DEX: Scaling Applications Beyond Machine Boundaries

Sang-Hoon Kim
Ajou University
Suwon, South Korea
sanghooonkim@ajou.ac.kr

Ho-Ren Chuang
Virginia Tech
Blacksburg, USA
horenc@vt.edu

Robert Lyerly
Virginia Tech
Blacksburg, USA
rlyerly@vt.edu

Pierre Olivier
The University of Manchester
Manchester, UK
pierre.olivier@manchester.ac.uk

Changwoo Min
Virginia Tech
Blacksburg, USA
changwoo@vt.edu

Binoy Ravindran
Virginia Tech
Blacksburg, USA
binoy@vt.edu

***Abstract*—Increasing the computing performance within a single-machine form factor is becoming increasingly difficult due to the complexities in scaling processor interconnects and coherence protocols. On the other hand, converting existing applications to run on multiple nodes requires a significant effort to rewrite application logic in distributed programming models and adapt the code to the underlying network characteristics.**

This paper presents DEX, an operating system-level approach to extend the execution boundary of existing applications over multiple machines. DEX allows the threads in a process to be relocated and distributed dynamically through a simple function call. DEX makes it trivial for developers to convert any application to be distributed over multiple nodes and for applications to transparently utilize disaggregated resources in a rack-scale system with minimal effort. Evaluation results using a running prototype and eight real applications showed promising results – six out of the eight scaled beyond the single-machine performance on DEX.

***Keywords*-Thread migration; distributed execution; distributed memory; RDMA;**

I. INTRODUCTION

Computer systems have undergone a fundamental shift to process exponentially increasing volumes of data despite the limitations in current semiconductor technologies. Data volume is projected to grow by 40% every year for the next decade [1], implying that more computational power is necessary in order to process such large volumes of data in a reasonable amount of time. Processor designers, however, are increasingly facing fundamental limitations in making faster processors under reasonable power budgets owing to the slowdown of Moore's Law, the end of Dennard Scaling, and the Dark Silicon effect. Thus, processor technologies are quickly moving toward many-core, heterogeneous, and specialized-core architectures.

Meanwhile, many memory-intensive applications such as in-memory database systems and in-memory graph processing engines have emerged to provide the ability to produce analytic results within a short latency and to process an enormous amount of data efficiently. These types of applications achieve the best performance on *scale-up* machines, which have extreme processing power and memory capacity in a single-machine form factor. However, building such a high-performance scale-up machine is expensive and becoming more difficult in practice due to the complexities of scalable interconnects and coherence protocols between CPU cores [2], [3].

Thus, we seek to explore the following question: *Can we boost applications beyond the single-machine performance while keeping the convenient programming models for a single machine?* We argue that confining the process execution boundary within a single machine limits application performance, as normal applications (in opposite to distributed applications) are inherently written to utilize only local system resources. Previous approaches to extending the execution boundary of a process to multiple nodes (i.e., utilizing remote system resources) require rewriting applications according to a distributed programming model or execution paradigm. Converting existing applications, however, requires significant modification, making it costly and oftentimes infeasible in practice.

This paper proposes a simple yet effective operating system-level approach to remove the limitations of the single-node execution boundary. DEX, short for "**D**istributed process **eX**ecution environment" is an operating system extension that enables a process to freely migrate threads on multiple machines. Any thread in a process can dynamically relocate itself to any remote node at any time by simply calling a function. Although each thread might be on a different node, threads can run as if they were on the same node; the distributed threads can transparently access consistent memory through regular load and store instructions *as is*. Moreover, the threads can synchronize with each other without modifying existing synchronization primitives, regardless of the threads' locations. These capabilities make it trivial for developers to convert existing applications written for a single machine into ones that extend their execution boundaries over multiple machines, allowing the applications to easily leverage the system resources dispersed in a rack-scale cluster and to perform beyond the performance of a single machine.

To realize this idea, we implemented a typical page-level memory consistency protocol in the Linux kernel. However, it is challenging to implement such a memory consistency protocol in the performance-critical and highly concurrent virtual memory subsystem of a contemporary operating system. In addition, using a page-level approach inevitably incurs false page sharing, in which a page containing data objects for different threads bounces between nodes, impairing application performance. We present our design and implementation of the protocol and propose application optimization techniques to mitigate false page sharing in our system. We also present the challenges to leverage emerging networking technologies (i.e., RDMA over InfiniBand) to minimize the memory consistency protocol overhead. The evaluation result using eight real applications on a running prototype indicates that our approach is practical and effective; we easily achieved a performance increase of up to 10.06× from seven applications on an eight-node rack-scale configuration.

The rest of this paper is organized as follows. Section II describes our motivation to build DeX. Section III describes the design principles and decisions for DeX and the implementation challenges and details. Section IV discusses a number of techniques to reduce the memory interference between nodes, known as false page sharing. Section V evaluates the applicability and performance of DeX using the prototype and real applications. Section VI reviews related work and compares them with the approach used by DeX. Section VII concludes the paper.

II. Motivation

We have been looking for a way to scale application performance to multiple machines while maintaining the convenience of developing applications for a single machine. Our initial idea was to provide a single system image over multiple machines so that applications could utilize system resources regardless of the locations of the resources. Obviously, such an approach is not new and has been intensively discussed in the literature, as explained in Section VI. However, the majority of these systems, including traditional distributed shared memory (DSM) systems, nowadays do not have any mindshare nor are they widely deployed in practice despite their promising features; to the best of the authors' knowledge at the time of this writing, there is no OS-level DSM system that keeps supporting contemporary OSes such as Linux. Many single-system image (SSI) system projects have been discontinued or have been inactive for years [4], [5].

We attribute the failure to the inherently poor *programmability and performance* of the systems. These systems have been proposing memory models, execution semantics, and/or programming models, which are oftentimes heavily customized for their assumptions and settings. To write an application using these systems, developers should thoroughly understand the complicated models and programming constraints, and adapting existing applications usually involves drastic modification. Such a high effort and cost are not acceptable for developers and industry in practice, and therefore these systems eventually lost attention [6].

Why did these systems propose such complicated semantics and memory models to begin with? We argue that the main reason for these complicated approaches is long network latencies. During the 1980s and 1990s, accessing data over the network took several orders of magnitude longer than accessing local data [7]. Thus, these works proposed relaxed memory consistency models and lock/release-oriented semantics in the hope of avoiding significant latency [8]–[14]. However, fast-evolving network technologies have been closing the gap between nodes, and modern interconnect technologies such as InfiniBand, RoCE, Omni-Path, and GenZ provide extremely high bandwidth (e.g., 400 Gbps) and low latency (e.g., 300 ns) [15] that approach those of inter-socket networks such as Intel UPI and AMD Infinity Fabric. In this sense, *the boundaries between a local machine and remote nodes are blurring, and we argue that it is the right moment to realign previous approaches with the modern context.*

In addition to the fast-evolving network technologies, the advancement of software architectures should also be considered. Multi-threaded applications exploiting high-core counts are ubiquitous these days. Developers should consider data placement and sharing to maximize the performance of an application even in a single machine setup. For example, non-uniform memory access (NUMA) architectures introduce bimodal memory access times, and, therefore, data location should be considered for optimally placing computation. Reckless use of locking mechanisms can easily destroy the scalability of an application [16], [17]. On the basis of this observation, we conclude that *modern performance-critical applications are already likely to be designed for scalability by considering data placement and sharing*. If this is the case, these *scale-ready* applications may easily leverage extended execution boundaries once a system enables the feature.

