KLNK: Expanding Page Boundaries in a Distributed Shared Memory System

Yi-Wei Ci, Michael R. Lyu, *Fellow, IEEE*, Zhan Zhang, De-Cheng Zuo, and Xiao-Zong Yang

***Abstract*—Software-based distributed shared memory (DSM) allows multiple processes to access shared data without the need for specialized hardware. However, this flexibility comes at a significant cost due to the need for data synchronization. One approach to mitigate these costs is to relax the consistency model, which can lead to delayed updates to the shared data. This approach typically requires the use of explicit synchronization primitives to regulate access to the shared memory and determine the timing of data synchronization. To circumvent the need for explicit synchronization, an alternative approach is to manage shared memory transparently using the underlying system. While this can simplify programming, it often imposes a fixed granularity for data sharing, which can limit the expansion of the coherence domain and increase the synchronization requirements. To overcome this limitation, we propose an abstraction called the elastic coherence domain, which dynamically adjusts the scope of data synchronization and is supported by the underlying system for transparent management of shared memory. The experimental results show that this approach can improve the efficiency of memory sharing in distributed environments.**

***Index Terms*—Operating system, inter-process communication, distributed shared memory.**

I. Introduction

COMPUTATION environments deployed across distinct physical machines can be interconnected, facilitating data exchange among processes in such isolated environments often presents a significant challenge. Message passing [1], [2], [3], [4] is a widely adopted approach. This method employs a set of communication routines that utilize efficient transport mechanisms to improve system throughput. Despite its advantages, this strategy introduces additional complexity in coordinating shared data across distributed nodes.

In virtual environments, these challenges persist. Despite the numerous advantages of virtualization technology in managing computing environments, effective data exchange among isolated processes remains a problem. To address this, approaches like GiantVM [5] have been proposed. GiantVM introduces a single system image (SSI) abstraction at the hypervisor level by integrating distributed shared memory (DSM) into Kernel-based Virtual Machine (KVM). The DSM is based on the IVY protocol [6].

Unlike a hypervisor-level implementation, the DSM can be natively supported by the operating system. This enables traditional inter-process communication mechanisms to be extended to operate within a distributed environment. With this implementation at the operating system level, memory sharing becomes possible across both physical machines and virtual environments. SSI systems [7], [8] often facilitate distributed inter-process communication at the operating system level, allowing for the transparent management of shared memory at the page level. The shared memory paradigm is vital in facilitating data exchange by providing memory abstraction and enabling access to distributed data through a uniform interface. This uniformity greatly simplifies the task of accessing distributed resources, even when these resources span across both physical and virtual machines.

Contrasting with transparent approaches, explicit primitives can be used to regulate access and maintain the consistency of shared data [9], [10], [11]. These primitives can be utilized to delay synchronization until it is possible to modify the updater, thereby reducing the need for frequent synchronization following each data update. However, managing these explicit synchronization primitives can demand substantial effort. An alternative approach to guide synchronization involves redirecting computation through delegate operations [12]. This approach allows for remote operations, but recognizing these specialized operations necessitates a user-level runtime system.

To circumvent the need for helper primitives, the synchronization of shared data can also be determined implicitly [6], [13], thereby making the management of shared data transparent. Typically, the operating system handles page-level data management [6], [13], [14], capturing data updates at the page granularity. As the paging mechanism provides data access protection, the operating system can identify the readers or writers of shared data. Sharing data through small-sized pages, such as 4 KB, can reduce the cost of retrieving remote data during each synchronization. However, if the required data size significantly surpasses the page size, this approach can result in frequent synchronizations. To address this issue, operating system support for transparent huge pages (THPs) provides the

Manuscript received 26 August 2023; revised 25 May 2024; accepted 28 May 2024. Date of publication 5 June 2024; date of current version 19 July 2024. This work was supported by the National Natural Science Foundation of China under Grant 62372438. Recommended for acceptance by K. Gopalan. *(Corresponding author: Yi-Wei Ci.)*

Yi-Wei Ci is with the Institute of Software, Chinese Academy of Sciences, Beijing 100045, China (e-mail: yiwei@iscas.ac.cn).

Michael R. Lyu is with the Department of Computer Sciences and Engineering, Chinese University of Hong Kong, Shatin, Hong Kong (e-mail: lyu@cse.cuhk.edu.hk).

Zhan Zhang, De-Cheng Zuo, and Xiao-Zong Yang are with the School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China (e-mail: zz@ftcl.hit.edu.cn; zuodc@hit.edu.cn; xzyang@hit.edu.cn).

This article has supplementary downloadable material available at https://doi.org/10.1109/TPDS.2024.3409882, provided by the authors.

Digital Object Identifier 10.1109/TPDS.2024.3409882

capability to utilize pages of a larger size. By reducing the number of memory pages required, frequent synchronizations of shared data can be minimized. However, updating a large-sized page, such as a 2 MB or 1 GB page, can incur substantial costs when retrieving updated data from remote processes over the network. Furthermore, the employment of large-sized pages can lead to false sharing because of data synchronization, as a thrashing effect occurs when different processes repeatedly update different parts of the same page.

The Virtual Memory Area (VMA) represents a segment of memory within the virtual address space. Given that VMAs can be considerably larger than a page, maintaining coherence when VMAs map to the same shared memory presents challenges. In a distributed environment, the memory pages within a VMA can be synchronized independently. However, traditional mechanisms lack the flexibility to adjust the granularity of data sharing effectively. In this paper, we introduce a data management approach based on virtual regions. These regions facilitate the dynamic merging of contiguous pages, differing from the VMA abstraction by specifically aiming to maintain coherence among selected pages in coordination. The set of merged pages can be adjusted, forming a flexible coherence domain that extends the granularity of data management beyond the page boundary. This method allows data synchronization to be transparently handled by the underlying operating system.

Recent enhancements in network performance [15] have significantly improved the feasibility of managing DSM through a unified network infrastructure. Improvements in optical networking, RDMA (Remote Direct Memory Access), and CXL (Compute Express Link) technologies are crucial for the efficient operation of interconnected DSM systems. Despite these advances, challenges persist in the management of DSM, including issues such as traffic storms arising from write-invalidation mechanisms. Studies [16], [17] have indicated that the use of shared memory can facilitate state sharing in distributed machine learning applications. Unlike these studies, this paper focuses on the construction of a general-purpose DSM system. We present the following contributions for building a DSM system that provides transparent management of shared memory:

- We introduce the concept of resizable coherence domains, a strategy that enables the seamless expansion of page boundaries in a distributed environment. This approach facilitates the maintenance of coherence over an extended scope, thereby reducing the need for frequent coherence management.
- By advocating for content-based sharing within each coherence domain, this approach enables multiple nodes to concurrently contribute different parts of the necessary updates. As a result, the unit of data synchronization can differ from the unit of coherence management.
- We have developed a DSM system at the operating system level. This allows existing applications that use System V IPC shared memory and run on shared-memory multiprocessors to operate in a distributed environment with minimal or no modifications.

The remainder of this paper is organized as follows: In Section II, we describe related work. In Section III, we introduce the elastic coherence domain. Section IV elaborates on the content-based memory sharing. Section V describes the implementation of our approach. In Section VI, we present experiments, and in the last section, we provide conclusions.

II. Related Work

Sharing memory in a distributed environment can be achieved through either hardware or software. Hardware-based DSM [18], [19] provides a lower-level abstraction of distributed memory than software systems, leading to improved memory operation latency. However, hardware-based DSM can be costly and challenging to verify. In contrast, software-based DSM simplifies system design. Software-based approaches enable distributed memory sharing through non-transparent, partially transparent, and transparent schemes.

Nontransparent Schemes: In nontransparent schemes, shared data is declared using explicit primitives, which means that shared data can be dynamically recognized based on explicit declarations. For instance, Midway [20] implements entry consistency, where each shared datum is explicitly associated with synchronization objects such as locks and barriers, which can be used to label synchronization locations. During synchronization, any modifications made to shared data can be delivered to the process that acquires the synchronization object. To address the problem of single ownership, the synchronization object can be acquired in a non-exclusive mode. Non-exclusive mode access to a synchronization object enables concurrent reading, subject to the restriction that if a process holds a synchronization object in exclusive mode, no other process is allowed to hold the synchronization object in non-exclusive mode. However, it can be challenging for programmers to convert programs to support nontransparent schemes.

