Cooperative Memory Expansion via OS Kernel Support for Networked Computing Systems

Pisacha Srinuan, *Member, IEEE*, Xu Yuan, *Member, IEEE*, and Nian-Feng Tzeng, *Fellow, IEEE*

Abstract—The growing popularity of in-memory computing for bigdata analytics often causes performance bottlenecks to memory subsystem resided in operating systems (OS). This article purposes cooperative memory expansion (COMEX), an OS kernel extension. COMEX establishes a stable pool of memory collectively across nodes in a cluster and enhances OS's memory subsystem for memory aggregation from connected machines by allowing process's page table to track remote memory page frames without programmer effort or modifications to application codes. COMEX employs Remote Direct Memory Access (RDMA) for low-latency data transfer with destination kernel bypassed and does not rely on an old design of the I/O block subsystem usually adopted by all known remote paging. COMEX fits soundly in the emerging system design approach of resource disaggregation which breaks hard walls between server-centric machines into a new design paradigm of separated resource pools. The new architecture facilitates both system scaling-up and scaling-out, also eliminates imbalance resources existing in datacenters. We have implemented COMEX based on Linux kernel 3.10.87 and deployed on our 32 networked servers. Performance evaluation results under ten applications from two benchmark suites reveal the speedup of up to 170 times when application execution footprints are 10 times larger than available system memory.

Index Terms—I/O block devices, memory management, networked computer systems, operating systems (OS), page tables, remote direct memory access (RDMA)

1 Introduction

Performance of a computer system is critically affected by the system's bottleneck during job execution. Given the data-intensive compute paradigm has gained its growing popularity lately for bigdata processing, the memory subsystem of a legacy OS (operating system) kernel often exhibits as the system's bottleneck since its physical memory size is usually dwarfed by the bigdata requirements. As a result, the memory subsystem relies on slow block device storage to hold most data during its processing. In addition, temporary swap space created on the storage device by OS for staging excessive pages evicted from main memory [4], [5] tends to become an execution performance bottleneck. Solutions have been pursued for addressing the performance bottleneck of bigdata processing due mainly to extensive accesses to slow I/O block device storage. Noticeably, they widely adopt in-memory computing by keeping all data in system main memory (DRAM) for speedy processing.

Since it can drastically boost bigdata application performance, in-memory computing realized by huge DRAM in a machine was considered [1], [2], [3] in support of speedily executing applications with large execution footprints. However, such a machine is cost-ineffective potentially as its memory size can be way excessive for some applications but be still insufficient for bigdata applications. Hence, it is challenging for a computing system to achieve effective in-memory computing across a range of applications, urgently calling for feasible commodity solutions. Dynamic memory allocation on-demand for bigdata applications executed on a server cluster thus has been explored [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], utilizing physical memory of other connected servers to stage excessive data of local applications during execution.

This article addresses OS kernel support for establishing immense memory collectively across nodes of a networked computing system to hold excessive cold memory pages evicted from the main memory of a host node when executing applications with huge execution footprints well beyond what the host's main memory can hold. Realizing cooperative memory expansion (COMEX), our proposed design permits applications in any node to expand its address space on-demand onto remote DRAM of other connected nodes, utilizing OS process page tables with appropriate extensions as detailed in Section 3 and shown in Fig. 2. Applications with large working sets under COMEX accelerate their execution runs transparently without code modifications, as a result of avoiding much slower secondary storage for swap space. COMEX is featured with locality-aware memory paging which keeps continuous swap-out pages of a given process in the same remote node at all possible to accelerate future page-fault handling due to effective page prefetching. This feature exploits the low-level Linux kernel data structure of reverse mapping uniquely, to ensure high performance even under multi-tasking when multiple concurrent applications run simultaneously on separate compute nodes,

• The authors are with the School of Computing and Informatics, University of Louisiana, Lafayette, LA 70504. E-mail: {pisacha.srinuan1, xu.yuan, tzeng}@louisiana.edu.

Manuscript received 1 Nov. 2019; revised 8 Feb. 2020; accepted 25 May 2020. Date of publication 3 June 2020; date of current version 17 June 2020. (Corresponding author: P. Srinuan.) Recommended for acceptance by W. Yu. Digital Object Identifier no. 10.1109/TPDS.2020.2999507

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

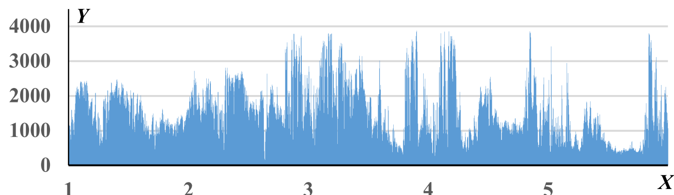

Fig. 1. Number of servers with memory utilization below 90 percent for the 4000-server Alibaba cluster over five days (along the X-axis).

as illustrated in Fig. 16 (on pp. 14). In contrast, earlier demand paging methods realized in the user-level often rely on I/O block devices [24], [27], [28], [29], [30], [31], [35], rather than working on kernel data structures (e.g., page tables and reverse mapping) for fine-grained remote memory management to raise DRAM utilization.

COMEX is attractive with growing adoption of Remote Direct Memory Access (RDMA) [7], [8], [11], [53] by operating systems (e.g., Linux and Windows) and by NICs (network interface cards) coupled with fast switches (like 10 GbE [50], [51] and InfiniBand [52]), making it possible to achieve low latency data transfer across nodes of a networked computing system formed using commodity hardware and software [48], [53]. RDMA represents the core enabler for resource disaggregation across a pool of resources resided in separate servers/blades [7], [8], [46], because of its low latency and high bandwidth [47]. Datacenters have deployed RDMA over large-scale Ethernet-based network fabrics using the RoCEv2 protocol [48], [53].

COMEX aims at RDMA-enabled networked computing systems, including those in general purpose server-centric datacenters, where each machine is equipped with specific amounts of resources (CPU, memory, storage, and accelerators). Memory imbalance exists in such a system routinely when executing diverse applications concurrently on its different nodes [7], [8], [24], [46]. Real-world data of memory utilization for a large-scale Alibaba cluster comprising 4,000 servers gathered and published in 2019 [33], [36] is demonstrated in Fig. 1, where the Y-axis denotes the number of servers with memory utilization below 90 percent. The result of this production cluster over five days reveals that on an average, 38 percent of cluster servers have memory utilization less than 90 percent, with their mean memory utilization (of those 38 percent servers) equal to only 79 percent. It is evident that, even with modern job schedulers and dispatchers, memory imbalance and underutilization usually exist in computing systems, with unused remote memory available for exploitation via COMEX transparently. In fact, COMEX fits soundly in the emerging system design approach of resource disaggregation [6], [7], [8] by building pools of separate resource types (e.g., CPU, memory, storage, communication, etc. [38]) that are allocated on-demand to applications dynamically. Resource disaggregation facilitates flexible system scaling and cost-effective hardware upgrading. COMEX facilitates aggregate memory resources for effective memory utilization during job execution, avoiding memory underutilization or overprovision in individual sever nodes.

This work builds a swap-based remote memory system, COMEX, that is transparent to applications by an OS extension to manage the swap space resided on remote memory. To guarantee high performance, RDMA network is used as the network backend. This enables low-latency data transfer, which is critical for a swap backend. COMEX differs starkly from other approaches considered previously for accelerating local application execution [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40] via utilizing idle remote memory, in multiple key aspects. First, COMEX is completely transparent to applications and requires no programmer effort nor modifications to application source codes. Second, COMEX is compatible with most commodity platforms and devices (with RDM support for low-latency networking [7], [8], [46], [47], [48]). Third, COMEX does not rely on any new networking hardware or protocol stack for resource aggregation, as does the case earlier [37], [38], [39], [40]. Fourth, COMEX manages its memory space and handles data transfer effectively, without utilizing such commonly adopted mechanisms as swapping and I/O block subsystems found in most remote memory paging studies previously [24], [25], [26], [27], [28], [29], [30], [31],

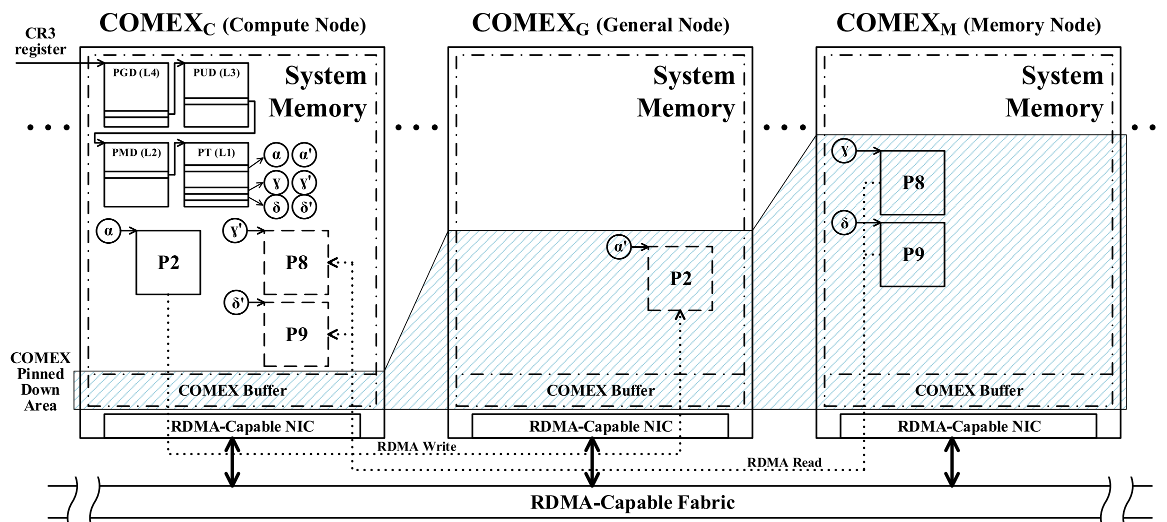

Fig. 2. Overall memory management and page data transfer under COMEX.

[32], [33], [34], [35]. This is because those two legacy subsystems were designed for backing storage made of slow hard disks [29], [41], [42], [43], [4], [45], [67], [68], [69]. Fifth, COMEX differs from virtualization that partitions physical hardware resources to logical compute units, called virtual machines (VMs) [9], [10], [37], [38], [39], [40] for coarse-grained VM-level resource sharing; instead, COMEX extends process page tables for fine-grained page-level memory aggregation and sharing.

We have implemented COMEX based on Linux kernel 3.10.87 and deployed on a testbed system comprising 32 networked Dell servers. Performance evaluation results under ten applications from two benchmark suites reveal that COMEX exhibits higher execution speedups when the ratio of the application execution footprint size to the local memory size rises, with the speedup to reach 170$\times$ when the ratio equals 10.

2 Pertinent Background and Prior Work

Virtual Memory. Virtual memory is adopted widely by the modern OS [4], [5]. It allows a process to have larger (virtual) address space than real physical memory existing in the system. OS maintains a page table for each process to keep the mappings of its virtual memory pages (of say, 4 KB or 2 MB each in Linux) to physical memory page frames, with a Page Table Entry (PTE) denoting one virtual memory page. For every virtual memory page that has not yet allocated a physical memory page frame or has been "swapped out" from its earlier allocated page frame to disk storage, its associated PTE records such information accordingly. A long latency occurs when a swapped out (or an unallocated) page is brought back to (or into) a DRAM page frame by OS upon a page fault, causing a marked performance penalty, which grows as the swapping activities intensify for bigdata processing.

Remote Direct Memory Access (RDMA). RDMA allows hosts to directly read/write data from/to physical memory in other connected machines [49]. It permits OS kernel bypassing at the destination machine to lower remote access time overhead and energy consumption, by means of the RDMA-enabled network interface card (RDMA-enabled NIC). Such a widely affordable NIC caches virtual memory address translation in its on-card memory and requires to pin the designated physical memory area down through "RDMA Memory Region registration" [11], [54]. In fact, RDMA is available for Ethernet (i.e., RDMA over Converged Ethernet or RoCE [55], which is commonly adopted for networking in datacenters [48] and enterprise server clusters), besides InfiniBand. RDMA is popularly adopted in datacenters [48], [53] because of its support from abundant hardware vendors [50], [51], [52] and various operating systems (including Linux). It facilitates computer system resource disaggregation [8], [9], [46].