III. Design and Implementation of DeX

The primary objective of DeX is to transparently extend the execution boundary of a process over multiple nodes so that applications can easily utilize the dispersed resources in a rack-scale system. To this end, DeX focuses on enabling threads to be migrated across machine boundaries and to provide a consistent execution environment to these distributed threads. Figure 1 illustrates an example of how an application executes using DeX and how DeX implements the required features. From the perspective of process A, four threads are running on a single node. However, the process is actually distributed across three nodes (nodes 0,

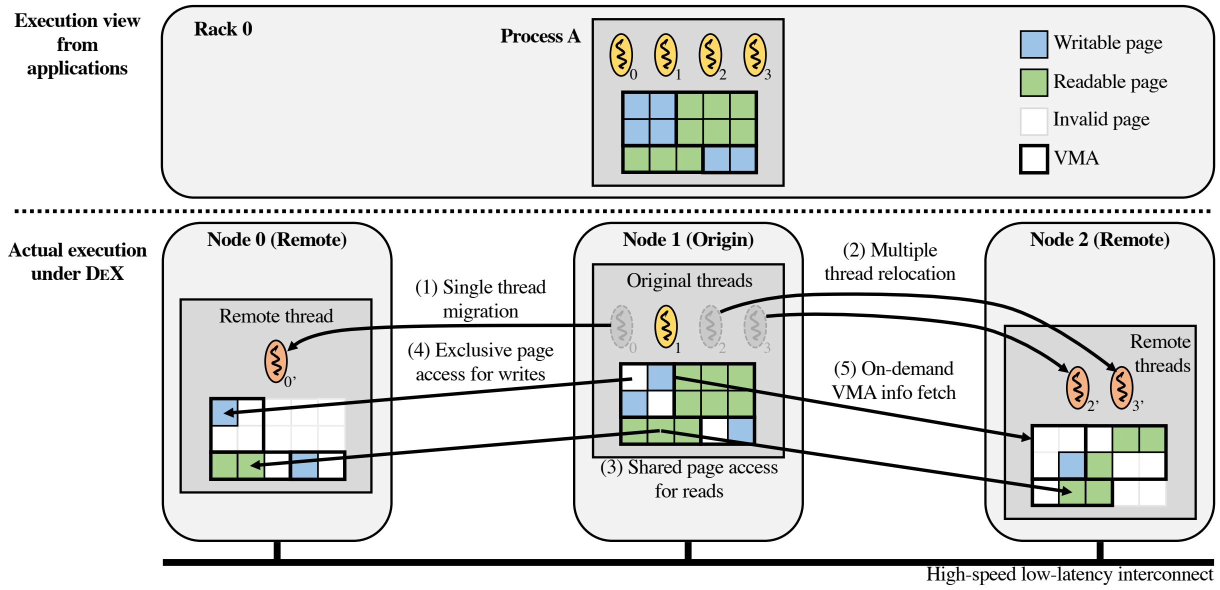
Execution view from applications
Rack 0
Process A
Writable page
Readable page
Invalid page
VMA
Actual execution under DEX
Node 0 (Remote)
Node 1 (Origin)
Node 2 (Remote)
Remote thread
Original threads
Remote threads
(1) Single thread migration
(2) Multiple thread relocation
(3) Shared page access for reads
(4) Exclusive page access for writes
(5) On-demand VMA info fetch
High-speed low-latency interconnect

Figure 1: With DEX, the threads of an application can be freely migrated to any machine at any time (① and ②; §III-A). A page-based read-replicate write-invalidate memory consistency protocol provides sequential memory consistency to the distributed threads (③ and ④; §III-B and §III-C). DEX is tightly integrated with the virtual memory subsystem inside the operating system kernel (⑤; §III-D), allowing applications to access consistent memory via regular load and store instructions. DEX utilizes a custom messaging layer to effectively exchange data between nodes using InfiniBand VERB and RDMA (§III-E).

1, and 2) through operations ① and ②. These threads can be relocated again to any node at any time. DEX provides consistent memory to distributed threads through operations ③, ④, and ⑤.

A. *Migrating Execution Contexts*

To migrate a thread across a machine boundary, we need to obtain the execution context that describes the current state of the thread at the original node. Fortunately, modern operating systems maintain such an execution context to preserve the thread state across system calls and context switches. In particular, Linux uses `struct pt_regs` and `struct mm_struct` for preserving registers and virtual address states, respectively, across system calls and context switches. We leverage these mechanisms to obtain the execution context.

DEX transfers the execution context between nodes via a messaging layer (see §III-E for details). We will refer to the node where the threads for a process are created as *the origin* of the threads. All threads in a process have the same origin, which is the node where the process is created for the first time. We will refer to the node onto which a thread is migrated as *the remote*. For example, in Figure 1, node 1 is the origin of the threads, node 0 is the remote for thread 0, and node 2 is the remote for threads 2 and 3. After sending the execution context to the remote, the *original thread* at the origin waits for incoming requests from its paired thread at the remote.

At the remote node, DEX reconstructs the original thread from the received execution context. It first creates a *remote thread* initialized using the received context. The remote thread is then put into the run queue of the job scheduler so that it can be scheduled eventually. Note that DEX only transfers the minimal execution context that is essential to start executing the thread at the remote; the virtual memory data of the application is not transferred at this stage.

Remote threads can ask their corresponding original threads to *work* at the origin on their behalf. For example, a remote thread may require virtual memory area (VMA) information from the origin to replicate its address space as shown in ③ in Figure 1. When the remote thread sends a *work request* to the origin, the request is dispatched to the original thread that was put to sleep after the migration or handling the last work request. The original thread is awaken, processes the request in the context of the original thread, and returns the result. The original thread is put to sleep again, waiting for the next request from its peer remote thread.

This *work delegation* design minimizes changes to the kernel yet transparently supports stateful OS features in the distributed environment. Practically, it is infeasible to re-implement all OS features (such as futexes and file I/O) to support a distributed execution environment. Instead, DEX reuses existing implementations through the work delegation. When a remote thread requires a stateful kernel feature, the request is handed to the original thread, performed at the origin, and only its result is transferred back to the remote thread. From the perspective of the kernel, this is identical to handling the request from a local thread. In this way, DEX can transparently provide many OS features to distributed threads without significant kernel modification. For example, DEX supports futexes (fast user-space mutex) which is the core mechanism for implementing thread synchronization primitives on Linux [18]. When a remote thread calls a thread synchronization operation, the operation is effectively translated to one or more futex system calls. The futex operations are forwarded to their original threads and handled at the origin through the original futex implementation. Thus, applications can use thread synchronization primitives based on the futex *as is, regardless of their locations*.

