On Optimizing and Leveraging Distributed Shared Memory: High-Performant Sequential Memory, Relaxed-consistent Memory, and Distributed Hypervisor for Resource Aggregation

Ho-Ren Chuang

Preliminary Exam

Doctor of Philosophy
in
Computer Engineering

May 06, 2020
Blacksburg, Virginia

Keywords: Distributed Shared Memory, Virtualization, Scale-out, Non-cache-coherent Domains

On Optimizing and Leveraging Distributed Shared Memory: High-Performant Sequential Memory, Relaxed-consistent Memory, and Distributed Hypervisor for Resource Aggregation

Ho-Ren Chuang

(ABSTRACT)

This dissertation focuses on the problem space of heterogeneous-ISA multiprocessors – an architectural design point that is being studied by the academic research community and increasingly available in commodity systems. Since such architectures usually lack globally coherent shared memory, software-based distributed shared memory (DSM) is often used to provide the illusion of such a memory. The DSM abstraction typically provides this illusion using a reader-replicate, writer-invalidate memory consistency protocol that operates at the granularity of memory pages and is usually implemented as a first-class operating system abstraction. This enables symmetric multiprocessing (SMP) programming frameworks, augmented with a heterogeneous-ISA compiler, to use CPU cores of different ISAs for parallel computations as if they are of the same ISA, improving programmability, especially for legacy SMP applications which therefore can run as-is on such hardware.

Past DSMs have been plagued by poor performance, in part due to the high latency and low bandwidth of interconnect network infrastructures. The dissertation revisits DSM in light of modern interconnects that reverse this performance trend. The dissertation presents Xfetch, a bulk page prefetching mechanism designed for the DEX DSM system. Xfetch exploits spatial locality, and aggressively and sequentially prefetches pages before potential read faults, improving DSM performance. Our experimental evaluations reveal that Xfetch achieves up to ≈142% speedup over the baseline DEX DSM that does not prefetch page data.

SMP programming models often allow primitives that permit weaker memory consistency semantics, where synchronization updates can be delayed, permitting greater parallelism and thereby higher performance. Inspired by such primitives, the dissertation presents a DSM protocol called MWPF that trades-off memory consistency for higher performance in select SMP code regions, targeting heterogeneous-ISA multiprocessor systems. MWPF also overcomes performance bottlenecks of past DSM systems for heterogeneous-ISA multiprocessors such as due to significant number of invalidation messages, false page sharing, large number of read page faults, and large synchronization overheads by using efficient protocol primitives that delay and batch invalidation messages, aggressively prefetch data pages, and perform cross-domain synchronization with low overhead. Our experimental evaluations reveal that MWPF achieves, on average, 11% speedup over the baseline DSM implementation.

Finally, the dissertation presents PuzzleHype, a distributed hypervisor that enables a single virtual machine to use fragmented resources in distributed virtualized settings such as CPU cores, memory, and devices of different physical hosts, and thereby decrease resource fragmentation and increase resource utilization. PuzzleHype leverages DSM implemented in the host operating systems to present an unified and consistent view of a continuous pseudo-physical address space to the guest operating systems. To transparently utilize CPU and I/O resources, PuzzleHype integrates multiple physical CPUs into a single VM by migrating threads, forwarding interrupts, and by delegating I/O. Our experimental evaluations reveal that PuzzleHype yields speedups in the range of 355%–173% over baseline over-provisioning scenarios which are otherwise necessary due to resource fragmentation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last decade, the computer architecture landscape has changed dramatically [20, 80, 83]. Single-threaded CPUs made the way for chip multiprocessors of ever-increasing core counts; these in turn saw the rise of heterogeneous computing using GPUs and programmable hardware [35]. These changes arose from the limits of single-threaded CPU performance and the difficulty of packing more cores in the same chip: after reaching a plateau in clock speeds, architects packed more transistors into the same chip, but as the transistor count increased, the heat dissipation became unmanageable. The "end of Moore's Law" [63, 79] forced chip vendors to advance performance and energy efficiency boundaries elsewhere. This has resulted in an array of radically different hardware: multicore [67, 157, 171] and manycore chips [35, 75, 152, 172] that exploit hardware parallelism; CPUs with heterogeneous micro-architectural properties [95, 146], partially overlapping instruction-set-architectures (ISAs) [92], and various forms of accelerators and programmable hardware [71] that exploit heterogeneity.

Yet despite the trend toward increased heterogeneity, commodity heterogeneous CPUs largely deploy CPU cores of the same ISA, or different ISA extensions, or microarchitecture (e.g., ARM's big.LITTLE [74], NVidia's Kal-El [139], Intel's Foveros [88]). A notable exception is MPSoC [177]. The academic research community, however, has explored alternative points in the architecture design space, including heterogeneous-ISA cores. Exploration in this particular design space includes many forms – shared-memory chip multiprocessors [25, 183], multiprocessors with multiple cache-coherent domains (and no coherence between domains) [113], and composite-ISA cores [185] – across many settings – ranging from cluster architectures [25, 140] to mobile platforms [105]. All of these works explore architectures that use cores belonging to different ISA families such as x86 [122], Alpha [50], ARM [1], and ARM Thumb [81].

Industry trends are fast changing. With the advent of ARM-based high-end servers [1, 16, 120, 121] capable of powering high-performance computing (HPC) applications, third-party organizations such as datacenter providers and cloud providers are increasingly integrating machines of different ISA families in their computing installations [14]. Chip vendors are also increasingly integrating processors of different ISA families in the same SoC – e.g., Intel Skylake processor with in-package FPGA [64, 90] enables synthesizing RISC-V and x86 soft cores, with hardware support for virtualization, AMD's new generation x86 processor integrates ARM cores; or on the same platform – e.g., smart NICs integrate ARM [77, 133], MIPS64 [119], or Tile cores [129].

This radical change in hardware has profound implications for software systems: hardware-agnostic programming – the traditional programming approach – cannot improve performance anymore. Improving performance now requires exposing greater degrees of parallelism in software that is suited for the hardware at hand – e.g., task parallelism for homogeneous multicore architectures [142]; SIMD parallelism for GPU-based architectures [166]. This challenges programmability. The reduced programmability is a serious detriment to software development, verification, and maintenance.

Infrastructure software, in particular, compilers, language run-times, runtime libraries, operating systems, and hypervisors are foremost impacted by emerging hardware trends as they directly run on, or close to, bare metal and export programming abstractions for building application software. Thus, compilers, run-times, operating systems, and hypervisors are continuously evolving to exploit various types and degrees of hardware parallelism and heterogeneity, and export convenient to use programming abstractions [24, 25, 27, 100, 167, 168].

Motivated by the trend of heterogeneous ISAs and the need to address the ensuing programmability challenges, the literature presents a number of efforts. Example architectural design points that have been studied include shared-memory chip multiprocessors [25, 184, 186], multiprocessors with multiple cache-coherent domains [113], and composite-ISA cores [185]. This dissertation focuses on the space of *heterogeneous-ISA* multiprocessors with multiple cache-coherent domains [113], wherein each domain has hardware supported cache-coherency, but no coherency exists between domains. This model is exemplified in architectures such as Intel SCC, Intel's Xeon/Xeon-Phi [47], and mobile SoCs such as OMAP4 [178], OMAP5 [179], and Samsung Exynos [48, 159]. In this model, efforts such as Popcorn Linux [25], K2 [113], Reflex [112], and ADSM [69] consider a shared-nothing operating system (OS) model that runs multiple OS kernel instances in each coherency domain, but gives the illusion of a monolithic, task-based, single OS image abstraction. Thus, application software is unaware of the distributed nature of the underlying OS, and is presented with an OS interface that is indistinguishable from the traditional symmetric multiprocessing (SMP) operating system interface.

The single OS abstraction of Popcorn Linux [25], K2 [113], Reflex [112], and ADSM [69] is implemented using a software-based *distributed shared memory* (or DSM) abstraction. The DSM abstraction – a mechanism that has a very long history and extensive literature [17, 28, 59, 65, 69, 108, 114, 132, 194, 196] – provides a logically global shared memory model of a physically distributed memory system. The abstraction provides programmers with the illusion of a single address space across multiple machines (or domains), each of which has its own memory, often cache-coherent. The consistency of the global memory state across such multiple cache-coherency domains is often ensured using a reader-replicate, writer-invalidate memory consistency protocol [17, 108], typically operating at the page-level granularity. This allows SMP programming frameworks, which have become popular as shared memory multiprocessors have become mainstream, such as Intel TBB [154], Intel

Cilk [153], and OpenMP [54] to view the multiple domains as a shared memory SMP system and use the CPU cores of the different domains for data- and task-parallel computations as if the cores belong to a single coherent domain. Without DSM, developing applications for such architectures would require using programming models that expose the separate physical memory regions, such as the message passing interface (MPI) [76], compute unified device architecture (CUDA) [160], or the partitioned global address space (PGAS) [39, 43, 132] models. For legacy SMP applications, this can be significantly expensive as it will require a full application re-write [24]. Thus, DSM is the key abstraction that enables systems such as [25, 69, 112, 113] to run SMP applications, including legacy ones, as-is over non-coherent domains, yielding their high programmability. (Chapter 2 overviews DSM.)

1.1 Motivations

The performance limitations of DSM systems are well known [17]: they have been historically plagued by network infrastructures with high latencies and low bandwidth, yielding poor performance when large amounts of application data and memory synchronization traffic are sent over such networks. However, new networking technologies continue to improve in both latency and throughput – e.g., commodity InfiniBand adapters provide 200 Gbps bandwidth [87]. With increasing high-speed interconnects, we believe that it is time to revisit DSM in order to improve its performance. This trend is already evident in the literature [9, 24, 25, 38, 69, 112, 113, 132, 140, 163]. Along the same line, this dissertation revisits DSM and proposes mechanisms to improve performance, specifically targeting heterogeneous-ISA multiprocessor systems with non-coherent domains. The dissertation proposes an aggressive page prefetching mechanism to improve the performance of the DEX DSM system [161].

SMP programming models such as Intel TBB [154], Intel Cilk [153], and OpenMP [54] provide a set of primitives to easily spawn multiple threads, distribute parallel work, and synchronize execution. When using these primitives, developers must write their applications in accordance with the programming model's memory consistency semantics. Oftentimes they use weak memory consistency, where updates are only made visible after synchronization operations. For example, OpenMP defines a "flush" operation [118] that determines when writes by a thread must be made visible to other threads; reads and writes to the same memory address not ordered by a flush operation result in undefined behavior. This forces developers to write computations that avoid such data races and thus are amenable to parallelization. Inspired by such SMP operations that permit weaker memory consistency, we develop a DSM protocol called MWPF that trades-off memory consistency for higher performance in select (SMP) code regions, also targeting heterogeneous-ISA multiprocessor systems.

DSM can be used to aggregate fragmented resources in virtualized settings, especially in cloud computing environments. Resource fragmentation in datacenters is a well known problem [70, 73, 136, 151, 176] that is caused by multiple factors. First, datacenter jobs have wide variety of resource requirements. Second, the granularity of their resource requirements (i.e., cores, RAM, etc.) is different from the granularity at which resources are usually allocated in datacenters (i.e., physical servers [164]). In addition to basic resource needs, datacenter jobs have platform dependencies such as the host types they can only run on, the OS kernel version they need, and microarchitecture constraints [155, 165]. Fragmentation is further aggravated by the difficulty of precisely estimating a job's resource needs [155]. This often results in over-provisioning [56], a common practice employed by many cloud vendors to ensure sufficient quality of service (QoS) during workload spikes. State-of-the-art solutions to this problem include optimized Virtual Machine (VM) placement methods, resource defragmentation through VM migration, and hardware disaggregation [66, 78, 110, 111, 164, 176]. These solutions are either not entirely effective [70, 147, 164, 176] or require newer hardware [164, 176]. The dissertation presents PuzzleHype, a distributed hypervisor that enables a VM to leverage fragmented resources – cores, memory, devices – belonging to different physical hosts, and thereby decrease fragmentation and increase resource utilization. PuzzleHype leverages DSM implemented in the host OSes to present an unified and consistent view of a continuous pseudo-physical address space to the guest OSes.

1.2 Dissertation Contributions

We summarize the dissertation contributions as follows:

1.2.1 The Xfetch Prefetch Mechanism for DEX DSM

We present a transparent bulk page prefetch mechanism called Xfetch that improves DSM performance (Chapter 4). We consider the DEX DSM system [161] which implements DSM as a first-class Linux OS kernel abstraction. DEX provides sequential memory consistency using a page-level, single-writer multiple-reader invalidation-based protocol. We consider the InfiniBand RDMA networking infrastructure. We exploit spatial locality, aggressively and sequentially prefetching pages before they are accessed, reducing read page faults in subsequent execution. Our experimental evaluations using an eight-node, 56 Gbps InfiniBand-based rack-scale system reveal that Xfetch achieves up to 142.74% speedup over the baseline DEX that does not prefetch page data.

