


# Providing scalable single-operating-system NUMA abstraction of physically discrete resources

Baik Song An<sup>1</sup>  | Myung Hoon Cha<sup>1</sup> | Sang-Min Lee<sup>2</sup> | Won Hyuk Yang<sup>3</sup> | Hong Yeon Kim<sup>1</sup>

<sup>1</sup>Artificial Intelligence Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea

<sup>2</sup>SMEs and Commercialization Division, Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea

<sup>3</sup>Department of Computer Science and Engineering, POSTECH, Pohang, Republic of Korea

## Correspondence

Baik Song An, Artificial Intelligence Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea.  
Email: bsahn@etri.re.kr

## Funding information

This research was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP), Government of Korea (MSIT), Grant/Award Number: 2022-0-00498.

## Abstract

With an explosive increase of data produced annually, researchers have been attempting to develop solutions for systems that can effectively handle large amounts of data. Single-operating-system (OS) non-uniform memory access (NUMA) abstraction technology is an important technology that ensures the compatibility of single-node programming interfaces across multiple nodes owing to its higher cost efficiency compared with scale-up systems. However, existing technologies have not been successful in optimizing user performance. In this paper, we introduce a single-OS NUMA abstraction technology that ensures full compatibility with the existing OS while improving the performance at both hypervisor and guest levels. Benchmark results show that the proposed technique can improve performance by up to 4.74× on average in terms of execution time compared with the existing state-of-the-art open-source technology.

## KEYWORDS

hypervisor, kernel, single-OS abstraction, virtualization

## 1 | INTRODUCTION

The volume of data required by computing systems for analysis and processing is increasing at an explosive rate, and this trend has been further accelerated by the recent emergence of large-scale machine learning technologies. For example, deep learning recommendation models generally require high computational power and a large memory bandwidth with central processing units (CPUs) and graphics processing units during embedding [1]. Therefore, the design and implementation of systems that can handle large volumes of data have been extensively

explored, and multicore or manycore systems equipped with many processor cores and high-capacity memory in a single node have emerged. Most of these systems are implemented as scale-up solutions with a nonuniform memory access (NUMA) architecture owing to physical limitations such as the limited number of memory channels per CPU. Consequently, scale-up systems contain multiple physical NUMA nodes in a single chassis. However, they have a low cost efficiency, and their setup costs increase exponentially with scaling. Owing to the rapid development of interconnected technologies and improved price competitiveness, scale-out systems

containing clusters with low-cost small servers are being increasingly used. Although scale-out systems have superior price competitiveness compared with scale-up systems, they have limitations because programs must be developed using a complicated programming model such as MapReduce or MPI.

Recently, virtualization-based single-operating-system (OS) NUMA abstraction has attracted the attention of researchers. It ensures compatibility with easy-to-use single-node programming interfaces across multiple physical nodes and improves the cost effectiveness compared with scale-up systems. Furthermore, it leverages the existing OS as a guest. The major challenge of single-OS NUMA abstraction is to minimize performance degradation owing to abstraction overheads. Remarkably, a huge gap in memory performance between local and remote accesses hinders the overall system performance. To prevent this problem, each layer in a software stack must be optimized to single-OS abstraction environments, but existing technologies do not have this capability. For instance, existing OSs are not designed for use as guests in single-OS abstraction environments. In addition, hypervisors are suboptimal, especially with distributed shared memory (DSM) that abstracts discrete memory resources in multiple physical nodes.

We introduce a virtualization-based single-OS NUMA abstraction architecture that enables cooperation between the guest OS and hypervisor while maximizing the compatibility with existing OSs. The DSM overheads are mitigated to overcome the major bottleneck of system performance. The proposed architecture achieves a performance improvement by a factor of up to 4.74 on average compared with state-of-the-art open-source technologies on micro and macro benchmarks.

The remainder of this paper is organized as follows. In Section 2, we explain the overall architecture of the proposed single-OS NUMA abstraction. Section 3 and 4 present the optimization techniques for the hypervisor and guest OS, respectively. In Section 5, we evaluate and analyze the proposed techniques in comparison with existing technologies, and related work is presented in Section 6. Finally, Section 7 summarizes our work and presents conclusions.

## 2 | ARCHITECTURE OF PROPOSED SINGLE-OS NUMA ABSTRACTION

In this section, we explain the overall architecture of the proposed single-OS NUMA abstraction and briefly describe the role, function, and interactions of the components that constitute the corresponding system.

Figure 1 depicts the overall structure of our proposed system. Multiple physical nodes are abstracted to create a large virtual machine (VM), and each physical node exports its hardware resources, such as CPUs and memory devices, to make them available to the VM. All the physical nodes are interconnected via a commercial off-the-shelf network infrastructure (for example, InfiniBand). Like existing virtualization technologies, abstracting physical hardware resources is accomplished by a hypervisor. However, existing virtualization splits a single physical resource into multiple virtual resources, whereas the proposed single-OS NUMA abstraction combines multiple discrete resources into a single one.

With the VM, hardware resources exported from multiple physical machines can be used as if they belong to a single machine. All the resources are functionally transparent and an existing OS can run as a guest on the VM without modification. To optimize the performance, the hypervisor and guest OS must be tuned. As the hypervisor is in charge of hardware abstraction, it cannot access the information in the guest OS. Therefore, the guest makes the hypervisor aware of it through hypercalls or some paravirtualized resources, and all these implementation details are hidden from users. The hypervisor abstracts resources belonging to the same physical machine as a NUMA node for the guest OS to manage the resources and make decisions based on the NUMA awareness aiming to minimize unnecessary data movement between nodes. In addition, user applications must

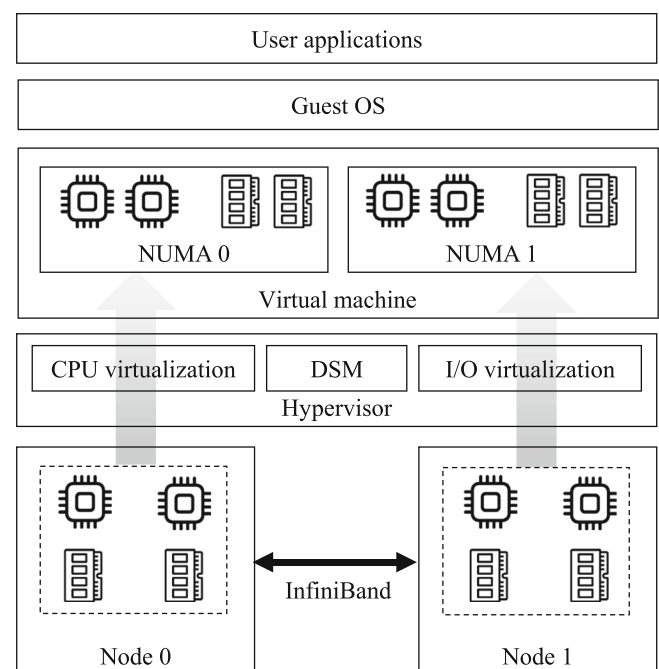


FIGURE 1 Overall architecture of proposed single-OS NUMA abstraction.

be aware of the NUMA architecture for optimal performance. We assume that applications are already tailored for NUMA awareness in this study.

