterns in the dataset. We observe that without the MS enabled, memory usage remains almost constant during the application's execution, ranging from 280 MB to 380 MB, with occasional spikes and drops in memory usage, falling below 100 MB when the pattern changes.

When the MS is enabled, the kernel reclaims an amount of underutilized memory from the container running the Grep application. This results in small fluctuations in memory usage, caused by the mechanism's efforts to free the underutilized memory, combined with the continuous loading of new data into memory by the MapReduce

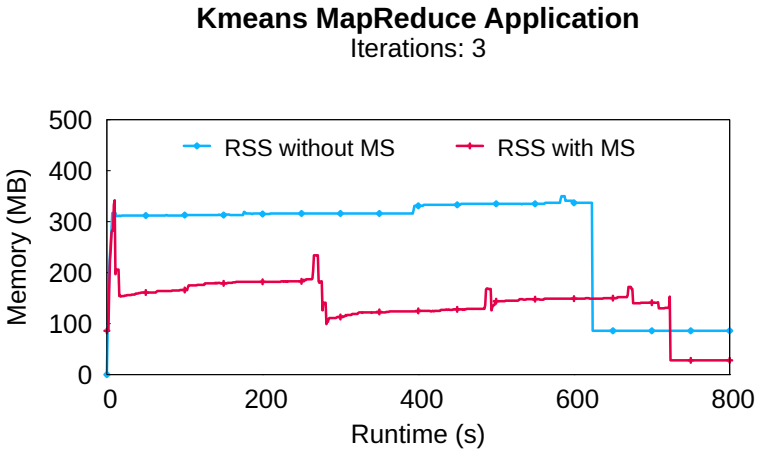

Figure 6.14: KMeans MapReduce application memory usage with and without the use of the MS.

process. We observe that the RSS of the container decreases and aligns with the memory usage patterns of the executing Grep application. More specifically, the average memory usage during execution with the MS enabled is 167.53 MB, compared to 297.16 MB when the MS is disabled.

Figure 6.14 shows the memory usage of the container over time during the execution of the KMeans application, which we configure to run for three iterations. Without the MS enabled, the memory usage remains relatively stable, fluctuating between approximately 280 MB and 380 MB throughout the runtime, with minimal variations. When the MS is enabled, the memory usage is reduced to about 170 MB. The memory usage exhibits noticeable fluctuations, with periodic slight drops in the RSS. This behavior occurs intermittently, and the RSS tends to increase at the beginning of each iteration as the application loads new data into memory, peaking at 170 MB. The MS forces the container to release unused memory, reducing the memory usage to 100 MB.

In Figure 6.15, we observe the performance of the Grep and KMeans MapReduce applications, which is minimally affected by the MS initiation. Without the MS enabled, the Grep application completes in 932s, while with the MS enabled, it completes in 917s. Additionally, we observe that the performance of the KMeans application remains nearly constant, with a slight delay when the MS is enabled. More specifically, the completion time of the application increases from 502.33s (without the MS) to 515.67s (with the MS).

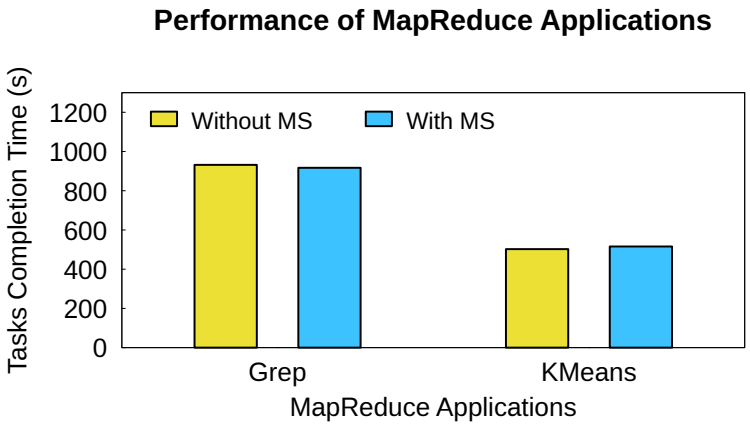

Figure 6.15: Performance of MapReduce applications without and with the use of the MS.