1.2.2 The MWPF DSM Protocol

We present MWPF, an efficient multiple writers DSM protocol that allows many writers to concurrently write to the same page without coherency (Chapter 5). MWPF is inspired by OpenMP's flush operation that allows a thread to determine when its results are visible to other threads in an OpenMP work-sharing region, permitting delayed synchronization, greater parallelism, and thus higher performance. Since a multiple writer protocol may not pay off in all scenarios, MWPF uses a heuristic called *smart regions* that select the best consistency protocol for a given OpenMP work-sharing region.

MWPF also overcomes several performance bottlenecks that plague past DSMs that implement DSM as a first-class (Linux) OS kernel abstraction [25, 140, 158], targeting heterogeneous-ISA multiprocessors with non-coherent domains: significant number of invalidation messages, false page sharing, large number of read page faults, and large synchronization overheads. MWPF overcomes these performance challenges by using efficient DSM protocol primitives that delay and batch invalidation messages, aggressively prefetch data pages, and perform cross-domain synchronization with low overhead.

Our experimental evaluations, conducted using a two-node, non-coherent heterogeneous-ISA multiprocessor with 56 Gbps InfiniBand interconnect, reveal that MWPF achieves, on average, 11% speedup over the baseline DSM implementation.

1.2.3 The PuzzleHype Distributed Hypervisor

We present PuzzleHype, a distributed hypervisor that enables the classical unit of job execution in a cloud setting – the Virtual Machine (VM) – to transparently use fragmented resources belonging to different physical hosts (Chapter 7). PuzzleHype runs as one instance per host and allows the creation of virtual CPUs (vCPUs), pseudo-physical memory, and I/O devices, and exports them to guest OSes as an unified host abstraction. PuzzleHype leverages DSM, implemented as a first-class (Linux) OS kernel abstraction of the hosts, to export an unified and consistent view of a continuous pseudo-physical address space to guest OSes. To transparently utilize CPU and I/O resources, PuzzleHype integrates multiple physical CPUs together into a single VM by adopting thread migration and supporting interrupt forwarding. In addition, PuzzleHype enables a vCPU to access remote host hardware I/O resources (e.g., NIC) by supporting I/O delegations. This is accomplished by modifying the *virtio*/*virtio-net* paravirtualized architecture [102, 143].

By abstracting the distribution of cores, memory, and devices into a system-level hardware interface similar to that of a classical VM, PuzzleHype is *transparent*: it can execute fully unmodified guest operating systems within a distributed VM, although some slight modifications to these OSes can bring significant performance improvements. More importantly, PuzzleHype does not require any modification to legacy user-space software, which can run as-is. By leveraging scattered resources that naturally occur in datacenters, PuzzleHype decreases resource fragmentation and thus increases resource utilization. It can either help to increase throughput by running more jobs on a given set of hosts, or it can reduce cost and power consumption by running a given set of jobs on a smaller set of hosts.

Our experimental evaluations, conducted using a rack-scale system with four virtual CPUs, reveal that PuzzleHype yields significant advantages: a speedup of up to 355%, and on average 285%, for the SNU NPB benchmark suite [162]; a speedup of 225% for a customized LEMP [3] stack; and 173% speedup for the OpenLambda [82] benchmark suite, all over a baseline over-provisioning scenario such as [56].

1.3 Dissertation Organization

The rest of the dissertation is organized as follows: Chapter 2 presents background information necessary to understand the dissertation contributions including the DSM abstraction and virtualization technologies. Chapter 3 discusses past and related works in the dissertation's problem spaces. Chapter 4 describes and evaluates the Xprefetch mechanism for the DEX DSM. Chapter 5 describes the MWPF DSM protocol; its evaluation results are presented in Chapter 6 . Chapter 7 and Chapter 8 describe PuzzleHype and its evaluation results, respectively. Finally, Chapter 9 concludes the dissertation and proposes post-preliminary examination work.

Chapter 2

Background

This chapter provides background information that will allow readers to better understand the main contributions of this dissertation. Our designs and implementations of Xfetch, MWPF, PuzzleHype are based on Distributed Shared Memory (DSM) (Section 2.1) and messaging layer over InfiniBand/RDMA (Section 2.5). In Chapter 5 and 6, MWPF prototype is built on Popcorn (Section 2.3). DEX (Section 2.2) is leveraged for building a prototype of PuzzleHype (Chapter 7 and 8). General virtualization techniques widely used in many areas are introduced in Section 2.4. Our PuzzleHype (Chapter 7 and 8) is designed and implemented based on the virtualization context and techniques.

Section 2.1 gives an introduction of DSM architecture and its theory and implementation. System overviews of DEX and Popcorn are presented in Section 2.2 and Section 2.3, respectively. Section 2.4 introduces different types of virtualization technologies. Section 2.5 explains prior knowledge of InfiniBand and RDMA.

2.1 Distributed Shared Memory (DSM)

Distributed Shared Memory (DSM) [137, 150, 169] abstraction is logically a shared memory model on a physical distributed memory system where memories are physically distributed among multiple nodes. This abstraction provides programmers with a single address space illusion for a Distributed Global Address Space (DGAS) where developers work as if they were coding in a multiprocessor computer sharing a main memory. Implementation categories of DSM are: hardware [57, 107], software [13, 17, 22, 28, 65, 108, 196, 196], and hybrid [33, 41, 46, 101, 192]. Hardware implementations usually rely on special network or caching hardware [57, 107]. It extends traditional cache coherence techniques to multiple domains which do not have a hardware supported cache-coherency between them. Development time takes longer. It maintains coherence with more fine-grain data size. Similar to hardware implementations, most of hybrid implementations [33, 41, 46, 192] relies on hardware-based coherence protocol.

Unlike hardware and hybrid DSMs, software DSM does not require specific hardware to assist DSM protocol in helping distributed memory coherency. Since hardware and hybrid DSMs require special non-commodity components and thus cost more [110]. This makes hardware solutions less promising and considered in recent academic researches. On the other hand, software DSM is more flexible and portable. For example, it can be implemented on commodity hardware. The implementation can be easily updated, such as synchronization granularity, and it is not constrained by special hardware features. Having said that, this dissertation focuses on software DSM.

2.1.1 Software DSM

Software DSM (SDSM) implementations can be further split into compiler, library, and operating system (OS) level implementations. Language/compiler implementations require programmers to declare shared data by using annotations. Compilers will translate shared data accesses into synchronization and coherence primitives [13, 22]. Besides, library-level implementations of DSM are implemented in a run-time library linked with applications at compile-time [17, 18, 40, 108, 109, 170, 173, 196]. It provides a set of primitives which allow programmers to use for programming. Library routines are provided to manage shared memory across nodes and synchronization between processes. Nevertheless, these user-level implementations must cross the OS layer to get interrupt events (e.g., page fault handler and interrupt handler triggered when a message received) or to perform system calls. Also, SDSM implemented at programming and library levels are usually not transparent to programmers and it requires programmers to be familiar with the memory consistency and primitives provided by the programming model.

On the other hand, OS-level DSM [114, 181] is implemented atop the OS virtual memory subsystem. It is directly adjacent to the hardware such as MMU and network devices without having a abstraction layer in between, leading to mitigate mode switching overhead. One of the advantages of this approach is the semantics of OS remains the same. As a result, legacy SMP binaries can be seamlessly ported from single-node system to a distributed system without neither modification nor recompilation. It also gives developers a full transparent single address space across multiple distributed memory regions.

There are three main DSM design choices for the DSM layer:

- Granularity: the minimal size of a shared memory unit. With a larger data size, false sharing would be more likely to happen. But it can reduce the network data transfer initialization overhead and better utilize the network bandwidth. Conversely, with a smaller data size, false sharing would occur less. Nevertheless, it has to suffer the network startup time for transferring a package. This latency is not affected too much by smaller or larger package sizes.
- Data location and access: In a DSM system, each node must know where to find the right data from the right node. For example, one way is centralized manager approach. A centralized manager is located on a machine and keeps track of all the ownership information. Its performance is relatively predictable. The first hop is always delivered to the centralized manager. In a If the centralized manager does not own the page but it knows where the page is located. So, the second hop then finds the real owner and

get the right data. Nevertheless, this approach always has to go through the centralized manager. Plus, the centralized node serializes all the accesses and becomes the single-node bottleneck. Another other way is dynamic distributed manager approach. Each node stores a table of the probable owner of each page. The advantage of this is that it does not have to ask the centralized manager. When the page is not frequent accessed. Requester can likely get a page with 1 hop. However, in a worst case, this may cause more request messages forwarding before reaching the owner.

- Coherence semantics/protocol: coherence semantics define when does a memory update propagate between nodes. There are many memory coherence protocols such as sequential, weak, release, and etc. For instance, one of them is Sequential Consistency [65, 108, 196] (SC) which guarantees execution order of a thread should obey its program order. Execution order between threads is undefined and thus can be in any order, depending on runtime. Another example is weak consistency [28, 40] which requires programmers using synchronizations operators to guarantee the data is sequentially consistent. One more example is release consistency [17, 107] which is an even more relaxed version of weak consistency. Unlike weak consistency, which directly uses synchronization points, it further break down into lock and unlock operations. Coherence protocol is implemented in lock primitives respectively to acquire more design flexibility. Addition, coherence protocol decides how the permissions are maintained such as write-invalidate and write-update. Write-invalidate [57] allows a single write exclusive data or multiple shared read data in the system. In write-update [33] protocol, a write updates all copies of a piece of data.

Many works have integrated with DSM to deal with problems such as programmability [1] , scalability, energy efficiency, etc. To scale out an application, another choice is MPI [76]. However it does not support legacy traditional SMP parallel applications. That is application rewriting is required to take advantages of distributed systems. Previous works [24, 25, 132, 161] have argued that rewriting a SMP application with MPI requires lots of effort. Thus, the DSM layer is adopted to make SMP application scale-out without changing a single line of code. Furthermore, recent systems such as K2 [113], Reflex [112], and ADSM [69] adopt software DSM layer to ease programming efforts in multiple non-cache-coherence heterogeneous-ISA domains.

There are many advantages in using the DSM abstraction. Legacy SMP applications can be run on DSM systems as-is, without changing a single line of code. The DSM abstraction mitigates the effort of programming by providing an abstraction for developers to access remote memories as if they were accessing local memories. Programmers can easier develop applications to leverage the advantages of distributed systems. For instance, remote computation and remote memory resources are easier to be utilized and thus programmers can more focus on the algorithm instead of message passing and synchronization. This also reduces

[1]We define ”programmability” as the additional programming effort required to run existing shared-memory applications in newer distributed architectures. The more programming effort is required, the lower programmability it is.

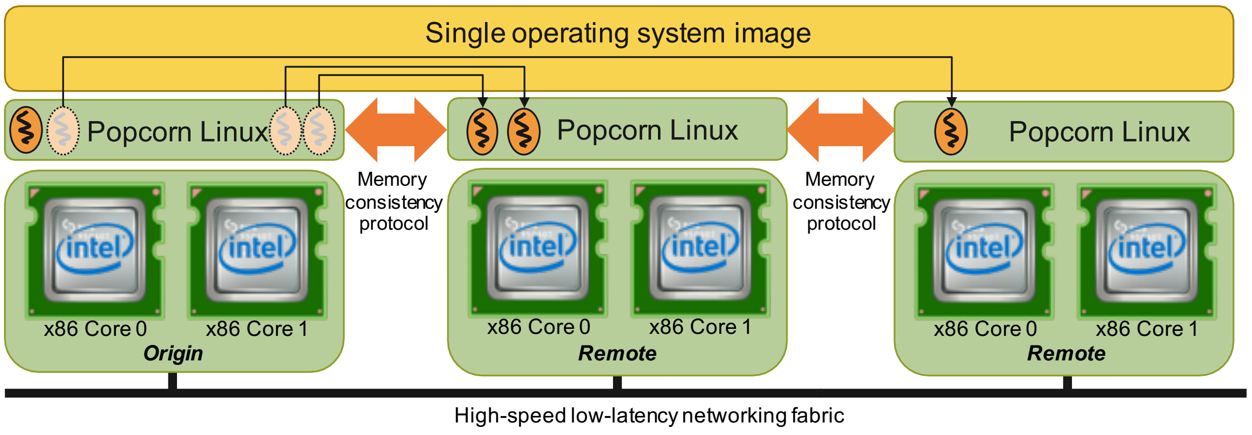

Single operating system image
Popcorn Linux
Popcorn Linux
Popcorn Linux
Memory consistency protocol
Memory consistency protocol
x86 Core 0
x86 Core 1
Origin
x86 Core 0
x86 Core 1
Remote
x86 Core 0
x86 Core 1
Remote
High-speed low-latency networking fabric

Figure 2.1: Overview of Distributed process EXecution environment (DEX) system.

the development time. The DSM is the cornerstone of transparent distributed execution for shared memory applications. Without the DSM layer, the execution (processes/threads) cannot migrate and run on remote nodes as-is [140, 158, 161]. Once the execution is migrated to another node, it will need to process data on the memory. However the memory is not migrated. DSM plays an of important role by bringing the data from the right node on-demand. Hence, the execution can really keep proceeding the execution. Also, programs can use larger physical memory, and are no longer constrained by local memory size and slowdowns caused by swapping.