2.1 Remote Memory

Design and implementation effort on remote memory has aimed at execution performance improvement for years [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35]. It can be categorized into two groups, as reviewed separately in the following subsections.

2.1.1 Programming Remote Memory

Programming techniques have been introduced for overcoming the memory limitation within a single machine. They employ a privilege for accesses to the memory area outside a local machine, known as remote memory. Some of them rely on the native network stack of TCP/IP [16], [17], [18], [22], [25], [26], whereas others either exploit such features as RDMA [11], [12], [13], [14], [15], [19], [20], [21] or adopt new network protocols [53]. In particular, RDMA and its variants [11], [12], [13], [14], [15] expose remote memory to users directly via a collection of programming APIs, allowing users to manipulate remote memory by means of considerable application modifications. However, users have to deal with an RDMA low-level abstraction [12], [13], [46], often involving steep learning and high programming skills in order to unleash the full potential of RDMA. In addition, RDMA library is available only in a few limited programming languages.

Key-value storage [16], [17], [18], [19], [20], [21], [22] and RDMA key-value storage [19], [20], [21] offer simpler API abstractions, allocating DRAM chunks from connected machines to form a large global pool of DRAM memory for storing application data. They permit applications to read their data directly from a DRAM memory pool, avoiding a huge latency from slow storage. However, applications may not follow key-value characteristics, making them difficult to adopt key-value storage [23].

This group of techniques usually has high flexibility for programmers to decide which data to be kept at remote memory for good memory usage and performance. However, they are not readily applicable to applications whose source codes cannot be modified or re-written easily.

2.1.2 Remote Memory Paging

Remote memory paging studies and implementations mostly employ traditional memory paging, swapping mechanisms, and I/O block subsystems found in operating systems [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35]. Swapping moves (1) cold memory pages from local memory to 'swap-out partitions' in storage and (2) miss pages from 'swap-out partitions' back to local memory, in order to expand address space beyond system's actual physical memory. However, it relies on asynchronous (for swapping pages out) and read-ahead (for swapping miss pages in) procedures to hide lengthy disk seek and write/read times [12], [13], [44], [45], [67], [68], [69]. Under memory paging to/from remote memory (of a connected machine), however, care must be taken to fully benefit from much faster DRAM for a low latency [41], [42], [43], [45].

A recent attempt at remote paging, called INFINISWAP [24], is based on swapping and I/O block subsystems. INFINISWAP mounts its block device to the system as a swap partition to cache cold pages at remote memory areas. INFINISWAP works at the "SlabSize" granularity, e.g., 1 GB per slab. When the number of references to a 1 GB (i.e., the slab size) address range reaches a set threshold, the INFINISWAP daemon allocates one slab of memory pages at a remote machine to hold swapped out pages. Coarse granularity is chosen by INFINISWAP to contain the tracking overhead of memory accesses, but it naturally leads to remarkable underutilization of allocated slab memory. In

addition, the I/O block subsystem adopted for page transfer has low efficiency [10], [41], [42], [43].

INFINISWAP relies on a separate mapping table in each compute node (being an extra data structure) to manage removed pages read/write activities for its decision on remote slab allocation, choosing a very large slab size (of 1 GB [24]) to contain overhead involved in managing extra data structures. Unfortunately, such a large granularity leads to low remote storage utilization, since (1) a slab is allocated at one remote node, as soon as page swap-in/swap-out activities (within the 1GB address range) per second exceed a specified threshold (HotSlab), irrespective of how many distinct pages are involved, and (2) a slab, once allocated, is not deallocated until its remote host node experiences local memory pressure, often seen a very small fraction of its page frames actually taken up by swap-out pages. In contrast, our COMEX makes use of page tables existing in any OS (including Linux) which supports virtual memory, to track all pages swapped out onto remote nodes, and to work at the fine grain page size (of 4 KB). COMEX achieves far better remote memory utilization than INFINISWAP without resorting to any extra data structure for remote memory management, resulting from (1) fine-grained page allocation to lower fragmentation and (2) periodical page frame reclaiming at each involved node to free up unused page frames proactively (via a light-weight RDMA verb, to be detailed in Section 4.5).

2.2 Resource Disaggregation and Virtual Machines

Growing adoption to the resource disaggregation architecture prompts active studies lately on *full disaggregation*, by which components of a computer system are redesigned and redeveloped from scratch. New design and/or prototype computer hardware and software in support of full resource disaggregation have been revealed [37], [38], [39], [40], and they usually require huge implementation effort and capital investment while discarding all otherwise functional computer hardware and software components. Full resource disaggregation may be attractive for new datacenter construction by adopting clean-slate computer hardware and software components to build compute nodes specific for the datacenter needs and to permit future node upgrades via replacing target node components. However, its suitability for general computer system installations is questionable.

COMEX takes a different approach known as *partial resource disaggregation*, aiming at existing server clusters without doing away with any hardware component nor developing the whole system software stack afresh. Specifically, COMEX extends the OS kernel to let its virtual memory subsystem track swap-out pages staged in remote physical memory via page tables (instead of adding extra data structures for tracking remote memory use like INFINISWAP [24]). We have implemented COMEX on Linux kernel 3.10.87 and made it easy for patching COMEX-specific software modules into mainline Linux kernels. Partial resource disaggregation under COMEX permits a server cluster to add compute nodes or memory nodes' heterogeneously in response to its growth needs, where a compute (or memory) node has powerful computation cores (or a limited computing ability) and a small amount of memory (or large memory), with the same COMEX-provisioned OS running on every cluster node.

Note that COMEX offers memory resource sharing on demand among compute nodes in a server cluster, it differs from resource virtualization, which share all server resources (compute, memory, storage, etc.) dynamically by adjusting the container size for good utilization [9], [10], [37], [38], [39], [40]. Instead of sharing physical hardware resources in the form of virtual machines (VMs), COMEX enables fine-grained page-level memory resource sharing across a stable pool of DRAM. On the other hand, virtualization offers coarse-grained resource sharing at the logical unit (virtual machine) level.

2.3 Alternatives to COMEX

Three alternatives to the COMEX design are stated in sequence. The first alternative can exploit native OS's swap mechanism by mounting the remote memory block device as a swap partition for a system for holding evicted cold pages. This falls into remote memory paging as reviewed in Section 2.1.2 [24], [25], [26], [27], [28], [29], [30], [31], [32]. It can be achieved through either the traditional software network stack (TCP/IP) alone [25], [26], [27], [28], [29], [34], [35] or together with hardware (e.g., RDMA) support [24], [27], [30], [31], [32], without modification to the OS kernel. However, this method usually exhibits poor performance because of its two shortcomings: (1) high time overhead and (2) reclaiming one single page frame at a time asynchronously [41], [43]. In fact, Linux foundation plans to improve those shortcomings [44], [45].

Another alternative may hold swapped out pages in remote memory, which is allocated and managed by extra data structures in coarse granularity (of 1 GB per slab, as adopted by INFINISWAP [24]). It is subject to the same shortcomings of swapping and I/O block subsystems stated in the previous alternative.

The third design alternative lets the OS kernel employ dedicated remote memory page-mapping, which is additional to (and separate from) existing process page tables. However, this alternative requires excessive effort and high space overhead, clearly inferior to COMEX that expands existing page tables to keep mapping information of remote memory page frames.

3 COMEX Design Consideration

The process page table is a key software structure to enable virtual memory for efficient physical memory management and for secure memory sharing and isolation, allowing a process to have address space larger than the physical memory of a system that runs the process [4], [5]. Due to emerging data-intensive applications with huge execution footprints of late, the proposed COMEX becomes indispensable to accelerate execution of such applications. COMEX fits soundly in the system design approach of resource disaggregation, which facilitates easy system scaling and component upgrading. It permits flexible memory resource aggregation on-demand during job execution and lets commodity hardware and software components be added to existing server clusters (or upgrade existing ones individually) for cost-effective scaling, in support of speedy execution of applications with big execution footprints.

Design Overview. Aligning with the resource disaggregation design, COMEX is built on a connected server cluster,

involving compute nodes and memory nodes. A compute node has high computation power (possibly from multiple sockets), responsible for application execution with swap-out pages staged at remote memory nodes. On the other hand, a memory node has abundant physical memory and contributes a big portion of its memory to the shared COMEX memory pool. A general node may also exist in a COMEX cluster, with modest computing power and sizable memory. Such a node may execute applications and also contribute a small fraction of its memory to the shared memory pool. Each node (be a compute, memory, or general one) in the server cluster runs exactly the same OS, with its kernel extended with COMEX-specific software modules.

COMEX is realized by page table extension for tracking remote physical memory (in other connected nodes of a server cluster) allocated to an application on-demand during its execution. A connected node is referred to as a "staged node", since it is for staging the swap-out pages of a process executed on another node. COMEX aims at transparent and efficient memory resource aggregation in server clusters connected by RDMA-capable networking gear without any user effort, application modification, or dedicated hardware support. An example COMEX configuration is illustrated in Fig. 2, where COMEX harbors three evicted pages (P2, P8, and P9) of a process run on Node C (denoted by $COMEX_C$) as the host node. Page P2 in $COMEX_C$ is written transparently (without invoking the destination OS kernel) over the established RDMA connection to $COMEX_G$ in Page Frame P2 of its pinned down zone upon evicting P2 from $COMEX_C$. Similarly, Pages P8 and P9 resided in $COMEX_M$ can be transferred over the established RDMA connection directly back to their corresponding Page frames in $COMEX_C$ upon an execution page fault (that triggers an RDMA Readback). This transfer does not involve the OS kernel or CPU of $COMEX_M$, and it is much faster than conventional page fault handling, by which a swapped out page (say, P8) would have to be brought into main memory of Node C from a swap partition in secondary storage.

COMEX lets its related PTEs (page table entries) point to COMEX page frames employed for harboring evicted pages in remote nodes. As shown in Fig. 2, three evicted pages (P2, P8, P9) are pointed by three corresponding PTEs of the process executed in $COMEX_C$. After P2 is evicted to $COMEX_G$, its associated PTE is changed from α (a page frame in $COMEX_C$) to α'. Likewise, after P8 and P9 are RDMA-transferred from $COMEX_M$ back to $COMEX_C$, their associated PTEs are changed from γ and δ (two page frames in $COMEX_M$) respectively to γ' and δ' (page frames in $COMEX_C$), as depicted in the figure. Here, a 4-level page table is assumed to map the virtual addresses of an execution process to physical addresses of allocated page frames (for a 64-bit system).

COMEX is subject to multifaceted design choices that dictate its performance and complexity in various degrees, with three major ones being (1) design in the kernel layer versus in the user layer, (2) RDMA connection setups, and (3) COMEX memory management. Details and trade-offs of those three design choices are provided in sequence next.

3.1 Kernel Layer Versus User Layer

In general, remote memory paging may be realized either in the kernel layer alone for high performance or in the user layer (plus some kernel modules) for good portability and flexibility. The kernel layer design usually has lighter overhead than its user layer counterpart, to yield better execution performance.

COMEX is designed in the kernel layer alone, realized by a loadable kernel module along with kernel page table enhancement. A kernel module is offered by OS (Linux) to let users add extra functionalities to the kernel. Modifications and additions for COMEX realization are all confined to the kernel layer of Linux, due to the following two reasons. For the *first reason*, a user layer design for memory aggregation is not transparent to applications and has to add the offered library primitives to the proper locations of an application code. Such a design typically relies on programming libraries, which enable users to exploit remote memory via a malloc-like memory allocation library, DLM [25], [26]. It requires the application's source code be available to make modifications explicitly, but the source code may be unavailable.

The *second reason* results from the need of PTE (page table entry) content changes when tracking remote page frames that hold swap-out pages of a process during its execution. Given that page tables belong to the kernel data structures (accessible only to kernel codes), it is preferred to realize COMEX functions in the kernel layer for direct accesses to PTEs. On the other hand, if COMEX is resided in the user layer, it must employ kernel modules to communicate with kernel data structures, usually suffering from unacceptably high time overhead. This is because every PTE access request from the user layer then has to pass down to the kernel layer with a kernel module, with the requested outcome sent back to the user layer with another kernel module. This way incurs excessive time overhead even with careful communication channel design, based on our studies. For example, the use of netlink socket (a datagram-oriented service provided by kernel for communications to the user space [57], [58]) is seen to cause frequent packet losses and corruption (according to our experiments), since the netlink socket is an unreliable service [59]. Needed retransmission handling and retransmitted data escalate communication traffic to hurt performance greatly. Another communication mechanism between the two layers by means of memory-based file systems [58], [60], was also found to be inefficient. Since a memory-based file system aims to let kernel convey information to the user layer in an asynchronous manner, notification from the user layer back poses serious bottlenecks and poor performance, no matter whether it is via polling (due to massive notification packets, in the order of up to 50 packets per microsecond according to our measurement) or signaling (due to limited capacity per real-time signal of up to 64 bits [60], but COMEX needs at least 64 bits for both local and destination addresses to perform one RDMA operation) at the kernel layer. Hence, COMEX is unsuitable for user layer design, determined to be in the kernel layer alone.