The ideal single-OS NUMA abstraction should address the following questions.

- Can single-OS NUMA abstraction provide the same programming interface as that of a single scale-up NUMA machine to users?
- Can performance overheads of the single-OS NUMA abstraction owing to data movement be minimized or hidden? In other words, can communication overheads between NUMA nodes be reduced even with slower networks compared with intra-node interconnects (for example, UPI or Infinity Fabric)?

We answer the first question above, stating that an existing OS can run as it is on top of hardware resources abstracted as virtual NUMA nodes. The amount of guest modification for performance optimization is minimized, and all changes are completely hidden from users. To answer the second question, we explain various optimization techniques in Sections 3 and 4.

### 3 | HYPERVISOR DESIGN OPTIMIZATION

The hypervisor abstracts two major components: virtual CPU (vCPU) and memory. vCPUs are usually implemented as user threads running on hosts and can migrate between physical cores inside a machine. However, we pin all vCPUs to physical cores to impede migration. In a typical bare-metal NUMA system, thread migration between different sockets may cause unnecessary performance degradation. For a single-OS NUMA abstraction system, the situation becomes even worse because scheduling for optimal performance is complicated. For instance, we should jointly consider user processes or threads, vCPU threads, and physical cores to obtain the best results.

Discrete memory devices located in multiple physical nodes are provisioned to the guest as a single large memory device abstracted into DSM. A guest physical page may be duplicated, migrated, or invalidated as necessary in multiple physical nodes with DSM. An ownership is managed per guest physical page, such that a DSM manager in the hypervisor can handle the guest pages based on the ownership information. In multithread programs, all threads share an identical address space. Hence, if multiple threads are concurrently running in different physical nodes, some pages may be shared between nodes, and communication overheads owing to page

movement between nodes should be minimized for optimal performance.

Figure 2 illustrates the DSM operation when a guest page fault occurs. First, an EPT violation causes a VM exit on the hypervisor in the physical node where the faulting process is running, and it is handled by `handle_ept_violation()` in a Linux KVM. Eventually, the DSM module in the hypervisor takes control of the page fault handling through `tdp_page_fault()`, which makes a request to the DSM module running on the owner node of the corresponding page. After receiving the request, DSM in the owner fetches the requested data from the local memory it manages and sends the data to the requestor.

#### 3.1 | `clear_page()` optimization

`clear_page()` is a function implemented in the Linux kernel to fill newly allocated pages with zeroes. When a guest runs Linux, this function is called during a page fault handling procedure starting from `handle_mm_fault()`. A guest process or thread may try to allocate a physical page that is not owned by the node it is currently running on. Then, the hypervisor must request the owner node to fetch the page whose data should be all zeroed immediately. As it has no information about whether the requested page would be zeroed, the hypervisor should adhere to the memory coherency policy. Serious performance degradation is expected under memory-intensive workloads. To handle this situation, we make the hypervisor aware of `clear_page()` and take special actions to mitigate overheads. Figure 3 depicts the `clear_page()` optimization mechanism in the proposed single-OS NUMA abstraction. Instead of requesting and fetching the remote page synchronously, the hypervisor allocates a local page once it detects that the kernel is running `clear_page()`. Then,

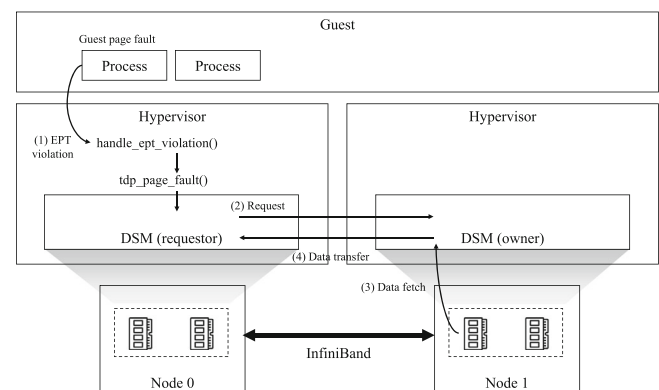


FIGURE 2 Guest page fault handling through DSM.

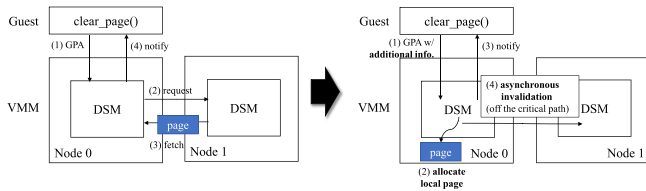


FIGURE 3 `clear_page()` optimization.

it zeroes the newly allocated page and makes it available to the user instantly while invalidating the remote page asynchronously, which is off the critical path.

### 3.2 | RDMA communication optimization

Page transmission between physical nodes is done via RDMA operations. One-sided RDMA is preferable for performance, but two-sided RDMA verbs should be used because the requester has no information of the address to fetch data from. Multiple copies of a guest physical page in various physical nodes may have various host virtual addresses. Therefore, transmission is performed using send/receive pairs between a requester and responder. For optimal performance, we implement different `receive()` mechanisms for the requester and responder. The requester performs a busy waiting during `receive()` execution to improve its responsiveness. The responder performs `poll()/sleep()` to save CPU utilization of the kernel threads it runs on. Hence, both responsiveness and CPU efficiency are achieved.

### 3.3 | Page ownership initialization

The ownership of all the guest physical pages is initialized with the identifier of the NUMA node that each physical page belongs to. When a guest process or thread requests memory allocation with a NUMA API (for example, `numa_alloc_onnode()`), it should get the page from the NUMA node specified by the request for optimal memory performance. If the page ownership is not properly initialized without considering NUMA as in [2], unnecessary communication may occur during memory allocation owing to ownership changes between NUMA nodes. Furthermore, once pages are allocated in a specific NUMA node, they are not likely to migrate to other nodes. When the pages are reused by another process or thread after reclamation, the ownership remains at that node. Thus, we can continue managing the pages efficiently.