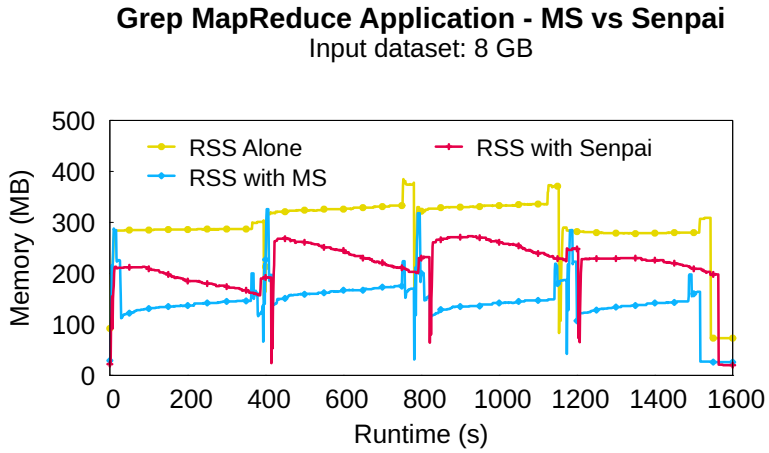

Figure 6.16: Grep MapReduce application memory usage alone, with MS, and with Senpai.

6.5 Comparison with other approaches

We further compare our Contention Service solution to a similar user-space mechanism called Senpai [23, 24]. Senpai runs as a user-space agent and aims to detect cold pages in local memory and move them to swap space by triggering memory reclamation. It determines the amount of memory to reclaim based on PSI (as described in Section 3.2.5) and aims to maintain a low memory pressure threshold. We use the open-source implementation of Senpai that adjusts the cgroup high memory limit during execution within a cgroup. Senpai dynamically adjusts a cgroup's memory range between MIN_SIZE and MAX_SIZE, based on PSI memory pressure

Table 6.8: Grep and KMeans MapReduce applications memory usage Alone, with the MS, and the Senpai.

MapReduce Application	**Alone**	**with MS**	**with Senpai**
Grep	240.70 MB	148.98 MB	231.41 MB
KMeans	179.55 MB	83.76 MB	118.15 MB

information, as outlined in [24].

Figure 6.16 illustrates the RSS (Resident Set Size) of the container during the runtime of the Grep MapReduce application under three scenarios: no external mechanism intervention, with MS enabled, and with Senpai enabled. More specifically, in the case where there is no intervention, the application runs without any memory-saving mechanisms like MS or Senpai. The memory usage remains relatively high and stable throughout the runtime, without significant fluctuations, as it can fully utilize the allocated memory, which exceeds its peak memory demands, without constraints.

With Senpai, the memory usage shows noticeable drops at periodic intervals. The memory starts at a higher memory usage demand, which periodically decreases. This indicates that Senpai forces the container to release the underutilized memory back to the system. The drops in memory usage occur as Senpai directs the kernel to perform memory reclamation and move the cold pages to disk. Senpai achieves average memory usage equal to 231.41 MB, compared to the execution of the Grep application without an external mechanism intervention, which achieves an average memory usage of 240.7 MB, helping to conserve memory system resources.

When the MS is enabled, we observe lower average memory usage in comparison to Senpai. The RSS exhibits sharper and more frequent drops, showing the MS's aggressive memory management, which triggers the memory reclamation process that releases the memory that is no longer actively in use by the application. The average memory usage during runtime is equal to 148.98 MB and smaller than when we use Senpai. The MS ensures that the application operates with minimal memory overhead, likely targeting the underutilized or redundant memory pages for reclamation more frequently than the Senpai.

Figure 6.17 depicts the RSS of the container while executing the KMeans MapReduce application under the three scenarios. When no external mechanism intervenes, the application's memory usage is 179.55 MB. With Senpai's intervention, the mem-

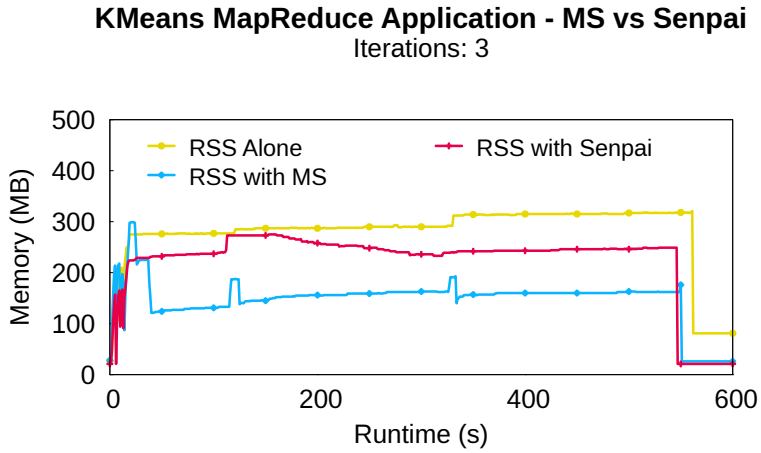

Figure 6.17: KMeans MapReduce application memory usage alone, with MS, and with Senpai.