Note that implementing COMEX in the kernel layer calls for patching COMEX-specific software modules and codes in the kernel of a server's OS, but requires no programmer effort or modifications to application source codes executed on the server. We have implemented COMEX on Linux kernel 3.10.87 and made it easy for patching COMEX software into mainline Linux kernels in support of quick and wide deployment.

3.2 RDMA Connection Setups

RDMA connection establishment is expensive, as revealed in prior reports [11], [24], [27], [30], [31], [32]. It involves memory pin-down to prevent allocated RDMA memory regions from being reclaimed by OS so that virtual address translation can be cached in the RDMA-capable NIC for OS-bypassing address translation, avoiding repeated memory region key distributions. Through loss-free communications (achieved by PFC [55], DCQCN [53], or DSCP [56]), COMEX establishes RDMA-connections from every compute node (and every general node as well, if existing) to its connected memory nodes (which contribute most of their physical memory to the COMEX sharing pool) upon its initialization. After established, connections are held durably so that expensive connection re-establishment overhead is not to re-incur later on.

An RDMA connection is established after two steps: (1) creating its control block to include the address and the size of pre-allocated memory space to serve as the RDMA buffer and (2) executing all_init to complete the initialization of all its connection sockets. The time taken to establish an RDMA connection varies widely and can be large (in tens of μs, according to our experimental measurements on our testbed composed of Dell servers under Linux CentOS 7). With its RDMA connections established permanently, COMEX can have the DRAM area of each server that contributes to the shared pool be "pinned down" so that those DRAM pages are "not swappable" in support of fast RDMA reads/writes without extra costs for pinning down individual pages. It was found that a server in our testbed could hold up to 300+ permanent RDMA connections reliably, without depleting excessive DRAM for DRMA buffers.

On-demand connection setups are possible, without holding multiple concurrent connections from each compute node persistently. However, COMEX operates at the fine granularity of 4 KB memory pages and involves very frequent data transfer mainly related to kernel's page frame reclaiming and page fault handling (up to 50 pages per microsecond observed).

INFINISWAP reports its large-slap allocation time of 55 μs, due mainly to RDMA memory registration, whereas we observe 18 μs for individual page allocation of our testbed. However, this lengthy delay could be even higher in real large-scale deployment, with control and payload packets to traverse multi-layer network switches. High overhead for each RDMA connection setup, coupled with frequent transfer of fine-grained 4 KB pages, makes it unsuitable for COMEX to establish RDMA connections on-demand during job execution. This is in contrast to other situations that operate at coarse granularity (like 1 GB slabs in INFINISWAP [24]).

Hence, at COMEX initialization, every COMEX compute node establishes durable RDMA connections with memory nodes to facilitate low-latency page writes (and reads) to (and from) remote page frames resided in those memory nodes during job execution. Similarly, every general node, if existing, also establishes its RDMA connections.

3.3 COMEX Memory Management

COMEX design involves memory management that deals with both local and remote memory while cooperating with the existing OS kernel memory subsystems. Key memory subsystem functions central to COMEX realization are outlined next.

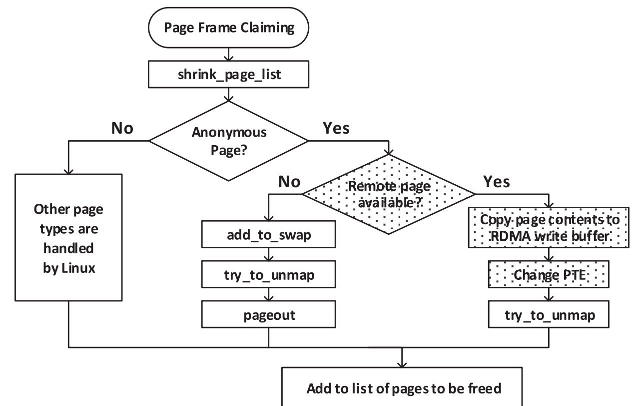

Fig. 3. Flowchart of the kernel page frame reclaiming process, with modified parts shaded.

3.3.1 Page Frame Reclaim

In need of promptly satisfying requests for various memory sizes during job execution, the OS has to maintain an adequate free memory pool proactively. To this end, it relies on page frame reclaiming, that periodically examines the system memory pool availability level, replenishing the pool more aggressively if the availability level is lower. Page frame reclaiming collects rarely used cold memory pages from execution processes, based on the access history [4], [5]. These reclaimable pages are then evicted out, to secondary storage under a typical OS (like Linux) with considerable latencies.

The flowchart of kernel page frame reclaiming is shown in Fig. 3, with its shaded portions denoting such COMEX-specific functions as (1) holding reclaimed pages in the pre-registered RDMA write buffer and (2) transmitting reclaimed pages from the buffer eventually to the pindown memory page frames via RDMA write. This way lets reclaimed page frames be freed immediately after their contents are copied to the COMEX RDMA write buffer. It exhibits much faster page frame reclaiming than under a typical OS, which asynchronously waits for slow confirmation from the I/O block device driver. In addition, COMEX issues an RDMA write to transmit all reclaimed pages destined to the same remote node in one batch.

3.3.2 Pindown Memory Area Management

Each memory node's pre-allocated pindown memory area (contributing to the COMEX global memory pool) has to be managed properly for its efficient utilization in satisfying requests from other compute nodes for holding swap-out pages during job executions on those nodes. To this end, the memory area is managed by a buddy system similar to Linux physical memory management (mm/page_alloc.c), using one list to track groups of contiguous page frames with a given group size. If eleven lists exist in the buddy system, the ith list ($0 \leq i \leq 10$) is to keep track of groups of contiguous page frames, with each group having 2^i contiguous page frames. Upon receiving a memory request from a compute node, the COMEX buddy system allocates one element from the list that matches the requested size for the request. On the other hand, when allocated pages to remote compute nodes become unneeded (after their contents are fetched back to their respective compute nodes where page

faults occur), they are periodically released back to its original buddy system maintained at the memory node, with possible coalescing to get larger contiguous groups.

3.3.3 Remote Page Frame Management

At a compute node, COMEX maintains free memory pools of remote page frames allocated to the compute node, making use of linked lists, with one list entry for a chunk of contiguous page frames (not limited to 2^i page frames per chunk from a memory node, as a number of page frames from an initially allocated 2^i page-frame group may have been used to stage swap-out pages). Multiple swap-out pages of a given process are transferred by one single RDMA write to contiguous page frames in the same staged memory node (which is determined in a way explained next). With swap-out pages staged in contiguous memory page frames, a later page fault (due to a swap-out page) is fulfilled by fetching the target swap-out page plus those in its neighboring multiple page frames (as prefetching, for a total of, say, 16 pages) via one RDMA read. Those fetched page and prefetched neighboring pages are held in one entry of the RDMA read buffer of the compute node (as stated later in this subsection).

The numbers of allocated free pages maintained in local pools (by linked lists) need to be considered carefully, since COMEX prefers to have enough pages ready for page-reclaiming. Insufficient pages may lead to poor COMEX page-reclaiming performance, whereas excessive pages can harm overall memory utilization. With a threshold assigned to each list, COMEX proactively sends out a replenishing request to a corresponding remote node for page refilling if the list sees its remaining page count to drop below the threshold. Naturally, proper thresholds and refilling page counts depend on the message round trip time (*RTT*), aiming to avoid exhausting free pages in any list. Message *RTT* includes network latency, service time, and queueing time, i.e., $RTT = T_{NW} + T_S + T_Q$. Network latency comprises all networking hardware latencies, service time indicates the time a memory node taken to reply one request. The waiting time of a request issued by one compute node of interest to a given memory node, T_Q, depends on the service time, T_S, and the mean number of requests issued from all other compute nodes to the same memory node at the same time. The mean number of requests to the given memory node, $\boldsymbol{R}$, is derived next.

Given a system with N compute nodes and M memory nodes, where each compute node has a probability P to send one request out at a time independently and uniformly to any memory node, we have the probability for a request to arrive at the given memory node of $P' = (\frac{1}{M})\ P$. The binomial probability distribution function of a random variable $\boldsymbol{R}$ that represents the number of requests sent by all $(N-1)$ compute nodes (other than the one of interest) to the given memory node can be expressed by $P\ (\boldsymbol{R} = k) = \binom{N-1}{k}\ P'^k(1-P')^{N-1-k}$. The expected number of requests issued from $(N-1)$ compute nodes to a memory node, $E[\boldsymbol{R}]$, then equals

$$\sum_{k=1}^{N-1} k \cdot P(\boldsymbol{R} = k) = \sum_{k=1}^{N-1} k \binom{N-1}{k} P'^k (1-P')^{N-1-k} = (N-1)\ P' = \frac{N-1}{M}\ P,$$

which indicates that $E[\boldsymbol{R}]$ is dictated by the ratio of N and M, as expected. The queueing time rises with more compute nodes or fewer memory nodes.

Our established testbed with 32 Dell servers is networked by one Mellanox RDMA-enabled switch and thirty-two 10 GbE NICs, as detailed in Section 5.1.2. According to its whitepaper [62], Mellanox MSX1024B-1BFS-RF (RDMA switch) has its lowest latency of 270 *ns*. Mellanox's evaluation [61] reveals that its 10GbE ConnectX-3 RoCE NIC has $\sim$1 μs latency. The end-to-end basic hardware latency of our testbed involves four trips over NICs (two at each server end) and two switch trips for a minimal of 4.54 μs in total. Note that the actual end-to-end latency can be higher, as it includes the times for realizing such network policies as RDMA, VLAN, and PFC in both end NICs. The service time (T_S, which includes kernel interrupt handling and work queuing times) is measured on our testbed via Linux Kernel Timer API [4], and it ranges from 4.0 μs to 6.0 μs. The page reclaiming rate varies, with the measured rate on our testbed to reach up to 50 pages per microsecond. In order to keep up with 50 pages/μs for the duration of *RTT*, a COMEX compute node that issues a memory request has to cache *at least* (50 $\times$ *RTT*) pages locally, with expected *RTT* given by

$$4.54\,\mu s + 6.0\,\mu s + T_Q = 10.54\,\mu s + 6.0\,\mu s \times \left(\frac{N-1}{M}\right) \times P, \tag{1}$$

where the last term indicates T_Q that is due to memory requests sent by all other $(N-1)$ compute nodes at the same time.

COMEX takes advantage of locality-aware page staging at remote memory nodes, by keeping swap-out pages of the same process at the same memory node as best as possible. To this end, COMEX assigns a dynamic threshold to each pool because PIDs change over time. Such a threshold can be derived according to estimated *RTT* to ensure that each buffer has adequate page frames assigned for good performance, while avoiding excessive over-provision to the buffer. An insufficient buffer size leads to low execution performance. Excessive buffer over-provision is especially damaging when the physical memory of compute nodes is scarce, as a result of performance degradation due to heightened memory pressure unnecessarily. Those situations are confirmed by our testbed evaluation results illustrated in Fig. 10 (on pp. 11). All benchmarks shown in the figure are seen to exhibit poor execution performance for small buffer sizes. Their performance levels near peak for the buffer replenishing threshold of some $750 (\approx 50 \times (10.54 + 6.0 \times \frac{9}{12})$ according to Eq. (1) under 12 memory nodes and 10 compute nodes with P approaching 1.0), and they start to drop if the threshold rises beyond 16K. More details of threshold values on execution performance are provided in Section 5.1.3.