### 3.4 | Page prefetching

We use a simple analyzer to detect sequential memory access patterns. Each vCPU in the VM keeps track of its own page access pattern at the guest physical address (GPA) level to detect its sequential patterns. If the sequential behavior exceeds a predefined threshold, the vCPU makes a prefetch request for a next page at the GPA level.

## 4 | GUEST KERNEL DESIGN OPTIMIZATION

This section presents the optimization techniques to design a guest kernel for the proposed single-OS NUMA abstraction. Guest kernel optimization aims to let the guest recognize the hypervisor implementation for resource management and function extensions, thus enabling optimized guest operations. All the optimization procedures are hidden from users, ensuring compatibility with existing user applications without requiring changes in the API.

### 4.1 | DSM-friendly readers–writer synchronization

The most important issue in single-OS NUMA abstraction is to minimize the performance overheads of a memory system implemented as DSM. A major cause of performance degradation with parallel workloads is the synchronization overheads of data accesses shared by multiple processes or threads. Data synchronization with locking is a common problem that seriously degrades the performance of manycore systems. When lock contention becomes serious, several processes attempt to access a single lock variable concurrently, causing severe cache line bouncing that leads to the rapid degradation of the overall system performance.

Readers–writer lock is a synchronization mechanism that allows simultaneous access to a critical section for read-only requests while impeding access during a write operation occupying the lock. It is widely used because it helps to improve the scalability of manycore systems for read-intensive workloads. However, when a readers–writer lock is used in DSM, the system performance may decrease severely even under read-only workloads. In fact, although concurrent read accesses to a critical section are possible, the lock variable is highly contended because all the threads try to simultaneously write the lock variable. This problem occurs in all NUMA systems, but it becomes worse in the case of multi-node systems

managed by DSM because inter-node interconnects are much slower than local system buses. Thus, a novel synchronization mechanism optimized for DSM is needed to prevent this problem.

We propose a new readers–writer synchronization technique that aims to minimize performance degradation for simultaneous read accesses to critical sections in DSM by efficiently managing lock variables to avoid simultaneous writes to a shared lock variable. Figure 4 shows the difference between an existing readers–writer lock technique and the proposed one. With the existing technique, threads running on different machines share a single lock variable. Thus, the page containing the lock must be updated and bounces between nodes, even under read-only workloads. The proposed technique uses as many lock variables as the number of physical nodes to avoid page bouncing. Multiple lock variables are implemented as an array with as many entries as the maximum number of physical nodes allowed in the system. Each entry is aligned to fit the size of a page management unit in DSM to prevent false sharing between nodes. For example, if the page management unit size in DSM is equal to that of an x86 structure (that is, 4 KB), each lock entry is aligned with 4 KB.

When a thread attempts to occupy a reader lock, it simply checks the physical node identifier it runs on and updates the corresponding entry in the lock array with an atomic operation when the lock is available. For a writer lock, a thread occupies it by checking and updating the values of all entries in the lock array one by one. If any entry is occupied by another thread, the writer restores all the updated entries and abandons the attempt. Deadlock can be prevented by updating the lock variables in a predefined order when occupying or releasing the lock entries. Our implementation prioritizes reader over writer locks to help improving concurrency and system scalability for read-mostly critical sections.

We first apply this lock implementation to `mmap_sem` (`mmap_lock`) of the `mm_struct` data structure in the Linux kernel. `mmap_sem` is a readers–writer semaphore

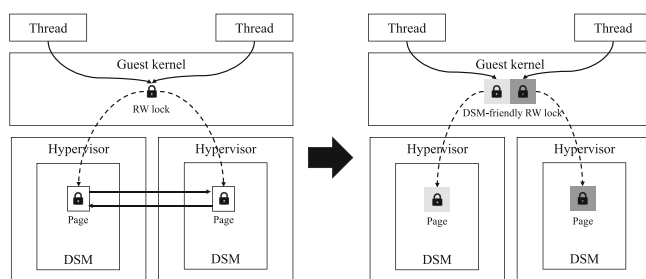


FIGURE 4 DSM-friendly readers–writer synchronization.

to protect VMA structures organized as a red–black tree and synchronizes readers for VMA search during page fault handling. With the existing `mmap_sem` when multiple threads simultaneously generate page faults, they simultaneously attempt to get `mmap_sem` with reader locks, causing page thrashing in DSM and enormous performance degradation. The proposed implementation avoids page thrashing.

The proposed readers–writer synchronization technique helps to improve the memory performance of existing multi-socket NUMA scale-up systems in general and not only for the systems with DSM. The page-aligned, multiple-entry lock structure enables each thread to access its own lock entry in the NUMA node it runs on without interfering other entries in different nodes. Hence, inter-socket communication overheads are reduced, leading to better memory access performance. The corresponding evaluation results are presented in Section 5.

## 4.2 | Spinlock alternative

Spinlock is a synchronization technique for threads achieved through busy waiting on a condition variable. It is widely used in various software packages, including OS kernels, because it can be easily used given its small memory footprint. Generally, it is implemented with atomic instructions such as compare-and-swap. The degree of spinlock contention increases with the system scale, causing the overall system performance to drop sharply. The MCS lock has emerged to prevent this problem. It avoids the lock contention by allocating local memory to each thread and busy waiting on the memory owned by the thread.

The spinlock implementation in the Linux kernel consists of two parts with different purposes according to the degree of contention. Under low contention, spinlock is acquired and released by an atomic instruction to a shared variable. Under high contention, it switches to the MCS lock mechanism. Using spinlock in a guest of a single-OS NUMA abstract system severely degrades performance, even under low contention. In fact, simultaneous access to a lock variable through atomic instructions cause enormous memory overheads owing to excessive communication in DSM.

We aim to improve the performance of a guest kernel by adopting cache coherent synchronization (CC-Synch) [3], a type of flat-combining lock, to the guest kernel. CC-Synch allows a designated thread (combiner) to execute critical sections on behalf of other threads instead of sharing the lock variable like spinlock. Thus, performance degradation owing to the spinlock



contention can be avoided. With CC-Synch, when multiple threads simultaneously make synchronization requests, they wait on a queue performing local spinning while the combiner thread handles the requests in the queue. An upper bound on the number of requests that each combiner serves is defined to prevent the combiner from traversing a continuously growing queue. After serving the predefined number of requests, the combiner identifies the next active thread as the new combiner.