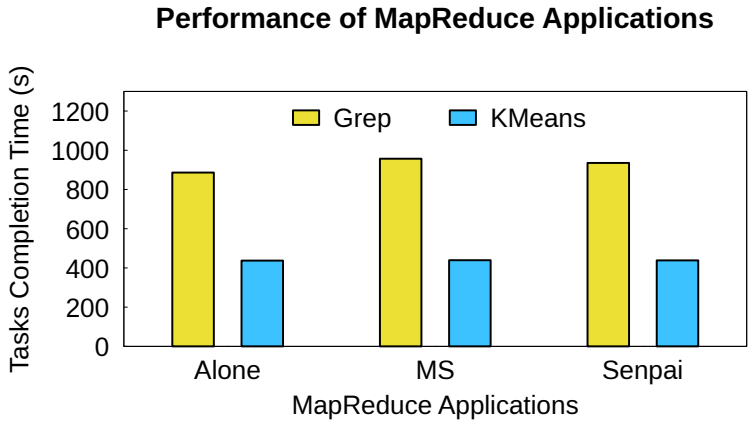

Figure 6.18: Performance of Grep and KMeans MapReduce application alone, with MS, and with Senpai.

ory resident in the cgroup is reduced to 118.15 MB, while our approach achieves a further reduction to 83.76 MB. In overall (table 6.8), the MS tracks the actual memory demands of the applications over time with higher accuracy and releases up to 36% more underutilzed memory than Senpai.

Figure 6.18 presents the completion time of the Grep and KMeans MapReduce applications when there is no mechanism intervention, with MS enabled, and with Senpai enabled. The completion time of the Grep application when a mechanism is applied to release underutilized memory decreases compared to the case where no external mechanism intervention is applied. More specifically, the application completion time is 886.24 seconds without any intervention, 956.91 seconds with MS

Table 6.9: Memory pressure on the Grep and KMeans MapReduce applications under MS.

MapReduce Application	**PSI (s) with MS**	**PSI (s) with Senpai**
Grep	2	1.58
KMeans	1.13	0.40

enabled, and 935.32 seconds with Senpai enabled. While enabling the MS introduces a slight overhead (7.9%), our focus is on how much memory is released. In comparison to Senpai, which adds 5.5% overhead, our approach efficiently reclaims more underutilized memory. We prioritize memory savings and optimize resource utilization by reducing the number of machines required for a given number of tenants. Our approach allows for the support of more tenants without significant overhead. The trade-off between overhead and the resulting benefits is minimal, making our approach efficient.

The completion time of the KMeans application for the three scenarios demonstrates the performance differences introduced by memory management mechanisms. More specifically, the completion time is 437.42 seconds without any intervention, 439.32 seconds with MS enabled, and 438.48 seconds with Senpai enabled. The results indicate that a slight overhead introduced by the memory management mechanisms when reclaiming underutilized memory.

To track memory pressure, PSI records the time spent on events that occur exclusively when there is a shortage of memory. The memory.pressure ”some total” metric provides an aggregate measure of memory pressure over time, indicating how frequently and for how long processes experience memory delays due to contention. We measure the PSI metric provided by the *memory.pressure* file in *cgroup v2* for both the MS and Senpai mechanisms when running the Grep and KMeans MapReduce application.

The results in Table 6.9 indicate that with MS enabled, the application experiences a total of 2 seconds of stalls during runtime, compared to 1.58 seconds with Senpai. While our approach introduces slightly more overhead, it prioritizes the proactive reclaim of underutilized memory to improve memory usage efficiency. By generating controlled memory contention, our approach adapts the RSS of the container to the application’s changing demands, ensuring that underutilized memory is released with

minimum impact on performance.

In contrast, Senpai periodically adjusts the amount of memory to be reclaimed to match the application's peak demand. However, it fails to dynamically align the container's RSS with the application's memory patterns over time, leading to fewer delays but reclaiming less underutilized memory.

Overall, our approach achieves lower memory usage compared to Senpai without adversely affecting application performance. It also performs more efficient memory reclamation by reclaiming idle memory faster, as shown in Figures 6.16 and 6.17.

6.6 Sensitivity Analysis

We conduct a sensitivity analysis to evaluate how the configurable parameters, such as the *Refault Ratio Threshold (RRT)*, the *MIN_CHUNKS*, and the memory contention size (*mc_size*) influence the performance of both synthetic and real-world applications, aiming to iteratively optimize the Contention Service configuration.

We conduct the experiments using a synthetic workload generated by the workload generator described in Section 4.11. The workload is configured with a high memory demand (HMD) of 500 MB, a low memory demand (LMD) of 100 MB, a high demand duration (HDD) of 40 seconds, and a low demand duration (LDD) of 10 seconds, with a total runtime of 600 seconds.