Despite pairing threads for work delegation, DEX still requires an additional thread at the remote for processing node-wide operations. Consider the case when a thread at the origin shrinks a VMA. The update should be applied to all remote threads in order to prevent illegal memory access operations. However, it is unclear from the perspective of the origin which remote thread should update the VMA on each remote node. For this reason, DEX creates a thread called *the remote worker* for each distributed process in each remote node. Node-wide operations such as VMA modification and original process exit are delivered to the remote worker and processed in the context of the remote worker.

Migrated execution can also be brought back to the origin. This *backward migration* is almost the same as the *forward-migration*; DEX collects the execution context of the remote thread at the remote, transfers the context to the origin, updates the context of the original thread with the up-to-date context, and resumes execution of the original thread. The remote thread exits upon completion of the backward migration. The resumed original thread may continue on to the origin or be migrated to any node again.

In the current implementation, both forward and backward migration are initiated by a system call. We believe that it can be easily extended so that OS schedulers or user-space libraries automatically initiate the migration. When a thread in a process is migrated to a node for the first time, DEX starts the remote worker with the given address space information, and forks a remote thread from the remote worker with `CLONE_THREAD`, allowing them to share the address space. Subsequent migration requests from the same process can be handled by simply forking a thread from the remote worker. In this way, DEX deals with multiple and repeated migrations with low overhead, which are commonly found in applications with multiple parallel execution regions.

B. Memory Consistency Protocol

Because threads share the same virtual address space, all threads must have the same view of memory even though they are distributed across remote nodes. There have been extensive studies on providing a consistent memory space to distributed execution contexts in DSM systems. However, as discussed in Section II, the vast majority focus on utilizing remote memory through custom memory management APIs to explicitly grab, lock, and release shared memory regions, which complicates application development and debugging significantly. Thus, we did not adopt a relaxed memory consistency model even though it is a popular, and maybe an effective, optimization method in traditional DSM systems. Instead, we fell back to the sequential memory consistency model in which distributed threads can transparently access shared memory with conventional load/store instructions.

To provide up-to-date data to distributed threads, DEX tracks the location of up-to-date pages by maintaining ownership of pages. Overall, it is similar to the multi-reader single-writer memory model in a DSM context [19]. Initially, the origin exclusively owns all pages of the process and remote nodes must contact the origin to obtain page data and page ownership. Each page can be owned by one or more nodes, and the ownership is tracked on a per-page and per-node basis at the origin. A node can continue accessing a page without contacting the origin as long as it has proper ownership of the page. If the request is for read access, the origin grants a common ownership to the remote so that both the origin and the remote can access the page simultaneously. When a remote requests a page for writing, the origin grants an exclusive ownership to it by revoking ownership from other nodes (including itself) by sending ownership revocation requests. To minimize network traffic, the origin simply grants ownership without transferring the page data when the remote already has the up-to-date one. Information such as the list of owners and page state is maintained in a per-process radix tree which indexes the information by the virtual page address.

C. Handling Concurrent Faults

The memory consistency protocol is triggered through the page fault handler in the kernel. DEX sets up page table entries (PTEs) so that access operations to pages not owned by the current node are trapped in the page fault handler. It checks whether the fault should be handled by the memory consistency protocol, and if so, the protocol handles the fault by resolving the ownership and acquiring page data from other nodes.

Modern operating systems including Linux rely on the page fault mechanism to provide various virtual memory features such as paging, copy-on-write, page sharing, kernel same-page merging (KSM) de-duplication, zero-page mapping, and page caching. Moreover, threads in a process access the address space concurrently, meaning that several page faults can be triggered simultaneously. For these reasons, page fault handling lies on the performance-critical path and operating systems are highly optimized for handling concurrent faults efficiently. To this end, the Linux kernel handles page faults optimistically and concurrently; when a fault happens, the kernel first prepares for the page by allocating physical memory, reading data from storage, or receiving data over the network, and only later updates the corresponding PTE. Page preparation can be performed concurrently, whereas the PTE update is serialized with a spinlock for correctness. If the PTE value is changed by other threads during the page preparation, the prepared page is simply discarded.

This optimistic and concurrent fault handling complicates ownership tracking in DEX. In the memory consistency protocol, page ownership must be updated during page preparation. However, it is possible that multiple threads in a node request the same page simultaneously. This can initiate multiple protocol requests, even though all per-thread requests are for the same page. This becomes more complicated if a thread must discard the fetched page owing to a changed PTE or to conflicts between threads; to safely discard the fetched page or resolve the conflicts, the kernel should maintain per-PTE metadata to identify which thread changed the PTE for whatever reason. This can slow down the page fault handler, which is not acceptable.

To effectively tame this inherent concurrency in page fault handling, we employed a *leader-follower model* in the page fault handler. DEX maintains a per-process hash table to track all ongoing fault handling. The thread that triggers the first fault for a page becomes the *leader* for the page fault handling of the page. Subsequent threads requiring the same page with the same access type (i.e., read or write) become the *followers* of the leader. The leader performs the operations to prepare the page; it brings the page from the other nodes by sending requests and invalidating. After handling the fault, the leader updates the corresponding PTE as well. Before the leader resumes its execution, it wakes up any followers. Followers do not process the fault again; instead, they simply resume execution with the updated PTE. In this way, DEX coalesces similar types of page faults and handles them with a single page fault handling.

D. *On-Demand VMA Synchronization*

The VM subsystem in Linux manages memory at two levels: at the virtual memory area (i.e., VMA) and at the page table entry (PTE). VMAs maintain the permissions, backing file, offset in the file, etc., for an address space range. On the other hand, PTEs maintain the current per-page status. Because VMA information must be shared by threads in the process, DEX requires a mechanism similar to the memory consistency protocol explained in the previous section. Threads, however, usually use disjointed VMAs (e.g., thread-local storage), making it unnecessary (even prohibitive) to fully synchronize VMAs across all threads. For these reasons, we deploy on-demand VMA synchronization.

During the execution context migration, no VMA information is transferred to the remote. When a remote thread sees a missing VMA (i.e., the address being accessed does not belong to any VMA it has), it contacts the origin to check whether the access is legitimate. If the access is to a legitimate address range and there is a VMA corresponding to the address, it implies the remote node has stale VMA information. In this case, the origin replies with the up-to-date information about the VMA, and the remote updates its VMA accordingly. If the access is invalid, the origin sends an error code to the remote which terminates the remote threads as if it performed an illegal memory access.

All VMA manipulations are performed at the origin by using the work delegation explained in §III-A. The origin only broadcasts updated VMA information when the operation shrinks a VMA region (e.g., `munmap`) or downgrades (e.g., `mprotect`) its access permissions; permissive operations (e.g., `mmap`) are not eagerly synchronized but are updated through the on-demand VMA synchronization mechanism.