On the contrary, the messaging passing latency and traffic are notoriously well known and problematic. The overhead can be larger than the benefits derived from programmability and thus make the system useless. To avoid the performance penalty, many works use many ways to reduce the number of messages used in the DSM layer. For example, for weaker consistency model [28, 40], the total number of messages exchanged is reduced by postponing synchronization operations for aggregating the messages to obtain better performance results. However, this requires programmers to follow the coherence semantics to write correct programs. Last but not least, although the DSM abstraction provides programmability advantages, it is usually not as efficient as explicit message-passing implementation.

2.2 DEX

PuzzleHype (Chapter 7 and 8) and Xfetch (Chapter 4) respectively develop two prototypes for evaluating this dissertation's contributions. PuzzleHype is built atop DEX by leveraging its thread migration features and OS-level DSM to provide a distributed guest VM. Xfetch provides a messaging layer by leveraging RDMA over InfiniBnad to DEX in order to improve DSM performance. Xfetch also proposes a prefetch mechanism based on the messaging layer and integrate it with the DSM to transparently enhance performance.

DEX (Figure 2.1) shorts for “Distributed process EXecution environment” is a rack-scale system design transparently scaling SMP applications. The system overview is shown in 2.1. It is a OS level extension allowing SMP applications to expand its execution boundary beyond a single machine. It enables a thread within a process to be dynamically relocated its execution to another remote homogeneous machine. It also adopts DSM to transfer data between machines to have a coherent memory view among multiple machines.

Different from Popcorn [25], DEX demonstrates how a cluster (at lease more than two machines, up to 8 nodes) can benefit from the system. Whereas, Popcorn only presents 2-node setup. Popcorn only supports process level dynamical migration. On the contrary, DEX supports a finer-grain thread level migration, which allows legacy SMP multithreaded processes to benefit from the system.

DEX built a prototype based on Linux kernel version 4.4.137 and evaluated on a rack cluster with 8 servers connected by InfiniBand through Mellanox ConnectX-4 VPI HCA in 56 Gbps through a Mellanox SX6012 switch. Each node is equipped with an Intel Xeon Silver 4110 processor which has 8 cores with two-way hyper-threading (16 hardware threads in total) running at 2.10 GHz and 48GB of RAM.

DEX supports a thread to migrate from the originating (*origin*) node to a destination (*remote*) node. The migrated thread can move back again from the remote node to the origin node. Thread migration is initialized by a system call, which can be easily adopted by developers or user-space libraries. Upon invoking the migration function, it collects current thread’s execution context such as registers’ state and virtual address space. Then, the execution context is transferred to the destination node. Once the thread’s execution is on the destination node, the destination node re-instantiates a thread according to the transferred thread context. The thread returns back to user space and resumes execution.

Upon the execution is resumed on the destination node, thread data is not migrated yet. To make memory consistent across nodes, DEX adopts DSM to maintain the memory coherency. Linux memory management routine can be simply split into two main parts: virtual memory area (VMA) management and page table (PTEs) management. VMA regions maintaining permissions, backing file, offset in the file, etc. define the address space for a process [36]. Each VMA has multiple PTEs. PTE maintains up-to-date per-page status such as permission, present, dirty. accessed, etc. DEX takes an on-demand virtual memory area and page data migration approach. The implementation relies on memory management unit (MMU) hardware raised page fault exceptions to the CPU. Linux page fault exception routine handles virtual memory area and page table separately. DEX implements the DSM layer in the page fault exception handling routine to makes VMA and PTEs consistent across nodes.

During the migration, no VMA information is transferred to the remote node. When the thread access to an address that is not belonging to its address space , it triggers a page fault exception which triggers DEX’s VMA *work delegation* mechanism. The VMA *work delegation* serializes VMA operations on origin node and then synchronize the VMA layout across nodes by broadcasting the updated VMA layout to the rest of nodes in the system.

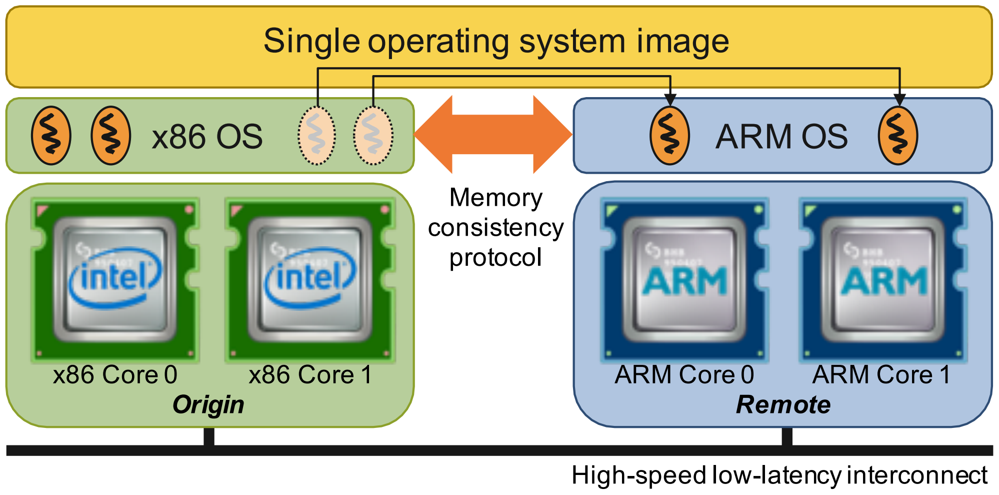

Single operating system image
x86 OS
ARM OS
Memory consistency protocol
x86 Core 0
x86 Core 1
Origin
ARM Core 0
ARM Core 1
Remote
High-speed low-latency interconnect

Figure 2.2: Overview of Popcorn Linux.

The same mechanism is used for all the VMA operations such as expanding and shrinking VMA layout.

After the VMA region is synchronized, the execution still cannot proceed yet in that the faulting page's content and permission need to be fixed as well. This resolving process is called PTE fault handling and it happens way more frequent than the VMA fault. The DSM in the PTE fault handling routine provides up-to-date page contents and permissions across notes. The DSM layer guarantees sequential memory consistency (SC) using a page-level, single-writer multiple-reader invalidation-based protocol. Access to pages not owned by the current node are trapped in the PTE fault handler. If the fault cannot be fixed locally, the DSM layer will help to resolve it such as fetching the page from a remote node and updating the permission of the page in the system if needed. PTE faults usually concurrently happen in performance-critical path. It brings significant communication traffic to the system. To this end, DEX also provides a *leader-follower model* for reducing redundant page transfer and network traffic. The first thread triggering the first fault for a page becomes the *leader*. A subsequent thread requiring the same page with the same access type becomes a *follower*. Once the *leader* resolve the fault, it will wake up the *followers* if there is any. The *followers* can simply resume executions without going through a redundant page fault routine again.

2.3 Popcorn Linux

Systems Software Research Group at Virginia Tech builds an advanced version of Popcorn Linux (Popcorn) [25]. This new version of Popcorn replaces the old on-demand paging implementation with DEX's DSM and replaces the old messaging layer with Xfetch's messaging layer. The system is running on a new setup where more powerful machines are used. The setup details are listed in Table 6.1. For the rest of the dissertation, we refer to this upgraded version of Popcorn as Popcorn. MWPF (Chapter 5 and 6) is built based on top of this Popcorn.

Popcorn Linux is mainly composed by Replicated-kernel OS and compiler/runtime framework for enabling execution migration between heterogeneous-ISA chip multiprocessors with no inter-domain cache-coherency. The system overview is shown in Figure 2.2. There is no commodity heterogeneous-ISA chip multiprocessor with cache-coherent shared memory nowadays. Popcorn approximates such a machine by connecting an Intel Xeon server (x86-64) to a Cavium ThunderX server (ARMv8) via InfiniBand. Popcorn exploits OS, compiler, and runtime extensions such as heterogeneous binaries and dynamic stack transformation [25] for dynamic and transparent cross-ISA execution migration. This gives great improvements in power efficiency [25] and performance [96].

Such a system software has to provide the ability to migrate threads and its data between servers. In order to support thread migration between heterogeneous-ISA machines, Popcorn includes a compiler which generates machine code for all available targets and injects metadata describing function stack layouts into a *multi-ISA binary*. The binaries for each ISA has a different virtual memory layout because of different symbol sizes and padding, etc. To make DSM work for the multi-ISA binaries, Popcorn Linux's alignment tool aligns the symbols such as global data, function addresses, padding, thread-local storage (TLS), etc. across all architectures.

Popcorn supports threads to migrate between nodes at *migration points* which are equivalence points [188] for the heterogeneous binaries. Function boundaries are naturally equivalence points. The *migration points* can be inserted manually by the developer or automatically by the compiler. Popcorn also inserts *migration points* inside the OpenMP runtime. Applications written with OpenMP API can transparently enjoy the benefits of dynamical migrations.

A thread context consists of live register values and the virtual process address space. The migration process is triggered by a system call. The thread invoking the system call will migrate to a remote node – the kernels cooperate to transfer the thread context to the destination node. Once the system call is invoked, the state transformation runtime converts the thread's execution context to the destination ISA's format (according to metadata generated at compile time) and passes the transformed register set to the kernel for migration. The thread invoking the system call will then migrate to a remote node by transferring the thread context to the destination node. The destination kernel then creates a new thread and reconstructs the thread context by using the transferred information. Then, the thread resumes execution by returning back to user space.

As the thread begins execution on the destination node, it triggers page faults caused by the thread accessing remote data. Inside the page fault handler, the DSM system works with the messaging layer to acquire page access permissions and data; once acquired, the DSM system installs the page into the thread's page table and resumes the thread. In this way, the kernel transparently fetches data on-demand by simply observing normal reads and writes. The memory consistency model of the DSM is the same as explained in Section 2.2.

2.4 Virtualization Technologies

Two virtualization architectures are widespread: *Type-1* where an hypervisor software is the only layer sitting above the hardware, for example Xen [26], and *Type-2* where the hypervisor co-exists with, or it is a service of an operating system – e.g., Kernel-based Virtual Machine (KVM) [99]. This section provides general background on two mainstream virtualization architectures. And finally mainly focus mechanisms such as virtual CPU, memory, and device on KVM which is adopted by PuzzleHype (Chapter 7 and 8).

Xen has a special privileged domain named *Domain0 (DOM0)*. The other unprivileged domains in Xen are called *(DOMU)*. *DOM0* has the highest privilege and manages the other DOMUs. Drivers for hardware are in *DOM0* as well. *DOM0* is the only vulnerable area or attacker surface in Xen. Plus, each *DOMU* provides a VM instance. *DOMU* also called Xen guest has two main types Xen Paravirtualization (PV) and Xen Full virtualization (HVM). PV required a modified guest OS to be aware of the underlying host hypervisor. This approach is a full software virtualization technique and does not require any hardware support. In HVM, guest VMs are virtualized by leveraging host CPU's hardware virtualization extensions such as Intel VT, AMD SVM. It uses QEMU to emulate hardware. Combining these two approach, recently there is a guest type called Enhancements to PV (PVH) approach which is lightweight HVM-like. It takes advantages of PV and HVM together. In PVH, the guests are lightweight HVMs utilizing hardware virtualizaion support for memory and privileged instructions. Unlike HVM, PVH does not use QEMU to emulate any devices. It leverages PV drivers, and thus, it also requires guest OS to be modified and aware of the host hypervisor. On the other hand, KVM is a virtualization module integrated into the Linux kernel. This approach usually relies on hardware acceleration built on modern CPUs such as Intel VT-x and AMD AMD-V hardware accelerations [12]. It is also built with Linux OS-level component's supports such as scheduler, memory management, I/O stack, etc. In particular, virtual CPUs are emulated by OS threads. Since it is integrated with Linux kernels, its flexibility makes it more prevalent. And its security level fully relies on commodity Linux kernels. Many existing implementations can be easier adopted in *Type-2* virtualization architecture compared with *Type-1* virtualization architecture. PuzzleHype (Chapter 7 and 8) takes advantage of this benefit to build a system providing a distributed VM equipped by resources from different machines. Thread migration is used for placing a thread on a remote node, meaning this allows vCPU in a VM to be located on a remote node in *Type-2* virtualization architecture. OS-level DSM is utilized to provide a global memory address space to the guest VM. vCPU from different nodes can transparently work on the same memory address space. PuzzleHype therefore spends the minimal time and focuses on other important components to build the system.

2.4.1 Virtualizing Memory

In KVM, each VM is implemented as a regular process. Virtualized CPUs are running as OS threads. Code in VM will be executed by them. Their physical address space (guest physical address, GPA) is the threads' address space (host virtual address, HVA). In order to achieve memory virtualization for the guest VM, and in order for the guest to address its own virtual memory (guest virtual address, GVA), a virtual memory management unit (vMMU) is needed.