3.3.4 Page Frame Discovery

COMEX design aims to have participating nodes work independently and to avoid a single point of failure. Given that multiple memory nodes together contribute to the shared memory pool for staging swap-out pages from all execution

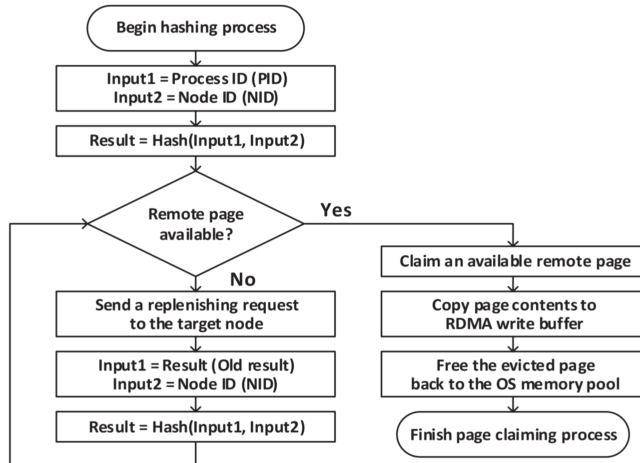

Fig. 4. Page frame discovering and claiming process.

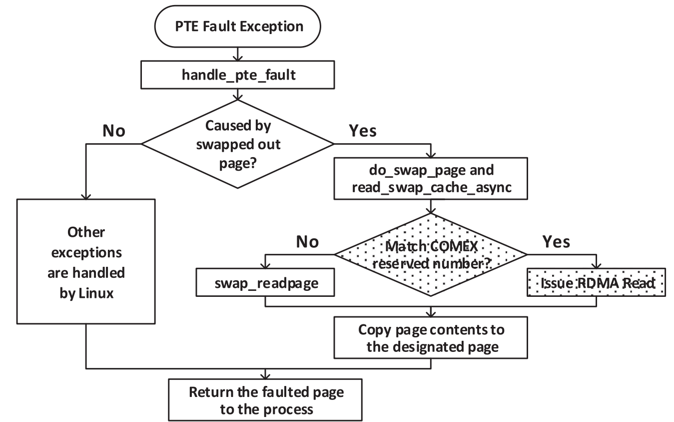

Fig. 5. Kernel page fault exception handler process flowchart, with modified parts shaded.

processes in the system, it is highly desirable to balance traffic and memory utilization across contributing memory nodes to ensure high performance. There are different design options for such load-balancing in a distributed manner, without involving broadcasts, probes to all memory nodes, or a dedicated management node. The *first option* distributes swap-out pages uniformly across memory nodes, irrespective of which processes they are associated with (like [4], [5]). This way achieves load-balancing in page granularity but disregards page locality, which could have boosted performance substantially resulting from fewer RDMA operations and effective prefetching after page faults. In addition, COMEX is designed to support heterogeneous clustered system where nodes can have different memory capacity and general purpose nodes may share only a slim portion of their memory. Thus decentralized uniform page distribution cannot balance the memory in such situations. Unlike INFINISWAP which holds swap-out pages in GB-sized slabs in the same or different memory nodes (due to far larger memory granularity) [24], COMEX favors locality-aware page frame allocation to swap out pages that belong to the same execution process, due to its fine granularity (of 4 KB pages). Such locality-aware page frame allocation is the *second option*.

This second design option discovers available memory page frames by involving the process ID (PID, obtained from the Linux page frame reverse mapping), so that swap-out pages from a given execution process are staged in the same memory node (called its preferred node). In a clustered system, different compute nodes may involve execution processes with an identical PID, making it desirable from the load-balancing standpoint to include the node ID (NID) for deciding page frames. Hence, our design hashes the NID plus PID of a given execution process to identify its preferred memory node in which its swap-out pages are staged, to balance traffic and maintain locality of swap-out pages (to the same remote memory node). If the linked list of a preferred memory node (maintained in the compute node) is exhausted (typically signifying that the memory node no longer has large enough contiguous page frames available in its pindown memory region), the next memory node is identified by rehashing the NID of the preferred memory node plus PID for alternative memory node discovery, as shown in Fig. 4. This way fixes the sequence of memory nodes examined for a process executed on a given compute node, likely to *expose staging locality of swap-out pages and exhibit good load-balancing* during multiple-process execution. As a result, it is preferred over other means for deciding alternative memory nodes, e.g., choosing the next available linked list, the longest linked list in the compute node, etc.

3.3.5 Page Fault Handling and Prefetching

As depicted in Fig. 5, COMEX handles a page fault by not only fetching the faulty page but also prefetching its neighboring pages from the same staged memory node in one RDMA read, with all fetched and prefetched pages held in an RDMA read buffer entry. For high performance, an RDMA read buffer is pre-allocated statically at each compute node during initialization, as stated earlier. The number of read buffer entries is a design parameter. Buffer entries are shared by all RDMA reads, no matter where they are from. This shared buffer design is preferred over its dedicated counterpart (where each staged memory node is provided with a separate buffer slice statically), promising better buffer utilization. If an RDMA read to a buffer with its entries all taken, one suitable buffer entry must be chosen to accommodate those prefetched pages transferred by the RDMA read. Different policies for choosing a replacement entry are possible, including LRU (least recently used), LFU (least frequently used), FIFO, a random choice, etc. The current COMEX prototype adopts the LRU policy.

According to remote page frame management described earlier, the swap-out pages of a process are usually staged in the same memory node, with those pages reclaimed at a time held in consecutive physical page frames. Hence, those pages prefetched in one RDMA read mostly belong to the same process, and they are available to fulfill subsequent page faults until their buffer entry is replaced. Once fulfilling associated page faults, those pages are kept in a separate list, waiting for periodical notifications via RDMA verbs to free up their associated page frames in remote memory nodes. Upon receiving an informed packet, the memory node then releases those corresponding page frames back to its maintained buddy system (as stated earlier under pindown memory area management).

Fig. 6. Linux page table entry (PTE) layout.

4 COMEX IMPLEMENTATION

This section describes COMEX implementation through enhancing functionality of the virtual memory subsystem of mainline Linux kernel 3.10, which is a base stable version adopted in various enterprise-grade Linux distributions, such as Red Hat Enterprise Linux 7 (RHEL 7) and Community Enterprise Operating System 7 (CentOS 7).

4.1 Page Table Entry (PTE)

COMEX leverages on Linux's original page table. When a page is claimed with its contents sent to a swap partition, the associated PTE's present bit and dirty bit are cleared (set to zero). In general, a PTE has up to 62 bits available for a "swapped out page identifier", as shown in Fig. 6 with six bits reserved for a "swap_type" field to indicate a swap partition (that keeps swap-out pages) and the remaining 57 bits for a "swap_offset" field to denote the partition's starting point [4], [5].

To simplify kernel modification, COMEX exploits the existing PTE format by reserving one unused swap_type number for COMEX identifiers. The 57-bit swap-offset field is for storing a remote node ID (NID) and the page frame number (PFN) of its physical memory allocated to hold a swap-out page of a job executed on the local node. In the current prototype, 16 bits of the field are for remote node ID and 32 bits are for the memory PFN, able to accommodate up to 65,536 COMEX nodes, each contributing up to 4 terabytes physical memory.

4.2 COMEX Memory Management

Every participating node allocates a portion of its physical memory in support of COMEX functionality during initialization, depending on its available physical memory size. For example, a compute node may allocate only a small memory area mainly for RDMA buffers. On the other hand, a memory node can allocate a majority of its physical memory for COMEX use, with (1) the allocated memory region registered to its RDMA-capable NIC and (2) the associated memory region key (produced after registration) sent to other participating nodes.

Each registered memory region is pinned down, partitioned into 3 fragments, as illustrated in Fig. 7. Note that the sizes of all fragments are specified in the configuration file for initialization use. The first fragment contributes to the COMEX shared global memory pool, formed by aggregating the memory fragments of all participating nodes. This fragment varies among nodes, with a compute node possibly allocating little to the shared pool but a memory node allocates its major physical memory for sharing. Memory nodes may contribute different amounts of memory for heterogeneous datacenters. The second fragment serves as the RDMA write buffers, one per connection (to a remote memory node), for efficient RDMA writes. Each buffer consists of a small number of frames (say, 256-512 frames of 4 KB

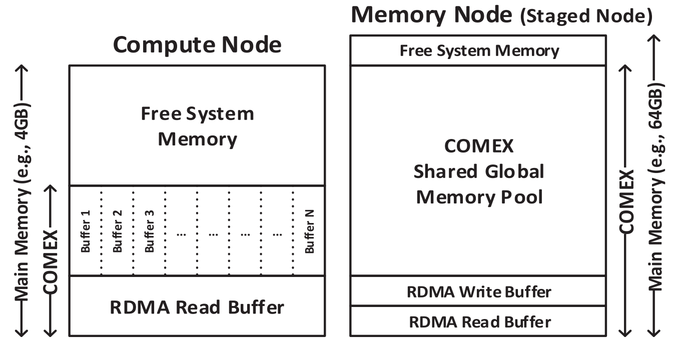

Fig. 7. COMEX Pin-down area layout.

each in our COMEX prototype) to house page frames that are reclaimed by kernel's page frame reclaiming function, waiting for RDMA-transfer. These dedicated RDMA write buffers also serve as buckets for sorting page frames, so that reclaimed pages with the same destination node are placed in physically contiguous buckets of the same buffer. The write buffer for a page is determined by its owner PID (process ID) and NID (ID of the node on which the process is executed). For a process with large memory footprint, its virtual address can be involved for buffer determination as well. Multiple contiguous pages in a buffer are transferred in one RDMA write onto the connected node, resulting in high performance and low network traffic. The third fragment is designated as the RDMA read buffer, for receiving faulted pages and accompanied prefetch pages transmitted by RDMA reads from staged nodes upon page faults. The read buffer is shared dynamically by all remote memory nodes, with its buffer entries managed by the LRU replacement policy, as discussed earlier.

The buddy system is employed at each memory node to manage the first memory fragment that contributes to the shared global memory pool (see Fig. 7). The buddy system provides contiguous memory allocation efficiently while collecting unused page frames returned from compute nodes and merging them to large sizes (of powers of 4 KB) when possible. COMEX buddy system implementation follows page_alloc.c of the Linux kernel source code with modifications. Upon receiving a request for memory page frames to stage the swap-out pages of an execution process, the buddy system allocates a linked list entry that matches the requested size. If the requested memory node has no large enough contiguous page frame chunk available, a negative reply is sent to let the requestor node try another memory node. This way prevents small contiguous memory chunks being allocated, avoiding excessive page frame requests and replies unnecessarily.

4.3 Page Frame Reclaiming

Page frame reclaiming under Linux is invoked periodically and also on-demand if system's free memory level drops below a threshold, where a more aggressive scheme is followed as the free memory amount shrinks (and becoming very aggressive when the amount is down toward 10 percent). It identifies candidate page frames in local memory for reclaiming, based on the well-known LRU-based algorithm. Identified page frames are passed to "shrink_page_list" (the

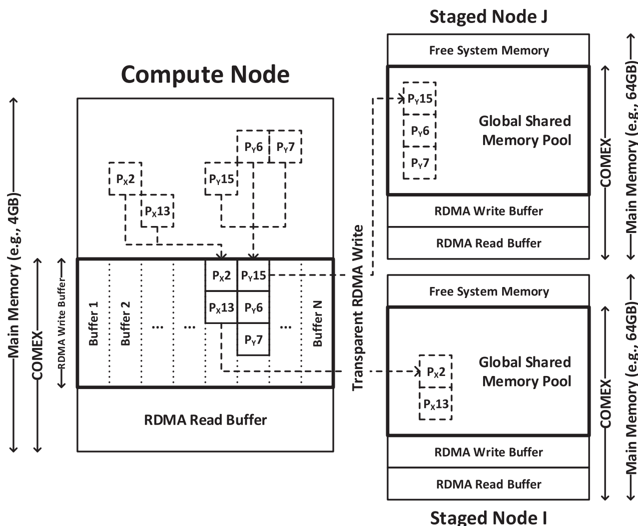

Fig. 8. COMEX page frame reclaiming and transparent RDMA write operation.

page reclaiming function in Linux; see Fig. 3), and they belong to four major types: (1) unclaimable pages, (2) file pages, (3) anonymous pages, and (4) discardable pages [4], [5]. The first page type includes kernel pages and pindown pages that cannot be swapped out from system physical memory. The second page type refers to those page frames which host copies of pages resident in storage, and hence their contents are either written back to corresponding storage copies (if dirty) or simply dropped (if clean), upon reclaiming. The last page type generally signifies page frames that keep cache and buffering pages, and they are simply reclaimed (with their contents discarded). Typically, only the anonymous page type has to be held in the swap partition, and that type can account for a significant portion of pages in the list of shrink_page_list during job execution of data-intensive applications. According to our evaluation for various benchmark runs, the total numbers of anonymous pages reclaimed during execution hike rapidly as the benchmark execution footprints grow larger than host's physical memory size. From Fig. 13, it reveals that the reclaimed page count rises above 185 million over the course of execution for all benchmarks examined.