E. *Inter-node Communication*

The communication layer is performance critical in distributed systems. Moreover, the communication layer should be flexible enough to support highly-concurrent and complicated communication usage in the implementation of DEX. To this end, we designed and implemented an inter-node messaging system over InfiniBand to leverage the high bandwidth and low latency of modern interconnect technologies.

At system boot-up time, nodes read in a configuration to establish a communication channel for each node pair under the InfiniBand Reliable Connection (RC) mode. Messages are routed to destination nodes through the corresponding connections, and message handlers on the nodes process the incoming messages.

Unlike in traditional sockets, I/O buffers for send and receive over InfiniBand must be explicitly mapped to a DMA-capable address space range so that an InfiniBand host controller adaptor (HCA) can perform DMA from/to the buffers. In addition, to perform remote DMA (RDMA), we must also associate the buffer with an *RDMA memory region* with a remote key, with which a remote node can perform RDMA from/to the buffer. Previous studies showed that DMA mapping and RDMA region association are costly [20]–[22]; therefore, so our design aims to minimize these operations.

In DEX, messages are bimodal in size; control messages are small, ranging up to tens of bytes, whereas page data is transferred in 4 KB messages. Owing to the high overhead of RDMA region association and the RDMA completion control path [21], transferring small messages over RDMA is too costly for DEX. Instead, DEX transfers small messages using InfiniBand VERB. To avoid the costly DMA mapping, we employ a *send buffer pool* in the messaging layer. Each connection has its own dedicated send buffer pool, which is configured during the initial setup. Internally, the pool is composed of chunks of physically contiguous pages that are mapped to DMA-capable address space ranges, and the pool manages the chunks as a ring buffer. A context (e.g., a thread trapped to the page fault handler) can allocate a buffer from the pool and compose an outbound message in the buffer. Because the buffer is DMA-ready, the message can be sent without the DMA mapping. The buffer is reclaimed by the pool when the send is completed.

Similarly, DeX maintains *receive buffer pools* for inbound messages. Each connection sets up a receive buffer pool during the initial setup phase by posting receive work requests built with DMA-mapped memory regions. An InfiniBand HCA writes the incoming data to the buffer through DMA, and notifies the host of the event through a completion queue. After processing the incoming message event, DeX recycles the DMA-ready buffer by reinitializing the receive work request with the buffer and posting the request again. In this way, DeX eliminates both DMA mapping and memory copies for small messages.

DeX leverages RDMA for transferring large messages such as page data. Unlike in domain-specific RDMA work [20]–[24], DeX must support arbitrary user applications. Thus, it is impossible to predetermine the virtual memory footprint and lifetime of the processes, which dynamically change over time and are different from each other. In addition, it is practically infeasible to keep the virtual memory address space of an application contiguous in physical memory. Thus, we rule out static approaches for RDMA memory region association which are commonly used by these domain-specific systems. On the other hand, dynamic RDMA region association is so costly that it can offset the benefit of RDMA.

Based on these observations, we devise a hybrid approach using both RDMA and memory copying. Each connection has an *RDMA sink*, which is set up during the connection initialization. The RDMA sink is composed of chunks of physically contiguous pages, and the chunks are associated with an RDMA memory region during the setup. To perform RDMA, DeX allocates a buffer from the RDMA sink and asks its peer to perform RDMA to the location. When the peer notifies an RDMA completion, the data in the RDMA sink is copied to its final destination (i.e., a page in the application virtual memory) and released. Even though this approach involves one memory copy, it is faster than performing RDMA association for each page in the memory consistency protocol.

IV. Adapting Applications

As we will show in Section V, many applications are scale-ready, so they can easily scale beyond the single machine performance on DeX. However, some applications do not scale unmodified since DeX provides memory consistency at a *page granularity*. This can lead to (1) contended pages, where conflicting accesses (read/write and write/write) to program objects on the same page cause cross-node interference, and (2) memory consistency protocol overheads, where multiple-node read, single-node write patterns cause the consistency protocol to flood the network with ownership invalidation messages. DeX provides a set of tools that help the developer identify and remove these bottlenecks in applications. Because DeX provides a shared memory programming model, developers can spend minimal effort when profiling by using the tools provided by DeX and by tweaking applications to achieve scalability on a cluster as opposed to other programming models such as MPI, which may require overhauling entire applications.

A. Profiling Page Faults

First, applications were profiled to determine which components caused the most cross-node traffic. DeX provides a profiling tool that collects a *page fault trace* containing a six-tuple for each observed page fault requiring the memory consistency protocol in DeX. Each tuple contains the system time when the page fault occurred, the node ID where the fault occurred, the task ID for the faulting task, the type of the fault (i.e., read/write/invalidate), the memory address of the faulting instruction, the memory address that caused the fault, and a user-specified identifier for tagging individual pieces of the application. DeX can be configured to collect this information for each fault and to hand it over to user-space via `ftrace`.

For profiling, applications were built with debugging information and run using DeX's profiling tool. After the execution, the profiling tool post-processes the trace in conjunction with the binary to provide a rich set of analyses, such as identifying the program objects or source code locations that caused the most page faults, page fault frequency over time, per-thread memory access patterns, etc. Using these traces, we could identify the sources of cross-node traffic and apply a set of small yet effective optimizations for better scalability.

Application data can be generally broken down into two categories: (1) global data used by all threads, which usually consists of arrays or custom data structures (e.g., graphs), and (2) per-node data, which includes per-thread data for all threads on a node and other data structures (e.g., filtered graphs in NUMA-aware applications [16] or logically partitioned heaps). Approaches for optimizing data access patterns vary for each category. For per-node data, the profiling tool helps identify data from multiple nodes placed on the same page; developers can then easily separate this data onto different pages to avoid contention. For global data, the tool helps developers find sub-optimal data access patterns that can then be optimized for scaling out. In addition, developers can express these patterns to the DeX system through data access hints to reduce protocol overheads.

B. Reducing False Page Sharing

There are several sources of false sharing common in many shared-memory applications that are easy to identify and remove using the page fault trace. The following approaches help remove false sharing caused by co-locating per-node data from multiple nodes onto the same page.

Stack. Each application thread is allocated its own local runtime stack for execution. Oftentimes, when forking child threads, however, the parent thread will pass data to children using its own stack, e.g., data pointers passed to `pthread_create` or OpenMP `shared` variables. This causes false sharing when the child threads read/write the shared program objects and the parent thread writes to its own stack at other locations on the same page. To eliminate this type of false sharing, we identified and relocated problematic stack data into global memory or pushed the data down to the thread-local storage of the child thread. For OpenMP specifically, we modified the compiler to automatically offload shared variables to global memory for the duration of parallel regions.