Partially Transparent Sharing Schemes: In partially transparent sharing schemes, shared data does not need to be explicitly labeled but can be synchronized implicitly. However, data modification must be protected through explicit primitives. Munin [21], [22] supports multiple consistency protocols through release consistency [11]. To efficiently achieve consistency, Munin uses sharing annotations that describe the access pattern of shared data to select a suitable consistency protocol. Within the release consistency model, shared data does not need to be bound to a synchronization object, and potential synchronization locations can be inferred through explicit primitives. Munin can detect modifications to shared data through virtual memory hardware, and the propagation of these modifications can be delayed until a release primitive is encountered. TreadMarks [23], [24], [25] implements a lazy release consistency model that improves performance by delaying the synchronization of shared data from the release time of a lock to the next acquisition time of the lock. Each page can only be modified by a process after a copy of the page has been created, thus enabling individual processes to be aware of their own modifications, and different processes to simultaneously write a page. When writers of a page start to synchronize with each other, the modifications of each writer can be detected by a word-by-word comparison with the unmodified copy. Scope consistency (ScC) [26] has been

proposed to improve the performance of the release consistency model. The ScC model guarantees that the modifications of a scope are visible to the process using the scope. The partially transparent sharing schemes still require explicit primitives that rely on the compiler.

Transparent Sharing Schemes: In transparent sharing schemes, data sharing is not intervened by external primitives, making programming easy. IVY [6] introduced user-transparent DSM systems to provide sequential consistency for a loosely coupled multiprocessor architecture. IVY is based on a write-invalidation protocol where the latest copy of each page is maintained by an owner, and ownership of pages is recorded through managers. A request is sent to a manager, who can locate the owner of the required page and request the owner to provide the page. In the fixed distributed manager algorithm, every processor is allocated a predetermined set of pages to manage, and managers can be directly located. In contrast, the dynamic distributed manager algorithm allows ownership information of each page to be maintained by uncertain nodes, which requires additional communication steps to find the node maintaining ownership information of a page. Mirage [13] is another user-transparent DSM system that implements sequential consistency using a write-invalidation protocol at the operating system level. A paged segmentation scheme is used for data management, and the locations of page data are tracked through library sites. Each request for page retrieval is dispatched to a library site that plays the role of the manager of the associated segment, and the page requests can be sequentially processed. The page request is then forwarded to the current writer for providing the requested page, allowing different pages to be synchronized separately. In conventional transparent sharing schemes, data sharing is often performed at a fixed granularity, such as the page-level. With an increase in cross-page memory access, more data synchronization times can be required. Although the data management of shared memory is handled by the underlying system, allowing the process of locating the latest version of the shared data to be transparent to applications, these transparent schemes still rely on advanced algorithms to manage shared data efficiently.

In this paper, we propose a strategy for the transparent management of shared memory that extends beyond the page-level scale. This approach enables shared memory to be managed without depending on specific cache coherence protocols [27], [28]. We demonstrate how a user-transparent DSM system can facilitate remote memory accesses [29], [30], [31], [32] via the underlying operating system. This transparent management approach can reduce the programming complexity of applications in a distributed environment. Essentially, existing shared memory applications can be adopted in a distributed environment with minimal or potentially no modifications. To facilitate the transparent management of coherence that extends beyond page boundaries, we introduce the concept of an elastic coherence domain. This concept allows for the scope of coherence maintenance to be dynamically adjusted, enabling proactive synchronization of shared data that is expected to be accessed in the near future. To minimize synchronization costs, we present a content-based memory sharing strategy. This strategy allows the necessary data to be concurrently collected from potential providers.

III. Page Boundary Expansion

The shared segment is assumed to be attached to the address space of each process that accesses the shared memory. These processes can be distributed across different nodes, which are connected through interconnection facilities. Maintaining coherence of the shared memory is required across nodes. To provide flexibility in managing shared memory, the granularity of data for coherence maintenance is not constrained to the page scale.

A. Elastic Coherence Domain

To achieve transparent data management, it is crucial for the underlying operating system to track the access to shared memory at the page granularity. Accessing each protected page triggers a page fault, ensuring the retrieval of the latest data for the page. The process of maintaining coherence of page data distributed among nodes is analogous to the cache coherence provided by processors. The single-writer/multiple-reader (SWMR) property is essential for tracking the latest version of updated data, and the write-invalidate protocol can be employed to accomplish this. Ensuring coherence involves exclusively updating each page, which necessitates additional invalidation alongside the synchronization of page data.

To improve page fault delays and reduce data retrieval and invalidation, huge page techniques can be leveraged. However, the inter-node communication bandwidth can be overwhelmed for synchronizing page data in a distributed environment. In addition, the efficiency of using huge pages can be harmed by the thrashing effect. Consequently, attention is turned towards the conventional page (4 K page), which can be seen as an atomic coherence domain (as shown in Fig. 1(a)). To allow data sharing granularity to vary, it is desirable to adjust the coherence domain provided by the underlying operating system dynamically. A coherence domain can be dynamically formed if certain pages nearby can be accessed subsequently.

To facilitate the concatenation of shared pages, the operating system can seamlessly group together continuous pages. This group of concatenated pages is also identified as a coherence domain. This coherence domain can be dynamically expanded as required, allowing for uninterrupted simultaneous access to a larger number of pages. Considering that the coherence domains have K different scales, each coherence domain can form a virtual region, which, if necessary, can be further divided into several sub-regions. Let $\mathcal{D}_i$ denote a coherence domain. $\mathcal{D}_i^k$ represents a coherence domain at the k-th scale ($k < K$). These coherence domains, which have the ability to resize and are thus referred to as elastic coherence domains, can be constructed as follows:

- $\forall u, v : \mathcal{D}_u^{k-1}, \mathcal{D}_v^{k-1} \in \mathcal{D}_i^k$, if the coherence domain $\mathcal{D}_u^{k-1}$ is accessed by a visitor p after the coherence domain $\mathcal{D}_v^{k-1}$ is accessed by p within a period of time Δ, then the coherence domains within $\mathcal{D}_i^k$ can be merged.
- If the sub-regions of a coherence domain $\mathcal{D}_i^k$ cannot be merged, then $\mathcal{D}_i^k$ is split into a set of coherence domains.

In this system, Δ represents the predetermined observation window used to monitor the interactions of each process with

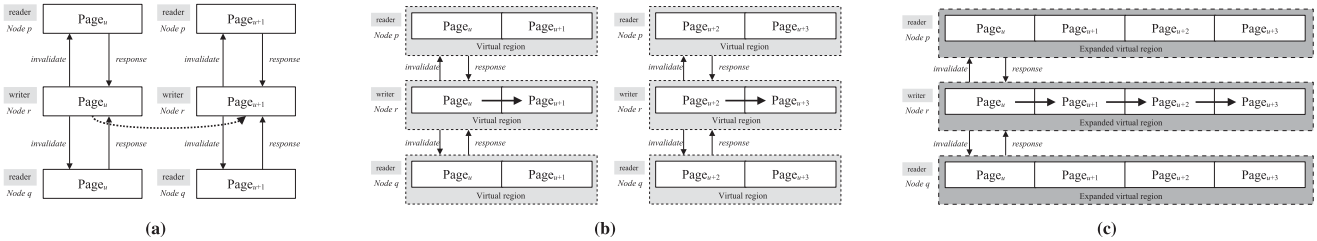


Fig. 1. (a) Page-level coherence management. (b) Virtual region. (c) Virtual region expansion.

shared memory. Increasing Δ can accommodate more extensive memory accesses, which allows these memory areas to be merged to facilitate uninterrupted access to larger memory areas. Δ is a threshold that can be set in advance to meet the particular requirements of the system.

To provide different scopes of data sharing in a distributed environment, we can adjust an elastic coherence domain denoted by $\mathcal{D}_i$. To simplify the explanation, assume that there are two sizes of coherence domains, namely $K = 2$. The smaller coherence domain is called a child region (as shown in Fig. 1(b)), while the larger coherence domain is referred to as a parent region (as shown in Fig. 1(c)). By merging child regions, coherence can be maintained in a larger memory area. On the other hand, if a parent region is split, the coherence of its child regions can be maintained separately.