Using CC-Synch without modification does not improve performance in single-OS NUMA abstraction because DSM still shows several EPT violations owing to the data structure used by CC-Synch. DSM-friendly CC-Synch should first use a synchronization instance per NUMA node. The memory used by each CC-Synch must be allocated from its local NUMA node and aligned to a page in DSM to avoid false sharing. We use spinlock for global synchronization between CC-Synch instances in multiple NUMA nodes. Contention in the global spinlock is not serious because the number of contenders is at most that of NUMA nodes. We apply a backoff delay to access the global spinlock to further reduce contention. Figure 5 illustrates the implementation of DSM-friendly CC-Synch for the proposed single-OS NUMA abstraction.

### 4.3 | Adjusting alignment of guest kernel data

When multiple pieces of data with different owners and access patterns are in the same page, false sharing may cause unnecessary memory performance degradation. The Linux kernel provides mechanisms (for example, CONFIG\_X86\_VSMP) to guarantee page size alignment, but the scope of kernel data affected by the option is very limited.

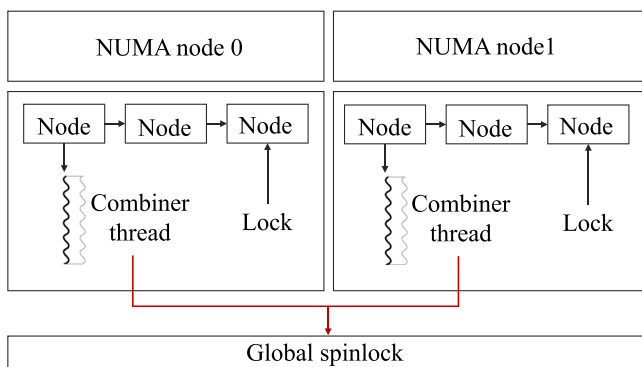


FIGURE 5 DSM-friendly CC-Synch implementation.

We analyze all the page faults generated by the guest kernel, choose kernel data that cause most page faults, and remove them by assigning separate pages to data with false sharing. This slightly increases the usage of guest memory. Nevertheless, considering the tradeoff between performance and memory footprint, our approach seems feasible.

## 5 | PERFORMANCE EVALUATION

### 5.1 | System configuration

We evaluate the proposed techniques constituting single-OS NUMA abstraction to examine the improvement in the overall system performance on various benchmarks.

We compare the proposed single-OS NUMA abstraction with GiantVM [2], a state-of-the-art open source many-to-one virtualization technology. The comparisons are made using the same number of CPU cores and memory devices. Two different hardware platforms are used for evaluation. The first platform is a 2-node system, with each node equipped with two Intel Xeon Gold CPUs. For both GiantVM and the proposed single-OS abstraction, each physical node exports 10 CPU cores and 128 GB of memory to create a virtual machine with 20 vCPUs and 256 GB of memory. The second platform is another 2-node system, with each node equipped with two Intel Xeon E5 CPUs and running a VM with 32 vCPUs and 64 GB of memory.

GiantVM adopts the Linux kernel (KVM) version 4.9.76 and QEMU for the hypervisor, both modified to create a multi-node VM. We modify the Linux kernel 4.18.20 and QEMU for the hypervisor of our single-OS NUMA abstraction system. As GiantVM does not consider a guest OS, we use an unmodified Ubuntu kernel 4.15.0-54 as the guest for GiantVM. The guest optimization techniques described in Section 4 are implemented as kernel patches, and we apply them to the Ubuntu kernel for use in our single-OS abstraction system. Tables 1, 2 and 3 list the detailed configurations used for evaluation.

TABLE 1 Hardware configurations for Xeon Gold.

Components	Specification
CPU	2 × Intel Xeon Gold 5215M (10 physical cores, no hyperthreading)
Memory	384 GB
Interconnect	Mellanox ConnectX-5 HCA (EDR)
Number of nodes	2

TABLE 2 Hardware configurations for Xeon E5.

Components	Specification
CPU	2 × Intel Xeon E5-2623 v3 (16 physical cores, no hyperthreading)
Memory	64 GB
Interconnect	Mellanox ConnectX-5 HCA (EDR)
Number of nodes	2

TABLE 3 Software configurations.

Type	Components	Specification
Hypervisor	Host OS	Ubuntu 18.04 LTS
	Host kernel	4.9.76 (GiantVM) 4.18.20 (proposed system)
	QEMU	2.8.11
	Guest	
Guest	Guest OS	Ubuntu 18.04 LTS
	Kernel	Ubuntu 4.15.0-54-generic
	Kernel parameters	nokaslr
Workloads	Compile options	-O3 -lpthread
	Benchmarks	In-house memory benchmark
		In-house KV-store benchmark
		PARSEC parallel benchmark
	NAS parallel benchmark	

## 5.2 | Performance improvement under memory-intensive workloads

First, we conduct a performance comparison with memory-intensive workloads in the Xeon Gold platform. We implement an in-house multithread memory benchmark that is simple but memory-intensive. In this benchmark, a master thread first allocates a source memory shared by multiple worker threads through a `mmap()` system call and populates the allocated memory area by touching it, thereby causing page faults. Then, it creates as many worker threads as the number of vCPUs in a system. Each worker thread allocates and populates its thread-local destination memory, and then it sequentially copies data from the shared source to the destination using the `memcpy()` operation. The execution time of mapping and population for both source and destination memory as well as the elapsed time with `memcpy()` are measured per thread. All worker threads are pinned to their corresponding vCPUs, and each vCPU thread is also

