Sharry: An Efficient and Sharing Far Memory System

Chen Chen
Zhejiang University
Hangzhou, China
chenc68@zju.edu.cn

Yuhang Huang
Zhejiang University
Hangzhou, China
22351093@zju.edu.cn

Shuiguang Deng
Zhejiang University
Hangzhou, China
dengsg@zju.edu.cn

Jianwei Yin
Zhejiang University
Hangzhou, China
zjuyjw@cs.zju.edu.cn

Xinkui Zhao*
Zhejiang University
Hangzhou, China
zhaoxinkui@zju.edu.cn

Abstract

Far Memory System(FMS) allows applications to access memory on remote machines(called memory nodes). However, existing FMSs can't deal with large loads and have low efficiency in utilizing far memory, which leads to the inability to share memory nodes among multiple processes, limiting the scalability of FMS. In this paper, we propose Sharry, an efficient Sharing FMS. Sharry manages memory objects from multiple processes within a unified address space, avoiding the overhead of space switching. Sharry also optimizes the utilization of far memory with fine-grained memory management. Additionally, Sharry offloads memory allocation to dedicated CPU core in order to handle larger loads in the sharing scenario. Compared to state-of-the-art FMS, Sharry improves memory utilisation by 45%, causing only 9% performance degradation when multiple processes sharing single memory node.

ACM Reference Format:
Chen Chen, Yuhang Huang, Shuiguang Deng, Jianwei Yin, and Xinkui Zhao. 2024. Sharry: An Efficient and Sharing Far Memory System. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3656508

1 Introduction

Memory is the most critical hardware resource in today's data centres. However, data centres face two major challenges in providing memory. Firstly, the memory capacity of a single machine is insufficient to meet the growing demands of modern data centre applications, such as Deep Learning and Data Analysis. The data sets are often too large to fit into the RAM, which is known as the ***Memory Capacity Wall***. Secondly, commodity servers combine memory capacity and the number of CPU cores in a fixed ratio, which leads to ***Memory Stranding***[7]: memory fragments on servers cannot be utilized either due to the CPU cores being fully occupied or the memory fragment being smaller than requirements. A report from Google's data centre[14] reveals that the average memory utilization per server is only 60%, with a significant amount of memory sitting idle. The root cause lies in the inability of applications to utilize memory across multiple machines.

To address this dilemma, One promising solution is ***Memory Disaggregation***[1] in data centres. CPU cores in disaggregated data center can be paired with either local or remote RAM modules, and a high-speed network interconnect ensures low access latency for far memory. Currently, hardware vendors are developing new hardware architectures and interconnects to support memory disaggregation. However, rolling out new hardware in data centres can be costly and time-consuming. In contrast, far memory (FM) solution is built on a set of existing commodity server machines, while computing nodes(CN) running computing processes and memory nodes(MN) providing far memory.

Far memory can be accessed by applications through various approach. In ***Page-Based approach*** (such as FastSwap[2]), the OS kernel treats far memory as a swapping device, swapping between local and far memory at the granularity of 4KB memory pages. However, page-based approach suffers form read-write amplification, as well as kernel overheads (such as page-fault and address space switching). Furthermore, developers are unable to control performance, cause page-based FMS is transparent to applications. ***Object-based approach*** (like AIFM[13]) provide developers with remoteable pointer abstraction through user-level runtime, which triggers swapping at object granularity when the remoteable pointer is dereferenced or when high memory pressure is detected. The object-based approach effectively mitigates the read-write amplification and avoids kernel overheads. Moreover, it offers developers more flexibility in optimizing their applications.

Existing FMSs follow the assumption that far memory has sufficient capacity. Therefore, they utilize a coarse-grained far memory management pattern(as shown in Figure 1), where memory on local machine serves as a cache for far memory. When applications construct a remoteable object, FMS runtime generates copies of the data in both local memory and far memory. These copies won't be recycled in far memory until the remoteable object is deconstructed, we call it ***Multi-Copy Pattern***. However, Multi-Copy Pattern introduces significant data redundancy and significant synchronization overhead between the multiple copies, which not only leads to lower utilisation of far memory, but also imposes limitations on the scale of remoteable objects in FMS.