B. Dynamic Adjustment

Prefetching is typically employed to proactively retrieve data in anticipation of future access, effectively minimizing data access latency. In order to optimize the associated costs of prefetching, data is generally fetched at a fine-grained granularity. The coherence of each fine-grained block is managed independently during the prefetching process. However, there exists an opportunity to group prefetching requests together, in a manner similar to the write combining technique [33] used in modern processors. This grouping decreases the need for frequent synchronization.

Contrary to batching prefetching requests, maintaining data coherence in an expanded memory area allows for retrieving required data with fewer requests. However, managing such an enlarged area may involve gathering more data during each synchronization if all the data in the area is retrieved, which is unnecessary when only part of the area is updated.

To reduce sharing costs, it is possible to partition a coherence domain into smaller domains if the visitor does not continually access different parts of the data within the coherence domain. Moreover, while coherence is typically carried out at a coarse-grained granularity, data synchronization can also be performed at a finer-grained level, as explained in the following section.

In the context of memory pages within a coherence domain, differentiating between modified and unmodified pages is crucial. This differentiation is achieved by maintaining the region content (RC) and accessed content (AC) separately. The RC signifies the latest copy of the content prior to incoming updates, while the AC represents the content that a node has accessed. Considering that data in the AC can be altered during a write access, the AC can hold data that is more recent compared to the corresponding RC. Upon receipt of a new remote read/write request, the modified portions of the AC are combined with the RC. As a result, other nodes can access the latest version of the content.

To manage changes in the coherence states of the RC and AC when the scope of the coherence domain changes, a tiered coherence protocol is adopted. This protocol allows for asynchronous adjustments to the coherence states of the RC and AC, which enables a lazy alignment of these states. This implies that the states of the AC can be altered upon actual access, reducing the synchronization cost between the RC and AC.

In order to maintain exclusive updates to the shared data, when the RC is downgraded to read-only permission, the associated AC is immediately downgraded to provide read-only access. If the RC is invalidated, the related AC must immediately switch to the invalidated state. When the AC needs to be upgraded to read/write permission and the corresponding RC already has that permission, the AC can synchronize with the RC and switch to the required read/write permission without any further remote access. Regardless of the size of the coherence domain, the states of the AC can be downgraded immediately and upgraded in a lazy manner.

To ensure the coherence of shared data within a flexible coherence domain, the coherence states of child regions need to be adjusted when the scope of the coherence domain changes. After child regions are merged, the RCs of the invalidated child regions are upgraded to read-only permission during the reading of the parent region. When write access to the parent region is performed, all the covered RCs of child regions that are invalidated or read-only are upgraded to write permission. Before the parent region is modified, the remote copies of the parent region are downgraded to an invalidated state, and the most recent content of the parent region is collected. Further details on the coherence protocol are elaborated in Appendix A, available online.

IV. Content-Based Memory Sharing

A coherence domain is essentially a group of memory pages. If only a small fraction of the data is updated, acquiring the entire data from a coherence domain can be a high-cost operation. Identifying modifications, which are the specific pieces of data that requesters are interested in, is key to reducing synchronization overheads. By separating the unit of data synchronization

from the unit of coherence management, it becomes possible to gather the needed data more efficiently. This separation allows multiple providers to independently supply different portions of the required data.

A. Content Detection

Versioning can be used to identify incremental updates within a coherence domain. This approach involves creating a new version of the coherence domain each time an update is initiated by a writer. The version of the coherence domain remains unchanged until another writer modifies it. When the version changes, all copies of the previous version must be marked as invisible for invalidation, as discussed in Section III-B. However, if these copies have been preserved, they can still be utilized. By tracking updates between two versions of a coherence domain, we can incorporate the necessary changes, also referred to as the content, into a previously saved version. This content represents the portions of data that have been altered since the last access by the requesting node.

For efficient content delivery, the content can be divided into distinct parts. These parts can then be transmitted simultaneously by different nodes, each responsible for maintaining the most up-to-date data. The initial step in this process is to identify the necessary content. To achieve this, every coherence domain $\mathcal{D}_i$ is further divided into a series of blocks. The desired content is then defined at the block level.

Let $\delta_i(v)$ represent the difference between versions v and $v+1$ of $\mathcal{D}_i$ $(v > 0)$. In this case, $\delta_i(v)$ is a difference vector where each element, $\delta_i(v)[j]$, is assigned a value of 1 if the j-th block in $\mathcal{D}_i$ has been altered in version v, and 0 if it remains unchanged. Furthermore, let $\delta_i(u, v)$ denote the difference between versions u and v $(u < v)$ of $\mathcal{D}_i$. $\delta_i(u, v)$ details the content needed after the version of $\mathcal{D}_i$ is updated from u to v. The expression for $\delta_i(u, v)$ is as follows:

$$\delta_i(u, v) = \delta_i(u) \mid \delta_i(u+1) \mid \cdots \mid \delta_i(v-1)$$

where $\mid$ is an element-wise OR operation. Given that a coherence domain may experience significant alterations over time, it is unnecessary to monitor every single update. When the difference between versions u and v surpasses a certain threshold N, it is appropriate to assign a value of 1 to each element of $\delta_i(u, v)$. Only the blocks validated by $\delta_i(u, v)$ are needed for transmission if there is a need to update the coherence domain $\mathcal{D}_i$ from version u to version v.

B. Redundant Hashing

Within a coherence domain, certain nodes, referred to as holders, keep the most recent data. These holders participate in the process of supplying content to a requester. To alleviate the content burden on each holder, the total content is divided among them. Each holder is responsible for providing a part of the total content, which results in more efficient content provision and a reduced workload for each individual holder.

To determine the content that each holder is responsible for, they need to be aware of each other. There is no need for them to have identical perceptions of the other holders. The only necessity is that they all observe each other in the same order, which provides a consistent view of the content providers. To ensure this consistency, every data retrieval request from a requester is passed along in sequence by a unique owner. As a result, the requester becomes a new holder and can be added to the holder list of each content provider upon receiving this notification from the owner. Note that if data retrieval requests can be speculatively sent from the requester to potential holders in parallel, it is possible that some of these holders have not received prior notifications from the owner. As a result, when these speculative data retrieval requests are received, the number of perceived holders can be different among nodes.

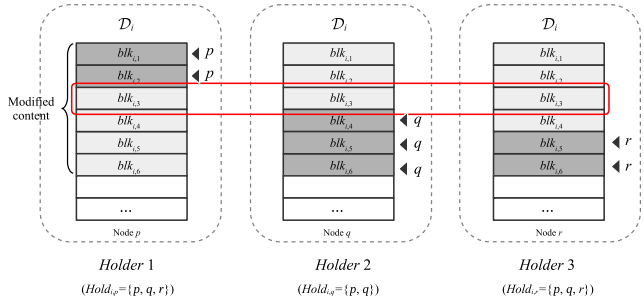


Fig. 2. Missed content (For $C_i = \{blk_{i,k} | 1 \le k \le 6\}$ in $\mathcal{D}_i$, the blocks provided by nodes p, q, and r are $h_1(C_i, 2) = \{blk_{i,1}, blk_{i,2}\}$, $h_2(C_i, 3) = \{blk_{i,4}, blk_{i,5}, blk_{i,6}\}$, and $h_3(C_i, 2) = \{blk_{i,5}, blk_{i,6}\}$, respectively).

Before we introduce the specific hashing approach used to determine the content each holder is responsible for, it is important to address a potential issue. Merely dividing the content evenly among holders can be inadequate if the perception of the holders is not identical across all content providers. Let $Hold_{i,p}$ denote the set of holders observed by a holder p of $\mathcal{D}_i$. For any two nodes p and q of $\mathcal{D}_i$, it is possible that $Hold_{i,p}$ is a subset of $Hold_{i,q}$ when receiving requests for providing the modified blocks of $\mathcal{D}_i$. If the modified blocks are hashed according to the number of observed holders, nodes p and q are possible to obtain different hashes of required content. Let h_n denote the blocks required to be provided by the n-th holder of a coherence domain, where $n > 0$. The k-th block of $\mathcal{D}_i$ is denoted by $blk_{i,k}$, where $k > 0$. Let C_i denote the set of required blocks of $\mathcal{D}_i$. Assume that the version of $\mathcal{D}_i$ that was last accessed by the requesting node is u, and the current version of $\mathcal{D}_i$ is v. The set C_i can be obtained from $\delta_i(u, v)$, which represents the changes between these two versions of D_i. Suppose p is the n-th holder of $\mathcal{D}_i$. A method to obtain h_n is as follows:

$$h_n(C_i, size_p) = \{blk_{i,k} \mid \lfloor (n-1) size_p \rfloor < k \le \lfloor n \cdot size_p \rfloor, blk_{i,k} \in C_i\}$$

where $size_p = |C_i|/|Hold_{i,p}|$. This hashing method can potentially lead to scenarios where the provided content does not encompass all updates within a coherence domain, as depicted in Fig. 2. In such a situation, the required block $blk_{i,3}$ is not included.