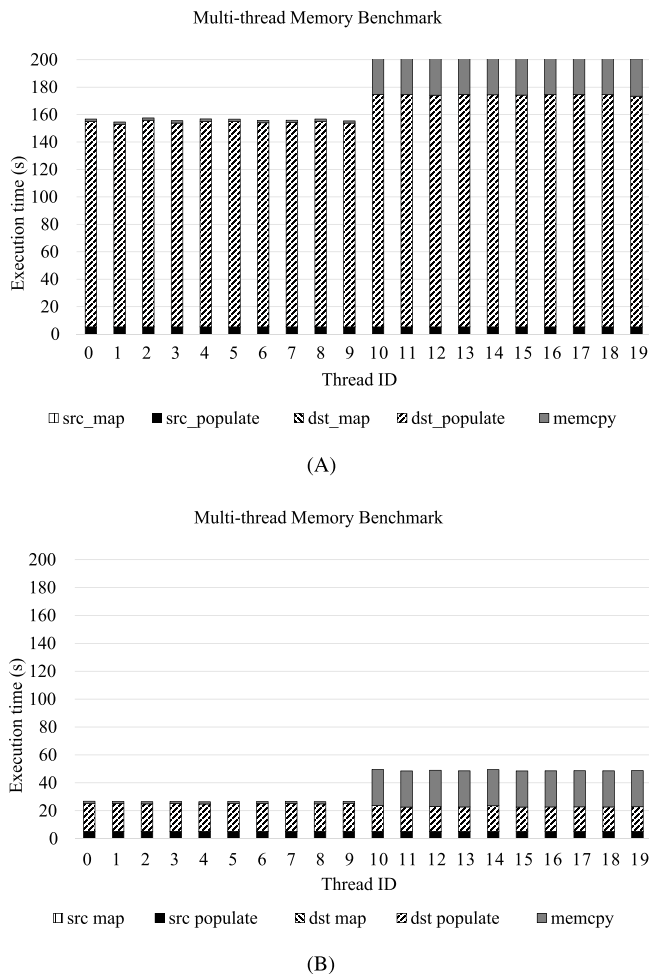


FIGURE 6 Memory-intensive benchmark results: (A) GiantVM and (B) proposed architecture.

pinned to its corresponding physical core. Therefore, threads 0 through 9 run on a primary physical node (node 0) and threads 10 through 19 run on a secondary physical node (node 1) without migration between the nodes.

Figure 6 shows results of the memory benchmark using GiantVM and the proposed single-OS NUMA abstraction. As a VM with 20 vCPUs is configured to use 10 cores per physical node, 20 worker threads are spawned and executed. The x axis represents the worker thread identifier, and the y axis represents the breakdown of execution time per thread. The mapping and population of source memory are performed by a separate master thread, and the corresponding overheads are equally added to those of worker threads. The performance overheads of memory mapping of both source and destination are marginal, and most overheads come from the memory population and `memcpy()` phases.

The proposed architecture improves the performance by a factor of 4.74 on average in terms of execution time

compared with GiantVM. Specifically, it drastically reduces the time consumed by destination memory population, which takes most of the execution time. When multiple page faults are handled simultaneously by various threads, `mmap_sem` located in `mm_struct` in the guest kernel is concurrently accessed. It causes severe thrashing in DSM for the corresponding kernel page containing `mmap_sem`. The DSM-friendly readers–writer synchronization in the single-OS abstraction solves this performance issue. Other optimization techniques introduced in Sections 3 and 4 also contribute to the substantial performance gain.

Further, we attempt to examine the effect of the guest optimization techniques explained in Section 4 with a multi-socket NUMA scale-up system. Thus, we configure another Xeon Gold server with traditional virtualization using vanilla QEMU/KVM. A VM is configured with the same number of vCPUs and memory devices but covering two NUMA sockets in a single physical node. Figure 7 shows the results of the memory

benchmark with a vanilla Ubuntu guest and with the optimized one. The optimized guest outperforms the vanilla guest by approximately 28% on average. The amount of improvement is smaller than that achieved with single-OS NUMA abstraction. This is reasonable because memory overheads in scale-up systems are inherently smaller than those in multi-node scale-out systems, resulting in a relatively smaller performance gap. Nevertheless, we believe this is a remarkable result that demonstrates the applicability of the proposed techniques to general NUMA architecture as well as single-OS NUMA abstraction.

### 5.3 | Evaluation with in-memory key-value store workloads

To evaluate the performance with more realistic memory-intensive workloads, we implement a simple tree-based in-house key–value store (KVS) that operates as follows. First, multiple worker processes are forked, and each worker generates a predefined number of key–value pairs to be inserted into the tree. Then, each worker iteratively generates a random key and looks it up in the tree. Finally, all trees are deleted from memory and the program terminates. This benchmark is multi-process (not multi-thread), and the volume of shared memory is relatively small compared with multi-thread workloads. However, the memory for inter-process communication in synchronization among processes is extensively shared. Furthermore, memory performance issues originating from the guest kernel persist. Figure 8A,B illustrates the performance of GiantVM and the proposed single-OS abstraction in the in-memory KVS workload, respectively. The x axis represents the worker process ID, and the y axis shows the breakdown of process execution time for the user and kernel. Each worker individually creates a KVS of approximately 400 MB. Subsequently, workers iteratively perform search operations on random keys, repeating the process for the same number of times as the count of entries within the KVS. The single-OS abstraction technique outperforms GiantVM by approximately 23% on average. A substantial portion of the performance enhancement can be attributed to the reduction in guest kernel overheads. Hence, the proposed scheme improves the performance even in realistic in-memory workloads.

### 5.4 | Impact of DSM-friendly CC-Synch mechanism

To verify the performance improvement provided by the proposed CC-Synch implemented in a guest kernel

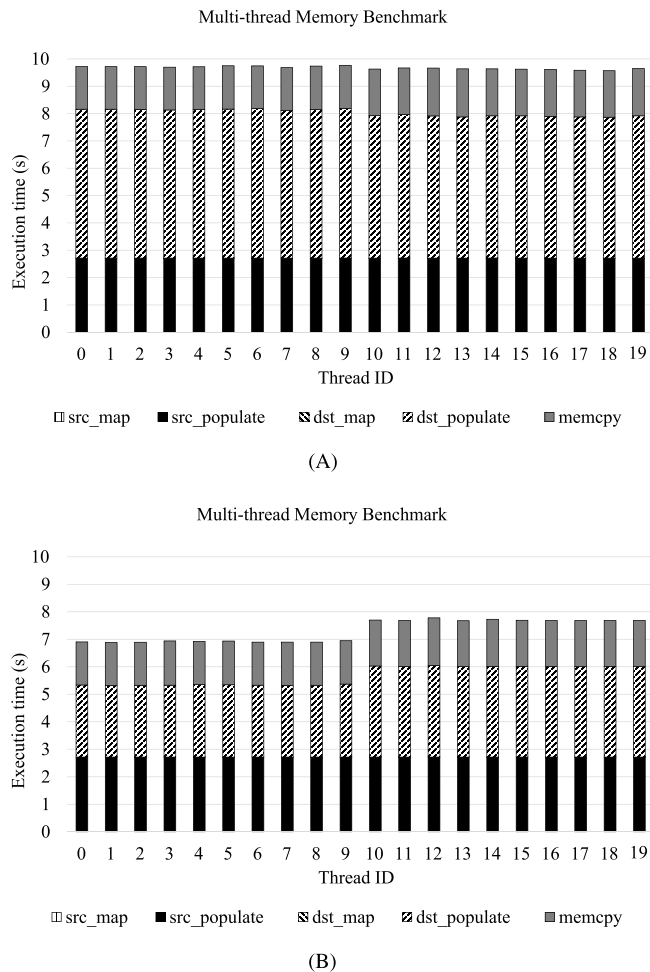


FIGURE 7 Results with multi-socket NUMA system: (A) vanilla Ubuntu guest and (B) optimized guest.