Multi-Copy Pattern also presents a potential issue where during object swapping, far memory managements(like allocation and free) are not frequently executed as the objects' data copies were already allocated in far memory. Therefore, existing FMSs adopt naive approaches for managing far memory. they utilizes locks to ensure metadata consistency between threads during concurrent

*Corresponding author: zhaoxinkui@zju.edu.cn

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3656508

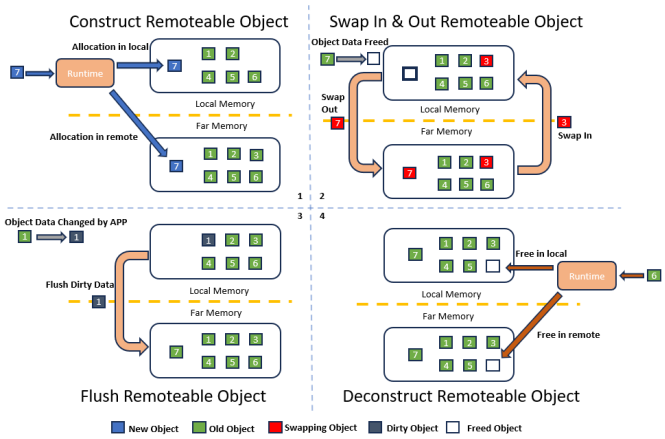


Figure 1: the lifecycle of remoteable objects in Multi-Copy Pattern

memory management. However, the overhead of inter-thread synchronization severely limits the efficiency of managing far memory, particularly in managing large-scale objects.

We propose ***Sharry***, a far memory approach that enables efficient sharing of far memory among multiple processes. Similar to AIFM, Sharry provides users with remoteable pointers at the application level to access far memory. In contrast to multi-process operating systems that allocate a virtual memory address space for each process, Sharry implements a unified far memory address space, which allows objects from different processes to be managed in single address space, avoiding the overhead of address space switching when serving different processes. Unified far memory address space also makes it possible for interconnection between processes through far memory.

In contrast to Multi-Copy Pattern, Sharry leverages ***Unique-Copy Pattern*** to improve the efficiency of utilizing far memory. In Unique-Copy Pattern, each object in the system only has a Unique data copy. When an object is swapped in to local memory, it's data copy in far memory is promptly freed. By adopting Unique-Copy Pattern, Sharry effectively resolves the data redundancy issue inherent in Multi-Copy Pattern and avoids the synchronization between multiple copies, which significantly improves the efficiency of utilizing far memory. However, Unique-Copy Pattern introduces a new challenge. In Unique-Copy Pattern, FMSs have to manage memory for each object swap, with a much higher frequency compared to Multi-Copy Pattern. Existing design of far memory allocator becomes a performance bottleneck of FMS.

To address this problem, we have implemented the ***Sharry Allocator***. Firstly, Sharry Allocator adopts the ***Separated Memory Layout***[4], which allows for faster identification of free memory blocks by decoupling the metadata from the data, ensures performance of Sharry when involving large-scale objects. Secondly, Sharry Allocator offloads memory management to an allocator thread that runs on a dedicated CPU core. Other threads have to request the allocator thread to do allocation or free. Sharry Allocator minimizes inter-thread synchronization overhead by eliminating redundant inter-thread synchronization, except the atomic primitives involved in interactions with the allocator thread.

Single CPU core struggles to handle the pressure of managing the entire unified memory address space, especially when far memory has a significantly large capacity. Sharry Allocator divides the unified address space into multiple Memory Sections of equal length. Each Memory Section is assigned a dedicated CPU core to run its respective section allocator thread. We have also implemented a simple request scheduler to balance the memory pressure between different memory sections. This approach not only alleviates the computational burden on a single CPU core but also maximizes the utilization of computational resources within MN.

We implemented Sharry on regular servers. Compared to AIFM[13] and FastSwap[2], Sharry saves on average 45% of far memory when multiple processes share a single memory node (whose request pressure and object scale multiple times greater than single process.), causing at most 9% performance degradation.

The major contributions of Sharry are as follows:

- We propose a unified far memory address space design for managing objects from multi-process and avoiding the overhead of address space switching;
- We propose Unique-Copy Pattern, which effectively avoid data redundancy and synchronisation overheads between multiple copies, improve the efficiency of far memory usage;
- We propose Sharry Allocator, which effectively handle the significant memory management pressure brought by Unique-Copy Pattern through offloading memory management to multiple CPU cores, ensuring the performance of Sharry when multiple processes share single memory node;
- We used multiple benchmarks to comprehensively evaluate the performance of Sharry.