More specifically, we examine a range of values for the *Refault Ratio Threshold (RRT)*, the *MIN_CHUNKS*, and the *mc_size* parameters, which primarily influence the termination of memory contention, and the intensity of memory contention, which consequently affects the application's performance.

Table 6.10: Synthetic application performance achieved under various values of the MIN_CHUNKS parameter, with Accessed=420 MB, and RRT=0.04.

MIN_CHUNKS	**RSS (MB)**	**PSI (s)**	**RSS/Accessed**
128	453.21	11.78	1.08
256	400.14	12.60	0.95
512	387.45	18.33	0.92
1024	370.51	21.92	0.88

Table 6.11: Synthetic application performance achieved under various values of the RRT parameter, with Accessed=420 MB, MIN_CHUNKS=128 and LSR=0.4.

RRT	**RSS (MB)**	**PSI (s)**	**RSS/Accessed**
0.01	643.23	4.10	1.53
0.02	515.80	4.87	1.23
0.03	476.19	6.65	1.13
0.04	453.21	11.78	1.08
0.05	404.11	40.14	0.96

Table 6.12: Synthetic application performance impact under different intensities of the memory contetion, with Accessed=420 MB, and MIN_CHUNKS=128.

mc_size	**PSI (s)**
mem_free - wm_low	0.4
mem_free - wm_min	18.83

Table 6.10 shows that the optimal configuration is achieved when the value of the *MIN_CHUNKS* parameter balances reduced *PSI (Pressure Stall Information)* with efficient memory usage, indicated by the lowest *RSS/Accessed* ratio. As the *MIN_CHUNKS* value decreases, the PSI value decreases, indicating reduced stalls caused by memory contention. However, the *RSS/Accessed* ratio slightly increases but remains close to the optimal value of 1. An *RSS/Accessed* value below 1 suggests that memory pages belonging to the application's working set are reclaimed by the Memory Stresser. The optimal configuration is observed when refaults are checked every 128 memory chunks, where PSI is minimized, and average memory usage aligns closely with the application's actual memory demands.

Table 6.11 presents the impact of varying the *RRT (Refault Ratio Threshold)* on memory contention performance. The results indicate that as the *RRT* value increases, the *RSS/Accessed* ratio tends to be closer to the optimal value of 1, reflecting more efficient utilization of memory. However, higher *RRT* values are also associated with increased *PSI* values, which signal for increased memory pressure due to contention. The balance point, where *PSI* is reduced while keeping memory usage close to the application's actual demands, occurs when the *RRT* is equal to 0.04. For values larger than 0.04, we observe that the memory pressure increases to 40s.

Table 6.13: Contention Service Configuration.

Parameter	**Setting**
List Size Ratio (LSR)	0.4
List Size Difference (LSD)	1.05
File List Size (FLS)	100 MB
Refault Ratio Threshold (RRT)	0.04
MIN_CHUNKS	128
Memory Contention Size (mc_size)	mem_free - wm_low

The memory contention size (mc_size) is a critical parameter, as it represents the minimum amount of memory that must be allocated to trigger the kernel's background page reclamation process in order to reclaim the underutilized memory and avoid degrading the performance of the applications.

We examine the impact of the memory contention size on application performance by analyzing two scenarios, as shown in table 6.12. The results indicate the corresponding stalls the application experiences during each scenario. When the memory contention size is sufficient to touch the low watermark (mem_free - wm_low), the application experiences 0.4 seconds of stalls due to memory contention. However, when the memory contention size is increased to reach the minimum watermark (mem_free - wm_min), the stalls increase to 18.83 seconds. Therefore, it is necessary to define the appropriate memory contention size that leads to reclaiming as much underutilized memory as possible without significantly impacting the performance of the applications.

Table 6.13 summarizes the chosen configuration of the Contention Service, presenting the parameter settings that increase the memory reclamation efficiency while minimizing the application performance impact. Note that we set the List Size Ratio (LSR) and List Size Difference (LSD) parameters empirically in order to activate the Contention Service when there is new application activity on the system or there is a transition in the workload of the application. However, we leave further experimentation to identify the optimal values for these parameters as a future work.

6.7 Summary

The evaluation of the Memory Stresser (MS) mechanism in both single-tenant and multi-tenant scenarios demonstrates its effectiveness in managing memory contention in cloud environments. The MS is capable of dynamically adapting to changes in memory demand, effectively reclaiming the underutilized memory and ensuring minimal memory waste. In a multi-tenant environment, the MS activates the Linux kernel page reclamation process, in order to release the underutilized memory from one tenant and provide sufficient resources to other tenants.