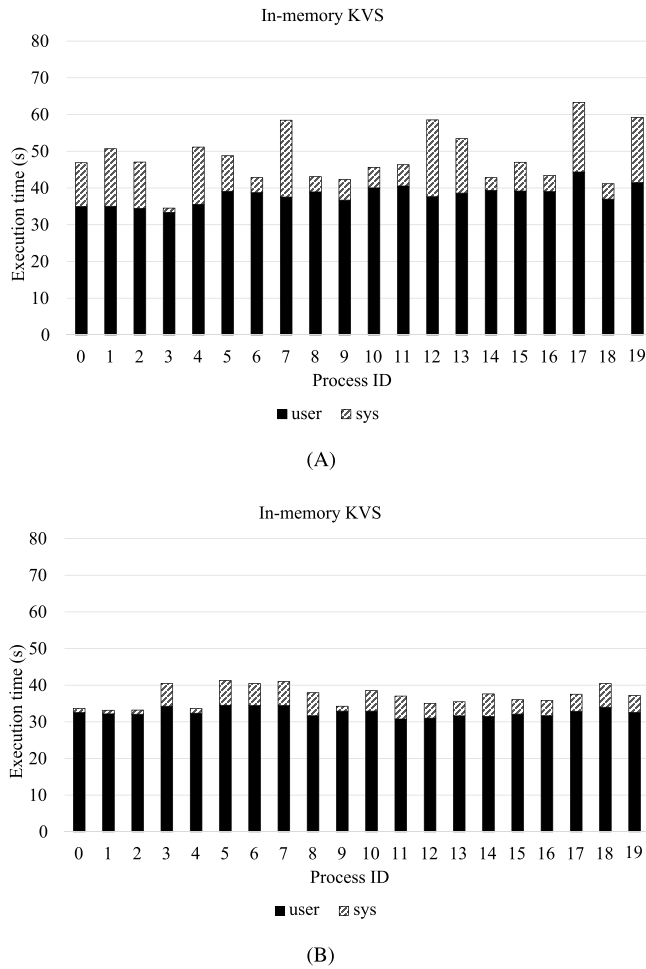


FIGURE 8 In-memory KVS benchmark results: (A) GiantVM and (B) proposed architecture.

compared with conventional spinlock, we implement two simple kernel-level benchmarks.

The first benchmark generates multiple kernel threads, and each thread attempts to increase a global counter using either spinlock or CC-Synch for synchronization. Figure 9A shows the performance of the global counter benchmark with the two synchronization techniques. The x axis represents the number of kernel threads, and the y axis represents the average execution time. As the number of kernel threads increases, the performance gap between spinlock and CC-Synch increases rapidly. CC-Synch improves the performance by a factor of 5.5 compared with spinlock for 32 kernel threads.

The second benchmark also spawns kernel threads, which simultaneously update the data structure of a shared linked list. Like the first benchmark, it uses either spinlock or CC-Synch for synchronization. Figure 9B compares the performance of the shared list update for the two synchronization techniques, where the y axis represents the average execution time on a logarithmic scale. As the system scale increases, the relative performance of

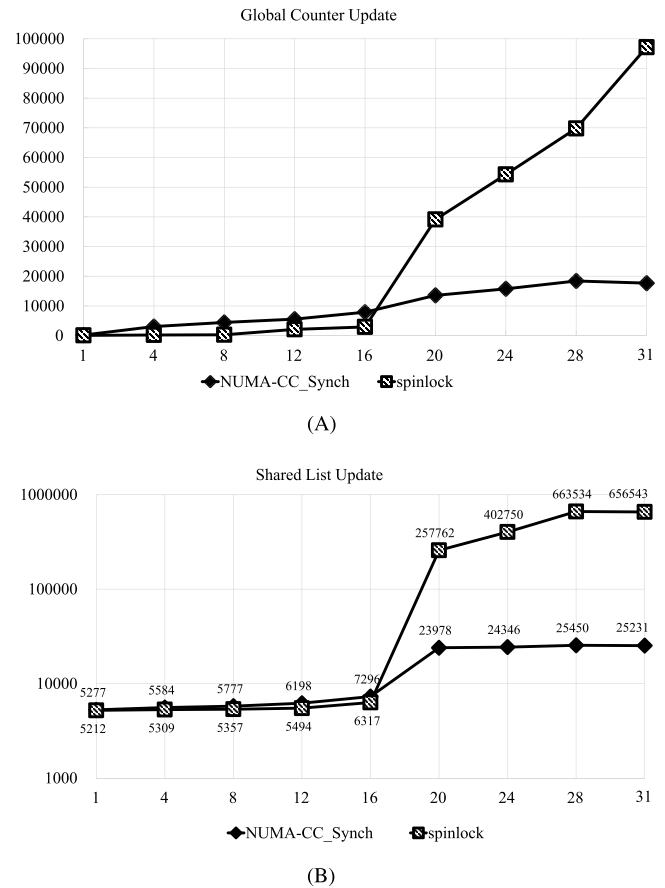


FIGURE 9 Performance improvement with DSM-friendly CC-Synch over spinlock synchronization: (A) global counter update and (B) shared list update.

CC-Synch increases impressively over spinlock. Eventually, CC-Synch outperforms spinlock by a factor of 26 for 32 kernel threads.

## 5.5 | RDMA optimization

To highlight the impact of RDMA optimization on memory accesses in DSM, we measure the average latency of associated function calls in the hypervisor. We evaluate `tdp_page_fault()`, which is called during guest page fault handling when an EPT violation occurs, and `krdma_receive()`, which is an RDMA receive function called inside `tdp_page_fault()`. The workloads and experimental environments used are the same as in the previous experiment reported in Section 5.2.