Global Data and Heap. Global program state, including statically and dynamically allocated data, can cause false sharing if two program objects with conflicting access types are allocated to the same page. This problem is easy to rectify: the user can simply add padding and align objects to page boundaries using the `aligned` declaration attributes for static data and allocating dynamic data using `posix_memalign`. However, blindly applying these fixes to all program objects could cause significant memory bloat. Moving every declared program object to a separate page would cause the binaries to balloon in size, and dynamically allocating every object in its own page would cause extreme internal memory fragmentation and out-of-memory errors for programs that allocate large numbers of small objects. Even worse, moving all objects to separate pages could cause performance degradation by reducing spatial locality and polluting caches. Instead of applying page alignment to every program object, we identified and selectively aligned per-node objects that caused the most interference according to the page fault trace.

C. Optimizing Global Memory

The profiling tool also helps developers optimize global memory usage for previously unseen applications by identifying source code locations where the most number of faults occur. Oftentimes two bottlenecking locations surface together – one location will incur a large number of write faults, while another incurs a correlated number of read/write faults. Another source of contention is globally-shared flags; some applications continue iterating while some condition holds (e.g., graph computation continues until no node data changed). Rather than blindly checking and setting the flag, it is often better to store each thread's flag updates locally and perform a global flag update at the end of an iteration. The tool helps identify data access patterns in the application which cause the bottleneck and correct them. Because of DeX's programmability advantages, developers can easily and progressively optimize these patterns for scaling out.

V. Evaluation

This section evaluates various aspects of DeX to answer the following questions:

- How much development effort is required to adapt existing applications to utilize DeX? [§V-A]
- How do the converted applications perform? What factors affect their performance? [§V-B]
- How can we optimize applications for DeX using our page fault profiling tool? [§V-C]
- How efficient are the key components of DeX in terms of performance and scalability? [§V-D]

Implementation. We implemented DeX in the Linux kernel version 4.4.137. The implementation consisted of 9,581 lines of added/modified kernel code, 474 lines of user-space migration library, and 1,094 lines of optimization toolchain. The full source code of DeX is publicly available as open-source as the part of the Popcorn Linux project: `http://popcornlinux.org/`.

Experimental setup. We evaluated DeX on eight servers, each of which was equipped with an Intel Xeon Silver 4110 processor running at 2.10 GHz and with 48 GB of RAM. The processor has eight cores with two-way hyper-threading (16 hardware threads in total). The nodes are connected over InfiniBand through a Mellanox ConnectX-4 VPI HCA and a Mellanox SX6012 switch, which supports bandwidths of up to 56 Gbps. The nodes mount a Network File System (NFS) share from a network-attached storage (NAS) so they can share an identical file system image, especially for executables and input data.

Benchmark applications. We evaluated DeX using eight applications in three different categories: (1) shared-memory data processing applications, (2) scientific applications, and (3) modern NUMA-aware applications.

We implemented two in-house data processing algorithms, namely, string match (GRP) and k-means clustering (KMN). String match looks up user-specified key strings from a file and counts their occurrences. The input file is divided into partitions, and each thread counts the occurrences of keys from the given partition in parallel. We used 8 GB of Wikipedia text and four key strings, each of which was 7 to 10 bytes. K-means clustering finds 100 centers of 5 million points in a three-dimensional space by repeating the computation until the vertices settle into the clusters.

For scientific applications, we picked BT, EP, and FT from the SNU NPB Benchmark Suite v3.3 [25], a C port of the NASA Parallel Benchmarks used for evaluating parallel system performance. We used the OpenMP multi-threaded implementation as our baseline and ran with the class C workload, the large problem size. We also used blackscholes (BLK) from the PARSEC Benchmark Suite v3.0 [26]. We used the gcc pthread implementation among

Application		Multithread	Changed LoC			
		Impl.	Initial		Optimized	
Simple	GRP	Pthread	+2	-0	+21	-12
	KMN	Pthread	+2	-0	+6	-3
NPB	Common		-		+1	-1
	BT	OpenMP (15)*	+38	-4	+5	-2
	EP	OpenMP (1)*	+2	-0	+2	-1
	FT	OpenMP (7)*	+28	-7	+1	-1
PARSEC	BLK	Pthread	+2	-0	-	
Polymer	Common		+18	-18	+86	-67
	BFS	Pthread	+6	-2	+10	-4
	BP	Pthread	+12	-11	+13	-10

Table I: Complexity to apply DeX to existing applications. *The number in parenthesis indicates the number of converted OpenMP parallel regions.

a number of multi-threaded implementation variants, and evaluated the 'native' workload, the largest workload the benchmark suite provides.

For modern NUMA-aware applications, we picked two graph processing applications from Polymer [16], namely, breadth-first search (BFS) and belief propagation (BP). Polymer is a graph analytic engine optimized for NUMA architectures; the applications are examples written using the framework. We ran the applications using a synthesized graph composed of 67 million vertices and 50 million edges, which is the maximum number of edges that fit in system memory. The graph was generated with the R-MAT generator in the Ligra framework [27] using the same configuration that used for the Graph500 benchmark ($\alpha = 0.57$, $\beta = 0.19$) for realistic graphs. The applications were set to iterate up to 64 iterations.

A. Adapting Applications to DeX

First, we analyzed the applicability of DeX. Initially, all applications were written for a single machine. We converted the applications to span over multiple nodes by distributing their worker threads. Each worker thread relocates itself to an assigned node at the beginning of the multi-threaded parallel execution region and returns to the origin at the end of the region. Applying this modification was straightforward, especially for those using pthreads (all but the NPB applications); we simply inserted thread migration function calls once the multi-threaded regions were identified. Each modification required only a few lines of code[1]. Specifically, as shown in Table I, we could convert GRP, KMN, and BLK by adding only one line each for the forward and the backward migration.

The NPB applications using OpenMP were converted similarly. We triggered thread migration at the beginning and end of the OpenMP parallel regions. EP has one OpenMP parallel region, and, therefore, we converted it by inserting two lines of code, one each for the forward and the backward migration. BT and FT have 15 and 7 OpenMP parallel regions, respectively, and we converted each region using the same way. This requires multiple code changes as shown in Table I. However, it requires only 2.5 to 4 lines of code per OpenMP region on average and can be done routinely, which we considered to be a negligible effort.

Because Polymer's applications are based on pthreads, we can apply the same strategy as that applied to GRP and KMN, i.e., by calling the migration functions in a worker thread. However, they require more lines than those required by other applications to replace libNUMA-specific functions (e.g., `numa_alloc_local()`) with their equivalent standard library functions (e.g., `malloc()`). Despite the additional modifications, each Polymer application can be converted with up to 12 lines of modification, most of which can be done with a simple search and replace operation.

Converting eight applications only required adding 110 lines and removing 42 lines of code in total, which is approximately 1.1% of the total lines of application source code. Because identifying the line to insert the migration function call only *requires analyzing the execution flow rather than understanding every detail of the applications*, the modification took only 3 days for one of the authors who had no prior knowledge about the applications. From this experience, we can conclude that DeX provides great programmability, allowing any application to easily extend its execution boundary with marginal effort.

B. Application Performance

We first analyzed the inherent scalability of the applications using a high-performance scale-up machine, which was equipped with eight Intel Xeon Platinum 8180 processors (224 cores in total). The times to complete application execution were inversely proportional to the number of threads for all applications. This result implies that the applications can inherently scale out as long as the system can provide more resources.