In the early ages of virtualization, vMMU was implemented by shadow page tables in software [12, 148]. Today, hardware mechanisms have taken over the slower software implementation [31]. These are Extended Page Tables (EPT) on Intel [131], Nested Page Tables (NPT) on AMD [30], Stage-2 MMU on ARM [55], etc., and provide what is called two-dimensional paging – i.e., two layers of hardware MMU translations. In simpler terms, a page fault in the guest OS can be handled by the guest MMU, but if there is no host backing page, the fault will be handled by the host MMU.

Finally, with the introduction of hardware mechanisms to support vMMU, CPU vendors introduced a new execution mode for the guest VM itself. Each CPU has its own instruction to switch the execution mode, and the environment descriptor has to be filled before switching (e.g., VMX in Intel and VMS in AMD). The descriptor includes a series of events that will make the CPU exit the execution in virtual machine mode. We refer to this as "full-virtualization", while virtualization without any hardware support that requires modification of the guest software is called "paravirtualization".

2.4.2 Virtualizing Devices

With the introduction of CPU and memory hardware assisted virtualization, manufacturers also added hardware virtualization to devices [61]. However, devices with hardware virtualization have hard constraints on the maximum number of VMs they support, or are expensive. In these cases, paravirtualized hardware devices are preferred.

VirtIO is the de-facto standard paravirtualized device technology, and implements different classes of devices, including the network interface card (*virtio-net*) and console (*virtio-console*). If independently adopted with *Type-1* or *Type-2* virtualization, *virtio-net* appears to the guest software as a PCIe device and requires the software to instantiate two in-memory ring buffers – a transmission ring (TX) and a receiver ring (RX) for outgoing and incoming network packets. With virtualization, performance is paramount. Due to *virtio-net*'s mediocre performance with *Type-2* virtualization, *vhost-net* was invented [143]. *vhost-net* moves the network emulation mechanisms from user space to kernel space to avoid user-kernel switches.

2.5 Network Technologies - InfiniBand/RDMA

This dissertation contributes a messaging layer over InfiniBand/Remote Direct Memory Access (RDMA) (Chapter 4), which is leveraged throughout the entire dissertation. This section explains background knowledge for understanding our contribution of the messaging layer.

InfiniBand is a networking communication standard which provides high throughput and low latency. In InfiniBand, the Network Interface Card (NIC) implementing RDMA features is called Host Channel Adapter (HCA). HCA can be connected directly or indirectly through a Switched Fabric.

There are different types of data transfer modes, including Reliable Connection (RC), Unreliable Connection (UC), Unreliable Datagram (UD). In datagram mode, one-sided RDMA operations are not supported. In unreliable connections, data transfer is not guaranteed to be delivered. Efforts of corner cases are required to handle. Instead, in RC, data is transferred in order and guaranteed to arrive. RC also provides on-sided operations. Therefore, RC is used in our design and implementation.

The connection is established by a bi-direction Queue Pair (QP) which contains a Send Queue (SQ), a Receive Queue (RQ), and a Complete Queue (CQ). SQ and RQ are called Work Queue (WQ) as well. A Work Request/Work Queue Element (WR/WQE) is passed to the SQ for sending out a package. WR provides a abstraction called Verb explaining what features a HCA provides. Once the connection is established, one node can send a message to another node via the connection by using SEND, RECEIVE (two-sided) messaging Verbs. InfiniBand RQ requires developers to register WRs before any data arrives. Once a buffer is received, the developer can check CQ and get the WQE from RQ. The WQE indicates which memory region is written by HCA. In addition for the RECEIVE operation, CQ is used to notify the developer when a SEND/WRITE/READ/ATOMIC operation is completed.

To perform RDMA operations via READ/WRITE (one-sided) memory Verbs, a Memory Region (MR) has to be registered to the HCA. Verb provides an interface for developers to create a MR which contains a local key, a remote key, and a memory region. The keys are used for security purposes. They are required when performing the operations on the registered memory region. The registered memory region cannot be swapped out during the operations as well. Protection domain (PD) is used for isolating different groups of MRs. Once all these are initialized, one-sided operations can be performed by inserting a WQE to WQ.

Chapter 3

Related Work

This chapter provides a literature review and compares them with our contributions.

Section 3.1 shows new trends in networking and many prefetching mechanism on hardware and software levels for differnt research areas. Traditional and recent DSM works are discussed in Section 3.2. Section 3.3 explores the recent works in Single System Image (SSI), resource disaggregated systems, and systems for solving resource fragmentation in datacenters.

3.1 Prefetch in DSM

Xfetch (Chapter 4) is inspired by recent fast network technologies and prefetching mechanisms to improve DEX's DSM performance. In this section, the dissertation introduces works related to prefetching and modern interconnect technologies.

Prefetching is widely studied in many research areas such as cache prefetching, I/O device prefetching, etc. The main idea of these studies is to utilize *spatial locality* and hide communication latency from processors. Data/instruction prefetching [145, 182] was initially proposed to reduce the speed gap between processors and memory by fetching data/instruction ahead of time. There are hardware and software based cache prefetchings. Hardware based prefetching requires special hardware mechanism in processor to achieve the prefetching. Both data and instructions in the cache line can be prefetched without causing an explicit fetch operation. Software based prefetching leverages the software such as compiler-inserted fetch instructions to perform the prefetching. In addition, a kernel-space read-ahead prefetching algorithm [193] is implemented to amortize block device data access latency. It reads data from disk to page cache before they are accessed.

Other than that, many works [32, 89, 104, 115] use prefetching mechanisms to reduce the number of remote page accesses in DSM systems. For instance, Adaptive++ [32] is a runtime data prefetching prediction-based strategy for software DSM. But these works do not consider high-speed and low-latency interconnects.

New trends in networking are making DSM systems and loosely coupled distributed applications feasible in datacenters. InfiniBand networking communication standard and its RDMA features offer very high throughput and very low latency. Many recent works are

Table 3.1: Comparison of related work on leveraging remote computational resources and how much programming effort is required by a developer.

	Traditional & Recent DSM systems [**17**, **28**, **40**, **65**, **198**] [**29**, **98**, **197**] [**69**, **112**, **113**]	**Dynamic Process Migration** [**24**, **25**]	**Dynamic Thread Migration** [**140**, **161**]
(A) Goal	Programmability	Migrate execution	Distribute execution
(B) Memory model	Shared	On-demand offloading	Shared
(C) Execution replacement	No	Yes	Yes
(D) Relocation unit	-	Process	Thread
(E) Concurrent execution	-	Single-node	Multi-node
(F) Programming effort	High / Low	No	No

leveraging the interconnect technologies to enhance DSM system performance. For example, Grappa and RING [124, 132] use prefetching mechanism to reduce cache miss rate and use batching to save network bandwidth. A recent work RAMP [123] utilizes RDMA and a prefetching mechanism to provide distributed applications running on multiple nodes better performance. However, to enjoy such benefits from RDMA, applications have to be rewritten or developers have to learn new primitives or memory consistency models.

To ease programming effort, researchers [91, 138] use InfiniBand and RDMA as their communication layer used by different memory consistency protocols. However, since the networking primitives are implemented in user space, kernel subsystems and extensions cannot adopt it. Other works [114, 181] instead provides the InfiniBand and RDMA primitives in kernel space. But it still does not provide any optimization for DSM systems. Besides, leveraging RDMA, MAGI [86] and VDSM [59] proposes a speculative fault approach to prefetch data pages before the pages are accessed in a user-space DSM. Nevertheless, it is unclear how RDMA buffers should be managed for better performance especially in kernel space. Therefore, we implement a transparent bulk page prefetch mechanism by using RDMA features in such a DSM system which transparently makes SMP applications scale-out.

3.2 DSM Systems

Scaling out SMP applications across nodes is not a new idea. MWPF's (Chapter 5) unique challenge is how to efficiently leverage existing programming frameworks while transparently scaling out a SMP application on multiple incoherent domains. Compared with previous DSM works [24, 25, 140, 161], MWPF endeavors to optimize the DSM protocol and implementation to reduce DSM overheads in non-cache-coherent domains. MWPF builds upon a variety of techniques and mechanisms [25, 161] such as DSM and thread migration. This section studies the related works for MWPF. Table 3.1 shows a comparison of traditional and recent DSM works, dynamic process migration works, and dynamic thread migration works including MWPF. They are traditional DSM works [17, 28, 29, 40, 65, 98, 197, 198] and recent DSM works [69, 112, 113]. Traditional works mainly discuss coherence protocol and design and implementation. Recent DSM works more focus on how to leverage DSM to ease programmability in non-cache-coherent domains. By leveraging DSM, some recent works support dynamical process-level migration [24, 25] and dynamical thread-level migration [140, 161] to make SMP applications transparently scale-out. The features of these systems are categorized into different aspects (rows in Table 3.1) for comparing these works. *(A) Goal* is the primary objective these systems designed for. *(B) Memory model* indicates the view of memory in the system. *(C) Execution replacement* means if execution can be dynamically replaced. *(D) Relocation unit* is the minimal migration unit. *(E) Concurrent execution* shows how much of the node(s) is concurrently utilized by threads in a process. *(F) Programming effort* explains how hard it is to use each system to concurrently leverage cross-node computing resources.

3.2.1 Traditional and Recent DSM Systems

DSM systems (Column 1 in Table 3.1) give the illusion of a single shared memory across non-cache-coherent nodes. Memory is automatically kept coherent across multiple incoherent domains by the DSM system software. Traditional DSM systems automatically transfer data across nodes for applications; developers however have to manually place execution at startup and synchronize/assign work along with execution. Additionally, cross-node execution synchronization mechanisms requires developers' involvement. For example, one way to make it happen is to break the application into finer-grand threads and manually place these threads on different nodes by executing the application on multiple nodes.

Heterogeneous architectures expose multiple discrete memory regions to developers. While DSM can be leveraged to handle remote memory accesses in such systems, traditional DSM frameworks do not support SMP programming models across heterogeneous architectures due to differing instruction set architectures (ISA), application binary interfaces (ABI), and execution models. Recent DSM systems such as K2 [113], Reflex [112], and ADSM [69] adopt DSM to ease programming efforts for these systems. For example, to leverage data parallelism, CPU-GPU programming models such as ADSM [69] require applications explicitly request data and perform data transfers from different memory spaces. Although ADSM significantly relaxes programming efforts for CPU-GPU systems, applications must be rewritten to use their *alloc/free/call/sync* APIs. Additionally, while ADSM automatically offers a variety of data offloading, aggressive, on-demand, and primitive-based offloading, it cannot execute computation concurrently on both CPU and GPU. Moreover, for leveraging heterogeneity in multi-node architectures, programming effort is still not trivial – heterogeneous-ISA architectures require much larger effort than homogeneous-ISA architectures. Programming in such a system coupled by multiple incoherent domains, explicit communication between the domains such as messaging passing is required. For different ISA domains, programs are usually not the same and at least have to be recompiled.

For instance, to leverage low-power processors, a developer has to learn new programming environment and synchronize data on non-cache-coherent processors. To this end, DSM is utilized to enable execution on multiple non-cache-coherence domains by transparently exchanging data K2 [113] and Reflex [112]. However, peripheral processor code still executes separately and communicates with remote procedure calls (RPC). Similarly, threads executing on an x86-64 processor cannot be migrated to an ARMv8 processor as ISA and execution state (registers, stack) are different. Similar problem happens for MPI programming models as well.

The aforementioned DSM systems require refactoring applications to exploit remote computing resources (row F in Table 3.1). DSM systems do not provide execution replacement, but rather require developers to take distributed execution into consideration while programming. Static execution placement (row C in Table 3.1) also loses the opportunity to dynamically migrate executions to utilize remote computation.

3.2.2 Dynamic Process and Thread Migration

Thanks to the DSM, memory access on a remote node can be seem as on the same address space. To leverage remote resources or to do load balancing, execution migration is required to resume an execution from one node to another. Execution migration replays execution context from one node to another to transparently run code and data on another node. Execution context describes the current state of a thread/process such as live value in CPU registers, and process address space layout.

Execution migration has been heavily studied. There are different types of execution migrations, e.g., stop/start, live-migration, and SMP-like migration with DSM. Checkpoint/restart systems [53, 60, 127, 128, 144] can save execution on one node and restore it on another node. Another example is virtual machine (VM) live migration [49, 85, 187, 190]. VM live migration moves the entire VM from one node to another by logging and restoring the entire VM by the underlying host OS or hypervisor. Additionally, while utilizing DSM, researchers [24, 25] (Column 2 in Table 3.1) provide dynamic process-level migration. By supporting dynamic migration capability, these works dynamically and transparently place a process on a different node. This gives greater improvements in power efficiency [25] and performance [96]. However, although Popcorn [24, 25] offers execution migration, it does not fully utilize the benefit of DSM in that these works only demonstrate process migration (D in Table 3.1) which cannot concurrently leverage multiple nodes' resources (E in Table 3.1) such as computational power and storage, which is especially necessary for HPC applications. These approaches share one common problem of migration granularity. The migration granularity of VM and process deactivate the ability for an application to concurrently leverage multiple non-cache-coherent domains at any given time.