Figure 10 compares the results of the existing GiantVM and proposed architecture with RDMA optimization. The proposed architecture improves the performance by approximately 67% for `krdma_receive()` and 17% for `tdp_page_fault()` compared with GiantVM. GiantVM uses a poll/sleep mechanism for both

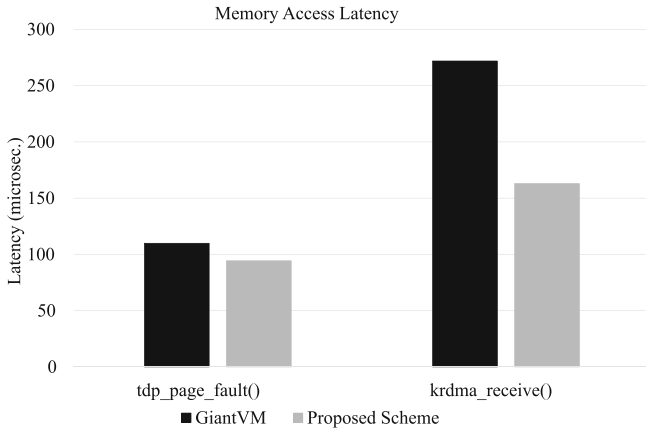


FIGURE 10 Performance of RDMA optimization.

the requester and responder during RDMA communication, whereas the proposed architecture applies selective busy wait to reduce the latency with marginal CPU overheads. The average latency of a callee, `krdma_receive()`, is longer than that of a caller, `tdp_page_faults()`. This is because `tdp_page_fault()` selectively calls `krdma_receive()` only when it requires remote memory access, and its latency becomes negligible for local accesses.

## 5.6 | Results for standard benchmarks

To verify the performance of the proposed scheme under reliable standard benchmarks, we evaluate our architecture using the PARSEC parallel benchmark suite [4] and NAS Parallel Benchmarks (NPB) [5]. We choose three workloads from PARSEC, namely, `blackscholes`, `fluidanimate`, and `ferret`, and three workloads from NPB, namely, `CG`, `FT`, and `MG`. All the workloads are compiled without modification. For PARSEC, `pthread` is used for parallel execution and native datasets are used as inputs. NPB workloads are compiled with OpenMP and a class B, except for `CG` with a class C. Figure 11 shows execution time of the benchmarks with GiantVM and the proposed architecture. Due to wide variations in execution times, the results of the proposed architecture are normalized to those of GiantVM. The proposed architecture offers superior performance from 9% to 48% and by approximately 19% on average compared with GiantVM. The results represent end-to-end performance, including disk input/output operations as well as initialization and wrap-up phases, and the proportion of memory operations is smaller than that in synthetic workloads. Accordingly, the amount of improvement becomes relatively small compared with that obtained from the memory-intensive workload reported in Section 5.2. However, we verify that

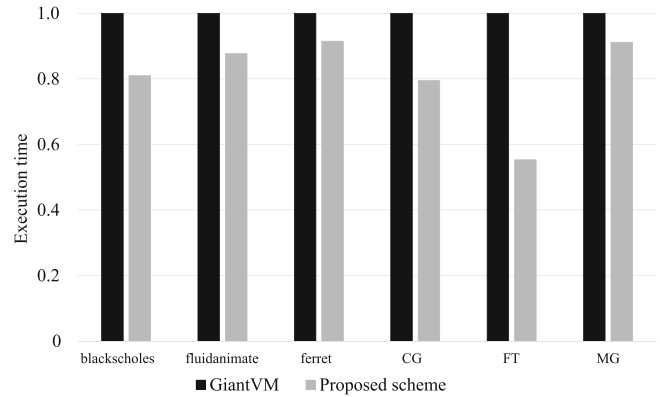


FIGURE 11 Performance for standard benchmarks.

memory access performance with each of the benchmark is enhanced owing to the optimized guest kernel and memory management of DSM.

## 6 | RELATED WORK

DSM integrates physically divided memory devices into a single address space and has long been studied at both the software and hardware levels [6-8]. IVY [7] is a software-based DSM technology that uses a dynamic distributed manager for memory coherence, thereby enabling the page owner to be located with only one messaging process. However, it has a problem with messaging overheads owing to the sequential consistency model. TreadMarks [6] is another software-based DSM technology that follows a lazy release constant model to reduce communication overheads. Nevertheless, the limitation of most conventional DSM technologies is that the application of a strict memory consistency model to ensure programming interface compatibility leads to poor performance, whereas the application of a relaxed consistency model to ensure high performance impairs programmability. Modern DSM technologies attempt to achieve high performance by using emerging hardware technologies such as RDMA [9] or persistent memory [10]. However, these technologies still fail to ensure the complete compatibility of the programming interface and have been evaluated under limited workloads.

Open-source single system image technologies such as Kerrighed [11] and openMOSIX [12] have been introduced. These technologies support load balancing through the migration of processes or threads between physically separated nodes. However, they are implemented at the level of the host OS running on bare-metal hardware and have limitations on the choice of OSs. Moreover, their development activities stopped a long time ago, hindering their use in modern OSs.

Resource disaggregation technologies that enable more flexible and efficient use and management of hardware resources have been recently introduced with the advent of fast interconnect and network fabric. EMP [13] is a hypervisor-based memory disaggregation technology that can access memory in a remote node connected via InfiniBand using RDMA. It guarantees the transparency of existing user applications and guest VMs and supports elastic blocks, thereby enabling dynamic adjustment of the memory block size to be fitted according to locality. Remote regions [14] is another disaggregated memory system technology that enables user-friendly file abstraction instead of difficult-to-use RDMA verbs. LegoOS [15] is a new OS technology for managing disaggregated systems, including major hardware resources such as CPU, memory, and storage. It achieves the flexibility of management while showing comparable performance to existing monolithic kernels.

Single-OS NUMA abstraction architectures integrate physically disaggregated nodes into one VM through a hypervisor. Commercial products include ScaleMP [16] and TidalScale Software-Defined Server. TidalScale products [17] can migrate vCPUs between remote nodes and make decisions of migrating vCPUs and memory pages for better performance based on machine learning techniques. However, the internal details of these commercial products are not open to the public. vNUMA [18] and GiantVM [2] are available as open-source technologies. vNUMA is implemented as a type-1 hypervisor running on bare-metal hardware without a host OS, but it can only support Itanium processors. GiantVM is implemented as a type-2 hypervisor running on the host OS and leverages the IVY-based DSM technology for discrete memory devices to be integrated and shared by users. However, it suffers from performance degradation owing to the lack of coordination between guest OS and hypervisor, especially when accessing memory.