To evaluate the performance of the converted applications, we measured the time to complete application execution while increasing the number of nodes from 1 to 8. We configured the applications to use $8\times n$ threads on an n-node configuration (i.e., 64 threads for an 8-node configuration). We used 8 threads instead of 16 to avoid any side effects of hyper-threading. The symbol key 'Initial' in Figure 2 shows the performance trend normalized to the unmodified application on a single node using 8 threads. Every point is an average of four runs; their standard deviation was negligible, and is therefore not shown.

We can clearly classify the applications into two categories: those whose performance scales with respect to the number of threads and those that do not. EP, BLK, and BP scaled linearly

[1]To focus on the actual burden of use for developers, we do not count the lines when including the API header files or setting up the evaluation environment such as the number of threads and nodes.

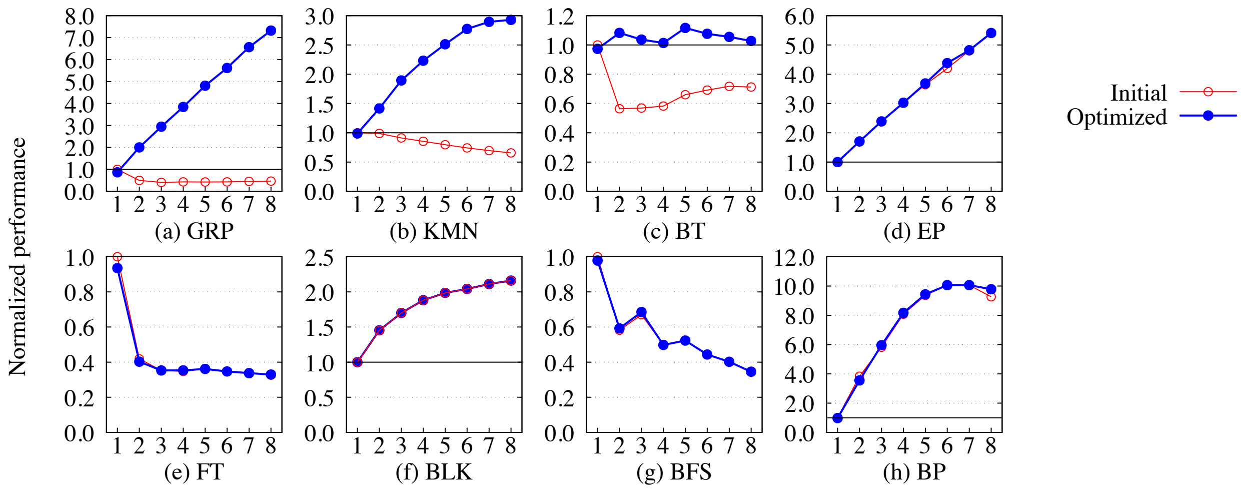

Figure 2: Scalability of applications on DEX. The x-axis is the number of nodes and the y-axis is the performance trend normalized to the original, unmodified application running on a single machine.

beyond the single-machine performance as we increased the number of nodes, with a speedup of up to a 10.06×. This implies that, while some applications are inherently scalable, but confined execution boundaries within a single machine have limited their performance. Once DEX removes this limitation, these scale-ready applications can utilize the resources of other machines immediately, achieving scalable performance. It is worth noting that BP scaled super-linearly, as its performance increased by 3.84× with the increase in nodes from 1 to 2. Further investigation showed that the CPUs were underutilized on a single-machine configuration and that the sum of CPU utilization across multiple nodes was larger than that of a single machine. This indicates that the performance of BP was not limited by the cores but by other system resources. We suspect that the limiting resource is memory channel bandwidth as BP continues accessing a large amount of memory without locality. Nevertheless, this supports our claim that DEX enables applications to utilize resources in other nodes in the rack-scale cluster.

GRP, KMN, BT, FT, and BFS performed worse on multiple nodes even with more processing resources than on a single node. Because these applications scale well in a single machine, we can infer that there is a performance bottleneck when they run on DEX. We attribute the performance degradation to the memory consistency protocol overhead between nodes. Note that, at this point, we have only converted applications to span over multiple nodes *without any optimizations to mitigate false page sharing*. Thus, these types of applications, which are not scale-out ready, continually shuffle pages between nodes with little execution progress. Moreover, we blindly inserted the migration triggers and set destinations without any in-depth analysis, making these applications run under suboptimal conditions.

C. *Optimizing Applications for* DEX

We applied the optimization techniques described in Section IV to our initial porting. We used our page fault profiling tool to identify contended access operations and other memory access patterns that could vastly degrade the performance in DEX. We found that our page fault profiling tool is quite effective, and, therefore, one author spent 4 days gathering profiling data and optimizing all applications. As shown in Table I, only 246 lines of code were modified in total across all applications. Overall, optimization does not take much effort, and we can conclude that adapting applications to DEX requires only marginal effort, similarly to converting the application to be distributed using DEX.

Figure 2 shows a comparison of the performance between our initial and optimized versions of the applications. Performance was normalized to single machine performance without any modification. Although the modifications were small, we observed drastic changes in application scalability; optimizing GRP and KMN allowed them to scale, BT achieved enhanced performance vs. its performance on a single machine, and EP, BFS, and BP improved their performance further. Using DEX, six out of eight applications scaled beyond a single machine. This indicates that many applications are almost scale-ready and that DEX allows developers to easily identify and remove bottlenecks to enhance the application to scale out to multiple nodes.

To detail our optimization, we had GRP and KMN place all thread arguments in an array contained on a single page, and allocate per-thread buffers from the heap without considering the locations of other thread buffers. We separated these thread arguments and buffers by replacing the calls to `malloc` with `posix_memalign` for page-aligned heap allocations. Additionally, the original implementations interfere with

	Forward migration			Backward migration		
	Origin→Remote		Total	Remote→Origin		Total
1st	12.1	800.0	812.1	6.4	18.3	24.7
2nd	6.6	230.0	236.6	8.4	17.4	25.8

Table II: Migration latency in microseconds

global variables — GRP updates a global variable when it finds an occurrence of a key, and KMN updates a global flag and the clusters for points. We changed the implementation so that each thread stages its updates locally before updating the shared global variables once after the computation.

Our tool identified that NPB applications continually read global parameters, especially variables containing for-loop ranges of parallel regions. These variables are intensively accessed and are read-only after the initial setup but are co-located with other global variables that are frequently updated. We relocated these read-only global variables onto separate pages so that they are replicated across nodes and not invalidated by nearby writes. Moreover, in BT, child threads in a number of parallel regions read their parent's stack variables. To prevent interference on the parent's stack, we explicitly passed these variables to child threads as arguments.

In Polymer, the framework allocates a number of data objects as arrays, which incur false sharing when different threads access each type of data object. We packed these data objects into a per-node data structure to avoid false sharing and isolate them within a node. In addition, we aligned the thread arguments and graph data to page boundaries in order to minimize cross-node references.