In order to ensure the provision of content even when the view of the holders is incomplete, we adopt a redundant hashing (R-Hash) scheme. This scheme allows for the intersection of

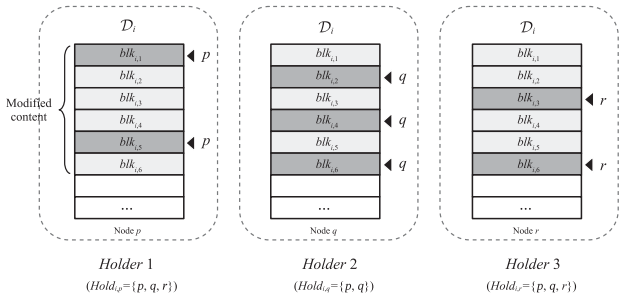


Fig. 3. Compensation (Using R-Hash, the blocks provided by holders p, q, and r can be identified as $h_1(C_i, 3) = \{blk_{i,1}, blk_{i,5}\}$, $h_2(C_i, 2) = \{blk_{i,2}, blk_{i,4}, blk_{i,6}\}$, and $h_3(C_i, 3) = \{blk_{i,3}, blk_{i,6}\}$, respectively).

content provided by different holders. The set of partitioned blocks assigned to the n-th holder is denoted as $part_n$ ($n > 0$). This can be expressed as follows:

$$part_n(C_i) = \{blk_{i,k} \mid k \bmod n = 0, blk_{i,k} \in C_i\}$$

R-Hash aims to ensure that the content provided by the n-th holder can compensate for the content that the potential holders do not provide. In R-Hash, the content to be provided can be calculated as follows:

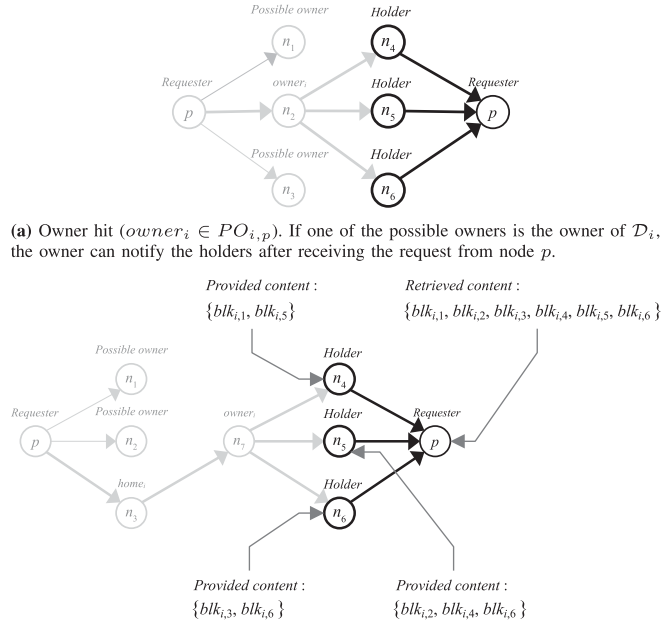
$$h_n(C_i, size_p) = part_n(C_i) - \bigcup_{m=n+1}^{size_p} part_m(C_i)$$

where $size_p = |Hold_{i,p}|$. Using this approach, the requested content can be assembled by combining the blocks sourced from different holders (as shown in Fig. 3). It can be shown that the content generated using R-Hash encompasses the modified blocks. The proof is provided in Appendix B, available online.

C. Multiple Providers Giving Content Together

In the management of coherence domains, each domain is assigned an owner. This owner is responsible for coordinating memory access and tracking the holders of the content within the domain. To ensure SWMR property for data updates, the owner issues write-invalidations. Given the dynamic nature of the list of content holders, the owner sequentially dispatches data retrieval requests to the current holders, ensuring consistent views of holder lists. The owner is initially determined by hashing the accessing addresses. To circumvent memory access bottlenecks, ownership can be dynamically adjusted at runtime. However, broadcasting changes of ownership to all nodes can be costly, so these details are made available only to nodes explicitly requesting access to the coherence domain. The precise ownership of each coherence domain is maintained by dedicated shared memory managers.

The content owner is empowered to dispatch a data retrieval request to the current holders. This prompts the holders to supply different portions of the content. Each requester can speculatively send the data retrieval request to potential owners, under the constraint that the content owner must be among the previous content providers. In a situation where a node is unaware of the current owner, the shared memory manager of the corresponding coherence domain can also be considered a potential owner.



(a) Owner hit ($owner_i \in PO_{i,p}$). If one of the possible owners is the owner of $\mathcal{D}_i$, the owner can notify the holders after receiving the request from node p.

(b) Home hit ($owner_i \notin PO_{i,p} \wedge home_i \in PO_{i,p}$). If none of the possible owners are the owner of $\mathcal{D}_i$, a shared memory manager $home_i$ can forward the request of node p to $owner_i$. On receipt of the request, $owner_i$ can notify the holders to provide content for node p.

(c) Home miss ($owner_i \notin PO_{i,p} \wedge home_i \notin PO_{i,p}$). If the set of possible owners does not contain the owner of $\mathcal{D}_i$ and the corresponding shared memory manager, $known_{i,p}$ can forward the request of node p to manager $home_i$, which can then notify $owner_i$. Finally, $owner_i$ can notify the holders to provide content for node p.

Fig. 4. Routing requests.

TABLE I
SUMMARY OF SYMBOLS IN DATA COLLECTION

Symbol	Description
$owner_i$	The owner of $\mathcal{D}_i$.
$home_i$	The node of the shared memory manager responsible for $\mathcal{D}_i$.
$known_{i,p}$	The owner known to node p with respect to $\mathcal{D}_i$.
$PO_{i,p}$	The set of nodes that node p considers as potential owners of $\mathcal{D}_i$.

We can distinguish a set of scenarios for the collection of shared data (as shown in Fig. 4). The roles of nodes are listed in Table I. Note that if the known owner $known_{i,p}$ does not correspond to the current owner $owner_i$ of $\mathcal{D}_i$, it indicates that the owner of $\mathcal{D}_i$ has been updated without immediate awareness by node p. This results in a temporary mismatch, but the request of node p is still forwarded to the updated owner $owner_i$. Once $owner_i$ receives the request, it forwards it to the relevant holders. Upon receiving the requests, the holders look up the updated blocks and provide different parts of the required content to node p. After node p collects all the required content of a coherence

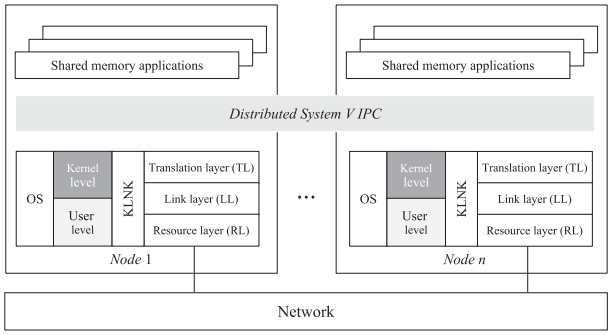


Fig. 5. Architecture of KLNK.

domain, it delivers the data and updates $known_{i,p}$ if necessary. The content provided by each holder is calculated using the R-Hash scheme mentioned in Section IV-B.

V. Implementation

KLNK is a DSM system that operates at the operating system level and enables the creation of elastic coherence domains through transparent management of distributed shared memory. This feature allows traditional shared memory programs to be reused in a distributed environment. Additionally, the programming interfaces of the DSM are compatible with System V IPC, which is supported by modern operating systems.