Even though researchers [140, 161] (Column 3 in Table 3.1) proposed a framework to simultaneously run threads of a process on multiple nodes, these works suffer from the smae problem of tremendous DSM overhead. While preserving programmability, they neglect cross-node performance, which stresses the DSM (B in Table 3.1). They simply utilize SC DSM protocol to transfer data pages, which leads to excessive communication overheads. We instead provide better DSM primitives to relieve the cross-node overheads that DSM creates. In addition, none of these works provide a solution to efficiently support cross-node synchronization. We propose a design to transparently synchronize execution across nodes with lower overhead. Compared to the previous works, MWPF instead leverages thread-level migration to transparently run a process fully distributed across multiple nodes and further enhances performance by presenting new DSM semantics such as a multiple writer (MW) protocol, aggressive data page prefetch mechanism and a low-overhead cross-node synchronization primitive. Finally we develop a runtime to transparently adopt these semantics/primitive.

3.2.3 Message Passing Interface (MPI)

Message Passing Interface (MPI) [76] and DSM are two extreme ways to scale out an application. MPI is not included in Table 3.1 because the effort of porting any single-node application to a distributed version is higher than any other programming model. The effort required is close to rewriting the application from scratch [24]. In MPI, tasks are spawned at startup and will not be migrated throughout execution. Memory is private to each task/node. Developers have to manually define message handling, transferring data and synchronizing data/execution across nodes via messaging. This programming overhead is extremely high. MWPF and the MPI programming model, respectively, target two opposite extremes in the spectrum of programming effort to scale out legacy SMP applications.

3.3 Disaggregated Computing

PuzzleHype (Chapter 7) is a distributed hypervisor aggregating remote fragmented resources for tackling resource fragmentation problems in datacenters. This section surveys similar works dealing with resource fragmentation problems and using similar approaches.

3.3.1 Distributed OSes

Aggregating resources from different physical machines while still providing a familiar programming interface, i.e., a Single System Image (SSI) [37], was realized through distributed OSes. Their primary goal is to allow workloads to scale out beyond the boundaries of a single physical machine. Notable past works include MOSIX [23], Amoeba [130], or LOCUS [189]. More recent examples include Helios [135], fos [191], and Popcorn Linux [24, 25, 140, 161]. Contrary to PuzzleHype, all these works allow an application rather than a VM to leverage remote resources. Take Popcorn Linux projects as an example. They provide a system transparently aggregating remote CPU resources. However, such works do not consider virtualization and network applications and thus far from being able to be adopted in cloud and edge datacenters.

3.3.2 Distributed Virtual Machines

Other works target virtualization and allow for the creation of distributed VMs [9, 42, 59, 156, 180, 194]. Software approaches providing custom *type-1* hypervisors such as vNUMA [42] or ScaleMP [9], or *type-2* as TidaScale [180], have the downside of being unpractical to deploy and maintain in production [194]. Systems such as Numascale [156] rely on special and expensive hardware. GiantVM [42, 194] are relatively close to PuzzleHype– in particular GiantVM targets the Linux/KVM, but uses QEMU vs kvmtool, and does not require custom hardware. In order to run scale-out compute-intensive workloads, they advocate running DSM at the user-space hypervisor level rather than at the OS level for simplicity not performance. However, they does not focus on fragmentation, and instead scales out compute-intensive workloads by aggregating the entirety of the resources from the physical machines involved. PuzzleHype targets general purpose VM workloads – e.g., contrary to GiantVM, we evaluate PuzzleHype on I/O intensive benchmarks, including a web server and FaaS.

3.3.3 Resource Fragmentation Works

One way to deal with such problems is to employ VM migration to rearrange the physical machine usages [6, 7, 10, 116]. This is also known as VM consolidation. Furthermore, elastic VM [136] resizes VMs to fix the problem by dynamically replacing big VM with small VMs to reduce empty holes.

In the long term, a solution to fragmentation is hardware disaggregation [66, 78, 110, 111, 164, 176], allowing compute, memory, and storage resources to scale independently from one another. Still, it is likely that software and hardware solutions in this domain will take years to mature before a potential integration in the datacenter. For example, LegoOS [164] is not a virtualization-based solution and relies on specific hardware supports. On the contrary, PuzzleHype can be deployed on traditional servers with standard software stacks.

StopGap [136] tackles a similar issue as PuzzleHype and addresses fragmentation with the concept of elastic VMs that can be dynamically resized. StopGap supports only certain types of applications, namely elastic multi-tier master-slave applications. Nevertheless these works still do not solve the root cause of resource fragmentation problem. On the contrary, PuzzleHype does not make any assumption on the application running within its distributed VMs. PuzzleHype endeavors to solve the problem by directly breaking the machine boundaries. Fragmented resources from multiple machines can be re-aggregated together to provide a single VM illusion to end users.

Chapter 4

The Xfetch Prefetch Mechanism for DEX DSM

This chapter presents the design, implementation, and evaluation of Xfetch, which provides transparent bulk page prefetch and a messaging layer over InfiniBand/RDMA to support DEX DSM system. A kernel-level messaging layer leveraging InfiniBand/RDMA is built for DEX and used for enhancing DEX DSM performance. A prefetch mechanism is proposed by leveraging the messaging layer to transparently and efficiently improve DSM systems performance by reducing the number of page faults.

The rest of this chapter is structured as follows: Section 4.1 gives a high level overview of Xfetch system which proposes two main contributions. They are messaging layer over InfiniBand/RDMA and a transparent prefetching mechanism. Section 4.2 explains Xfetch's key design and implementation in detail. Last, the evaluation for Xfetch is conducted in Section 4.3.

4.1 Overview

Distributed computing systems and fast network fabric technologies have been vigorously studied. For example, by leveraging high-bandwidth low-latency networks, researchers have developed systems providing easier API [132] and transparently running legacy SMP applications [25, 140] to enjoy distributed computational power. A common problem with these systems involves too many communication messages being sent because of the DSM protocol and high inter-node communication latency. Page prefetching [32, 89, 174] has been proposed to solve the problem a while ago. However, initial attempts were not very successful. Faster network technologies enable the rethinking of page prefetch in DSM systems because its high throughput and low latency. This is not well-discussed in the context of such a DSM system such as DEX.

Our contribution to DEX is a messaging layer which efficiently transfers messages of different sizes by leveraging InfiniBnad/RDMA. Additionally, we provide a transparent bulk page prefetch design which automatically sends prefetch requests along with the page fault handler (only if the fault cannot be solved locally) and the prefetch request is asynchronously processed in the background for minimizing the interference with foreground execution. To evaluate the performance of our design, we implemented a prototype in the context of Linux operating system. Our results show the page prefetch transparently enhance by reducing the number of page faults. The performance improvement is up to 42.74% compared to without enabling the page prefetch feature. Our contributions can be summarized as follows:

- We designed a messaging layer for Xfetch to efficiently prefetch pages in DSM layer by leveraging low-latency and high-throughput emerging networking technologies.
- Xfetch provides a bulk page prefetch design without interfering foreground execution too much. We implemented and evaluated it on a DSM system transparently scaling out SMP applications.

4.2 Design and Implementation

To transparently scale out a SMP application on multiple nodes without changing any line of code, a system leverages the DSM layer guarantees Sequential memory Consistency (SC) using a page-level, multiple-reader single-writer protocol, for cross-node memory consistency. Finding page data and page ownership in such a system is a time-consuming process since it involves (at least) one roundtrip communication over the network and an search/update of the per-page state index. Plus, the communication layer is of importance in terms of performance in such a distributed system. Recent networking fabric technologies can be utilized to solve the problems. To amortize the overhead and better utilize the network bandwidth, Xfetch therefore propose a transparent bulk page prefetch design in kernel-space DSM with a careful InfiniBand/RDMA page transfer messaging layer design; while the *origin* (Section 2.2) handles a page request sent from a remote node, it transfers additional pages which are next to the faulty page. In this way, Xfetch leverages the spatial locality of memory access that is a very common memory access pattern for many applications. Our prefetch features over InfiniBand/RDMA are listed below:

- Prefetching pages as a batch: it hides network latency.
- Asynchronous background prefetch: we send the request along with the normal page fault execution and recycle the existing worker to perform data page prefetch for reducing redundant context switches.
- All control messages are sent by our InfiniBand POST primitive, which places a request to the NIC and does not wait until the message is guaranteed to be sent. This save a few CPU cycles to work on things more important by reducing redundant wait/wakeup.
- The prefetching mechanism achieves minimal interferes to the system by operating on a best-effort basis; it does not spend time on waiting pages that is being processed by someone else. It instead move on and try to prefetch the next page.

4.2.1 Inter-Node Communication over InfiniBand/RDMA

We built a inter-node communication layer (messaging layer) over different network fabrics in the Linux kernel, which can be leveraged by other kernel subsystems, and user-space processes and libraries. The design can be split into two parts: first, we introduce a messaging layer abstraction which is agnostic to underlying hardware and integrated with the Linux kernel. Moreover, we provides implementations over Linux device driver modules to leverage different network fabrics. To be able to adopt common commodity systems, we first provide an Ethernet socket implementation. Ethernet device is one of the most common network devices on servers. For better performance and easier to leverage InfiniBand/RDMA, we also provide an InfiniBand/RDMA implementation. Particularly, native InfiniBand/RDMA provides very low-level abstraction close to hardware primitives. This makes developers hard to directly leverage its benefits [181]. We will focus on the design and implementation over InfiniBnad/RDMA in the rest of this chapter.

Before jumping into the implementation of data transfer, messaging layer also takes care of the connection between nodes. A global IP table is duplicated on every system. Our current implementation has the table in a header file but it can be stored in any text file in the Linux filesystem. Although InfiniBand provides many transport models, our design takes Reliable Connection (RC) [181] because it guarantees the correctness of data transfer and provides RDMA WRITE operation which will be leveraged for transferring data page in DEX DSM. After all the connections are established, the messaging layer assigns each node a node ID in accordance with the IP table. Once all connections are established, one node can send messages to other nodes via a particular connection channel.

InfiniBand provides a two-sided message passing interface (SEND, RECV) and a one-sided Remote Direct Memory Access (RDMA) interface (READ, WRITE) accelerating transmissions. One-sided RDMA provides abilities to directly transfer data from source buffer to remote destination buffer.

Different from traditional socket programming, I/O buffers for two-sided operations over InfiniBand has to be explicitly mapped to a DMA-capable address space range. InfiniBand HCA can then directly operate on the buffer via the PCI bus. However, in a DSM system, simply leveraging native InfiniBand/RDMA primitives may not fully take advantage of its benefits because page faults happen too frequent. Especially, these faults are in a performance-critical path. Dynamically mapping these buffers for transferring data in the performance-critical page fault handler may bring nonnegligible overhead. Same for the RDMA WRITE/READ operations, the buffer has to be registered to a RDMA memory region called *remote memory region* in order to generate a key for another node to access the memory region. The key has to be deliver to a remote node so as to perform READ/WRITE operations on the buffer transparently. The overhead of message construction overhead cannot be neglected. Thus, upon the messaging layer initialization, we pre-allocate a pool of DMA-mapped memory regions and associate them with one-/two-sided work requests.

In RC, the CQ can notify a processor when a work request is completed. It guarantees these work requests are competed and happend in order. Developers can decide to wait for the completion or not, depending on demands such as blocking or nonblocking. Thus, we provide two primitives to send a InfiniBand two-sided message. One is normal SEND primitive. It returns when the data is sent out. The execution can safely modify the buffer after invoking this function.Another is POST primitive which returns immediately after it place a work request to the HCA. The developer has to manage the usage of their buffer properly. For example, releasing or modifying the buffer too early may cause to a unexpected result for the messaging layer and also the system. Moreover, A *send buffer pool* is presented in the messaging layer. This pool is made up of chunks of physically contiguous pages. The pages are pre-mapped to DMA regions. The pool is implemented as a ring buffer structure. Each connection maintains its own pool to allow callers to save time for redundant memory allocation, DMA-mapping in critical paths. Once a SEND request is completed, the *send buffer pool* will reclaim the chunk for recycling it at a later time.

When a data arrives, HCA first stores the data into a memory region indicated by the receive WQE. The HCA triggers an interrupt. We provide an interrupt handler which finds the memory region and process the request. After completing the request, we provide a primitive to recycle the WQE and its memory region. InfiniBand requires developers to pre-register receiving buffers to HCA before any data arrives. Similar to SEND operation, the receiving buffers are required DMA-mapped before registering it. The messaging layer pre-register an entire *receive buffer pool* per connection during the messaging layer initial phase. At runtime, when an InfiniBand Host Channel Adapter (HCA) receives a message, it directly writes to the buffer via DMA. Once it is completed, InfiniBand HCA signals the kernel by raising a hardware interrupt. Our messaging layer will enter a top-half interrupt handler and a data receiving completion will be found in a completion queue. We immediately leave the task for a bottom-half kernel worker. It then handles the request according to the message headers. When the request is completed the messaging layer recycles the buffer by re-registering it again. This again avoids redundant memory allocation and DMA mapping time.