Under COMEX, a reclaimed anonymous page is assigned to a remote page frame (maintained in the local linked list determined by PID and NID), waiting for transfer in a batch later by one RDMA write. The address of the allocated page frame (comprising the NID and the PFN (page frame number)) is then put to the PTE of the corresponding page, before the reclaimed page frame is released back to the system.

An RDMA write is invoked in one of the three situations: (1) when contiguous pages reach a certain number (set to 64 pages in the prototype), (2) the copy page is not placed in a contiguous page frame location, or (3) the copy page exhausts an RDMA write buffer. After the write, its associated RDMA write buffer is ready for the next reclaiming batch.

Fig. 8 illustrates page frame reclaiming under COMEX, where P_x2 and P_x13 are two pages of Process X while P_y6, P_y7, and P_y15 are pages of Process Y. These five pages are cold and passed to the kernel reclaiming function (shrink_page_list). COMEX assigns them to remote page frames based on their PID and NID (as described in Section 3.3.4). Specifically, P_x2 and P_x13 are assigned to Node I whereas P_y6, P_y7, and P_y15 are assigned to Node J, since the first two (or last three) belong to the same process of X (or Y). After assignment, their contents are copied to the associated buffer slots (for Node I and Node J), waiting for RDMA-transfer in batches, with one batch of buffer slots transferred via one RDMA write to a remote memory node. If a page fault exception happens to any page in an RDMA write buffer, such an exception is handled directly from the write buffer, fulfilling the request via one kernel-level memory copying operation.

4.4 Page Fault Handling

A page fault exception is raised when an execution refers to a location absent in local physical memory, caused possibly by such kernel activities as on-demand paging, copy-on-write, swap-out paging, etc. In the Linux kernel, a page-fault exception is passed onto "handle_mm_fault", a handler implemented in memory.c of the Linux source code (see Fig. 5). The handler examines the faulted PTE (plus flags therein) before taking proper actions. A page fault caused by a swap-out page under Linux results in its associated exception being passed to the "do_swap_page" function, which brings in target page contents from the swap partition via a "read_swap_cache_async" call.

Like the Linux kernel, COMEX examines the faulted PTE to get the swap_type field in the event of a page fault. If the field designates COMEX, NID (node ID) and PFN (page frame number) kept in the associated swap_offset field are extracted. The faulted page in question can be found in three possible locations, by making use of extracted NID and PFN. First, the page is still resided in the RDMA write buffer that is waiting to be written to the staged memory node. In this case, COMEX simply fulfills the page fault with needed contents from the write buffer. Second, the faulted page was found in an RDMA read buffer entry, as a result of prefetching. In this case, COMEX simply moves the found page contents to the page frame allocated by the OS kernel. The page is then marked as "used". Third, if the faulted page is not in either location, it needs to be fetched from a staged node. COMEX in this case issues an RDMA read to bring the faulted page plus its neighboring pages back to the prefetch area (i.e., RDMA read buffer), held in one buffer entry. The faulted page is then used to satisfy the page-fault exception.

Fig. 9 explains a situation when COMEX handles a page-fault exception with respect to a memory page, P_y6, which is staged in a remote node. After examining the RDMA-read and the RDMA-write buffers to find that the failed page is not held there, COMEX issues an RDMA read to bring back the target page of P_y6 plus its three neighboring pages of P_y15, P_y7, and P_y9. Those four pages are placed in one RDMA-read buffer entry, and then P_y6 contents are copied to the page frame allocated by the page-fault exception handler. Finally, the PTE of P_y6 records the allocated page frame.

The page frames in a staged memory node can be released, as long as their corresponding pages (in the RDMA-read buffer maintained in a compute node, as shown

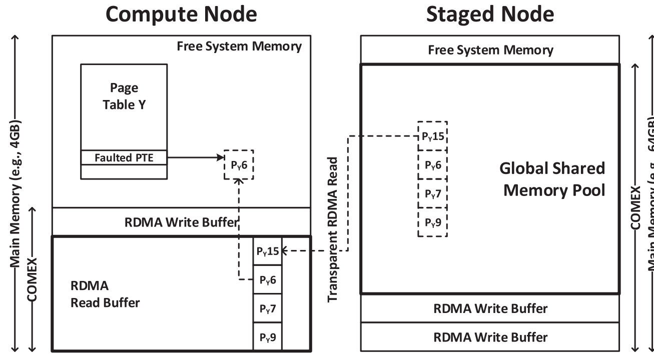

Fig. 9. COMEX page fault exception handler and transparent RDMA read operation.

in Fig. 9) have been used for fulfilling page faults, notified by the compute node. To contain traffic of release notifications sent to memory nodes, COMEX collects multiple used pages in a compact data structure before issuing an RDMA verb notification to the memory node for releasing them at once.

4.5 RDMA Functions

RDMA functions developed in support of COMEX data transfer were modified from krping's example RDMA programming code [63]. Being kernel loadable, the krping module utilizes the Open Fabrics verbs originally to implement a client/server ping/pong program. It is modified extensively for our use in the distributed system environment. At initialization, each node first reads its configuration file to obtain parameters which specify the NID (node ID), the total number of nodes in the cluster with which RDMA connections are to be established, and the size of pre-allocated memory space to serve as the RDMA read buffer, which is shared by all connections, and as an RDMA write buffer, which is dedicated per connection. It then creates control blocks for connections to complete the initialization of all its connection sockets. Two major types of RDMA operations are implemented in the module, as follows.

1) RDMA verbs aim to send control information to a connected node, which will then provide a reply that includes the necessary information based on the received verb. The sender fills a corresponding verb data structure before invoking do_send() to actually deliver the verb. Upon receiving the verb, the receiver prepares a reply data structure accordingly (or simply an acknowledge) before sending it back in a verb via do_send(). Initialization messages, page request/reply messages, and free page message are implemented in RDMA verb.
2) RDMA reads and writes are implemented by the functions of do_read() and do_write(), respectively, with respect to the specific areas in staged memory nodes. They are transparent to the receiving support nodes. However, a node which performs such reads or writes can operate asynchronously (i.e., non-blocking that exits the function afterwards immediately) or in a blocking way (which ensures write or read operations are completed before exiting the function). Since RDMA connections established for COMEX are reliable via Mellanox's Priority Flow Control (PFC) [55], non-blocking RDMA reads and writes are adopted.

TABLE 1
Benchmark Details and Execution Footprints

Benchmarks			Configuration	Footprint
NPB 3.3.1	BT - Block Tri-diagonal solver	Class	Customized D	19.6 GB
		Grid size	500 x 500 x 500	
	FT - discrete 3D FFT	Class	Customized D	40.0 GB
		Grid size	1024 x 1024 x 1024	
	LU - Lower-Upper Gauss-Seidel solver	Class	Customized D	21.4 GB
		Grid size	550 x 550 x 550	
	MG - Multi-Grid	Class	Customized D	26.5 GB
		Grid size	1024 x 1024 x 1024	
	SP - Scalar Penta-diagonal solver	Class	Customized D	35.5 GB
		Grid size	600 x 600 x 600	
PBBS	BFS - Breadth First Search Tree	# of nodes	50,000,000	25.9 GB
		# of edges	400,000,000	
	DICT - Dictionary	Type	Integer	23.3 GB
		# of entries	700,000,000	
	FSORT - Floating point sorting	Type	Floating point	33.9 GB
		# of entries	700,000,000	
	HULL - Convex Hull	Type	Uniform 2D	34.6 GB
		# of dots	350,000,000	
	MIS - Maximal Independent Set	# of nodes	50,000,000	20.0 GB
		# of edges	300,000,000	

5 Evaluation and Result Discussion

This section deals with COMEX performance evaluation under various benchmarks with big execution footprints. A given application can finish its execution on a host machine faster under COMEX than under its native Linux OS (with kernel 3.10.87) counterpart when its execution footprint exceeds host's memory capacity. The speedup results for various memory sizes under an application are gathered using our real testbed, which consists of 32 servers interconnected by an RDMA-enabled switch. Each result shown in subsequent figures is the average of multiple execution runs. Given that the speedup of COMEX results from lower-latency data page transfer between host DRAM and remote DRAM in connected servers, we also collect both (1) the rate of data pages swapped into remote DRAM (over all swapped pages) and (2) the page fault count and its breakdowns during execution, as two additional performance measures of interest. A page fault leads to one RDMA readback from remote DRAM, if it is not a hit to the COMEX prefetching buffer, prolonging execution.

5.1 Evaluation Benchmarks and Testbed

5.1.1 Benchmark Details

Benchmarks chosen for evaluation all involve large execution footprints, including NAS Parallel Benchmarks (NPBs) [64] and CMU's Problem Based Benchmarks Suite (PBBS) [65], as listed in Table 1. As an open source benchmark set from NASA Advanced Supercomputing Division, NPBs are designed to measure high-performance supercomputers.

Among three NPB variants, NPB-SER does not require any special library and is chosen for our benchmark use, since it can run on commodity machines, like networked servers of our testbed. Each benchmark in NPB-SER (Release 3.3.1) has its pre-defined parameters and input sets that lead to different workload classes. Five NPB-SER benchmarks are chosen for evaluation use, with their details provided in Table 1.

PBBS is an open source benchmark suite which defines problems in terms of their functional specifications to compare different programming methodologies [65]. Their execution footprints can be specified flexibly. The five PBBS chosen for our evaluation use have their execution footprints ranging from 23.3 GB to 34.6 GB, as given in Table 1.

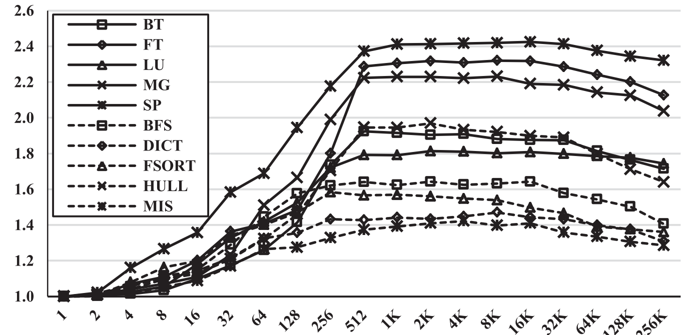

Fig. 10. Execution speedup versus replenishing threshold (T ranging from 1 to 256K pages) for multi-tasking with ten benchmarks executed concurrently on separate compute nodes of the COMEX testbed.

5.1.2 *Testbed Configuration*

Our testbed consists of 32 Dell PowerEdge 1950 servers networked by a Mellanox 48-Port 10GbE RDMA-enabled switch (Model MSX1024B-1BFS-RF). Each 1950 server has two processor sockets, each equipped with a dual core Intel Xeon Processor 5160 (Woodcrest, with a 4-MB shared L2 cache). Every server has eight 8-GB DDR2 DRAM modules (for a total size of 64 GB), with one Mellanox RDMA-enabled dual-port 10GbE ConnectX-3 network interface card (NIC) [66] installed. A server is connected to the MSX1024B switch via one port of its installed NIC. We also upgrade the backing storage to the recent batch (February 2018), Western Digital 2-TB 5,400RPM SATA hard disk drive with 64 MB buffer cache, because old disks can generate unrealistic results and backing storage is the sensitive parameter in our evaluation.

Each machine runs CentOS 6.10 Final, Linux-based distribution (under kernel 3.10.87), with all software updated to the date when our evaluation commenced, as the basis for gathering performance metrics of interest during benchmark execution on a configured DRAM size. Performance metrics of benchmark execution on the testbed with its constituent servers running COMEX (built under customized kernel 3.10.87) for the same DRAM configuration are then gathered. They reveal COMEX memory aggregation results under various situations ranging from partial disaggregation (where compute nodes equipped with reasonable size memory) to almost full disaggregation (where most memory is disaggregated). Note that when an application is executed on a compute node, its physical memory capacity is configured to various sizes through GNU GRUB boot-loader option. Compute node only pins down a small memory fragment for RDMA buffers (read and write) in support of RDMA operations while contributing zero memory to the shared global pool. Other idle nodes are performing supporting role as staged nodes (memory nodes) with majority of their physical contributed to the shared global pool and they are configured to contribute 48 GB memory in this experiment.