An isolated space is provided to maintain virtual regions and enable coherence domains to be managed separately. The KLNK system has a layered architecture that involves three layers (shown in Fig. 5): a translation layer (TL), a link layer (LL), and a resource layer (RL). The resource layer, implemented at the user level, provides abstractions of shared memory resources and other shared resources. Within this layer, shared memory is organized using virtual regions that describe coherence domains. The translation layer converts kernel requests and sends them to the resource layer through the link layer. The link layer passes requests from the kernel level to the user level and also delivers the results generated by the resource layer to the kernel.

To decouple the management of distributed resources from the operating system kernel and enhance security, our goal is to establish a dedicated user-space service for distributed resource management that ensures isolation. To achieve this, we utilize the user-level file system (FUSE) [34] to implement a daemon that is extendable for managing shared memory across distributed nodes. Leveraging FUSE allows for the execution of kernel requests through file operations. Each shared memory request is translated into a corresponding file write operation, with the necessary arguments encapsulated within. The RL is implemented as a file system based on FUSE.

When the kernel handles a page fault of a shared memory page (as shown in Fig. 6), the TL forwards the page fault request (❶) to the RL through the LL (❷), which allows kernel requests to be sent through file accesses. The RL captures these file accesses, enabling the handling of page faults of each shared memory page in user space. Within the RL, the coherence domain of the target page can be determined (❸). If the user-space daemon of the RL maintains the latest copy of the requested page in the relative virtual region, the page data can be delivered to the kernel through the LL. Otherwise, the RL sends a virtual region request (❹) that is routed to the holders of the virtual region as described in Section IV-C. Each holder provides necessary content (❺) through the R-Hash scheme upon receiving the request, as discussed in Section IV-B.

The recent update history is maintained within a matrix consisting of a series of difference vectors $\delta_i(v)$. This difference matrix is used to monitor the changes for the latest N versions of the specified memory region D_i. Data outside the range of these N versions is treated as completely revised. When making a request, the requester notes the last known version of D_i, enabling content providers to identify the necessary changes using the difference matrix. This matrix is also supplied to the requester by one of the providers.

Furthermore, each writer accessing the target region must update the most recent difference vector to reflect the latest changes. Upon receiving a request for a specific memory region under the control of a writer node, the system performs a comparison between the data in kernel space and its user-space counterpart. This comparison generates a difference vector, capturing the changes between the two data versions. This vector is then used to update the difference matrix, which records the changes across the most recent versions of the memory region. Consequently, it is possible to calculate the cumulative updates that have occurred from a previous version to the most current version of the memory region, as detailed in Section IV-A.

If the requester intends to update the virtual region, the holder must invalidate each page of the virtual region. The kernel of the holder can be informed of the invalidation through the notification sent via the LL (❻). With the helper handlers added to the kernel, shared memory pages can also be invalidated in kernel space (❼). Once the requester retrieves the data of the corresponding coherence domain, the data of the shared memory page can be delivered to the kernel through the LL.

The user-space service in RL monitors the time interval of page fault requests. When accessing a child region that has its own coherence domain, and the time interval for accessing the memory area within the scope of its parent region falls below a given threshold Δ, child region requests (CRRs) transition to parent region requests (PRRs). This transition helps maintain an expanded coherence domain, allowing the entire content of the parent region to be retrieved using PRRs. Both CRRs and PRRs are classified as coherence domain requests (CDRs), with CRRs specifically used to retrieve the content of child regions.

If the time interval for accessing the memory area within the scope of its parent region exceeds Δ and the requesting child region has already been merged into its parent region, the RL sends a CRR to obtain only the content of the child region, which results in a reduction of the coherence domain. In each CDR, the target read/write permission of the requesting coherence domain is assigned. The owner node can then forward the request to all the holders that maintain the content of the coherence domain.

Upon receipt of the request, the holder adjusts the coherence states of the corresponding RC in user space and, if necessary, simultaneously adjusts the states of the associated AC in kernel space. Each CDR includes a piggybacked version number of the

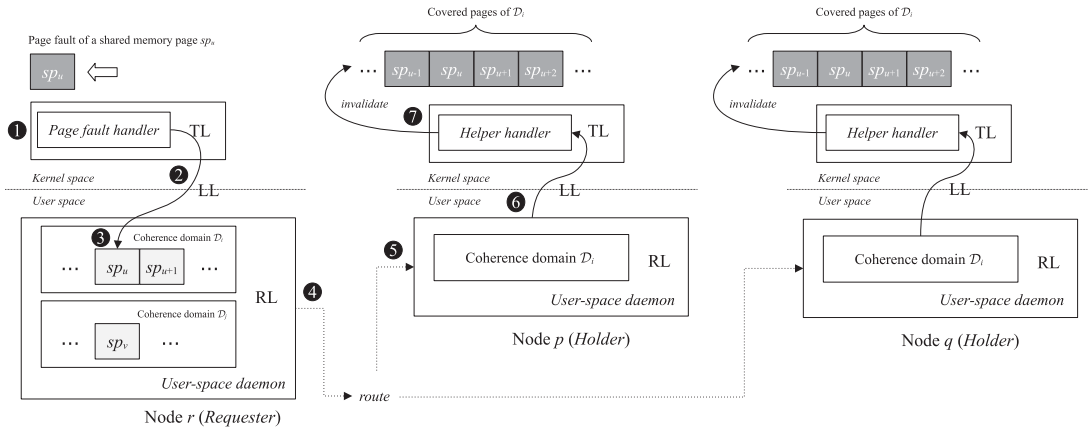


Fig. 6. Procedure for data retrieval.

content perceived by the requesting node. This allows the holders to detect any modifications in a coherence domain from that specific version. If modifications are detected, the holders can provide the distinct parts of the modification to the requesting node concurrently. Once all the required content is collected, the RC of the requesting node is updated, and the most recent data related to the page fault request can be committed to the kernel, resulting in an update to the AC of the requesting node.

When multiple nodes focus on updating particular memory areas, they may frequently request access to the same data items. If a node cannot hold these items while another node requests to update them, the computation can be interrupted frequently for data synchronization. This thrashing, caused by data contention instead of actual computation, can consume significant bandwidth for exchanging data. To mitigate this issue, nodes may need to take more time to update shared data. A thrashing control method [35] is employed to ensure that memory areas are fairly utilized by all nodes.

In the implementation of KLNK, minor modifications have been made to the Linux kernel[1], amounting to approximately 1790 lines of code. The primary functionalities are implemented within the user-level resource layer. This layer establishes an isolated environment[2], using a user-space file system, which facilitates the connection between distributed nodes and the management of distributed resources.

VI. Evaluation

This section assesses the performance of the DSM system under various memory access patterns. We aim to evaluate the following questions:

- What are the effects of merging conventional pages into a coherence domain?
- How does changing the maximum size of the coherence domain (also referred to as the parent region) impact the system?
- What are the implications of adjusting the granularity of data sharing (block size) within each coherence domain?

A. Experiments

For the evaluation, we utilized 8 cloud servers which provided memory resources for the DSM system. Each cloud server had 8 GB of RAM and 4 vCPUs, and was equipped with Intel Xeon Platinum 8369B processors. These cloud servers were connected through a 12.5 Gbps network. We deployed a modified 64-bit Linux kernel with version 5.4.161 on each cloud server. We explore the performance of the DSM system under various access patterns through a set of micro-benchmarks[3].

To provide the baseline performance, we implement page-based shared memory management through the IVY protocol [6] using a fixed distributed manager algorithm. IVY detects shared memory updates at the page level, whereas KLNK detects updates at the block level and collects them incrementally. We evaluate the speedup of KLNK compared to the experimental time of the baseline.

In KLNK, we explore parent regions (*PRs*) with sizes of 16 KB, 32 KB, 64 KB, and 128 KB. Each PR is composed of four child regions, with each child region covering several 4 KB memory pages. We set the time interval Δ for merging child regions to be $50\,ms$. We also examine different block sizes as the block size of coherence domain affects the fragments of the modified data. We refer to the data sharing granularity of an n-byte block as **DSG-n**.

1) Dynamic Access Pattern: With the dynamic access pattern, each process can randomly select shared pages to read or write. Each test round can be a read round or a write round. During the read round, a process continuously reads a randomly selected area of a page. During the write round, a process updates a randomly selected area of a page.