Large size data transfers, such as transferring a page, are completed via one-sided RDMA operation. Previous works have proposed many ways to leverage RDMA technologies [62, 94, 117, 163]. However, most of them are too specific optimization to their use cases, which are different from our DSM system and thus a new approach has to be reconsidered. For example, in DSM systems, it is almost impossible to predetermine the virtual memory footprint and lifetime of processes which change dynamically over time and are different from each other. Unfortunately, it is almost impossible to keep virtual memory addresses physically contiguous in general commodity systems. Moreover, as we mentioned earlier, dynamically allocating a DMA-mapped region and registering it as a *remote memory region* in InfiniBand HCA is time consuming. Therefore, we cannot simply use the ways these system used or traditional dynamic approaches to design our messaging layer.

To this end, we propose a hybrid approach which statically allocates buffers and makes them DMA-mapped with just one extra memory copy at runtime. We argue that memory copying is less expensive than DMA mapping and registering it to InfiniBand HCA. Each connection maintains its own physically continuous *remote memory regions*, which is created during messaging layer setup. A requester (sender) should send a *remote memory region* to the destination (receiver) node via a control message. And thus, the destination node can directly prepare the page(s) and directly write it to the memory region provided by the requester without any CPU effort. Due to one-sided WRITE operation is performed without any notification on a remote node. The destination node has to acknowledge the requester with a control message. When receiving the acknowledgment, the requester copies the page data to the real page frame and reclaims the *remote memory region*. Although this is not the ideal approach of using one-sided RDMA, faster than two-sided and dynamic mapping one-sided WRITE implementation.

We provide READ/WRITE primitives for performing RDMA WRITE/READ. A requester can ask the messaging layer for a pre-allocated *remote memory region* and send it to the destination node. The destination node takes the key and address to use our WRITE/READ primitive to perform a RDMA WRITE directly to requester's memory region. The WRITE/READ maps the data into DMA region and associates it with a pre-allocated WQE. The WQE is delivered to the SQ and a READ/WRITE operation is performed.

Leveraging messaging layer in DEX. In the DSM system, transferred data can be classified into two classes; small messages also known as control messages (e.g., tens of bytes) and large messages also called data page messages (lager and equal to a page size 4KB in DEX). Compared to one-sided operations, two-sided operations can provide lower latency for small messages. To this end, control messages transferred by the system are done by using two-sided send/receive operations. We also provide an non-blocking/asynchronous two-sided send primitive which returns immediately back to the execution without waiting for a completion synchronization signaled by the NIC. A data page is transferred via one-sided RDMA WRITE operation.

4.2.2 Transparent Bulk Page Prefetch

Remote page fault latency is a notoriously well known problem in DSM systems due to its slow message passing speed. Even for applications which are well written to partition data evenly and locally, there still exists data that must be shared across different nodes. Accessing this data generates unavoidable remote page faults. Thus, to fully utilize network bandwidth, reduce network latency, and reduce the number of remote page faults, a Transparent Bulk Page Prefetch (TBPP) mechanism is leveraged in our system. TBPP brings multiple pages in a single round trip via regular remote page fault. This eliminates unnecessary latency of messaging and avoids near future page faults.

There are three main design principals in our prefetch mechanism. First, TBPP issues prefetch along with regular fault executions. Although it does prefetch more data pages in a single remote page fault request, the native remote page fault routine does not wait until the prefetch process is completed. Instead, it responses immediatly to the requester. Piggybacking prefetch requests on existing fault executions reduces redundant overheads such as interrupt and message brought from asynchronous prefetch requests. Moreover, TBPP brings as many pages as possible in a single regular remote page fault to utilize network bandwidth and reduce network contention. In such a DSM environment, the network traffic is always inevitable high. The latency of data transfer initialization is very expensive especially for small messages. Therefore, it is worth to batch all pages and response on a single message since the network benefit it brings will be much higher than the latency of accumulated memory copy and page lookup.

Secondly, prefetch should be transparent to developers. One of the major challenges for developers is to decide which page to prefetch. We provide a TBPP DSM system, which automatically prefetches pages to alleviate the need for manual annotations or manual tuning which originally may have to be done by the developer.

Finally, sequential page prefetch policy obtains benefits from spatial locality as well as inducing minimal increase in overhead to page fault critical path. We found many applications such as graph or machine learning algorithms which are specifically designed to benefit from spatial locality. Such applications usually access the same shared data region over many iterations throughout the entire execution. For instance, pages around the fault address will be accessed in the near future. Therefore, we adopt sequential page prefetch policy, which benefits from the spatial locality. Ideally, the prefetch mechanism should not add on too much additional latency to the page fault handling. If a more sophisticated prefetch policy were used, it may bring higher latency in the critical page fault handling path. Sequential page prefetch gives us the most efficiency as it does not bring much computational overhead to the critical path. Thanks to the sequential locality, our prefetch policy is able to be achieved in a neat manner.

When a read page fault cannot be handled locally at on a *remote* node, the page fault handling process will try to prefetch pages surrounding the page fault address along with the regular remote page fault requesting message. The prefetch-page candidates are selected by complying with the following rules: (1) All of the prefetch-page candidates must be within the same VMA region as the originating fault address. Doing so, the entire prefetch process does not require to lookup VMA for each page. This also makes sure that no illegal pages will be prefetched. (2) The range of fetching size is from 1 (without batch) to 8 (with batch). (3) In some cases, we can simply skip a prefetch-page candidate immediately, moving to check the next prefetcg-page candidate. These cases are:

1. The requesting node has had the correct permission for the prefetch-page candidate. In current implementation, we only support prefetching shared-read pages.

2. The PTE of the candidate is not created yet. This means the page is not created in the system yet.

3. The page is not being handled by any other local or remote page fault process routine. Doing this the prefetch process can skip redundant wait time.

4. The native fault is asking for a write permission. We currently prefetch only for read permission because bulk invalidation (caused by write faults) will incur a deadlock problem if there is no sophisticated algorithm. However, such a complex algorithm may bring more overhead to the performance-critical page fault path.

When a *remote* node has a fault which cannot be solved by itself, it sends a control message to the *origin* node. The control message is composed by a list of expected pages to be prefetched. Once the *origin* node receives a remote page fault request, it first handles the native fault and then checks for the prefetch-page candidates sent along with a remote page fault request. It prepares the page data and maintain the permission of the page by following DEX DSM protocol. Once the regular remote page fault on the *origin* node is completed, if there is any prefetch-page candidate, the *origin* node starts to prepare the prefetch-page candidates. Prefetching process does not interfere with other local/remote page fault executions. If another fault handling process is dealing with the prefetch-page candidate (enumerated above), the prefetch-page preparation process will immediately skip prefetching the candidate rather than wait until it gets the lock and proceeds. Failure to do so will incur additional latency. Xfetch strives to incur as minimal overhead as possible to regular executions. When all the prefetch-page candidates are all checked, the *origin* node directly sends the results and pages back to the *remote* node. Once the *remote* node receives the message, it copies the successfully prefetched page data to the corresponding page frame.

In our design, the memory consistency protocol treats the *origin* as a page cache server. When the *origin* prepares a page for a remote node, if the *origin* does not own the page, it brings the page back from the current owner. The *origin* then sends the page back to the requesting node. As a result, the *origin* can immediately serve any future page fault on a remote node. This effectively prefetches the page for the *origin* as well. Although the *origin* does not issue prefetch requests, it still can enjoy the benefits derived from prefetching done by remote nodes. On the *origin*, prefetching requests from remote nodes may require it to fetch a page from another remote node because both the requester and the *origin* are not the owner of the page.

4.3 Evaluation

Xfetch is built on top of DEX, using the very same hardware configuration. Xfetch is a prototype based on Linux kernel version 4.4.137. We evaluate on a rack-mounted cluster built with 8 servers connected by InfiniBand through Mellanox ConnectX-4 VPI HCA in 56 Gbps through a Mellanox SX6012 switch Each node has an Intel Xeon Silver 4110 processor which has 8 cores with two-way hyper-threading to provide 16 hardware threads in total,

Table 4.1: Speedup in execution time by using TBPP- higher is better.

Applications \ Nodes	2	3	4	5	6	7	8
KM	0.66%	8.53%	11.26%	12.97%	13.05%	15.79%	17.95%
GRP	4.26%	4.40%	3.14%	3.84%	1.79%	0.03%	1.22%
PR	1.05%	27.41%	31.65%	35.73%	36.59%	42.74%	36.47%
BP	1.65%	3.82%	1.15%	3.06%	3.36%	4.26%	3.21%
BFS	15.95%	24.79%	14.84%	23.32%	15.86%	17.06%	17.63%
BT	3.08%	5.24%	6.10%	10.69%	12.92%	14.70%	14.74%
EP	-0.22%	-0.76%	0.73%	1.44%	1.46%	1.75%	-1.20%
FT	7.43%	11.54%	15.23%	17.61%	21.21%	21.87%	22.04%
BLK	1.29%	0.85%	-0.06%	-0.89%	-0.63%	2.91%	1.30%
lavaMD	3.33%	1.35%	3.45%	1.16%	1.41%	0.96%	2.02%

running at 2.10 GHz and 48GB of RAM. To execute on a remote node, a binary has to be distributed on all nodes. We share the binary through Network File System (NFS).

We carefully selected ten applications to represent different memory access patterns.

Two of them are in-house data processing applications implemented by us. They are string match (GRP) and k-means clustering algorithm (KMN). String finds and outputs a user-specified pattern of characters also called key strings, filtering out the other texts, from a given text file. GRP partitions the file into chunks so that each thread can process its own chunk parallelly without interference. In our experiment, we used a 8 GB Wikipedia text. Four key strings, each is 7 to 10 bytes, are used. KMN is a popular clustering algorithm used in many fields. It divides data into groups by repeatedly computing until all the data finds its closest group or a certain threshold of iteration. We configured it to calculate 100 groups among 5 million points in a 3 dimensional space.

For high performance computing (HPC) applications, we chose BT, EP, and FT from the SNU NPB Benchmark Suite v3.3 [162], a C version of the NAS Parallel Benchmarks [21] written with OpenMP programming model from Seoul National University [162]. Its class C workload version is selected for our experiment. We also selected Blackscholes (BLK) from PARSEC Benchmark Suite v3.0 [149], which is a GCC pthread implementation. We evaluate it with ‘native input’ workload, which is the largest workload the benchmark suite provides for large-scale experiments on real machines.

For recent NUMA-aware applications, we chose Breadth-First Search (BFS), Belief Propagation (BP), and PageRank (PR) from Polymer [195]. Polymer is NUMA-aware graph-structured analytics framework written in C++ co-locating graph data closer to the NUMA-node as much as possible. These applications are written by using the framework.

Table 4.2: Number of remote page faults for each application on four-node setup.

	KM	GRP	PR	BP	BFS	BT	EP	FT	BLK	lavaMD
Without TBPP	1315582	278	28389105	1469413	35419526	35514843	6263	26798105	52259	60127
With TBPP	545199	247	13618356	1471548	18966877	21576816	6381	16438699	52137	60100
Remote page faults reduction (%)	58.56%	11.15%	52.03%	-0.15%	46.45%	39.25%	-1.88%	38.66%	0.23%	0.04%

4.3.1 Experimental Results

We evaluate the speedup applications gain when using TBPP subsection 4.2.2 compared to without using TBPP as presented in Table 4.1. The maximum payload size per message in this experiment was set to 32KB since the nature of InfiniBand Receive Queue and our current implementation allocates a physically continuous memory region for *receive buffer pool*. Since the largest size of physically continuous memory is limited by the Linux kernel memory allocator, if we increase the size per message, the number of messages per connection will decrease. If the RQ does not have any WQE when a data arrives, the system will not function correctly. The speedup in execution time is up to 142.74%. To further explore TBPP trends, we also collect the number of remote pages faults on four-node setup (Table 4.2). The reduction in remote page faults is up to 58.56%.

Looking at these results, we observe a major divide among our benchmarks and classify them into the following two trends with our reasoning explained to follow. The applications in the first trend observe extremely minimal gains because they are highly parallel. For example, GRP, BP, EP, BLK, and lavaMD partition their workloads evenly to each node and do not share data with other nodes. Thus, there are very few remote page faults generated while running. As a result, it is hard to see obvious speedups from Table 4.1. The reason why the number of page faults in BP is not small is because graph processing applications in general (e.g. BP, PR, & BFS) load global graph data to each nodes' local region. Our prefetch mechanism currently does not proactively prefetch for pages which have never been touched. Thus, the number of remote page fault for BP is large and not reduced.

The rest of applications, KM, PR, FT, BFS, and BT, attain very good performance improvements from using TBPP. These applications share data over iterations throughout entire execution time. Thus, they often benefit from our TBPP. These applications usually require shared data which is written by other nodes from previous iterations. In such scenarios, our prefetch mechanism efficiently brings multiple pages automatically to reduce the number of future remote faults. For the applications in this trend, we also found that TBPP scales well, increasing in performance with larger maximum payload message sizes. It is worth mentioning that the benchmark KM, which already scales out with base optimizations applied, further benefits from the prefetch mechanism. This experimental evaluation shows that TBPP incurs minimal overhead while providing the benefit of reducing the overall remote page faults. This insight shows us that TBPP is highly beneficial to use with applications that spread and syncronize their data across many nodes. On top of this, TBPP provides these benefits without any additional needed code modifications.