By closely tracking the memory access patterns of the applications, the MS aligns the Resident Set Size (RSS) of the containers with the actual memory requirements of the workloads. This results in eliminating memory over-provisioning and ensures that containers receive the memory they need while the underutilized memory is reclaimed.

The results also highlight that the MS's ability to free up the underutilized memory while maintaining steady the application's performance, even during the execution of real-world MapReduce tasks such as Grep and KMeans.

Compared to Senpai, the MS consistently achieves better results, significantly reducing the memory usage. The MS reduces the average Resident Set Size (RSS) to 148.98 MB for Grep and approximately 83.76 MB for KMeans, outperforming Senpai's average memory usage of 231.41 MB and 118.15 MB. Additionally, the MS has a minimal impact on performance, with only a slight increase in completion time for KMeans. Overall, the MS reclaims underutilized memory more quickly than Senpai, conserving more memory resources

Chapter 7

Related Literature

In this chapter, we review comparative studies addressing dynamic memory management in containerized environments, focusing on methods to optimize resource allocation and prevent memory pressure. We also discuss significant works aimed at enhancing idle-page detection and reclamation of underutilized memory allocated by applications. Finally, we compare our proposed solution, the Contention Service, which identifies underutilized memory pages and applies controlled memory contention to activate the Linux kernel memory reclamation mechanism, enabling efficient memory recovery without performance degradation, to related research.

7.1 Memory management

Chen et al. introduce Pufferfish [30], an elastic memory manager that dynamically adjusts the memory limits of containers on-the-fly to accommodate data-intensive applications. Pufferfish returns memory to the system by first verifying that enough

memory is available on the node before launching new containers. If memory is insufficient, it reclaims memory from low-priority containers that experience memory pressure, starting with the lowest priority to minimize impact on critical applications. It uses a lazy approach, delaying reclamation until necessary, and in extreme cases, it terminates the lowest-priority containers to quickly free up resources and prevent out-of-memory (OOM) errors. By using CPU and CPUSET files from the cgroup interface, it reallocates CPU resources and sets CPUSET to a single core, enabling direct memory reclamation. However, this method cannot free unused memory from already running containers, potentially leading to memory waste. Pufferfish aims to improve flexibility in memory allocation among tasks and reduce out-of-memory (OOM) errors.

Autopilot uses the Borg [37] scheduler to dynamically resize the memory limits of container instances based on time-series measurements of CPU and memory usage, with the goal of minimizing the slack between the allocated and used memory. It leverages historical usage data and an ensemble of machine learning models to recommend optimal resource limits, optimizing cost functions based on job and infrastructure goals. Autopilot uses statistics, such as the peak usage, the weighted average resource usage, and a specific percentile of usage in order to ensure efficient resource utilization. The resource requirements of some applications may not always well captured by these statistics, leading to potential over-provisioning. Another limitation of Autopilot is that it is mainly designed and tuned for predicting the Google's workload needs [38].

Pi et al. develop Hermes [32], a mechanism that focuses on fast memory allocations for latency-critical services that use C libraries. Hermes addresses the inefficiencies in memory allocation by reserving resource slacks for latency-critical services, maintaining dedicated memory pools for each service, and advising the Linux OS to release file cache pages occupied by batch jobs. Hermes triggers memory reclamation when memory usage surpasses a predefined threshold. This approach focuses on proactive reclaiming file-backed pages trying to reduce the execution of direct memory reclamation.

Charon [31] is a cluster scheduling system that aims to detect and eliminate memory pressure caused by memory oversubscription. This approach activates the out-of-memory killer by examining memory pressure in terms of the major page faults and page evictions that occur within a specified time frame and aims to prevent memory thrashing by preemptively terminating processes. While this can prevent prolonged

thrashing, it can also lead to the abrupt killing of containers, which may not be desirable by tenants.

Laniel et al. introduce Memory Optimization Light (MemOpLight) [33], a system designed to reallocate memory from containers with lower resource requirements to those that are underperforming. MemOpLight dynamically adjusts memory allocations using application throughput and performance states, categorized into green, yellow, and red. When memory is scarce, it reclaims memory from green (fully satisfied) and yellow (adequately performing) containers, redistributing it to red containers that fail to meet their SLOs. This approach ensures critical containers receive necessary resources while minimizing system performance impact. However, MemOpLight requires kernel modifications and has only been evaluated with synthetic workloads, not real-world applications.

Weiner et al. propose Transparent Memory Offloading (TMO) [23], Meta's data center solution, which focuses on offloading unused memory pages allocated by the applications from main memory to heterogeneous devices through direct reclaim. They introduce a new metric called Pressure Stall Information (PSI) to measure the impact of memory pressure in the applications performance and a userspace agent called Senpai to decide how much memory to offload based on workload and hardware characteristics. Senpai proactively activates Linux kernel direct reclamation when the PSI metric exceeds a predefined threshold value by dynamically resizing container limits.