In the evaluation, we set the size of the shared memory to be 1 MB, and we perform 1×10^4 rounds of testing. DAP-2 and DAP-4 provide tests with read/write ratios of 2 and 4, respectively. For DAP-2 (as shown in Fig. 7), the updates of writers are only visible to limited readers, and the data can be

[1]A modified Linux kernel for KLNK is available at https://github.com/virthub/kern-vhub.

[2]A userspace daemon of KLNK is available at https://github.com/virthub/virthub/tree/main/modules/klnk.

[3]A micro-benchmark for the shared memory system is available at https://github.com/virthub/shm-bench.

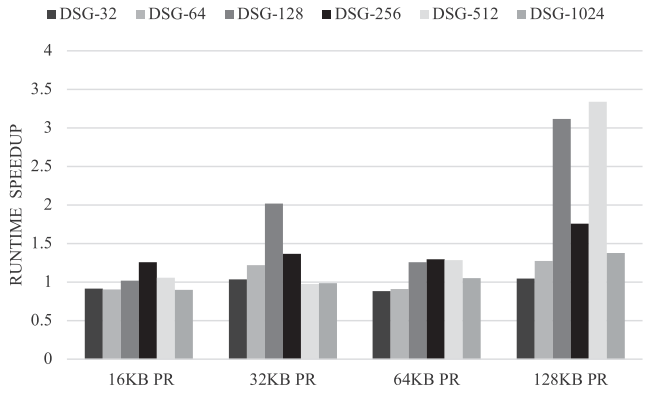


Fig. 7. Dynamic access. Read/write ratio is 2:1 (DAP-2).

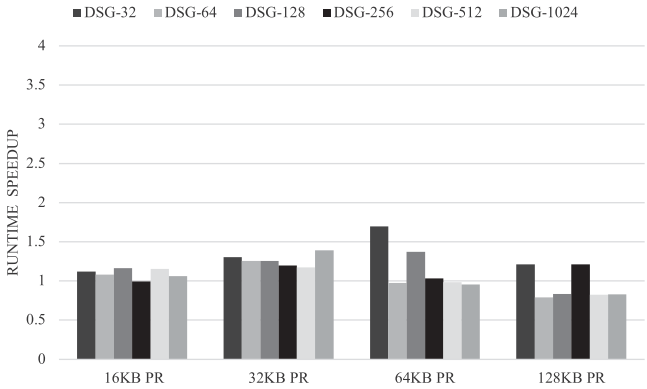


Fig. 8. Dynamic access. Read/write ratio is 4:1 (DAP-4).

TABLE II
CONTENT DETECTION OVERHEAD (128 KB PR)

	DSG-32	DSG-64	DSG-128	DSG-256	DSG-512	DSG-1K
DAP-2	0.24%	0.08%	0.01%	0.35%	0.06%	0.07%
DAP-4	0.26%	0.14%	0.07%	0.23%	0.36%	0.08%

frequently modified by processes located at different nodes. For DAP-4 (as shown in Fig. 8), the data can be shared on a larger scale.

In DAP-2 and DAP-4, capturing the locality of data access can be challenging due to the random access pattern. Sustaining continuous access within a small-sized coherence domain (e.g., 16 KB PR) is less likely, which can hinder the merging of child regions and reduce opportunities for pre-synchronization of the required data. However, for large-sized coherence domains, it is more probable that their child regions are accessed within a given time window. The retrieved data of a coherence domain can be accessed by the visitor without interruption caused by write invalidation. Moreover, during synchronization, only the updated portion of each coherence domain is collected. This allows the utilization of large-sized coherence domains (e.g., 128 KB PR) to enhance performance (as illustrated in Fig. 7). KLNK can achieve up to 3× speedup compared to page-based data sharing across different updating intensities. Table II presents the ratio between content detection duration and total time for data synchronization in the dynamic access patterns.

2) Exclusive Access Pattern: To ensure atomicity during shared data updates, mutual exclusion operations are utilized

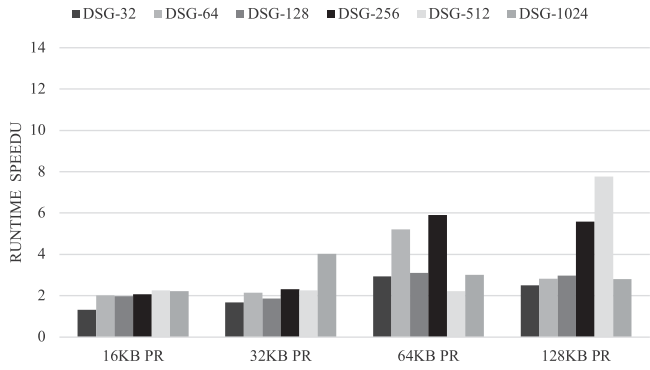


Fig. 9. Exclusive access to 2 KB sized SGs (EAP-2K).

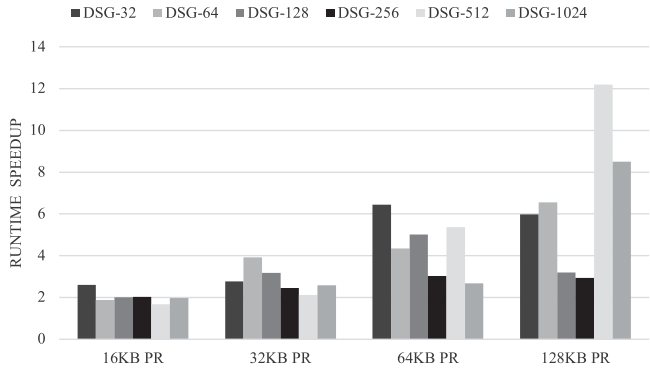


Fig. 10. Exclusive access to 4 KB sized SGs (EAP-4K).

for access control, providing exclusive access to the shared data. The shared memory is divided into groups, enabling seamless switching between different access areas. To manage exclusive memory accesses, each sharing group (SG) is equipped with an SG lock. The operations of the SG lock can introduce variations in global states. To synchronize these states across distributed processes, a mutual exclusion mechanism based on shared memory [36] is employed.

During each testing round, the data within an SG can only be accessed once the corresponding SG lock is acquired. The lock is released after the data access is complete. Each process can act as either a writer, responsible for updating a randomly selected SG, or a reader, obtaining data from an SG. To consider data locality, each process has a probability of P_{next} to continue accessing the SG adjacent to the previously accessed SG. Additionally, in each testing round, a randomly selected area within the SG is accessed. We conduct 1×10^4 testing rounds with 256 SGs, where P_{next} is set to 0.5, and the read/write ratio is 4. The tests are performed on two different SG sizes: EAP-2K, representing a size of 2 KB, and EAP-4K, representing a size of 4 KB. The shared memory utilized to store SG locks has a capacity of 1 MB.

In EAP-2K and EAP-4K (as shown in Figs. 9 and 10), processes have exclusive access to neighboring SGs. However, in EAP-2K, processes are less likely to remain within a sufficiently large coherence domain. By merging child regions, it becomes possible to increase the number of SGs accessed within a given time period, particularly for larger coherence domains like 128 KB PR. KLNK outperforms page-based data sharing, achieving up to 12× speedup.

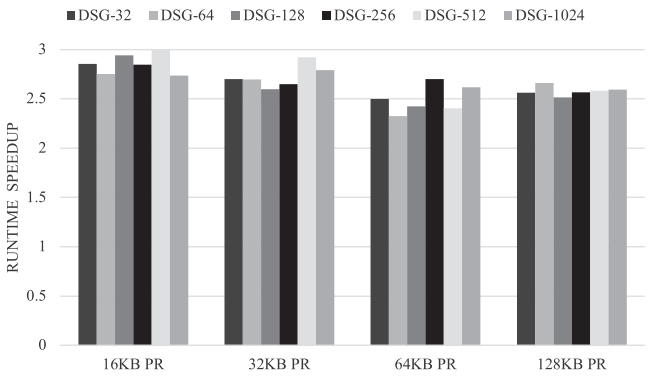


Fig. 11. Contention access. Read/write ratio is 2:1 (CAP-2).