Chapter 5

The MWPF DSM Protocol

Recently there has been increasing interest in coupling systems with non-cache-coherent domains, or simply domains or nodes, together in data centers and the cloud [25]. Systems with non-cache-coherent domains are usually heterogeneous. Our envisioned architecture design point is one with multiple cache-coherency domains, with each domain hosting cores of a different ISA and no inter-domain cache-coherency. For such an envisioned heterogeneous-ISA architecture platform, Multiple Writer protocol with Prefetch (MWPF) provides a relaxed DSM consistency model, *smart regions*, a page prefetch mechanism, and a low-overhead barrier for inter-node synchronization. MWPF reduces the number of page faults by delaying page data consistency. *Smart regions* is a mechanism to dynamically select which DSM protocol (MWPF/SC) should be used in a work-sharing region. Page prefetch mechanism reduces the number of page faults during work-sharing parallel regions. Our cross-domain barrier synchronization is used for reducing redundant data page exchanging compared with using traditional SMP barrier synchronization. All these benefits do not require any change to existing legacy code. Existing runtime libraries can easily adopt the primitives provided from the kernel space. Taking the OpenMP parallel programming model and its runtime library for example, we demonstrate not only our design but also our implementation. With very few Lines of Code (LOC) OpenMP runtime library modification, applications written with the OpenMP programming model can transparently enjoy all the benefits we propose.

Section 5.1 motivates the development of MWPF. Section 5.2 describes on our DSM protocol, *smart regions*, prefetch mechanism, and cross-node barrier synchronization designs in detail. In Section 5.3, the protocols and mechanisms are implemented in a first-class Linux OS kernel and the OpenMP runtime is instrumented to utilize the proposed primitives.

5.1 Overview

Because DSM transparently extends the shared-memory abstraction across discrete computing elements, with thread migration features, users can run existing applications across multiple domains as-is. Previous DSM systems, however, were plagued by networking limitations [17]. Additionally, many optimizations proposed to improve DSM scalability relied on weakening the memory consistency models of the DSM [17], making correctly programming such systems challenging. Besides, fast network technologies continue to evolve in both latency and throughput. Current commodity network interface cards such as InfiniBand connectivity provide 200 Gbps bandwidth [87]. This has motivated re-investigating DSM as many assumptions about the performance characteristics of such systems are changing.

As shared memory multiprocessors have become mainstream, developers have turned to parallel/fork–join programming models such as Intel Threading Building Blocks (TBB) [154], Intel Cilk [153], and OpenMP [54] to easily leverage multiple CPU cores for data- and task-parallel computation. These programming models provide a set of primitives to easily spawn multiple threads, distribute parallel work, and synchronize execution. For example, in OpenMP, a work-sharing region can be split into two main components, parallel execution and synchronization barrier. A synchronization barrier is applied across the entire system to synchronize execution before any thread can proceed. When using these primitives, developers must write their applications in accordance with the programming model's memory consistency semantics. Oftentimes they use weak memory consistency, where updates are only made visible after synchronization operations. This forces developers to write computations that avoid such data races and thus are amenable to parallelization. Because these shared memory parallel programming models are mature and widespread, there is a large body of applications built assuming a relaxed consistency model [21, 44, 45, 51, 149].

Together with the aforementioned trends in networking, we argue it is time to re-investigate using DSM to target systems with multiple non-cache-coherent domains. In particular, we argue that DSM systems can be optimized using weaker consistency semantics and run existing shared-memory symmetric multiprocessing (SMP) parallel programs with high performance on these emerging systems. Using DSM allows developers to run existing applications on non-cache-coherent systems as-is, rather than requiring a full re-write to a new programming model like MPI or PGAS languages [39, 43, 132].

In this work, we explore changing the DSM consistency model and implementation to better optimize cross-domain execution. We found several bottlenecks in previous works [25, 140, 158]: significant numbers of invalidation messages, false page sharing, large numbers of read page faults, and large synchronization overheads. To solve these problems, we propose efficient DSM protocol primitives that postpone memory synchronization points, batch invalidation messages, aggressively prefetch data pages, and perform cross-domain synchronization with low overhead. We follow the existing shared memory multiprocessing programming model to design our operating system level (OS-level) DSM primitives, which can be easily adopted in parallel programming runtimes. For developers familiar with programming in such relaxed consistency models or for existing legacy applications, our system transparently brings further performance improvement. To prove the applicability of the new primitives, we develop a runtime based on OpenMP [54]. We instrument the runtime to utilize the new primitives, allowing OpenMP work-sharing regions to efficiently distribute parallel computation across nodes and transparently leverage the new primitives. Thus, we seamlessly support cross-domain execution of existing OpenMP applications without changing any line of code. We evaluate a prototype on two heterogeneous-ISA nodes.

To conclude, MWPF introduces a design transparently improving the performance of shared memory multiprocessing applications running on no cache coherent domains without any code modification by leveraging Multiple Writer (MW) protocol, release consistency (RC), memory access pattern profiling and page prefetching, and multi-domain synchronization design. No line of code requires to be changed in order to run a legacy SMP application written in a parallel programming model.

5.2 Design

Our goal is to reduce DSM overhead so as to leverage remote computing power with minimal communication. Two large sources of overhead caused by using SC are invalidation messages and false page sharing. Because these are intrinsic to Sequential Consistency (SC), one way to reduce consistency traffic is to switch to a relaxed protocol that can delay invalidations and front-load page fetches (Sections 5.2.1 and 5.2.2). Also, to reduce redundant communication caused by traditional cross-node synchronization, the DSM includes a new cross-node synchronization primitive (Section 5.2.4).

The DSM system is redesigned to take into consideration these forms of overhead. In order to reduce the impact of invalidation messages, the DSM employs an Release Consistency (RC) protocol that delays invalidation messages until a synchronization point (Section 5.2.1). This not only allows the DSM to immediately upgrade permissions from shared to exclusive for resident pages, but also allows the DSM's underlying messaging layer to batch multiple invalidation messages together to better utilize the interconnect. To further improve the impact of the MW protocol, the DSM system also includes an aggressive page prefetching mechanism which creates a form of Snapshot Isolation at the start of work-sharing regions (Section 5.2.2). By prefetching all pages read during the work-sharing region, there is a much higher likelihood that the DSM layer does not need to fetch remote data and can simply provide the correct access permissions during a write fault. While these mechanisms reduce intra-region DSM overhead, they come with their own overheads during start or end-of-region processing. The DSM system includes a *smart region* heuristic that detects when using MW may in fact cause more overhead than performance benefit and falls back to SC in such cases (Section 5.2.3). Finally, the DSM system is extended to provide a new synchronization primitive that avoids redundant page transfers caused by traditional atomics and futex operations (Section 5.2.4).

5.2.1 Multiple Writer (MW) Protocol

The multiple writer protocol allows multiple nodes to concurrently write to the same page without cross-node consistency. This allows delaying invalidation messages till the end of a work-sharing region, or simply a region, where all invalidations are exchanged to fix up

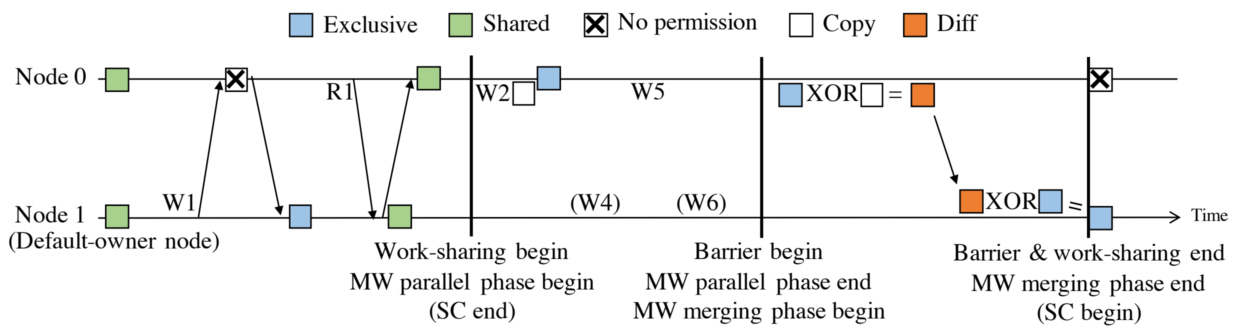


Figure 5.1: Example of DSM traffic before work-sharing regions (SC) and during work-sharing regions (MW).

pages and permissions. For pages written by two or more nodes, the DSM layer detects and propagates updates between nodes. It accomplishes this through copy-on-write (COW) – when a node encounters a write page fault, the DSM layer saves a copy of the original version of the page. When propagating writes, the DSM compares the updated page to the original and sends updates to other nodes. Allowing updates to be delayed till the end of the work-sharing region fits directly with parallel programming models.

Upon entering a work-sharing region, the DSM switches consistency protocols from SC to MW and execution enters the *MW parallel phase*. At this point the DSM layer designates one node as the *default-owner* and all other nodes as *non-default-owners*. This facilitates an optimization in propagating updates – rather than an all-to-all exchange for page updates, the default-owner pulls in updates from all other nodes (all-to-one) and performs the merge; subsequent accesses to the page by non-default-owners must reacquire the page and permissions. Because the default-owner merges updates from all other nodes, only non-default-owners perform COW to generate diffs. At the end of the work-sharing region, the DSM layer enters the *MW merging phase*. It begins by exchanging invalidations encountered during the region and determines which pages need merging. Then, the DSM layer generates "differences" (diffs) of the COW pages and sends them to the default-owner. The default-owner merges the diffs, and the DSM system transitions back to SC.

Take Figure 5.1 for example. It focuses on a single page case. Node0 represents a non-default-owner node and Node1 is a default-owner node. Both nodes are initially outside the work-sharing region and have the page mapped read-only, as per the SC protocol. When Node1 first attempts write operation W1, it causes a page fault due to the lack of write permissions. During fault handling, Node1 sends an invalidation message to Node0 to revoke Node0's page permissions. Node0 drops the page and permissions and sends an acknowledgement (ACK) message back to Node1. After receiving the ACK, Node1 gains exclusive write access and writes to the page. Next, Node0 performs a read operation R1. Since Node0 does not have any permissions for the page, it sends a message to Node1 asking for the page and shared permissions. Node1 downgrades its exclusive permissions to shared permissions and sends the page data back to Node0. At this point both nodes have the same shared read-only page. Both nodes have the latest version of the page.

When the nodes enter the work-sharing region, the DSM transitions from SC to the MW protocol. Node0 issues W2 generating a write fault. The DSM system first checks whether Node0 is the default-owner node. In this example, since Node0 is not, it makes a copy of the page (COW) and changes the permissions to be exclusive without cross-node coherency. After this initial duplication, the following writes before MW parallel phase end will not incur page faults. This saves the latency of sending invalidation messages and permission changes caused by different interleavings of writes W2, W4, W5, or W6.

When the application encounters either an explicit or implicit barrier (end of work-sharing region), the DSM system enters the MW merging phase. First, the DSM batches all delayed invalidation requests into a single message and distributes invalidations across all nodes. Pages written by only one node are simply invalidated on other nodes by the DSM. However, for pages written by multiple nodes (i.e., the page written by W2 and W5 on Node0 and W4 and W6 on Node1), the DSM layer must generate a diff from the copy created during the first write fault. Non-default-owner Node0 creates a diff by applying an exclusive or (XOR) operation between the original and updated version of the page. The DSM transfers the diff to Node1, which merges the diff with its copy of the page. After all invalidations and merges have been performed, the work-sharing region ends and the DSM transitions back to SC. Note that COW pages are only used during the MW merge phase if nodes write to the same page and therefore necessitate a merge; however, the DSM layer cannot predict whether this will happen and must copy the page, even for pages only updated by a single node.

5.2.2 Profiling Page Prefetching (PF)

Similar to the MW protocol, the primary goal of aggressive page prefetching is to reduce the number of read page faults. Resolving read page faults incurs long latency – the CPU must undergo a mode switch and the DSM system must transfer each page between nodes using the communication layer. Rather than fetching pages on-demand during the middle of computation, the DSM layer prefetches pages in batches at the beginning of a work-sharing region to both reduce interruptions during computation and better utilize the available network bandwidth. Currently, the prefetcher profiles read operation patterns during the initial execution of a work-sharing region and prefetches the same pages before the next invocation of the same region.

5.2.3 Smart Regions

The MW protocol's benefit does not come without its drawbacks. For a non-default-owner node, an extra memory copy is required upon the first write to a page. Also, the DSM must record all written pages and copied pages in a list. During the MW merging phase, the DSM iterates over the lists to find conflicts that need merging. If the number of invalidation messages is not significant, the MW overheads may cause a performance hit versus SC. Many compute-intensive applications have repetitive regions exhibiting similar DSM consistency traffic. The DSM system records each region's behaviour, and if the number of invalidation messages is not above a certain threshold, the DSM system smartly skips switching to MW protocol for subsequent executions of the region.