5.2 COMEX Sensitivity Analysis

COMEX performance is dictated by several configuration parameters. In this section, we evaluate COMEX key parameters, including the replenishing threshold (denoted by T), the prefetch size (denoted by P), and the RDMA readback buffer size (denoted by B). When the number available pages maintained in local pools drop below T, COMEX sends out a memory replenishing request to one memory node for memory refilling. P refers to the number of contiguous data pages to be prefetched into the COMEX read buffer upon each page fault when the page contents located in remote memory (DRAM), via one RDMA read operation. The RDMA readback buffer allocated for holding data pages prefetched from remote DRAM. Naturally, a larger buffer size (B) usually yields a better result because more data pages then can be held for future subsequent page faults, as a result of spatial and temporal locality. However, an excessively large B is undesirable. COMEX employs a prefetch table to keep track of those prefetched page frames. Any page fault that has a hit in the buffer is fulfilled immediately without invoking RDMA data transfer.

5.2.1 *Page Replenishing Results*

As discussed in Section 3.3.3, COMEX caches memory pages in its corresponding memory pools for good page-reclaiming performance, with the replenishing threshold of T. Obviously, without enough memory pages cached locally (under a small T), COMEX has to wait for the reply of a memory request back from a memory node before putting a swap-out page into its corresponding buffer when the memory pool is exhausted, degrading execution performance. Excessively large T, on the other hand, may see memory pool pages unused, while blocking applications from utilizing those allocated pool pages unnecessarily, lowering execution performance as well.

Execution speedup results versus a wide T range for multi-tasking with ten benchmarks executed concurrently on separate compute nodes of the COMEX testbed are shown in Fig. 10. Each compute node has 8 GB system memory, whereas a memory node contributes 48 GB of its DRAM to the global memory pool. Benchmarks are launched simultaneously on compute nodes, which concurrently share the global memory pool contributed by 12 memory nodes. The speedup gain is calculated against $T = 0$, referring to the baseline configure with proactive replenishing disabled.

When T is smaller than 256 pages, all benchmarks exhibit performance levels lower than their best potentials. Specifically, four benchmarks (i.e., BFS, DICT, FSORT, MIS) see

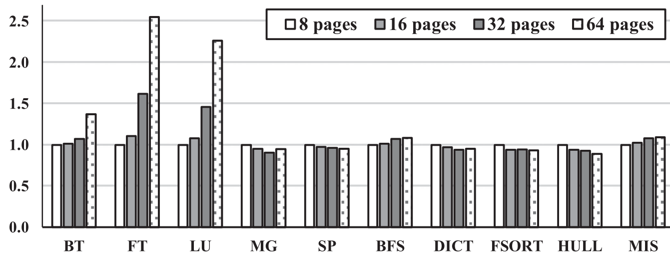

Fig. 11. Normalized benchmark execution times under various prefetch sizes when B = 8192 pages (with the result of P = 8 pages for each benchmark as the baseline).

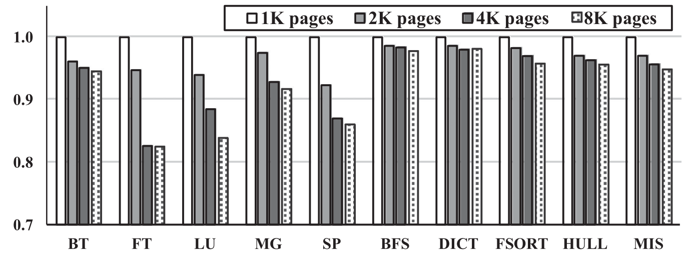

Fig. 12. Normalized benchmark execution times (with respect to those of B = 1K pages) under various buffer sizes for P = 16 pages.

their performance levels plateau for $T \geq 256$, since the memory pool can then keep up with the system (Linux) page frame-reclaiming rate. On the other hand, other benchmarks require T to reach 512 or even 2K before their performance levels peak. Results in Fig. 10 also reveal that benchmark performance starts to degrade if T goes beyond 8K. Overall, there is a favorable range of **T** (say, 512 to 8K) for which benchmarks tend to exhibit sound performance, agreeing with our derivation given in Section 3.3.4. In COMEX evaluation hereafter, we adopt T of 1024 as the proactive replenishing threshold for local memory pools.

5.2.2 Prefetching Results

For a given B (buffer size), the best P (prefetch size) is application-dependent, since sequential data access patterns are to favor a large P whereas random data access patterns benefit from a small P as a result of better buffer utilization and less energy waste caused by moving unused page contents.

The prefetching performance outcomes for ten benchmarks under different P values ranging from consecutive 8 to 64 pages are shown in Fig. 11, when benchmarks are executed under a compute node with 8 GB memory (DRAM) and under B = 8192 pages (to avoid it from causing a performance bottleneck). The baseline result for each benchmark in the figure is obtained under P = 8 pages. As can be found in the figure, five benchmarks (e.g., MG, SP, DICT, FSORT, HULL) exhibit slightly performance improvement for a larger P (likely due to their sequential data access manners) whereas five other benchmarks (e.g., BT, FT, LU, BFS, MIS) suffer performance degradation as P rises. However, 32-pages and 64-pages reveal severe deteriorated results under two benchmarks (FT and LU). In general, it is concluded that 8 and 16 pages are appropriate prefetch sizes since both are favorable across all benchmarks examined while benefiting from spatial and temporal locality reasonably. COMEX results presented in *subsequent figures are all based on* P = 16 (i.e., prefetching 16 pages).

In Linux 3.10, the OS kernel prefetches (also known as readahead in Linux) consecutive 8 swap-out pages at a time because the kernel I/O block subsystem is fundamentally designed and heavily tuned with optimizations for rotational devices (hard disk drive), such as buffering, reordering, and prefetching [4], [67], [68], [69]. Due to the fact that disk device is a non-uniform-access device returning different access latencies from physical data locations, disk's random access can be a magnitude slower than its sequential access [67], [68] because such a device needs to move its read/write head to the right location before accessing. Paying huge seeking time and spinning time for reading only one swap-out page is not worth the excessive price, so that Linux expands a small random access to a larger efficient sequential access. On the other hand, DRAM (remote memory) is a random access media returning uniform access latency across all locations storing data in the chip. Thus, COMEX does not need to worry about seeking and spinning delays like disk counterparts. COMEX prefetching mainly aims for high performance.

5.2.3 Buffering Results

It is natural that a larger COMEX buffer (B) can hold more prefetched pages from remote DRAM via RDMA reads, each for 16 contiguous pages. However, after B reaches a certain value, any further increase is subject to a quick diminishing return, making it unattractive to provision the COMEX buffer excessively. In addition, since the provisioned buffer counts toward overall COMEX memory consumption, it is desirable to set the buffer size appropriately to have high performance, avoiding aggressive system memory reclaiming.

COMEX buffering performance outcomes over ten benchmarks chosen for evaluation with 16-pages prefetching under four buffer sizes are depicted in Fig. 12, where each benchmark is run on a compute node with its DRAM sized to 8 GB and the results of B = 1K pages normalized to 1.0. The figure reveals that an increase in B from 1K to 2K and 4K benefits all benchmarks with lowered execution times. However, further increasing in B beyond 4K pages is found to have diminishing gains in five benchmarks (BT, FT, MG, SP, DICT). As a result, the COMEX buffer B *is sized to 4096 pages* for all results illustrated in subsequent figures.

5.3 COMEX Evaluation Results

We evaluate COMEX under different system memory sizes (denoted by M). Naturally, a smaller M results in higher system memory pressure, which causes Linux (OS) to claim memory more aggressively. For a given M, all benchmarks execute faster with COMEX support than without (under their native kernel counterparts), resulting in execution speedups.

5.3.1 Mono-Tasking Execution Results Under COMEX

For mono-tasking evaluation, each benchmark is run on one server, with on-demand paging support by staging servers (connected by an RDMA-enabled switch). The compute node is configured to allocate a small pin-down region, whereas each staged node pins down and contributes 48 GB of its DRAM to the shared global pool.

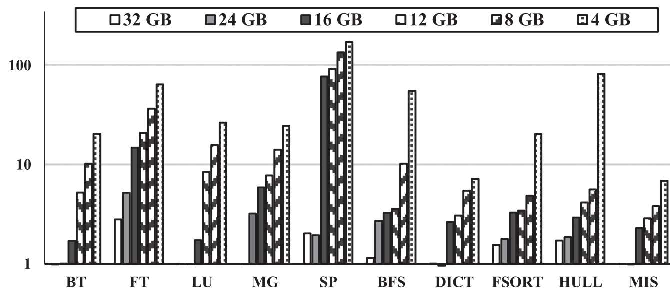

Fig. 13. Mono-tasking execution speedups versus M values under COMEX of ten benchmarks for T = 1024, P = 16, and B = 4096 pages.

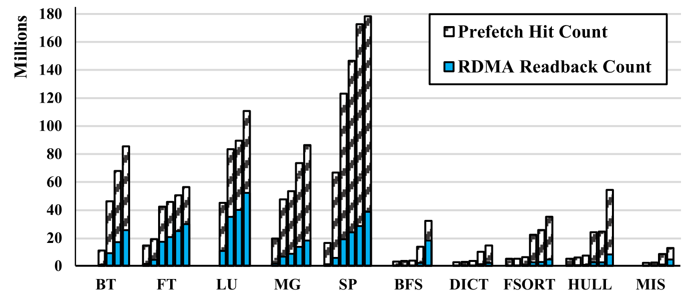

Fig. 14. COMEX Mono-tasking page fault activity break-downs under ten benchmarks for T = 1024, P = 16, and B = 4096 pages, with the results of each benchmark for M = 32, 24, 16, 12, 8, and 4 GB denoted respectively by its associated six bars.

Execution speedups versus M values for ten benchmarks under COMEX are shown in Fig. 13, where each benchmark exhibits a larger speedup for a smaller M value, as a result of longer execution under native Linux but limited execution prolongation with COMEX support, when the benchmark execution footprint exceeds M. If M drops further, increased system memory pressure pushes kernel page frame reclaiming to aggressively claim cold anonymous pages, resulting in stagnant execution without COMEX support. For M = 4 GB, COMEX accelerates SP execution by more than 170×, when compared with its execution under native Linux on the same host machine. All benchmarks under extremely scarce memory (of 4 GB) have their speedups well exceeding 20.0, except for DICT (with its speedup of 7.2) and MIS (with the speedup of 6.8).

5.3.2 Application Memory Footprint and Working-Set

On the other hand, different benchmarks for a given M, have their speedups dictated by their execution footprints, memory working-set sizes, and application memory access patterns. Some applications may allocate large memory chunks for storing their data (footprint) but do not require all data simultaneously during computation (working-set). Such memory access patterns technically can work at low system memory with negligible swapping penalties until available system memory drops below their actual working-set sizes. As can be found in Fig. 13, SP has a smaller execution footprint (of 35.5 GB) than FT (with 40.0 GB) and yet exhibits far bigger speedups for $M \leq 16$ GB, mostly due to its less regular memory accesses that result in more page faults during execution to amplify COMEX's benefits. In contrast, HULL has a relatively large execution footprint (of 34.6 GB) but exhibits a speedup spike only when M drops down to 4 GB, because of its regular memory access patterns to make it tolerate memory insufficiency well until memory is extremely small.

5.3.3 Page Fault Count and Prefetch Results Under COMEX

The number of page faults directly affects the execution time of a benchmark, with a larger page-fault count naturally resulting from both higher memory scarcity (i.e., smaller M) and less regular application memory accesses, as explained above. Under COMEX, page faults can be fulfilled in two possible ways: (1) from the RDMA readback buffer, if a faulted page is already brought back by a prior RDMA read, resulting in a prefetch hit with very low latency and (2) from DRAM of a staged node, if the faulted page is not resided in the RDMA read buffer, requiring COMEX to issue an RDMA read transfer for reading back the page. Break-downs of page fault activities under COMEX for ten benchmarks are depicted in Fig. 14, where the RDMA readback count typically represents a small fraction of all paging activities and the prefetch hit count includes those faulted pages found in the RDMA readback buffer. Six bars associated with each benchmark in the figure denote the page-fault counts for M = 32, 24, 16, 12, 8, and 4 GB, respectively.