Maruf et al present MemTrade [6], a system that removes and distributes memory among tenants. Memtrade achieves the above by dynamically resizing the cgroup memory limits considering the degradation of the applications performance, using refault statistics as a key metric. It focuses on reclaiming idle memory pages from applications through direct reclaim, which can provide free memory pages quickly, but at the cost of disrupting application performance.

7.2 Idle-page detection

In their work, Maruf et al. propose a Transparent Page Placement (TPP) mechanism [25] for CXL-tiered memory subsystems, which seeks to optimize memory usage by identifying and placing cold pages efficiently in slower memory tiers. Their ap-

proach involves cold page detection and proactive page reclamation using the Linux kernel LRU-based age management mechanism. Their approach requires kernel modifications.

Lagar-Cavilla et al. in [26] leverage the idle-page tracking method [29] to track the age of pages. Their approach uses a machine learning technique to tune the idle age threshold aiming to achieve a stable page swapping rate. Pages older than this threshold are marked as eviction candidates.

Multi-Generational LRU (MGLRU) [27] and Data Access MONitoring (DAMON) [28] frameworks track page activity over multiple generations to identify and reclaim unused pages. Both require kernel modifications and aim to optimize memory management by analyzing detailed page access patterns.

Idle page tracking-based approaches [29, 27, 28] require kernel or hypervisor modifications, along with the CPU and memory overhead. Identifying idle pages based on the accessed bits in page table entry (PTE) needs continuous access bit monitoring, which results in increasing slowdowns with the application's memory footprint. This approach cannot accurately track the memory access patterns, as soon as this mechanism cannot consider multiple accesses within a certain period of time or catch the reuses of one physical page by multiple virtual pages.

7.3 Comparison with related work

In contrast to the above works, our method reacts preemptively without requiring kernel modifications, by enabling the existing Linux kernel LRU-based memory reclamation process to remove idle memory pages by monitoring the size of the inactive LRU lists and the number of swap-in events. We dynamically adjust the intensity of an external pressure activity to wake the kswapd thread and terminate the memory pressure when significant amount of memory is reclaimed, such that the free memory in the system reaches the high watermark. Our approach avoids direct reclaim on tenant containers, reclaiming memory with minimum performance degradation without altering container memory limits. We aim to prevent memory thrashing and remove underutilized memory allocated by the applications running in the tenant containers.

7.4 Summary

The related literature on memory management highlights various strategies to address dynamic memory allocation, contention, and idle-page detection. One group of approaches focuses on dynamically adjusting container memory limits and reclaiming resources based on priority or usage patterns. Methods such as Pufferfish [30] and Autopilot [37] dynamically reallocate memory to prevent out-of-memory (OOM) errors or optimize utilization. While Pufferfish reclaims memory from low-priority containers or terminates them as a last resort, Autopilot uses machine learning and historical data to resize memory limits, albeit with limitations in handling diverse workloads and statistical inefficiencies.

Another set of methods aims to improve memory allocation for specific workload requirements. Solutions like Hermes [32] prioritize latency-critical services by reserving memory pools and proactively reclaiming file cache pages. Similarly, Charon [31] addresses memory pressure by preemptively terminating processes based on page faults and evictions to prevent thrashing, though at the cost of abrupt container terminations. Techniques like MemOpLight [33] dynamically reallocate memory across containers based on performance metrics, while solutions such as TMO [23] and MemTrade [6] employ advanced metrics like PSI and refault statistics to optimize memory usage and reclaim idle pages. However, these approaches often rely on mechanisms like direct reclaim, which can lead to application performance degradation.

For idle-page detection [29], techniques such as Transparent Page Placement (TPP) [25] and frameworks like MGLRU [27] and DAMON [28] track page access patterns and identify cold pages for reclamation. These methods enhance memory utilization by moving idle pages to slower memory tiers or reclaiming unused pages. However, they generally require kernel modifications, incur monitoring overhead, and may fail to capture the memory access patterns, especially with large application footprints.

In contrast, the proposed method introduces a preemptive approach that leverages the Linux kernel's existing LRU-based reclamation process to remove idle pages without modifying the kernel. By dynamically adjusting external pressure to trigger memory reclamation, the method avoids the need for direct reclaim and prevents excessive tenant container performance degradation. This approach uniquely balances proactive memory management with minimal system disruptions.

Chapter 8

Conclusions

In this chapter, we outline directions in which our work could be extended in the future and conclusions that we received during the research of this thesis.