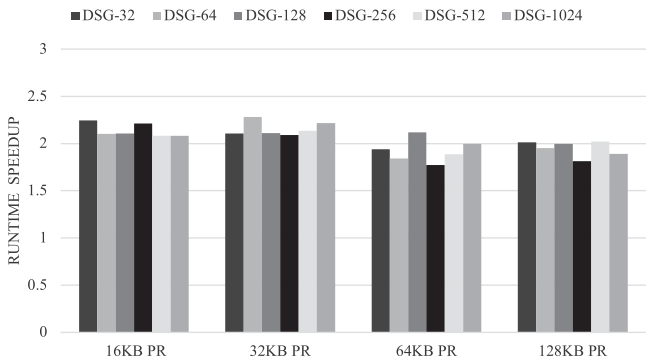


Fig. 12. Contention access. Read/write ratio is 4:1 (CAP-4).

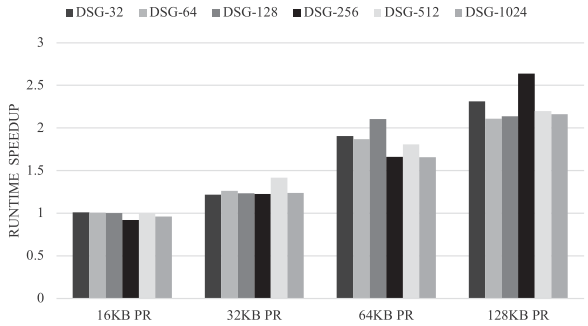


Fig. 13. Regular access (RAP-FFT).

3) Contention Access Pattern: To evaluate the contention that arises from multiple processes accessing shared memory, it is likely that the data required by different processes overlaps. While frequent access to the same area can increase data contention, the required data size can be varied to facilitate access to different data items. This can be achieved by altering the shapes of the areas being accessed so that the accessing area of each process overlaps with that of another.

The access area can be subdivided into smaller regions, granting each process the flexibility to access a specific subregion. During each testing phase, every process selects M areas for access. Let A_i denote the i-th area selected in a testing round, $start_i$ the starting offset of area A_i within the shared memory, and $Total_{pg}$ the total count of memory pages in the shared memory. The value of M is determined as $\log_2(Total_{pg})$. The size of area A_i is calculated using $size_i = 2^{M-i} \cdot pgsz$ for $i = 1, 2, \ldots, M$, where $pgsz$ signifies the size of each memory page. The offset for $start_i$ is computed as $start_i = start_{i-1} + r_i \cdot size_i$, with r_i being a randomly generated Boolean value. The initial offset, $start_0$, is set to 0.

Each area A_i functions as a circular buffer, permitting continuous access to k bytes of the buffer at any given time, where k is a value randomly selected between 1 and $size_i$. When accessing area A_i, a process may either read from or write to the area, with the starting point for the k bytes randomly determined within A_i. As the testing progresses and the access area diminishes in size, it enables processes to repeatedly access overlapping memory areas.

In the evaluation, we set $Total_{pg} = 64$ and $pgsz = 4\,KB$, and perform 1×10^4 rounds of testing. CAP-2 and CAP-4 provide tests with read/write ratios of 2 and 4, respectively. In contention access patterns, shared data items are frequently updated by processes within a certain period. Therefore, a coherence domain can cover the data required by different processes. It is possible that several concatenated pages within a coherence domain get modified before a process accesses them. For large-sized coherence domains (e.g., 32 KB, 64 KB, and 128 KB PRs), the synchronization cost may increase due to the frequent modification of shared data. In contrast, small-sized coherence domains (e.g., 16 KB PR) have the advantage of reducing the amount of data that needs to be collected for memory contention use (as depicted in Figs. 11 and 12). Since content-based memory sharing can be performed at the block granularity, the synchronization cost can be saved. In memory contention access, KLNK can achieve up to 3× speedup.

4) Regular Access Pattern: We employed the Stanford Parallel Application for Shared Memory (SPLASH) benchmark suite [37] to assess the shared memory performance of common workloads. We extended the PARMAC macros, which are components of SPLASH designed to adapt to different inter-process communication interfaces.

KLNK provides a distributed implementation of the System V IPC, facilitating the attachment of shared memory to the individual address spaces of separate processes. However, a limitation in the current implementation prevents the exposure of these individual address spaces across distributed processes. A potential solution, involving the attachment of shared memory to the virtual memory area using a uniform starting virtual address across nodes, is presently not supported. This constraint prompted us to evaluate the performance of shared memory in certain scenarios, employing the FFT and LU kernels, designated as RAP-FFT and RAP-LU, respectively. The RAP-FFT executes a 65536-point Fast Fourier Transform, necessitating comprehensive inter-process communications during the matrix transpose phases. The RAP-LU decomposes a dense matrix into a lower triangular matrix and an upper triangular matrix. To maintain workload equilibrium, the dense matrix is partitioned into blocks, typically of size 16×16, which can be distributed among different processes. For our assessment, we employed a matrix of dimensions 512×512.

As depicted in Fig. 13, the RAP-FFT workload significantly benefits from large coherence domains such as 32 KB, 64 KB,

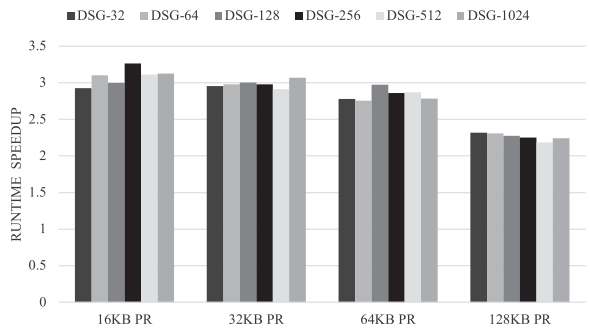


Fig. 14. Regular access (RAP-LU).

TABLE III
TOTAL COST OF SYNCHRONIZATION (128 KB PR)

	DAP-2	DAP-4	EAP-2K	EAP-4K	CAP-2	CAP-4	RAP-FFT	RAP-LU
DSG-32	1.46s	1.06s	1.63s	1.74s	44.46s	36.66s	0.90s	21.73s
DSG-64	2.03s	2.23s	4.37s	1.62s	46.71s	36.26s	0.96s	37.54s
DSG-128	0.66s	2.55s	3.25s	2.97s	44.52s	35.06s	0.97s	23.69s
DSG-256	0.85s	2.29s	1.12s	4.08s	43.60s	40.56s	0.82s	57.39s
DSG-512	0.61s	1.25s	0.99s	1.05s	47.13s	38.84s	0.82s	15.86s
DSG-1K	1.86s	2.93s	3.15s	1.56s	47.36s	34.39s	0.92s	22.10s
Baseline	2.18s	2.06s	9.14s	17.29s	158.39s	84.68s	2.85s	89.73s

and 128 KB PRs, which improve the locality of memory access. This effectiveness diminishes in the RAP-LU workload, as shown in Fig. 14. On the other hand, smaller coherence domains like a 16 KB PR manifest a superior ability to manage synchronization costs. Upon configuring KLNK to utilize 16 KB PRs, we observed a speedup of up to 3× in the RAP-LU workload.

B. Discussion

A coherence domain allows for the grouping of memory pages, which in turn facilitates efficient retrieval of updated pages by a requester in a batched manner. As a result, when a miss occurs for the required data, the process allows for the retrieval of not only the target data but also the surrounding data, functioning similarly to data prefetching. This strategy effectively reduces the number of remote requests triggered by data synchronization, compared to traditional page-level management of shared data. Although data prefetching can enhance performance, executing fine-grained prefetching in distributed environments can be costly due to the high synchronization demands. To address this issue, KLNK offers the flexibility to decouple the unit of data synchronization from the unit of coherence management. As shown in Table III, the data synchronization method is capable of achieving efficiency across a variety of workloads.

To minimize data-sharing costs, it is crucial to synchronize only the modified parts of shared data, avoiding the collection of unchanged, redundant information. By identifying these changes at the block level, redundancy is controlled. Using larger blocks reduces the number of blocks needed for data retrieval but leads to transmitting more redundant data during synchronization. The size of the coherence domain, along with data synchronization granularity, also contributes to data sharing efficiency.

When concurrent data collection is implemented, a larger data sharing scope can significantly impact performance. By adjusting coherence domains based on memory access patterns, data likely to be accessed in the future can be synchronized in advance, further improving efficiency.

Currently, changes in the sizes of coherence domains can be initiated after a fixed monitoring window Δ. The limitation of this approach is that the fluctuations in size are largely influenced by the memory access behaviors observed within the monitoring interval. A fixed time window may not be sufficient to capture relevant memory accesses. An alternative strategy involves predicting the locations of memory access, which are then utilized to determine the sizes of coherence domains. We leave this predictive method for future work.