From the breakdowns of each bar, it is evident that COMEX achieves a high hit rate to the RDMA readback buffer, which is filled with page prefetching upon page faults. The vast majority of page faults, which are on the critical path, can be satisfied by prefetched pages kept in the COMEX buffer (desirably sized to B = 4096 pages, as discussed earlier and illustrated in Fig. 12), thanks to COMEX page locality-aware handling which sorts and keeps pages of the same process in a given staged node as best as possible. Six benchmarks achieve high hit rates under M = 4 GB (specifically, 87 percent for FSORT, 86 percent for DICT, 85 percent for HULL, 79 percent for MG, 78 percent for SP, and 70 percent for BT), clearly benefiting from COMEX prefetching and page locality-aware handling. Only three benchmarks exhibit mediocre hit rates (with 44 percent for BFS, 47 percent for FT, and 53 percent for LU) likely due to their irregular memory accesses that limit prefetching gains. In fact, they can suffer more from larger prefetch sizes ($P > 16$) because then higher amounts of prefetched pages are wasted, as revealed in Fig. 11 on prefetching size evaluation. It can be seen in Fig. 11 that FT and LU exhibit markedly worse performance if P rises beyond 16. They favor small prefetch sizes (of $P \leq 8$).

5.3.4 Multi-Tasking Execution Results Under COMEX

COMEX multi-tasking execution performance is evaluated under a range of M for each compute node, with different benchmarks executed on separate compute nodes concurrently. Our testbed cluster is configured to have 10 compute nodes (to run ten different benchmarks launched simultaneously) and 22 memory nodes, with each memory node contributing 48-GB memory to the global pool in support of demand paging dynamically and transparently during multi-tasking execution.

Multi-tasking execution results are illustrated in Fig. 15, where the speedups are against times taken under native

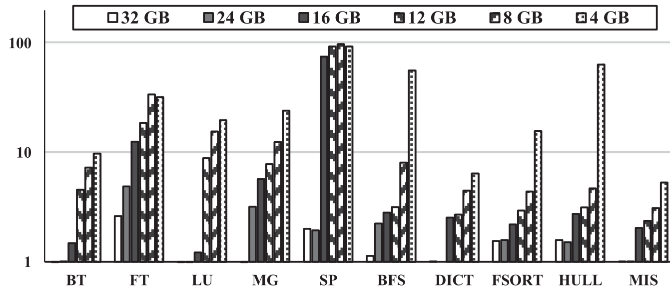

Fig. 15. Multi-tasking execution speedups under COMEX when ten benchmarks are executed concurrently on separate compute nodes, with each of its M ranging from 32 to 4 GB.

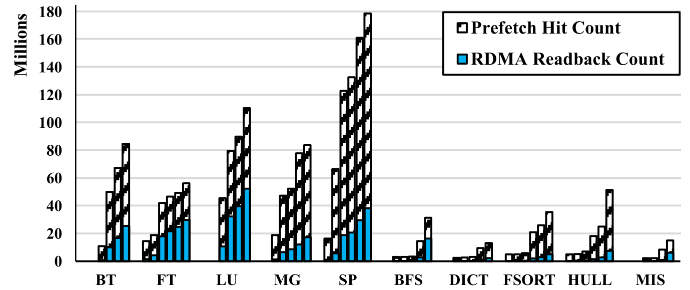

Fig. 16. COMEX multi-tasking page fault activity breakdowns under ten benchmarks for T = 1024, P = 16, and B = 4096 pages, with the results of each benchmark for M = 32, 24, 16, 12, 8, and 4 GB denoted respectively by its associated six bars.

Linux on same ten compute nodes to run benchmarks separately. Every compute node is equipped with the same amount of memory (M), which ranges from 32 GB to 4 GB. Benchmarks executed concurrently on compute nodes under COMEX are all found to enjoy various speedups, with more speedups for all benchmark under a smaller M. This is mainly because the memory requirements of benchmarks vary during the course of execution, with some having high memory demands at a time point when others have low memory needs (as observed also in prior work [7], [8], [24], [46]), making on-demand memory expansion effective. If compute nodes have extremely scarce memory (e.g., M = 4 GB), concurrent execution exhibits speedup upsurges for all benchmarks, resulting from their drastic execution slowdowns on computer nodes with native Linux. When executed on servers without COMEX support, applications are to experience performance degradation if their execution footprints exceed the physical memory sizes of servers, with bigger degradation under higher server memory scarcity then to better benefit from COMEX. The page fault activity count under COMEX multi-tasking scenarios are shown in Fig. 16, where COMEX is seen to still exhibit prefetching performance levels close to those under the mono-tasking counterpart (see Fig. 14). While aggregate memory offered by memory nodes is shared heavily under multi-tasking, COMEX still maintains effective prefetching due mainly to its key design feature of locality-aware remote paging discussed earlier.

A given benchmark is expected to have a smaller speedup under multi-tasking execution than under mono-tasking execution, as a result of competing the global memory pool and higher aggregate network traffic from multiple compute nodes. For example, under M = 8 GB, the mean execution speedup of ten benchmarks drops to 19× under multi-tasking (see Fig. 15) from 24× under mono-tasking (see Fig. 13). Since the global memory pool of COMEX is managed following the buddy system (as described in Section 3.3), memory fragmentation in the global pool is expected after demand paging from involved compute nodes to get continuous page frames (with prefetching) and page reclaiming to return mostly noncontiguous page frames repeatedly. The fragmentation degree is higher under multi-tasking execution than under mono-tasking, and it is more severe if the compute nodes have higher memory pressure (under a smaller M). A mechanism for alleviating this memory fragmentation is yet to be included in COMEX.

In line with the modern design approach of resource disaggregation, COMEX facilitates easy memory upgrading through both scale-up (increasing memory capacity) and scale-out (adding more memory nodes). Every compute node in a server cluster thus benefits from memory upgrading equally, without concerning about hardware limitations of individual server nodes.

5.3.5 *Execution Results With SSD as Swap Partition*

The solid-state drive (SSD) has become an alternative storage device even in datacenters where failures cannot be avoided [70], due to its far speedier reads and writes. To demonstrate the advantage of COMEX over a counterpart server with its swap partition served by an SSD (of SATA Transcend 64 GB SSD340), we conducted benchmark evaluation under **M** = 8 GB, with normalized execution time results shown in Fig. 17. The results were normalized with respect to those of COMEX under mono-tasking execution (given in Fig. 13). It is evident that COMEX clearly prevails, as a result of three key aspects. First, the remote memory (DRAM) access in our testbed is faster than the access to the SSD (of Transcend SSD340 with 530 MB/s read bandwidth), even with its dual-channel 667 MHz PC5300 DDR2 DRAM (with 10.6 Gb/s bandwidth). Second, COMEX employs superior data transfer channels over the 10GbE RDMA-enabled networking fabric, as compared to SATA's 1.5 Gb/s. Third, SSD suffers from heavy overhead inherent to the legacy swap design [41], [42], [43], [44], [45]. Under a larger M (say, 16 GB), it is found that the normalized execution time gaps shrink but COMEX always solidly outperforms its counterpart with an SSD-based swap partition.

Adopting SSD to improve swap performance in datacenters and HPC environments is possible, given that SSD and non-volatile memory have seen their capacities rise and costs drop. However, SSD has a well-known limited lifespan problem which can be undesirable for such write-intensive

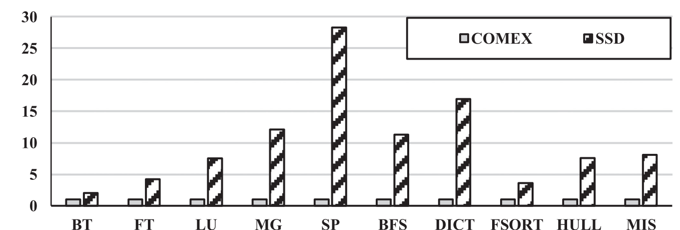

Fig. 17. Normalized execution time results of COMEX in comparison to those of SSD as a swap partition under M = 8 GB.

applications as swap partitions. Hence, adopting SSD for swap partition use must be considered thoroughly from the resilience perspective, besides its cost and performance.

6 Conclusion

Cooperative memory expansion (COMEX) aims to enable fine-grain kernel-level memory aggregation in networked computing systems in support of resource disaggregation design. COMEX has been developed and implemented by extending the OS kernel memory subsystem to let any machine in a networked computing system expand its memory size on-demand dynamically to hold evicted pages for accelerating execution. COMEX leverages page tables existing in Linux OS to manage data transfer between remote page frames (in memory nodes) and local page frames (in a compute node) over low-latency RDMA connections. It achieves execution speedups of applications with large execution footprints that dwarf the host's memory size, by avoiding much slower disks as swap space (commonly under a modern OS). COMEX is a light-weight kernel-level design and exploits kernel information for locality-aware page transfer, resulting in superior prefetching performance. It is completely transparent to applications and users, applicable to any commodity computing system networked by the RDMA-enabled fabric. Such a design approach is suitable for any OS that relies on page tables to handle virtual address mapping.

In support of applications with memory footprints larger than what is available to a node's physical memory alone, COMEX utilizes physical memory of remote nodes in a distributed system networked by an RDMA fabric (i.e., Mellanox RoCEv2) for page-fault swapping realized via the Linux OS virtual memory subsystem. It avoids the pitfalls of previous RDMA memory expansion solutions that rely on less efficient I/O-based Linux swap mechanisms, more granular slab-based approaches, or user-level remote buffer regions that require application integration.

With 16-page prefetching to a 4096-page RDMA read-back buffer at each compute node, COMEX exhibits high buffer utilization for almost all execution scenarios examined in our experimental evaluation, undertaken on a testbed with 32 Dell servers connected by an RDMA-enabled 48-port 10GbE switch (Model MSX1024B) via Mellanox 10 GbE ConnectX-3 NIC ports [66]. Evaluation results demonstrate that memory aggregation over networked computing cluster achieved by COMEX enjoys higher speedups as the execution footprint grows larger than machine's memory size. Under extreme memory disaggregation from compute nodes, one benchmark is seen to yield a speedup over 170× (or 86×) under mono-tasking execution (or multi-tasking execution with ten concurrent workloads). Hence, COMEX is particularly advantageous in support of memory disaggregation for executing diverse applications with large execution footprints.

Acknowledgments

This work was supported in part by National Science Foundation under Award Numbers: CCF-1423302, CNS-1527051, and III-1763620, and in part by Louisiana Board of Regents under Contract Number LEQSF(2018-21)-RD-A-24.