8.1 Concluding Remarks

In this study, we address the problem of the inefficient utilization of memory in multitenant cloud environments. This problem is caused by static resource allocation and overprovisioning to accommodate peak loads, which leaves large portions of reserved memory idle and unavailable for other applications. Our objective is to design a solution that efficiently identifies and reclaims underutilized memory, in order to optimize the usage of system resources and reduce waste, without affecting the application performance.

We propose Contention Service (CS), an application-agnostic user-space agent, which autonomously detects and reclaims the underutilized memory in containerized, multitenant cloud environments by relying on system metrics. CS employs controlled memory pressure to trigger the background memory reclamation mechanism of the Linux kernel, in order to identify and release idle memory without degrading

application performance. Our system leverages existing kernel interfaces, ensuring compatibility with existing applications and avoiding modifications to applications or the underlying kernel.

We evaluate our solution using both synthetic workloads and real-world MapReduce applicaitons like Grep and KMeans. We also evaluate the CS mechanism in both single-tenant and multi-tenant cloud scenarios. Our experiments assess the MS's ability to dynamically adapt to memory demands, reclaim the underutilized memory, and align the container Resident Set Size (RSS) with the actual application needs. Moreover, the MC in comparison to Senpai, the user-space agent introduced by Meta [23], reclaims up to 36% more memory without affecting the application performance. Overall our approach demonstrating effective memory management with minimal performance overhead in diverse workloads.

8.2 Future Work

In this thesis, we have proposed the Contention Service, a mechanism that proactively reclaims underutilized memory to enable memory saving and, in parallel, achieve minimum performance impact by applying the appropriate memory pressure according to the workload memory needs.

There are several directions for future work that could extend our system and improve our implementation. In this section, we list a number of interesting topics that need further research. Our experimental results are based on both synthetic workloads generated through a workload generator designed for emulating the memory access patterns of the MapReduce applications, and real-world MapReduce applications, such as Grep and KMeans. We also provide a comparison of our solution with a similar approach, Senpai introduced by Meta, which demonstrates that our solution faster reclaims the underutilized memory with minimum stalls in the application's execution. Further experimentation using benchmarks on diverse datacenter workloads (e.g., Redis, and Memcached) would be necessary to generalize our findings and validate the applicability of our approach across a broader range of scenarios.

In addition to controlled experiments, a crucial next step is the deployment and evaluation of our system in a real datacenter environment. Testing our approach on a large-scale infrastructure with production workloads would provide valuable

insights into its scalability, robustness, and real-world effectiveness. Such an environment would also reveal practical challenges, such as integration with existing resource management frameworks and handling diverse workload patterns.

One potential improvement is to dynamically adjust the configurable parameters based on the specific needs of each tenant. One approach to accomplish this is to rely on machine learning models to tune the parameters.

Last but not least, another potential topic of interest is to explore the energy efficiency of our solution. We expect that we can save memory resources through the reclaim of the underutilized memory, as fewer machines would be needed to support more tenants. This reduction in resource consumption can enhance the overall energy efficiency, particularly in large datacenter environments where energy costs are a critical factor.

Bibliography

[1] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2011.

[2] Kernel.org, *Describing Physical Memory*. [Online]. Available: https://www.kernel.org/doc/gorman/html/understand/understand005.html

[3] V. Babka, "Linux memory management (with focus on page allocations)." [Online]. Available: https://d3s.mff.cuni.cz/files/teaching/nswi161/2022_23/05_linux_memory.pdf

[4] Y. Liang, J. Li, R. Ausavarungnirun, R. Pan, L. Shi, T.-W. Kuo, and C. J. Xue, "Acclaim: Adaptive memory reclaim to improve user experience in android systems," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 897–910.

[5] Y.-Q. Chou, L.-W. Shen, and L.-P. Chang, "Rectifying skewed kernel page reclamation in mobile devices for improving user-perceivable latency," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 5s, Sep. 2023.

[6] H. A. Maruf, Y. Zhong, H. Wang, M. Chowdhury, A. Cidon, and C. Waldspurger, "Memtrade: Marketplace for disaggregated memory clouds," *ACM on Measurement and Analysis of Computing Systems*, vol. 7, no. 2, May 2023.

[7] M. Frisella, S. L. Sanchez, and M. Schwarzkopf, "Towards increased datacenter efficiency with soft memory," in *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. Association for Computing Machinery, 2023, p. 127–134.

[8] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Transactions on Computer Systems*, vol. 30, no. 4, Nov. 2012.

[9] Y. Zhang, 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 *USENIX Annual Technical Conference*. USENIX Association, Jul. 2020, pp. 785–798.

[10] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, 1995, p. 79–95.