2 Background

Far Memory Far memory is a promising solution in data centres' architecture. One common approach to building far memory systems is page-based memory swapping. InfiniSwap[5] is the first RDMA-based far memory swapping system, while FastSwap[2] improves its performance through better scheduling and polling mechanisms. Leap[9] takes a different approach by prefetching memory pages based on the majority access pattern of processes, thus avoiding far memory accesses on critical paths. The page-based scheme is application-agnostic and transparent to the application. In addition to the page-based approach, another far memory approach swaps at the granularity of objects and exposes far memory to developers as remoteable pointers. AIFM[13] introduces a new programming model for far memory, including features such as remoteable pointers and dereference scopes. Carbink[16] introduces an efficient fault-tolerance mechanism in the far memory system and swaps at a spanning granularity instead of the object granularity strategy used in AIFM. Carbink also performs reclamation of far memory. Mira[6] is the most recent work that utilizes program analysis to improve prefetching on top of AIFM.

Existing work often overlooks the scalability of far memory systems and fails to optimize the efficiency of far memory management. our work focuses on addressing these issues and aims to improve the scalability and efficiency of far memory management.

Memory Management Research on local memory management has been well-established. Today's application relies on user-level memory allocators, such as PTmalloc2[3](From glibc), TCMalloc[4] (From Google), and Mimalloc[10](From Microsoft), for efficient memory management. Domain-specific implementations, such as NVAlloc[15], are also available for non-volatile memory. These memory allocators typically support concurrent memory management for multi-threaded programs, efficiently handling simultaneous memory requests from different CPU cores. However, as the number of CPU cores continues to increase, software memory

allocators face challenges in managing ***Thread Contention***. The synchronization of inter-CPU core metadata through atomic operations or locks, which maintains the global free list, introduces significant performance overheads. To address this problem, TCMalloc utilizes per-CPU/thread caches to manage metadata associated with each logical core, reducing the need for locks during memory allocations and deallocations. Mimalloc employs a three-page locally-shared list of frees to enhance locality, minimize contention, and support fine-tuning on fast-path allocations and frees.

Inspired by NextGenMalloc[8], Sharry addresses the Thread Contention problem in large-scale request scenarios by offloading memory management to a dedicated CPU core. This approach allows for serial execution of memory allocation requests, eliminating the synchronization overhead associated with atomic primitives.

3 Motivation

Traditional memory management approach, such as the buddy allocator and slab allocator in the kernel, as well as user-level allocators, need to be reevaluated in far memory. In these scenarios, allocation requests come from different computing processes on different servers, requiring scalability to handle large memory capacities and process requests while managing fragmentation. There are two potential solutions, first one is ***Centralized Solution***, which centralizing all allocation requests on a huge memory node, second one is ***Distributed Solution***, which distributing memory allocation across multiple nodes, requiring efficient coordination.

Our KEY INSIGHT is that ***Centralized Solution using a high-capacity commodity machine as partial memory pool (e.g., at the rack level) is more suitable for today's data centres instead of Distributed Solution that clusters plenty small machines to construct a global memory pool***. This is because the Distributed Solution faces three major challenges:

- provisioning memory resources in a distributed cluster is a complex process, even for most advanced cluster scheduler;
- running far memory runtime on every memory node is resource-intensive, while a centralized scheme only requires running one far memory runtime;
- having multiple memory nodes corresponding to a computing process increases the complexity for the computing side runtime as it needs to determine which memory node to swap out;

hence Centralized Solution is simpler and more efficient, easier to rolling-out in data centres (requiring few machines) .

Sharry aims to effectively share far memory(provided by single memory node) among multiple computing processes and optimizes the performance of FMS in centralized and large-scale scenarios. We will introduce the design of Sharry in next section.

4 System Design

Figure 2 illustrates the overall architecture of Sharry, which comprises multiple computing nodes and a single memory node. Each computing node executes computing processes that may utilize far memory upon computing side runtime of Sharry. Memory node executes the memory side runtime of Sharry, exposing far memory through network communication. Notably, Sharry does not require any custom hardware or modifications to OS kernel, is able to run on any commodity servers. Computing processes swap out objects when runtime detects memory pressure, and swap objects back into the local memory when accessed.