## 7 | CONCLUSIONS

We propose a scalable hypervisor-based single-OS NUMA abstraction architecture with discrete hardware resources. Both the hypervisor and guest kernel are optimized for virtual NUMA environments to achieve performance improvements focusing on memory.

We evaluate the proposed architecture and its techniques, demonstrating performance improvement by factors of 4.74 on average for memory-intensive multi-thread workloads and by 19% on average with standard parallel benchmarks compared with a state-of-the-art open-source technology. In future work, we will perform

in-depth performance analyses of our architecture under various workloads and for larger-scale systems. In addition, we will attempt to apply a DSM-friendly CC-Synch mechanism to various synchronization points in a guest kernel to evaluate its performance in practice.

## CONFLICT OF INTEREST STATEMENT

The authors declare that there are no conflicts of interest.

## ORCID

Baik Song An  <https://orcid.org/0000-0002-4039-5380>

## REFERENCES

1. R. Jain, S. Cheng, V. Kalagi, V. Sanghavi, S. Kaul, M. Arunachalam, K. Maeng, A. Jog, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, *Optimizing CPU performance for recommendation systems at-scale*, (Proceedings of the 50th Annual International Symposium on Computer Architecture, Orlando, FL, USA), 2023, pp. 1–15.
2. J. Zhang, Z. Ding, Y. Chen, X. Jia, B. Yu, Z. Qi, and H. Guan, *GiantVM: a type-II hypervisor implementing many-to-one virtualization*, (Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Association for Computing Machinery, Lausanne, Switzerland), 2020, pp. 30–44.
3. P. Fatourou and N. D. Kallimanis, *Revisiting the combining synchronization technique*, (Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, New Orleans, LA, USA) 2012, pp. 257–266.
4. C. Bienia, *Benchmarking modern multiprocessors*, Ph.D. Thesis, Princeton University, 2011.
5. NASA, NAS parallel benchmarks. <https://www.nas.nasa.gov/software/npb.html>. Accessed: 2023-08-15.
6. C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, *Treadmarks: shared memory computing on networks of workstations*, *Computer* **29** (1996), no. 2, 18–28.
7. K. Li and P. Hudak, *Memory coherence in shared virtual memory systems*, *ACM Trans. Comput. Syst.* **7** (1989), no. 4, 321–359.
8. Y. Zhou, L. Iftode, J. P. Sing, K. Li, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood, *Relaxed consistency and coherence granularity in DSM systems: a performance evaluation*, (Proceedings of the sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Association for Computing Machinery, New York, NY, USA), 1997, pp. 193–205.
9. J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, *Latency-tolerant software distributed shared memory*, (2015 Usenix Annual Technical Conference (Usenix Atc 15), Santa Clara, CA, USA), 2015, pp. 291–305.
10. Y. Shan, S.-Y. Tsai, and Y. Zhang, *Distributed shared persistent memory*, (Proceedings of the 2017 Symposium on Cloud Computing, Association for Computing Machinery, Santa Clara CA, USA), 2017, pp. 323–337.
11. C. Morin, R. Lottiaux, G. Vallée, P. Gallard, D. Margery, J.-Y. Berthou, and I. D. Scherson, *Kerrighed and data parallelism: cluster computing on single system image operating systems*,

- (IEEE International Conference on Cluster Computing (CLUSTER 2004), San Diego, CA, USA), 2004, pp. 277–286.
12. M. Bar, Openmosix project. <http://openmosix.org>. Accessed: 2022-02-14.
  13. K. Koh, K. Kim, S. Jeon, and J. Huh, *Disaggregated cloud memory with elastic block management*, IEEE Trans. Comput. **68** (2019), no. 1, 39–52.
  14. M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novaković, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei, *Remote regions: a simple abstraction for remote memory*, (2018 Usenix Annual Technical Conference (Usenix Atc 18), Boston, MA, USA), 2018, pp. 775–787.
  15. Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, *LegoOS: a disseminated, distributed OS for hardware resource disaggregation*, (13th Usenix Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, USA), 2018, pp. 69–87.
  16. ScaleMP, ScaleMP, Inc. <http://www.scalemp.com>. Accessed: 2021-05-31.
  17. TidalScale, Tidalscale—scalable solutions for in-memory computing. <http://tidalscale.com>. Accessed: 2022-02-14.
  18. M. Chapman and G. Heiser, *vNUMA: a virtual shared-memory multiprocessor*, (2009 Usenix Annual Technical Conference (Usenix Atc 09), San Diego, CA, USA), 2009, pp. 349–362.

## AUTHOR BIOGRAPHIES



**Baik Song An** received the BS and MS degrees in computer science from Seoul National University and the PhD degree in computer science from Texas A&M University. He is currently a principal researcher of the Electronics and Telecommunications Research Institute. His research interests include system software, computer architectures, and large-scale machine learning.



**Myung Hoon Cha** received the BS and MS degrees in computer science from Kyungpook National University, Daegu, Republic of Korea, in 1995 and 1997, respectively. Since 2005, he has been with the Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea, where he has worked on developing the distributed parallel file system,

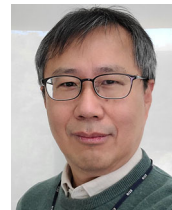
exascale file system, and memory-centric operating system. He is currently a principal researcher. His main research interests include artificial intelligence computing systems, operating systems, and distributed systems.



**Sang-Min Lee** received the PhD degree in computer science from the Korea Advanced Institute of Science and Technology, Republic of Korea, in 2019. Since 2002, he has been with the Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea, where he has worked on developing the distributed parallel file system, exascale file system, and memory-centric operating system. He is currently a principal researcher.



**Won Hyuk Yang** received the BS degree in computer science engineering from Konkuk University, Seoul, Republic of Korea, in 2020. Since 2022, he has been studying computer science engineering at POSTECH, Pohang, Republic of Korea. His main research interests include computer architectures and memory systems.



**Hong Yeon Kim** received the PhD degree in computer engineering from Inha University, Incheon, Republic of Korea, in 1999. Since 1999, he joined the Electronics and Telecommunications Research Institute and is currently working as a senior researcher. His main research interests include operating systems, distributed computing, and artificial intelligence.

**How to cite this article:** B. S. An, M. H. Cha, S.-M. Lee, W. H. Yang, and H. Y. Kim, *Providing scalable single-operating-system NUMA abstraction of physically discrete resources*, ETRI Journal **46** (2024), 501–512. DOI [10.4218/etrij.2023-0056](https://doi.org/10.4218/etrij.2023-0056).