References

[1] Intel, "INTEL Ed.: Innovating for the 'Data-Centric' Era," Aug. 2018. [Online]. Available: https://www.intc.com/investor-relations/investor-education-and-news/investor-news/press-release-details/2018/Intel-Editorial-Innovating-for-the-Data-Centric-Era/default.aspx
[2] DELL Technologies, "Dell technologies secures hyper-converged infrastructure certification for SAP HANA production environments," Oct. 2018. [Online]. Available: https://corporate.delltechnologies.com/en-us/newsroom/dell-technologies-hci-sap-hana-certification.htm
[3] Lenovo, "In-memory computing with SAP HANA," Jan. 2016. [Online]. Available: http://blog.lenovo.com/en/blog/in-memory-computing-with-sap-hana
[4] D. P. Bovet and M. Cesati, in *Understanding The Linux Kernel*, 3rd ed., Sebastopol, CA, USA: O'Reilly Media, 2006.
[5] M. Gorman, *Understanding The Linux Virtual Memory Manager*, Jul. 2007. [Online]. Available: https://www.kernel.org/doc/gorman/
[6] Intel, "Intel, facebook collaborate on future data center rack technologies," Feb. 2013. [Online]. Available: https://newsroom.intel.com/news-releases/intel-facebook-collaborate-on-future-data-center-rack-technologies/
[7] P. X. Gao *et al.*, "Network requirements for resource disaggregation," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 249–264.
[8] S. Han *et al.*, "Network support for resource disaggregation in next-generation datacenters," in *Proc. 12th ACM Workshop Hot Topics Netw.*, 2013, Art. no. 10.
[9] A. D. Papaioannou, R. Nejabati, and D. Simeonidou, "The benefits of a disaggregated data centre: A resource allocation approach," in *Proc. IEEE Global Commun. Conf.*, Dec. 2016, pp. 1–6.
[10] K. Koh, K. Kim, and S. Jeon, J. Huh, "Disaggregated cloud memory with elastic block management," *IEEE Trans. Comput.*, vol. 68, no. 1, pp. 39–52, Jan. 2019.
[11] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2016, pp. 437–450.
[12] S. Tsai and Y. Zhang, "LITE Kernel RDMA support for datacenter applications," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 306–324.
[13] E. Kissel and M. Swany, "Photon: Remote memory access middleware for high-performance runtime systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 1736–1743.
[14] R. Gerstenberger, M. Besta, and T. Hoefler, "Enabling highly-scalable remote memory access programming with MPI-3 one sided," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, Nov. 2013, pp. 1–12.
[15] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Operat. Syst.*, 2014, pp. 3–18.
[16] Memcached, "What is memcached?" 2009. [Online]. Available: http://memcached.org/.
[17] R. Nishtala *et al.*, "Scaling Memcache at facebook," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 385–398.
[18] J. Zawodny, "Redis: Lightweight key-value store that goes the extra mile," *Linux Mag.*, Aug. 2009. [Online]. Available: http://redis.io/
[19] C. Mitchell, Y. Geng, and J. Li, "Using one-sided RDMA reads to build a fast, CPU-efficient key-value store," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 103–114.
[20] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "FaRM: Fast remote memory," in *Proc. 11th USENIX Conf. Netw. Syst. Des. Implementation*, 2014, pp. 401–414.
[21] B. Cassell *et al.*, "Nessie: A decoupled, client-driven key-value store using RDMA," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 12, pp. 3537–3552, Dec. 2017.
[22] J. Ousterhout *et al.*, "The RAMCloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, Sep. 2015, Art. no. 7.
[23] RAMCloud, "Deciding whether to use RAMCloud," Nov. 2015. Available:https://ramcloud.atlassian.net/wiki/spaces/RAM/pages/6848537/Deciding+Whether+to+Use+RAMCloud
[24] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with INFINISWAP," in *Proc. 14th USENIX Conf. Netw. Syst. Des. Implementation*, 2017, pp. 649–667.
[25] H. Midorikawa, M. Kurokawa, R. Himeno, and M. Sato, "DLM: A distributed large memory system using remote memory swapping over cluster nodes," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2008, pp. 268–273.

[26] H. Oura, H. Midorikawa, K. Kitagawa, and M. Kai, "Design and evaluation of page-swap protocols for a remote memory paging system," in *Proc. IEEE Pacific Rim Conf. Commun. Comput. Signal Process.*, 2017, pp. 1–8.
[27] H. Choi, K. Kim, and D. Kang, "Performance evaluation of a remote block device with high-speed cluster interconnects," in *Proc. 8th Int. Conf. Comput. Model. Simul.*, 2017, pp. 84–88.
[28] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel, "Nswap: A network swapping module for linux clusters," in *Proc. Euro-Par Int. Conf. Parallel Distrib. Comput.*, 2003, pp. 1160–1169.
[29] T. Newhall, E. R. Lehman-Borer, and B. Marks, "Nswap2L: Transparently managing heterogeneous cluster storage resources for fast swapping," in *Proc. 2st Int. Symp. Memory Syst.*, 2016, pp. 50–61.
[30] S. Liang, R. Noronha, and D. K. Panda, "Swapping to remote memory over InfiniBand: An approach using a high performance network block device," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2005, pp. 1–10.
[31] K. Lim *et al.*, "Disaggregated memory for expansion and sharing in blade servers," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 267–278.
[32] K. Lim *et al.*, "System-level implications of disaggregated memory," in *Proc. IEEE Int. Symp. High-Perform. Comp Archit.*, 2012, pp. 1–12.
[33] Alibaba Developer, "Alibaba cluster data: Using 270 GB of open source data to understand alibaba data centers," Jan. 2019. [Online]. Available: https://www.alibabacloud.com/blog/alibaba-cluster-data-using-270-gb-of-open-source-data-to-understand-alibaba-data-centers_594340
[34] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets, "Cashmere-VLM: Remote memory paging for software distributed shared memory," in *Proc. 13th Int. Parallel Process. Symp.*, 2002, pp. 153–159.
[35] M. D. Flouris and E. P. Markatos, "The network RamDisk: Using remote memory on heterogeneous NOWs," *J. Cluster Comput.*, vol. 2, no. 4, pp. 281–293, 1999.
[36] Alibaba Open Source, "Alibaba cluster trace program," Sep. 2019. [Online]. Available: https://github.com/alibaba/clusterdata
[37] M. Bielski, I. Syrigos, K. Katrinis *et al.*, "dReDBox: Materializing a full-stack rack-scale system prototype of a next-generation disaggregated datacenter," in *Proc. Des. Autom. Test Europe Conf. Exhib.*, 2018, pp. 1093–1098.
[38] D. Syrivelis *et al.*, "A software-defined architecture and prototype for disaggregated memory rack scale systems," in *Proc. Int. Conf. Embedded Comput. Syst.: Archit. Modeling Simul.*, 2018, pp. 300–307.
[39] G. Zervas, H. Yuan, A. Saljoghei, Q. Chen, and V. Mishra, "Optically disaggregated data centers with minimal remote memory latency: Technologies, architectures, and resource allocation," *IEEE/OSA J. Opt. Commun. Netw.*, vol. 10, no. 2, pp. A270–A285, Feb. 2018.
[40] A. Peters, G. Oikonomou, and G. Zervas, "In compute/memory dynamic packet/circuit switch placement for optically disaggregated data centers," *IEEE/OSA J. Opt. Commun. Netw.*, vol. 10, pp. 164–178, Jul. 2018.
[41] A. M. Caulfield *et al.*, "Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, pp. 1–11, Nov. 2010.
[42] E. Seppanen, M. T. O'Keefe, and D. J. Lilja, "High performance solid state storage under linux," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–12.
[43] Y. J. Yu *et al.*, "Optimizing the block I/O subsystem for fast storage devices," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, Jun. 2014, Art. no. 6.
[44] Tim Chen, "The next steps for swap," in *Proc. Linux Storage Filesystem Memory-Manage. Summit*, 2017. [Online]. Available: https://lwn.net/Articles/717707/
[45] Jonathan Corbet, "The final step for huge-page swapping," Jul. 2018. [Online]. Available: https://lwn.net/Articles/758677/
[46] M. K. Aguilera *et al.*, "Remote memory in the age of fast networks," in *Proc. Symp. Cloud Comput.*, Sep. 2017, pp. 121–127.
[47] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's time for low latency," in *Proc. 13th USENIX Conf. Hot Topics Operating Syst.*, 2011, Art. no. 1.
[48] C. Guo *et al.*, "RDMA over commodity ethernet at scale," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 202–215.
[49] RDMA Consortium, "Architectural specifications for RDMA over TCP/IP," 2009. [Online]. Available: http://www.rdmaconsortium.org/
[50] Finteligent Trading Technology Community (FTTC), "Research report: 10GbE low latency networking technology review," Oct. 2012. [Online]. Available: http://www.mellanox.com/related-docs/whitepapers/FTTC_10GbE_Report_FINAL.PDF
[51] The RoCE Initiative, "RoCE interoperability list," Oct. 2019. [Online]. Available: http://www.roceinitiative.org/integratorslist/
[52] InfiniBand Trade Association, "InfiniBand product directory," IBTA 2013. [Online]. Available: https://www.infinibandta.org/infiniband-product-directory/
[53] Y. Zhu *et al.*, "Congestion control for large-scale RDMA deployments," in *Proc. ACM Conf. Special Interest Group Data Commun.*, 2015, pp. 523–536.
[54] Dotan Barak, "ibv_reg_mr()," *RDMAmojo – blog on RDMA Technol. and programming*, 2012. [Online]. Available: http://www.rdmamojo.com/2012/09/07/ibv_reg_mr/
[55] Mellanox Technologies, "Recommended network configuration examples for RoCE deployment," May 2017. [Online]. Available: https://community.mellanox.com/docs/DOC-2855
[56] M. Handley *et al.*, "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proc. ACM Special Interest Group Data Commun.*, 2017, pp. 29–42.
[57] Kevin Kaichuan He, "Why and how to use netlink socket," *Kernel Korner*, Jan. 2005. [Online]. Available: https://www.linuxjournal.com/article/7356
[58] Ariane Keller, "Kernel space, user space interfaces," *The Linux Documentation Project*, Oct. 2008. [Online]. Available: http://wiki.tldp.org/kernel_user_space_howto
[59] Corbet, "Extending netlink," Apr. 2005. [Online]. Available: https://lwn.net/Articles/131802/
[60] Linux programmer's manual, "Signal - Overview of signals," *Linux manual page*, Mar. 2019. [Online]. Available: http://man7.org/linux/man-pages/man7/signal.7.html
[61] Mellanox, "RoCE vs. iWARP competitive analysis," 2017. [Online]. Available: http://www.mellanox.com/related-docs/whitepapers/WP_RoCE_vs_iWARP.pdf
[62] Mellanox, "SX1024," 2017. http://www.mellanox.com/related-docs/prod_eth_switches/PB_SX1024.pdf
[63] Steve Wise, "Kernel mode RDMA ping module," 2009. [Online]. Available: https://github.com/larrystevenwise/krping
[64] NASA advanced supercomputing division, "NAS parallel benchmarks," 2017. [Online]. Available: https://www.nas.nasa.gov/publications/npb.html
[65] J. Shun *et al.*, "The problem based benchmark suite," in *Proc. 24th Annu. ACM Symp. Parallelism Algorithms Archit.*, 2012, pp. 68–70.
[66] Mellanox Technologies, Official Store, "Mellanox connectX-3 10 pro gigabit dual-port ethernet network interface card," Sept. 2017. [Online]. Available: https://store.mellanox.com/categories/adapters/ethernet-cards.html
[67] W. U. Fengguang, X. I. Hongsheng, and X. U. Chenfeng, "On the design of a new linux readahead framework," *ACM SIGOPS Operating Syst. Rev. - Res. Develop. Linux Kernel*, vol. 42, no. 5, pp. 75–84, Jul. 2008.
[68] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch," in *Proc. USENIX Annu. Techn. Conf.*, 2007, Art. no. 20.
[69] C. Li, K. Shen, and A. E. Papathanasiou, "Competitive prefetching for concurrent sequential I/O," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 2007, pp. 189–202.
[70] P. Sigdel and N.-F. Tzeng, "Coalescing and deduplicating incremental checkpoint files for restore-express multi-level checkpointing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 12, pp. 2713–2727, Dec. 2018.

Pisacha Srinuan received the BE degree in computer engineering from Kasetsart University, Bangkok, Thailand, in 2013, and the ME degree in computer engineering from the Center for Advanced Computer Studies (CACS), University of Louisiana, Lafayette, in 2016, where he is currently working toward the PhD degree, in computer engineering. Since 2013, he has been a research assistant and system administrator with the Computer Architecture and Networks Laboratory, in CACS. His research interests include high-performance computing, computer architecture and memory systems, operating systems, and machine learning.

Xu Yuan (Member, IEEE) received the BS degree from Nankai University, Tianjin, China, in 2009, and the PhD degree from the Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA, in 2016. From 2016 to 2017, he was a post-doctoral fellow of electrical and computer engineering at the University of Toronto, Toronto, ON, Canada. He is currently an assistant professor with the School of Computing and Informatics at the University of Louisiana at Lafayette, LA, USA. His research interest focuses on wireless communication, networking, cyber-physical system, and security and privacy.

Nian-Feng Tzeng (Fellow, IEEE) has been with Center for Advanced Computer Studies, School of Computing and Informatics, the University of Louisiana, Lafayette, since 1987. His current research interest is in the areas of high-performance computer architecture and systems, computer communications and networks, and parallel and distributed processing. He was on the editorial board of the *IEEE Transactions on Parallel and Distributed Systems*, 1998–2001, and on the editorial board of the *IEEE Transactions on Computers*, 1994–1998. He was the chair of Technical Committee on Distributed Processing, the *IEEE Computer Society*, from 1999 till 2002. He is the recipient of the Outstanding Paper Award of the 10th IEEE International Conference on Distributed Computing Systems, May 1990, and received the University Foundation Distinguished Professor Award, in 1997.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.