VII. CONCLUSION

Incorporating distributed shared memory into the underlying system can simplify the management of shared data and facilitate the programming of distributed shared memory applications. This can be achieved through a software-based approach that introduces an operating system-level DSM system. This system utilizes elastic coherence domains to enhance the flexibility of data management. The abstraction of coherence domains enables the dynamic concatenation of a continuous series of pages, thereby reducing the need for frequent synchronization. To accelerate data synchronization in a distributed environment, the required content within each coherence domain can be partitioned and collected concurrently. Our experimental results demonstrate that this approach to transparent management can effectively improve the efficiency of memory sharing in a distributed environment.

ACKNOWLEDGMENTS

We express our gratitude to the anonymous reviewers for their insightful and constructive comments, which have significantly improved the manuscript.

REFERENCES

[1] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, pp. 789–828, 1996.
[2] G. Burns, R. Daoud, and J. Vaigl, "LAM: An open cluster environment for MPI," in *Proc. Supercomputing Symp.*, 1994, pp. 379–386.
[3] E. Gabriel et al., "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proc. Eur. Parallel Virtual Mach./Message Passing Interface Users' Group Meeting*, Springer, 2004, pp. 97–104.
[4] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," *Australian Comput. Sci. Commun.*, vol. 10, no. 1, pp. 56–66, 1988.
[5] J. Zhang et al., "GiantVM: A type-II hypervisor implementing many-to-one virtualization," in *Proc. 16th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2020, pp. 30–44.
[6] K. Li, "IVY: A shared virtual memory system for parallel computing," in *Proc. Int. Conf. Parallel Process.*, 1988, Art. no. 94.
[7] C. Morin et al., "Kerrighed: A single system image cluster operating system for high performance computing," in *Proc. Eur. Conf. Parallel Process.*, Springer, 2003, pp. 1291–1294.
[8] R. Lottiaux, P. Gallard, G. Vallée, C. Morin, and B. Boissinot, "OpenMosix, OpenSSI and Kerrighed: A comparative study," in *Proc. IEEE Int. Symp. Cluster Comput. Grid*, 2005, pp. 1016–1023.
[9] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. C-28, no. 9, pp. 690–691, Sep. 1979.

[10] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," *ACM Trans. Comput. Syst.*, vol. 12, no. 2, pp. 91–122, 1994.
[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proc. 17th Annu. Int. Symp. Comput. Architecture*, 1990, pp. 15–26.
[12] J. Nelson et al., "Latency-tolerant software distributed shared memory," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 291–305.
[13] B. Fleisch and G. Popek, "Mirage: A coherent distributed shared memory design," in *Proc. 12th ACM Symp. Operating Syst. Princ.*, 1989, pp. 211–223.
[14] C. Morin et al., "Kerrighed and data parallelism: Cluster computing on single system image operating systems," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2004, pp. 277–286.
[15] P. Costa, H. Ballani, K. Razavi, and I. Kash, "R2C2: A network stack for rack-scale computers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 551–564, 2015.
[16] E. P. Xing et al., "Petuum: A new platform for distributed machine learning on big data," in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2015, pp. 1335–1344.
[17] A. Tootoonchian, A. Panda, A. Nematzadeh, and S. Shenkar, "Distributed shared memory for machine learning," in *Proc. Int. Conf. Mach. Learn. Syst.*, 2018.
[18] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the dash multiprocessor," in *Proc. 17th Annu. Int. Symp. Comput. Architecture*, 1990, pp. 148–159.
[19] D. Lenoski et al., "The stanford dash multiprocessor," *Computer*, vol. 25, no. 3, pp. 63–79, Mar. 1992. [Online]. Available: https://doi.org/10.1109/2.121510
[20] B. Bershad, M. Zekauskas, and W. Sawdon, "The midway distributed shared memory system," in *Proc. Dig. Papers. Compcon Spring*, 1993, pp. 528–537.
[21] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and performance of Munin," *ACM SIGOPS Operating Syst. Rev.*, vol. 25, no. 5, pp. 152–164, 1991.
[22] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Proc. 2nd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 1990, pp. 168–176.
[23] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, "Quantifying the performance differences between PVM and treadmarks," *J. Parallel Distrib. Comput.*, vol. 43, no. 2, pp. 65–78, 1997.
[24] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "Tread marks: Distributed shared memory on standard workstations and operating systems," in *Proc. USENIX Winter Tech. Conf.*, 1994.
[25] C. Amza et al., "TreadMarks: Shared memory computing on networks of workstations," *Computer*, vol. 29, no. 2, pp. 18–28, 1996.
[26] L. Iftode, J. P. Singh, and K. Li, "Scope consistency: A bridge between release consistency and entry consistency," *Theory Comput. Syst.*, vol. 31, no. 4, pp. 451–473, 1998.
[27] Q. Cai et al., "Efficient distributed memory management with RDMA and caching," in *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1604–1617, Jul. 2018. [Online]. Available: https://doi.org/10.14778/3236187.3236209
[28] Q. Wang, Y. Lu, E. Xu, J. Li, Y. Chen, and J. Shu, "Concordia: Distributed shared memory with in-network cache coherence," in *Proc. 19th USENIX Conf. File Storage Technol.*, M. K. Aguilera and G. Yadgar, Eds., 2021, pp. 277–292. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/wang
[29] M. K. Aguilera et al., "Remote memory in the age of fast networks," in *Proc. Symp. Cloud Comput.*, 2017, pp. 121–127.
[30] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with INFINISWAP," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 649–667.
[31] M. K. Aguilera et al., "Remote regions: A simple abstraction for remote memory," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 775–787.
[32] I. Calciu et al., "Rethinking software runtimes for disaggregated memory," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, New York, NY, USA, 2021, pp. 79–92. [Online]. Available: https://doi.org/10.1145/3445814.3446713
[33] N. P. Jouppi, "Cache write policies and performance," *ACM SIGARCH Comput. Architecture News*, vol. 21, no. 2, pp. 191–201, 1993.
[34] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE or not to FUSE: Performance of user-space file systems," in *Proc. 15th Usenix Conf. File Storage Technol.*, 2017, pp. 59–72.
[35] Y.-W. Ci, M. R. Lyu, Z. Zhang, D.-C. Zuo, and X.-Z. Yang, "Random priority-based thrashing control for distributed shared memory," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 3, pp. 663–674, Mar. 2020.
[36] L. Lamport, "A fast mutual exclusion algorithm," *ACM Trans. Comput. Syst.*, vol. 5, no. 1, pp. 1–11, 1987.
[37] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," *ACM SIGARCH Comput. Architecture News*, vol. 23, no. 2, pp. 24–36, 1995.
[38] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *Proc. 11th Annu. Int. Symp. Comput. Architecture*, 1984, pp. 348–354.

Yi-Wei Ci received the PhD degree in computer science from the Harbin Institute of Technology, PR China, in 2010. He is with the Institute of Software, Chinese Academy of Sciences. His research interests include distributed computing and computer architecture.

Michael R. Lyu (Fellow, IEEE) received the BS degree in electrical engineering from National Taiwan University, Taipei, Taiwan, R.O.C., in 1981, the MS degree in computer engineering from the University of California, Santa Barbara, in 1985, and the PhD degree in computer science from the University of California, Los Angeles, in 1988. He is currently a professor with the Department of Computer Science and Engineering, Chinese University of Hong Kong. His research interests include software reliability engineering, distributed systems, fault-tolerant computing, and machine learning. He is an ACM fellow, an AAAS fellow, and a croucher senior research fellow for his contributions to software reliability engineering and software fault tolerance.

Zhan Zhang received the PhD degree in computer science from the Harbin Institute of Technology, PR China, in 2008. He is currently a professor with the School of Computer Science and Technology, Harbin Institute of Technology. His main research interest is fault-tolerant computing.

De-Cheng Zuo received the PhD degree in computer science from the Harbin Institute of Technology, PR China, in 2001. He is currently a professor with the School of Computer Science and Technology, Harbin Institute of Technology. His main research interest is fault-tolerant computing.

Xiao-Zong Yang is currently a professor with the School of Computer Science and Technology, Harbin Institute of Technology, PR China. His main research interest is fault-tolerant computing.