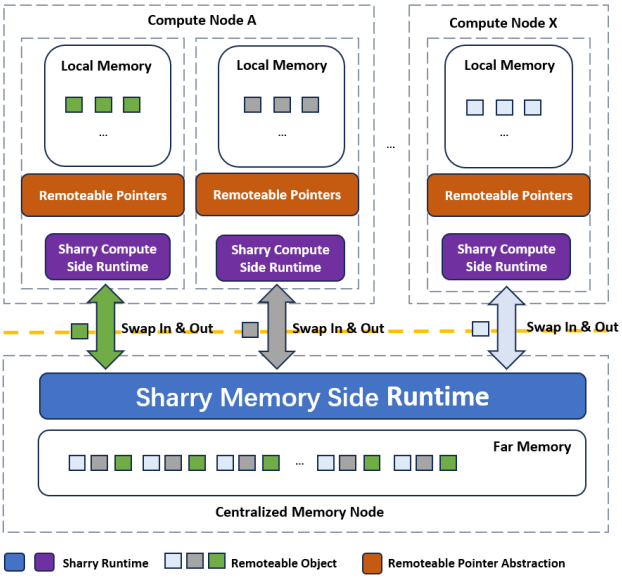


Figure 2: Sharry's Overall architecture.

Inspired by previous work AIFM[13], Sharry utilizes remoteable pointers abstraction to expose far memory at the object granularity, which also enables constructing remoteable data structures to execute more complex workloads.

When serving multiple computing processes, Sharry maintains a ***Unified Memory Address Space***(UMA) instead of creating separate address spaces and manage all objects in UMA. Unlike the previous works which follow the Multi-Copy Pattern, Sharry employs the ***Unique-Copy Pattern*** to increase the efficiency of utilizing far memory. Within UMA, Sharry manages far memory throuth ***Sharry Allocator***, which offloads memory management to a single dedicated CPU core and running as allocator thread. Sharry Allocator avoids multi-threads contention caused by concurrent memory management.

To further enhance the performance of Sharry Allocator running on a dedicated CPU core, Sharry divides UMA into multiple memory sections. Each memory section corresponds to an allocator thread running on a dedicated core, improving the efficiency of memory management through the divide-and-conquer strategy, without causing thread contention.

4.1 Unified Memory Address Space

Sharry's memory side runtime is tasked with managing all the memory within the memory node and making it accessible to computing processes through the network. When multiple computing processes share a single memory node, Sharry introduces an efficient design called ***Unified Memory Address Space***(as shown in figure 3), which aims to consolidate the memory management of remoteable objects from all computing processes.

Address Space Design Different from the design of the process address space in a multi-process operating system that each process has its own virtual memory address space and page table, ensuring isolation between different processes. In our design, Sharry has only one memory address space(UMA) and Sharry allocates addresses within UMA for objects from all processes, instead of creating a separate "remote address space" for each process, which would introduce the overhead of switching address spaces when serving different computing processes.

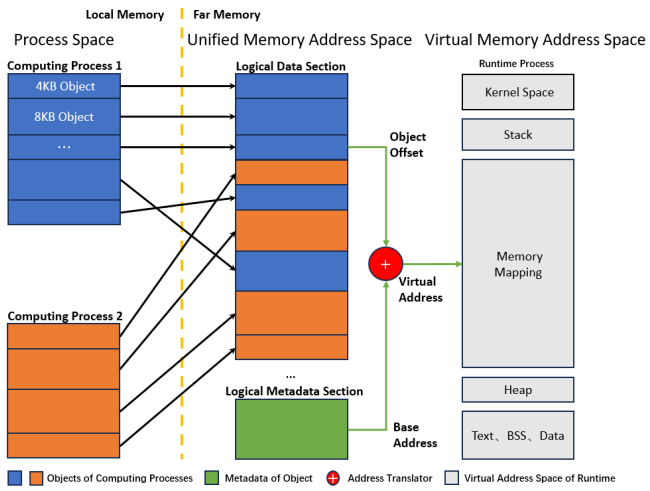


Figure 3: Unified Memory Address Space

As depicted in Figure 3, the unified memory address space is a logical address space provided by the Sharry runtime. It has a size equal to the physical memory size of the memory node(subtracting the size of the kernel space), in contrast to traditional memory address space, UMA only consists of metadata section and object data section, without separate divisions for heap, stack and others. The object data section stores data, and metadata section stores all objects' metadata. We will introduce this separate layout for metadata and data in Section 4.3.