Some applications showed degraded performance as the number of nodes is increased greatly. We attribute the degradation to the small working data size per core. Each core will be assigned with a partition of the graph, and the partition will get smaller as we put more cores. The smaller the graph partition given to a core is, the faster the core will finish the computation; however, at some point, distributing graph partitions incurs more overhead than improved processing performance. We observed a similar trend from KMN and BLK, and we believe that the performance improvement will be continue with a larger graph workload.

D. Performance of DEX*'s Mechanisms*

Thread migration overhead.

To measure the thread migration overhead, we used a microbenchmark that repeatedly migrates a thread every second. We measured the time for handling thread migration at both the origin and the remote node. Table II summarizes the results using the average of 10 thread migrations.

The first forward migration took 812.1μs in total, whereas the first backward migration only took 24.7μs in total. When the thread was migrated to the same node again, the second forward migration only took 230.0μs, which is only 28.3% of the total time of the first migration. The time of the second

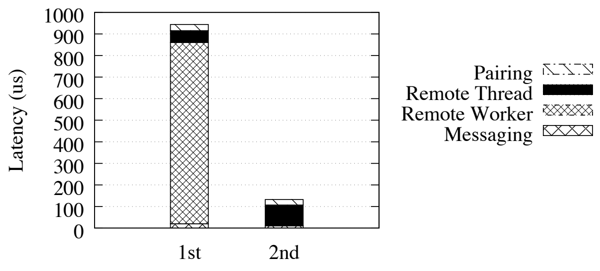

Figure 3: Breakdown of the migration latency at the remote node

backward migration was almost the same as that of the first backward migration. Subsequent migrations to the same node were similar to the latencies of the second migration.

The latency difference between the first and the second forward migration comes from the time to construct the per-process data structures. During the first migration, DEX handles most of the per-process procedures such as creating the remote worker, setting up the address space, and setting up other process-level data structures. The result of an evaluation that breaks down the latency is shown in Figure 3, which confirms that 620.0μs of the first migration was taken up by the per-process procedures, denoted by the symbol key "Remote Worker".

The latency discrepancy between the forward and the backward migration latency comes from the amount of work required for each migration type. During the forward migration, DEX creates threads and sets them up using the original execution context. In contrast, the backward migration simply requires updating the status of the original thread, which is significantly faster than in the forward migration.

Page fault handling overhead. To analyze the page fault handling performance, we created another microbenchmark that forks two threads and relocates one of them to a remote. Both threads then continually update a single global variable, stressing the memory consistency protocol to shuffle a page between the nodes for exclusive ownership. We collected around 154,676 page faults from the origin during a 30-second execution of the microbenchmark.

We observed a bimodal distribution of the fault handling time; although our messaging layer constantly took 13.6μs to retrieve a 4 KB page, 27.5% of the faults were handled in 19.3μs. However, when both nodes contended for the same page and one of the nodes had to fall back to retry, the fault handing extended to 158.8μs on average. This implies that reducing the false page sharing is critical in DEX because it not only accelerates fault handling but also reduces its occurrences.

VI. Related Work

Distributed shared memory (DSM) systems provide a consistent memory view to distributed execution contexts (i.e., processes and threads) across multiple machines, and have been thoroughly discussed in the past. The vast majority of these systems focus on utilizing remote memory through custom memory management APIs to explicitly grab, lock, and release shared memory regions [8]–[14]. Oftentimes, such APIs limit the type of virtual memory that can be shared between distributed contexts (e.g., only heap-allocated data can be shared) and only guarantee a relaxed memory consistency from shared memory regions. This requires developers to write applications using APIs and semantics tailored to the memory model of each system, which complicates application development and debugging. DeX is unique compared with previous DSM work in that distributed threads can transparently access consistent memory as-is, without rewriting applications for remote memory accesses and distributed synchronization. In addition, distributed threads can use synchronization primitives as is regardless of their actual location.

The majority of DSM systems focus on sharing data between processes; only a few consider threads in a DSM context [9], [28]. In general, the latter only support static thread placement, where a remotely created thread cannot be relocated once it is spawned on a node. Additionally, they require application refactoring to place data in shareable virtual memory regions for access after migration [28]. In contrast, DeX allows threads to dynamically place themselves, thereby greatly improving flexibility and programmability.

Recently proposed disaggregated memory systems and the like leverage modern high-speed low-latency interconnects to provide applications with a large volume of memory beyond what is available in a single machine. In particular, Grappa [29], LITE [22], HotPot [24], Remote Regions [30], and LegoOS [3] inspire us by exploring emerging memory system architectures with RDMA-capable interconnect and/or the DSM concept in the modern context. Although they show promising results, they do not allow developers to leverage the simplicity and efficiency of a scale-up design for a single machine; each application has to consider the low-level details of the underlying network [20], [21], [24], and/or developers must redesign entire applications from the ground up [21], [29], [30]. For example, in Grappa [29], developers must completely rewrite applications using their data addressing modes, delegation operations, and communication interfaces built on an MPI programming model. This impairs programmability [31]–[33], making it difficult to adapt existing applications to the framework. In addition, many disaggregated memory systems do not provide a mechanism to utilize remote system resources other than memory (i.e., computing power of processors and underlying storage); thus, applications must manually be distributed or else the system resources remain underutilized [3].

Single-system image (SSI) systems feature flexible process placement and migration to effectively utilize clusters by balancing the load between nodes. The majority of these systems, however, work at the process level; a process cannot simultaneously utilize multiple nodes but can only run on a single node at a given moment. Thus, application performance is limited to single-machine performance even if there are idle nodes in the cluster. Kerrighed [34], [35] uniquely supports thread-level migration to deal with this case; however, like traditional DSM systems, it requires explicitly declared memory regions to share data between distributed threads. This again impairs programmability and imposes overheads for rewriting applications. Relocating running contexts is also extensively studied in the context of virtualization [36]–[39] and checkpoint/restart systems [40]–[42]. However, they also cannot utilize multiple nodes simultaneously either, limiting the execution boundary to a single machine at any given moment. vNUMA [43] proposes a hypervisor-level DSM system, however, it is unclear how it handles crucial OS-level features such as futex in a distributed way. ScaleMP [44] allows a software-defined server from multiple nodes by leveraging virtualization technologies. Even though ScaleMP provides very similar features to those of DeX, its internals are unclear as it is proprietary software.

VII. Conclusion

We introduced DeX, a Linux kernel extension that allows an application to expand its execution boundary beyond a single machine. Any application can be converted to span its execution over multiple nodes through a simple function call. The evaluation result using a number of realistic applications confirms that DeX provides an intuitive yet effective way to utilize dispersed resources in a rack-scale system.

We believe that the execution relocation capability of DeX can be leveraged in a number of scenarios, such as relocating the computation near data, accelerating the computation through offloading, and saving energy by using nodes with heterogeneous power profiles.