[11] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation (3rd Edition)*. Prentice-Hall, Inc., 2005.

[12] Denning, Peter J., "Virtual memory," *ACM Computing Surveys*, vol. 2, no. 3, p. 153–189, Sep. 1970. [Online]. Available: https://doi.org/10.1145/356571.356573

[13] Peter J. Denning, "Working set analytics," *ACM Computing Surveys*, vol. 53, no. 6, Feb. 2021. [Online]. Available: https://doi.org/10.1145/3399709

[14] T. K. D. Community, *Documentation for /proc/sys/vm/*. [Online]. Available: https://docs.kernel.org/admin-guide/sysctl/vm.html

[15] Peter J. Denning, "The working set model for program behavior," *Commun. ACM*, vol. 11, no. 5, p. 323–333, May 1968. [Online]. Available: https://doi.org/10.1145/363095.363141

[16] Denning, Peter J., "The locality principle," *Communication Networks and Computer Systems ACM*, vol. 48, no. 7, p. 19–24, Jul. 2005. [Online]. Available: https://doi.org/10.1145/1070838.1070856

[17] D. C. Marinescu, *Cloud Computing: Theory and Practice*, 1st ed. Morgan Kaufmann Publishers Inc., 2013.

[18] B. K. R. Vangoor, P. Agarwal, M. Mathew, A. Ramachandran, S. Sivaraman, V. Tarasov, and E. Zadok, "Performance and resource utilization of fuse user-space file systems," *ACM Transactions on Storage*, vol. 15, no. 2, May 2019.

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

[20] P. Menage, *Control Groups*. [Online]. Available: https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html

[21] T. K. D. Community, *Cgroups v2 Documentation*. [Online]. Available: https://docs.kernel.org/admin-guide/cgroup-v2.html

[22] B. Singh and V. Srinivasan, "Containers: Challenges with the memory resource controller and its performance," in *Ottawa Linux Symposium (OLS)*. Citeseer, 2007, p. 209.

[23] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos, "Tmo: Transparent memory offloading in datacenters," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2022, p. 609–621.

[24] "Senpai." [Online]. Available: https://github.com/facebookincubator/senpai/blob/main/senpai.py

[25] H. A. Maruf, 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 *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2023, p. 742–755.

[26] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan, "Software-defined far memory in warehouse-scale computers," in *International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2019, p. 317–330.

[27] J. Corbet, "The multi-generational lru." [Online]. Available: https://lwn.net/Articles/851184/

[28] S. Dr Park, "Damon: Data access monitoring framework for fun and memory management optimizations." [Online]. Available: https://lpc.events/event/7/contributions/659/contribution.pdf

[29] T. K. D. Community, "Idle page tracking." [Online]. Available: https://docs.kernel.org/admin-guide/mm/idle_page_tracking.html

[30] W. Chen, A. Pi, S. Wang, and X. Zhou, "Pufferfish: Container-driven elastic memory management for data-intensive applications," in *ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2019, p. 259–271.

[31] W. Chen, A. Pi, and S. Wang, "Os-augmented oversubscription of opportunistic memory with a user-assisted oom killer," in *International Middleware Conference*. Association for Computing Machinery, 2019, p. 28–40.

[32] A. Pi, J. Zhao, S. Wang, and X. Zhou, "Memory at your service: Fast memory allocation for latency-critical services," in *International Middleware Conference*. Association for Computing Machinery, 2021, p. 185–197.

[33] F. Laniel, D. Carver, J. Sopena, F. Wajsburt, J. Lejeune, and M. Shapiro, "Memoplight: Leveraging application feedback to improve container memory consolidation," in *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*, 2020, pp. 1–10.

[34] "Apache hadoop system." [Online]. Available: http://hadoop.apache.org/core/

[35] "Puma benchmarks and dataset downloads." [Online]. Available: https://engineering.purdue.edu/~puma/datasets.htm

[36] "The 20 newsgroups data set." [Online]. Available: http://qwone.com/~jason/20Newsgroups/

[37] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: workload autoscaling at google," in *European Conference on Computer Systems*. Association for Computing Machinery, 2020.

[38] Z. Zhong, M. Xu, M. A. Rodriguez, C. Xu, and R. Buyya, "Machine learning-based orchestration of containers: A taxonomy and future directions," *ACM Computing Surveys*, vol. 54, no. 10s, Sep. 2022.

Short Biography

Rodopi Kosteli was born in Drama in 1998. She obtained a Diploma in Computer Science and Engineering from the Department of Computer Science and Engineering at the University of Ioannina, Greece, in 2022. She is currenlty a M.Sc. student at the same department. Her research interests include Cloud Computing, Virtualization, Resource Monitoring, and Machine Learning.