Flat Address Translation Sharry's memory side runtime is also responsible for translating addresses from the ***Unified Memory Address Space***(UMA) to the ***Virtual Memory Address Space***(VMA) of the runtime process. Then, MMU (Memory Management Unit) will translate the Virtual Memory Address to the physical address of the memory node. Since the size of UMA is similar to the size of VMA, Sharry chooses to build a direct mapping from UMA to VMA at the byte granularity(as shown in figure 3), instead of emulating traditional memory systems that use multi-level page table mappings at the page granularity. This flattened mapping approach limits address translation to a maximum of one address lookup.

4.2 Unique Copy

Sharry leverages the Unique-Copy Pattern to utilize far memory efficiently. In this Pattern, an Object has only one globally unique copy present in both local and far memory. When an Object is swapped out to far memory, the memory side runtime promptly allocates a memory block of the appropriate size on the memory node, leading to reclamation of the Object in local memory. On the other hand, when an Object is swapped into local memory, the memory side runtime reclaims the memory block in far memory. This mechanism enhances the efficiency of far memory utilization.

To avoid inefficiencies in object swapping caused by frequent memory block reclamations, Sharry employs an asynchronous reclamation approach. Memory block reclamation is performed only when the total number of memory blocks need to be freed reaches a threshold. By adopting the Unique-Copy Pattern, Sharry eliminates the synchronization overhead between multiple copies of data, which caused by Multi-Copy Pattern.

4.3 Sharry Allocator

Existing far memory systems, such as AIFM[13], typically adopt the Multi-Copy Pattern for far memory management. In this Pattern,

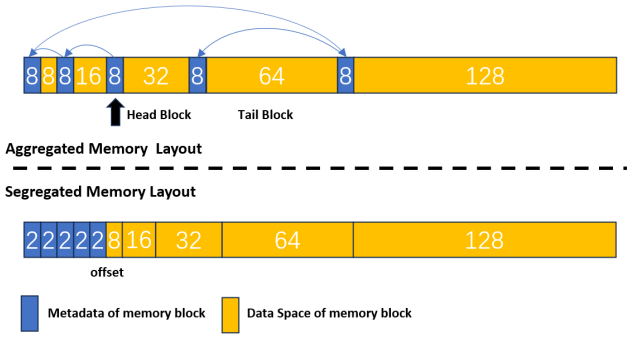


Figure 4: Segregated Layout and Aggregated Layout

memory allocation is performed once when a remoteable object is constructed, and deallocation is performed once when the object is reclaimed. However, Sharry introduces the Unique-Copy Pattern, which involves intense memory management throughout the lifecycle of a remoteable object. This distinction highlights that the primitive memory allocators utilized in existing FMS have become a performance bottleneck.

Exsiting memory allocators, such as PTmalloc2[3], TCMalloc[4], and Mimalloc[10], are designed for single process scenarios with limited memory block sizes and memory request intensities. Two key issues affecting memory allocator's performance are ***memory block scales*** and ***thread contention***. The performance degrades as the scales of the memory block increases. Thread contention arises from the concurrent memory management support in existing memory allocators, where multiple threads running on different CPU cores concurrently process requests. Synchronization between threads necessitates mutex locks to maintain a global free list, resulting in significant performance degradation. As the number of CPU cores continues to grow, memory managers face even greater challenges in handling thread contention. We introduces Sharry Allocator,which is designed to optimize the performance bottleneck of memory allocators, which is amplified in far memory scenarios.

Decoupled Memory Layout The Unified Memory Address Space design in Sharry provides a straightforward and comprehensible approach for managing the far memory address space. However, as the scale of memory management requests from computing processes increases, Sharry's efficiency in handling large memory blocks can potentially become a runtime performance bottleneck due to the growing number of memory blocks involved. To address this issue, Sharry adopts a metadata and data decoupled Segregated Memory Layout. This design enhancement aims to improve Sharry Allocator's performance when facing large-sacle Objects.