Acknowledgement

This work is supported in part by grants received by Virginia Tech including that from the US Office of Naval Research (ONR) under grants N00014-16-1-2711, N00014-16-1-2104, and N00014-18-1-2022, and from NAVSEA/NEEC under grant N00174-16-C-0018. Kim's work at Virginia Tech (former affiliation) was supported by ONR under grants N00014-16-1-2711 and N00014-18-1-2022.

This work is also supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government (20ZS1310).

References

[1] IDC, "The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things," Apr. 2014, https://tinyurl.com/ya8oasf8.

[2] T. P. Morgan, "HPE fills a NUMA server gap with SGI UV Iron," Feb. 2016, https://tinyurl.com/y8g8exvq.

[3] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A disseminated, distributed OS for hardware resource disaggregation," in *Proc. 13rd OSDI*. USENIX Association, Oct. 2018, pp. 69–87.

[4] M. Bar, "openMosix project shutting down," https://sourceforge.net/p/openmosix/news/2008/02/openmosix-project-ends/, Mar. 2008.

[5] Kerrighed, "Kerrighed: linux clusters made easy," http://www.kerrighed.org/wiki/index.php, Sep. 2010.

[6] W. Zwaenepoel, "P2P, DSM, and other products of the complexity factory," in *Proc. 2010 ACM SAC*, 2010.

[7] C. Vilett, "Moore's Law vs. storage improvements vs. optical improvements," *Scientific American*, Jan. 2001.

[8] Y. Zhou, L. Iftode, J. P. Sing, K. Li, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood, "Relaxed consistency and coherence granularity in DSM systems: A performance evaluation," in *Proc. 6th PPoPP*, Jun. 1997, pp. 193–205.

[9] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Proc. 2nd PPoPP*, Mar. 1990, pp. 168–176.

[10] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared memory computing on networks of workstations," *Computer*, vol. 29, no. 2, pp. 18–28, Feb. 1996.

[11] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems*, vol. 7, no. 4, pp. 321–359, 1989.

[12] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *Proc. 19th ISCA*, May 1992, pp. 13–21.

[13] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The midway distributed shared memory system," in *Compcon Spring '93, Digest of Papers*, feb 1993.

[14] Y. Zhou, L. Iftode, and K. Li, "Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems," in *Proc. 2nd OSDI*. USENIX Association, Oct. 1996, pp. 75–88.

[15] Mellanox, "ConnectX-6 EN 200Gb/s Adapter Card," 2016, https://tinyurl.com/yaaxt2yj.

[16] K. Zhang, R. Chen, and H. Chen, "NUMA-aware graph-structured analytics," in *Proc. 20th PPoPP*, Feb. 2015, pp. 183–193.

[17] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of linux scalability to many cores," in *Proc. 9th OSDI*. USENIX Association, Oct. 2010.

[18] H. Franke, R. Russell, and M. Kirwood, "Fuss, futexes and furwocks: Fast userlevel locking in linux," in *Proc. Ottawa Linux Symposium*, Jun. 2002.

[19] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed shared memory: concepts and systems," *IEEE Parallel Distributed Technology: Systems Applications*, vol. 4, no. 2, pp. 63–71, 1996.

[20] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *Proc. 11st NSDI*, Mar. 2014, pp. 401–414.

[21] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, scalable and simple distributed transactions with two-sided RDMA datagram RPCs," in *Proc. 12nd OSDI*. USENIX Association, Nov. 2016, pp. 185–201.

[22] S.-Y. Tsai and Y. Zhang, "LITE kernel RDMA support for datacenter applications," in *Proc. 26th SOSP*, Oct. 2017, pp. 306–324.

[23] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an RDMA-enabled distributed persistent memory file system," in *Proc. 2017 USENIX ATC*. USENIX Association, Jul. 2017.

[24] Y. Shan, S.-Y. Tsai, and Y. Zhang, "Distributed shared persistent memory," in *Proc. 8th SoCC*, Sep. 2017.

[25] Seoul National University Centers for Manycore Programming, "SNU NPB suite," http://aces.snu.ac.kr/software/snu-npb/.

[26] Princeton University, "The PARSEC benchmark suite," http://parsec.cs.princeton.edu/.

[27] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. 18th PPoPP*, Feb. 2013, pp. 135—-146.

[28] A. Itzkovitz, A. Schuster, and L. Shalev, "Thread migration and its applications in distributed shared memory systems," *Journal of Systems and Software*, vol. 42, pp. 71–87, 1998.

[29] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant software distributed shared memory," in *Proc. 2015 USENIX ATC*. USENIX Association, Jul. 2015, pp. 291–305.

[30] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei, "Remote regions: a simple abstraction for remote memory," in *Proc. 2018 USENIX ATC*. USENIX Association, Jul. 2018.

[31] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi, "Productivity analysis of the UPC language," in *Proc. 18th IPDPS*, Apr. 2004.

[32] J. Silcock and A. Gościński, *Message passing, remote procedure calls and distripbuted shared memory as communication paradigms for distributed systems*. Deakin University, 1995.

[33] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran, “Popcorn: Bridging the programmability gap in heterogeneous-ISA platforms,” in *Proc. 10th EuroSys*, Apr. 2015.

[34] G. Vallée, R. Lottiaux, L. Rilling, J.-Y. Berthou, I. D. Malhen, and C. Morin, “A case for single system image cluster operating systems: the kerrighed approach,” *Parallel Processing Letters*, vol. 13, no. 02, pp. 95–122, Jun. 2003.

[35] C. Morin, P. Gallard, R. Lottiaux, and G. Vallée, “Towards an efficient single system image cluster operating system,” in *Proc. 5th Int'l Conference on Algorithms and Architectures for Parallel Processing*, Oct. 2002, pp. 370–377.

[36] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand, “Parallax: managing storage for a million machines,” in *Proc. 10th USENIX Workshop on Hot Topics in Operating Systems (HotOS X)*, Jun. 2005.

[37] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, “Live migration of virtual machines,” in *Proc. 2nd NSDI*, May 2005, pp. 273–286.

[38] VMware, Inc., “Live migration for virtual machines without service interruption,” 2009.

[39] M. R. Hines and K. Gopalan, “Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning,” in *Proc. 2009 ACM SIGPLAN/SIGOPS VEE*, 2009, pp. 51–60.

[40] CRIU, “CRIU project website,” Online: http://criu.org, accessed Jan 2020.

[41] A. Mirkin, A. Kuznetsov, and K. Kolyshkin, “Containers checkpointing and live migration,” in *Proc. Ottawa Linux Symposium*, vol. 2, Jul. 2008, pp. 85–90.

[42] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, “Process migration,” *ACM Computing Surveys*, vol. 32, no. 3, pp. 241–299, 2000.

[43] M. Chapman and G. Heiser, “vNUMA: A virtual shared-memory multiprocessor,” in *Proc. 2019 USENIX ATC*. USENIX Association, Jul. 2019.

[44] ScaleMP, Inc., “Virtualization for high-end computing,” http://www.scalemp.com.