As illustrated in Figure 4, previous studies on memory space have proposed two prominent metadata layouts: the ***Aggregated Layout*** (utilized by Mimalloc[10]) and the ***Segregated Layout*** (used by TCMalloc[4]). In Aggregated Layout, the first 8 bytes of each free block (assuming a 64-bit size) serve as a pointer to the next free block. In contrast to Aggregated Layout, the Segregated Layout decouples the metadata from the memory block's data, using a smaller index (e.g., 16 bits) to indicate the location of the next free block. This design choice enhances memory utilization efficiency and improves memory management performance by solely traversing the metadata portion of the memory space, without the need to navigate through the entire memory space. While there is always a

trade-off between the Aggregated Layout and the Segregated Layout, the Segregated Layout is better suited for managing large-scale memory blocks in far memory scenarios.

Offload to Dedicated Core In the current design of memory allocators, contention can arise when two threads simultaneously manage memory metadata. To mitigate this contention and control the order of changes made to critical resources, read-write locks are commonly used. Sharry Allocator offloads memory management to a dedicated CPU core, uses a single-threaded mode to avoid synchronizing between threads.

In this design, Only allocator thread running on dedicated CPU core have the ability to operate on the metadata of memory blocks. This approach effectively eliminates most thread contention, minimize the overheads caused by locks. Ordinary threads perform memory management by modifying the synchronization variables using atomic primitives and requesting the allocator thread to handle the request on their behalf.

The performance advantages of Sharry Allocator stem from two key aspects. Firstly, since all memory management operations runs on a dedicated core, it ensures sequential execution of all request. As a result, there is no need for any locks in Sharry Allocator to synchronize metadata between threads, thus reducing the overhead. Secondly, the dedicated core exclusively runs allocator threads and does not perform thread switching, thereby preventing CPU cache line contamination.

4.4 Multi-Section Memory Management

Sharry employs a design where all memory management operations are offloaded to a single thread running on a dedicated core. This approach reduces thread contention during memory management and ensures sequential execution of all memory requests on that specific core. However, in large-scale scenarios, a single core may not possess sufficient computational capability to handle all requests, resulting in performance degradation. To overcome this limitation, Sharry introduces the Multi-Section design, which dividing the unified memory address space into multiple memory sections. For each memory section, Sharry allocates a dedicated core to run a allocator thread. Each thread is responsible for managing memory within the corresponding memory section(as shown in figure 5). This design allows for better utilization of computing resources on the memory node and alleviates computing pressure on allocator threads running on each dedicated core.

In the case of memory management requests from multiple processes, Sharry employs a request scheduler to dynamically select the most suitable memory section. The request scheduler performs request scheduling, and the corresponding allocator thread of the selected memory section handles the incoming requests. Request scheduler is able to monitor the usage pressure of each memory section and the request pressure during a fixed time window in real-time. This enables the scheduler to balance the memory usage and computing pressure across different memory sections.

5 Implementation

We implemented the Sharry runtime library in C++, comprising approximately 7.7K lines of code (LOC), which runs on both local nodes and memory nodes. Sharry is designed to run on unmodified generic operating systems. For the memory nodes, we implemented Sharry Allocator which offloads memory management to a dedicated core and handles the synchronization of request threads and allocator threads by atomic primitives. Sharry Allocator are used for memory management within unified memory address space. The communication between computing nodes and memory nodes occurs over TCP network stack. Currently, Sharry supports multiple computing nodes and a single memory node.

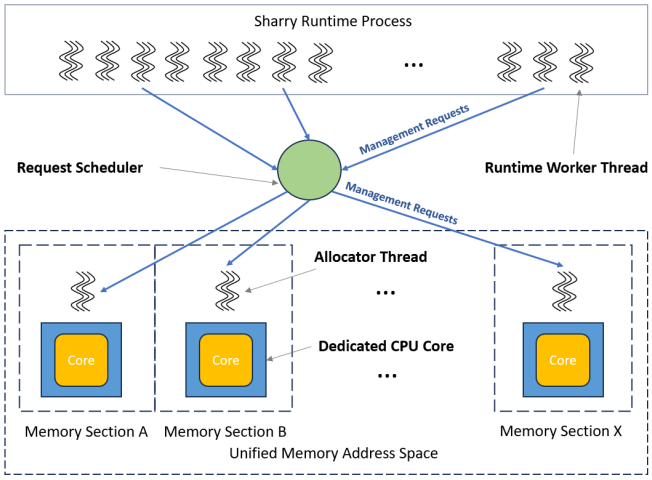


Figure 5: Multi-Section Memory Management

6 Evaluation

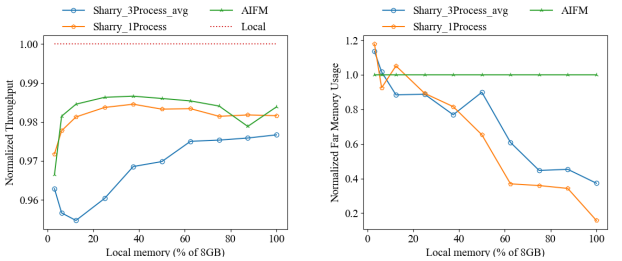
The experiments are designed to answer following questions:

- What is the throughput and far memory usage of Sharry when multiple processes sharing far memory(section 6.1)?
- How does sharry perform when there are massive objects in the system(section 6.2)?
- How does Sharry's performance compare to the to standard Linux and AIFM(section 6.3)?

Testbed setup Our experimental environment consists of four commodity servers, with three nodes serving as computing nodes and one node serving as a memory node. All nodes are equipped with 10-core Intel Xeon E5-2640 v4 CPUs (2.40 GHz), 64GB of memory, and connected by 25 GbE network. We are using Ubuntu 20.04 (kernel v5.14). We followed common practices to tune these servers for low latency[12], including disabling CPU frequency scaling, machine-check exceptions, and transparent hugepages.

6.1 Sharing vs. Not Sharing

We use a micro-benchmark to test the performance of Sharry when multiple computing processes share far memory. The benchmark is based on an array data structure, where we create 8 remoteable arrays. For each we set the input size to 1GB, including 260,000 objects of 4KB size. We run this benchmark on a cluster consists of three computing nodes and one memory node.



(a) Throughput in sharing **(b) memory usage in sharing**

Figure 6: 3 processes sharing single memory node

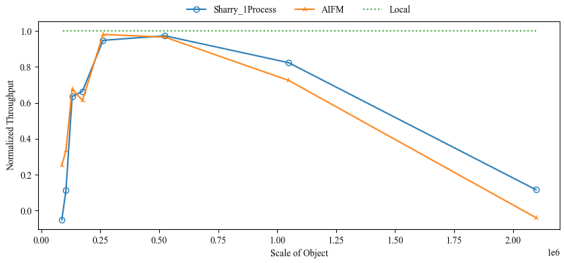


Figure 7: Performance in different Scale.

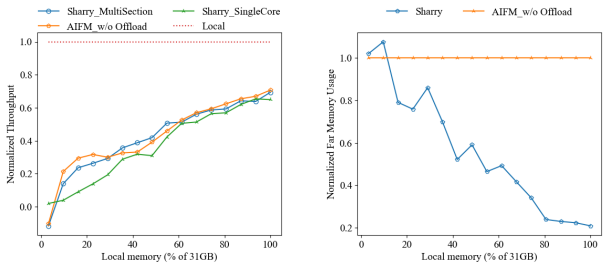
Figure 6 shows the performance of Sharry when sharing far memory and the efficiency of utilising far memory. As shown in Figure 6b, as the local memory size increases, the Unique-Copy Pattern helps Sharry save more far memory compared to AIFM, up to 50%. The delayed reclamation mechanism of Sharry Allocator causes Sharry to use slightly more far memory than the total workload size minus the local memory size. Figure 6a shows that Sharry's performance causes up to 9% performance degradation when sharing far memory, and the performance degradation gradually decreases as the local memory size increases.

6.2 Impact of Scale

We modify the benchmark above slightly by fixing the local memory size to 2GB and changing the object scale in the system. Figure 7 shows the performance of Sharry as the scale of the objects stored in FMS continues to grow. When the scale is large, Sharry has a visible performance improvement compared to AIFM(nearly 10%), which we believe comes from the Segregated Layout.

6.3 Real Application Benchmark

We also tested sharry's performance on a real application benchmark, DataFrame[11], a data analysis system written in 24.3K LOC C++, which is widely used as the current infrastructure in the field of data analysis. DataFrame can be both computationally intensive and memory intensive when operating on large datasets, making it well suited to use far memory. We build our benchmark on 2016's NYC Taxi Trip dataset, using DataFrame to analyses and predicts taxi travel times at different periods of day.



(a) Throughput in Dataframe (b) Memory usage in Dataframe

Figure 8: DataFrame Performance.

Figure 8b shows the usage of shared far memory. Figure 8a shows the throughput of Sharry and AIFM(without function offloading) on Dataframe. Unique-Copy Pattern does impact on Sharry's performance. When the local memory is small, due to intense swapping requests, Sharry's throughput is slightly lower than AIFM (about 9% performance degradation). The design of ***Multiple Sections*** brings significant performance improvement compared to ***Single Section***, as it distributes the computing load. As the size of local memory increases, Sharry's performance getting close to AIFM. Even local memory reaches 100%, both Sharry and AIFM still lag behind Local cause the high dereferencing cost of remoteable pointers.

Discussion: How to offload function in FMS? Existing FMS are discussing how to improve performance by offloading function to memory node, cause instead of swapping large amounts of data into local memory, processing data directly in far memory avoids network overhead and makes efficient use of the CPU cores on the memory node(AIFM achieves near-local memory performance when performing offloading functions of Dataframe).

For Unique-Copy Pattern, the key problem of function offloading is that objects need to be calculated are not all in the local or remote. To address this, we have two approach, Firstly, ***Gathering all objects in one place***. Secondly,***Calculating each part of data, and aggregating them***, we prefer the latter one, which avoids redundant network communication, but brings complexity. Our next work will discuss how to offload functions in FMS more efficiently.

7 Conclusion

In this paper, we propose Sharry, an object-based FMS. Sharry enables a single memory node to serve multiple computing processes and achieves efficient management of shared far memory through various techniques such as the ***Unique-Copy Pattern***, ***Separated Memory Layout***, and ***Memory Management Offloading***. Compare to AIFM, Sharry achieves a 45% improvement in far memory utilization while incurring 9% performance loss at most.

8 Acknowledgment

We deeply appreciate the comments from the reviewers. This work was supported in part by the National Key R&D Program of China (2022YFF0902702), and the Major Program of National Natural Science Foundation of Zhejiang(LD24F020014), and the Zhejiang Pioneer(Jianbing) Project (2024C01032), and part of the Key R&D Program of Ningbo(2023Z235), and part of the Ningbo Yongjiang Talent Programme (2023A-198-G), and part of the Beijing Life Science Academy (BLSA:2023000CB0020).

References

[1] Marcos K. Aguilera and Nadav Amit. 2018. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (ATC '18)*.
[2] Emmanuel Amaro and Branner-Augmon. 2020. Can Far Memory Improve Job Throughput?. In *15th European Conference on Computer Systems (EuroSys '20)*.
[3] Wolfram Gloger. 2022. PTmalloc2. http://www.malloc.de/en/.
[4] Google. 2023. TCMalloc. https://github.com/google/tcmalloc/
[5] Juncheng Gu and Youngmoon Lee. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*.
[6] Zhiyuan Guo, Zijian He, and Yiying Zhang. 2023. Mira: A Program-Behavior-Guided Far Memory System *(SOSP '23)*.
[7] Huaicheng Li and Berger. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2023)*.
[8] Ruihao Li and Qinzhe Wu. 2023. NextGen-Malloc: Giving Memory Allocator Its Own Room in the House *(HOTOS '23)*.
[9] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (ATC '20)*.
[10] Microsoft. 2023. Mimalloc-bench. https://github.com/daanx/mimalloc-bench/
[11] Hossein Moein. 2023. C++ DataFrame for statistical, Financial, and ML analysis. https://github.com/hosseinmoein/DataFrame
[12] Amy Ousterhout and Joshua Fried. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.
[13] Zhenyuan Ruan and Malte Schwarzkopf. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.
[14] Muhammad Tirmazi and Barker. 2020. Borg: The next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.
[15] Devesh Tiwari and Lee. 2010. MMT: Exploiting fine-grained parallelism in dynamic memory management. In *2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS '10)*.
[16] Yang Zhou and Hassan. 2022. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*.