5.2.4 Cross-Node Barrier Synchronization

User-space synchronization primitives are based on futex to synchronize by relying on a shared memory model. One way to provide synchronization across nodes is to delegate futex operations based on DSM. It allows transparently handling existing Shared Memory (SHM) synchronization primitives. When running across incoherent domains, SMP programming model is just an illusion provided by the DSM system; traditional futex operations transfer many redundant pages due to SC (i.e., single-writer). When multiple threads on different nodes try to write to a same page, the data page and its ownership will bounce back and forth. In this subsection, we will explain the entire process of a such traditional cross-node synchronization and its problem. Finally, how these overheads can be avoid by using our design.

When a thread enters a synchronization barrier, it atomically fetches and increases a shared counter to determine if it is the last arriving thread. During this process, the DSM layer grabs the page containing the counter and invalidates its permissions on other nodes. Subsequent threads entering the synchronization repeat this process until the last thread arrives. Threads waiting on a condition variable call a wait futex operation to wait in an in-kernel queue. Once awoken, they load a user-space futex variable to synchronize on the counter. The last arriving thread invokes a wake futex operation and increments the user-space futex counter (a write operation). The futex variable is automatically synchronized by the DSM layer, causing redundant page faults on multiple nodes.

To synchronize the in-kernel futex wait queue, futex operations must be delegated to the same node. Before waiting in the queue, futex waits verify that the user-space futex address still matches the wait condition and then sleep until a futex wake on this futex address. While verifying these conditions, the DSM layer locks the pages containing the user-space counters to prevent any access. This entire process may cause up to 2 more pages faults due to the lack of read permissions. All in all, when multiple threads on different nodes try to synchronize with each other, the data and ownership of the page will bounce back and forth.

The DSM layer provides a new primitive to synchronize cross-node threads in a single system call. This function does two things; first, it broadcasts a notification message to other nodes; second, it spins until receiving a response from all other nodes for the same synchronization point. Upon returning from this function, the cross-node synchronization is finished. This causes zero page faults.

5.3 Implementation

MWPF is built on top of the latest version of Popcorn [25], which implements inter-node thread migration and DSM in kernel space.

We implemented the MW DSM protocol (Section 5.3.1), page prefetch (Section 5.3.3), smart region (Section 5.3.2), and cross-node synchronization (Section 5.3.4) mechanisms at the operating system level. We also adopted an OpenMP runtime (Section 5.3.5) that transparently integrates the aforementioned mechanisms for work-sharing regions. Thus, OpenMP applications targeting simultaneous cross-domain execution in heterogeneous-ISA systems can benefit from these new mechanisms without changing any lines of application code.

5.3.1 MW Protocol

The MW protocol enables multiple writers to concurrently operate on the same page without coherency to amortize networking latency overheads. The MW protocol has to determine which node is the default-owner. In our setup (described in Table 6.1), because the Intel Xeon has better single-threaded performance than the Cavium ThunderX, it executes the serial phases of the application and therefore is chosen as the default-owner. Doing so lowers the number of page faults incurred during the serial phase since the default-owner gains exclusive ownership when merging pages.

The kernel maintains per work-sharing region (per-region) data structures to record relevant information (hashkey, pages written, statistics). The MW protocol's implementation can be split into two phases – the MW parallel phase and the MW merging phase.

MW Parallel Phase. When there is a write to a read-only shared page inside a work-sharing region (a write fault), the fault will be trapped by the OS-level page fault handler. During the write fault handling process, the DSM system records the virtual address of the faulting page in a hashmap contained in the per-region data structure; this hashmap is used to construct the delayed invalidation message and detect conflicts during the MW merging phase. Additionally, if it is the default-owner node, it simply adds write permissions to the corresponding page table entry (PTE) and returns to user space. If it is not, before correcting the PTE, the DSM layer duplicates the page and records the address of the copy in a per-node hashmap indexed by virtual address. After returning to user space, subsequent read or write operations to the page will not incur any further page fault.

MW Merging Phase. Once threads exit a work-sharing region, the DSM enters the MW merging phase. In the merging phase, nodes exchange lists of pages written during the work-sharing region to handle invalidations and detect which pages were written by multiple nodes and therefore need merging. The DSM layer batches invalidation requests up to a tunable threshold (4087) by iterating over the hashmap containing write-faulting pages.

Plus, due to OpenMP work-sharing semantics, no two threads will ever write to the same address without synchronization, meaning the merge process will never have to resolve conflicting writes to the same address. We adopt differential logging [106] to both create a differential between the current and copied page and apply the differential to the same page on the default-owner.

5.3.2 Smart Regions

The DSM layer can decide whether to use the MW protocol for each work-sharing region. The kernel records execution characteristics from previous invocations of the same work-sharing region to make protocol selection decisions for the next invocation of those regions. If a region is marked as non-beneficial, threads entering the work-sharing region avoid MW meta-data initialization and fall back to SC.

The DSM layer determines whether a region is beneficial at the end of the MW merging phase by recording the number of invalidation messages. If the number of invalidation messages is lower than an *invalidation threshold*, then using the MW protocol does not pay off and it is declared a *probational region*. If a region is determined to be a probational region for consecutive iterations, it is marked as a non-beneficial region and falls back to SC for the rest of execution. Our current invalidation threshold is set to be the core count on its node. That is, each thread should on average have more than one invalidation request in order for the MW protocol to be beneficial; if not, it is considered a probational region. The probation threshold is set to 10. We found that some applications have different numbers of invalidation requests for the same work-sharing region. The probation threshold is utilized to gather more profiling information across several invocations of the region to make more accurate decisions regarding whether the MW protocol will be beneficial.

5.3.3 Profiling Page Prefetching

The in-kernel per-region data structure also records all read faults. The DSM layer uses this record to prefetch those pages in the next execution of the same work-sharing region. The prefetcher relies on the fact that many HPC applications execute the same work-sharing region with the same input/output buffers multiple times, leading to repetitive page access patterns for executions. To identify repeated invocations of a work-sharing region, the DSM layer uniquely identifies regions with a key using the containing function's name, line number, file name, and iteration. This information is supplied to the kernel by the OpenMP runtime (Section 5.3.5). The key and per-thread hashmap can be used to quickly determine whether the region has been previously executed.

During the first program execution of a work-sharing region, the kernel records read-faulting pages in a hashmap. Upon subsequent program executions of the region, the prefetcher iterates over the faults observed from the previous execution. The prefetcher sends requests as batches, maximizing the number of pages fetched in a single message to better utilize network bandwidth. Upon receiving the response message, the prefetcher maps the pages by fixing permissions and copying the page content to the proper pages. Once pages have been placed across nodes, the threads are released to begin computation.

5.3.4 Cross-Node Barrier Synchronization

To replace the traditional fetch-and-add and futex synchronization used in the SHM programming model (5.2.4), we implemented a new system call invoked by threads to synchronize across nodes. Each kernel maintains a local barrier counter and a remote barrier counter per remote node. The system call does two things. First, it increases the local barrier counter by 1 and sends a message to the other nodes for increasing remote barrier counter by 1 on remote nodes. Second, the node spins until the remote barrier counter is equal to or larger than the local barrier counter (i.e., until the remote nodes have sent corresponding response messages for synchronization). Upon completion of spinning, the thread is released back to user-space to continue execution.

5.3.5 Runtime Support

To utilize the previously described mechanisms, we modified Popcorn's OpenMP runtime (which is derived from GNU's libgomp) to integrate the new MW protocol and prefetching. Popcorn's OpenMP runtime provides the ability to migrate threads of a team executing a parallel region between nodes to take advantage of remote compute resources. Our modifications to the runtime added/removed 11(+), 3(-) lines of code for replacing futexes with our primitive, 54(+) for the hash function, and 33(+) for our API library.

Hashmaps for Regions. We use hashmaps owned by every thread to identify when a thread is entering a previously seen region. This hashmap is different from the one used for recording written page addresses in subsection 5.3.1.

To identify repeated regions, the DSM layer constructs a key using the current function's name, line number and file name. The key and per-thread hashmap can be used to quickly determine whether the region has been previously executed.

Regions. Popcorn's OpenMP runtime marks the start of a work-sharing region with calls to `__kmpc_dispatch_init()` or `__kmpc_static_init()`. When entering a work-sharing region, the thread performs a system call to set the MW region flag inside the thread's in-kernel descriptor. At this point the DSM layer decides whether to enable prefetching (if the region has been previously seen) or logging of read faults (if it is the first time executing the region).

When a thread hits the work-sharing region exit point (denoted by calls to `__kmpc_dispatch_fini()` or `__kmpc_static_fini()`), it will unset the MW region flag. At this point the MW merge phase will begin and propagate writes between nodes.

In between an MW parallel phase begin and end, there may be barriers besides the implicit end-of-work-sharing barrier. In this case, the DSM does not directly exit the MW region. Threads reaching the barrier will instead invoke the MW merging phase to force a consistent memory view across nodes. It is required to propagate barrier state updates across nodes so that threads on one node see when remote threads arrive at the barrier, thus preventing deadlock.

Barriers. OpenMP uses explicit and implicit barriers to synchronize threads. We invoke our OS-level synchronization primitives inside the OpenMP runtime in place of the traditional SHM barrier to optimize cross-node synchronization, which are upgraded from single node barriers, GOMP_barriers().

Chapter 6

Evaluation of the MWPF Protocol

In this chapter, we evaluate MWPF (Chapter 5) and present its performance results. A prototype is built by adapting an OpenMP runtime library leveraging the proposed primitives in the Linux kernel. A micro benchmark and many real-world applications are used to evaluate the performance of the MWPF prototype.

Section 6.1 describes the hardware setup for evaluating MWPF. In Section 6.2, we present a micro benchmark for understanding performance improvement of adopting our synchronization design. Section 6.3 evaluates MWPF on different applications which cover a variety of different memory access pattern. Finally performance results are summarized in Section 6.4.

6.1 Experimental Setup

We evaluate our DSM system through a series of micro and macro benchmarks in order to understand how the MW protocol, aggressive page pre-fetching, and new synchronization primitive improve cross-domain performance. Table 6.1 displays our setup. All applications were executed by saturating the available cores in a given configuration. i.e., 16 threads for homogeneous x86-64 execution, 96 threads for homogeneous ARMv8 and 16 + 96 threads for heterogeneous setups. We envision in the near future heterogeneous domains (likely 2 domains) will appear in datacenters. So, we experiment on a X86-ARM combination integrated via a high speed interconnect to mimic the envisioned future architectures. We implement our prototype using Linux kernel 4.4.137 and OpenMP 4.5 [34]. In our evaluation,

Table 6.1: Experimental setup.

Machine	Intel Xeon E5-2620	Cavium ThunderX
ISA	X86-64	ARMv8
Cores	8 (16HT)	96 (48 * 2 socket)
Clock (Ghz)	2.1 (3.0 boost)	2.0
LLC Cache	L3 - 16MB	L2 - 32MB
RAM (Channels)	32GB (2)	128GB (4)
Interconnection	Mellanox ConnectX-4 56Gbps	

we answer the following questions. With zero lines of application code changed,

- How does the new cross-node synchronization primitive improve barrier performance? (Section 6.2)
- How much does the MW protocol improve performance vs. SC? Is the smart region heuristic able to accurately determine when the MW protocol will and will not improve performance? (Section 6.3.3, 6.3.2)
- How much performance improvement does aggressive data page prefetching provide? (Section 6.3.4)

Benchmark Applications. High performance computing (HPC) applications are prevalent in the cloud. We choose the following applications to represent HPC applications. We select Blackscholes (BLK) from PARSEC (native input) [149]. BT Class C (BT-C), SP-C, EP-C, CG-D from the C + OpenMP version of the NAS Parallel Benchmarks [21] (NPB) from Seoul National University [162]. Due to compiler limitations, CG-D is slightly modified to dynamically allocate memory instead of using statically allocated memory as originally written. This does not change the behaviour or the execution time of the application. We additionally select lava molecular dynamics (LavaMD/LAVA), lower–upper decomposition (LUD) and Hotspot2D (HS) from Rodinia [44], which is a benchmark suite for heterogeneous computing. Last, we include an in-house OpenMP version of Kmeans (KM) written by slightly modifying the pthreads version of Kmeans from Phoenix [175]. These benchmarks cover a variety of different computation and memory access patterns.

OpenMP loop-based work-sharing regions distribute parallel work by assigning loop iterations to threads participating in the thread team. Because we use a heterogeneous setup (Table 6.1) where each server consists of different numbers and different types of cores, we manually skew the ratio of work distributed to each thread based on each core's relative performance. For example, for a particular application a Xeon core may provide 3x the performance of a ThunderX core; the OpenMP runtime would give the Xeon core 3x as many loop iterations in this scenario. This ratio is determined experimentally per application. As the focus of this work is not to determine the optimal loop iteration scheduler policy, we leave exploring dynamic scheduling policies as future work.

6.2 Micro Benchmarks

Cross-node Synchronization. To understand the performance improvement gained from the new synchronization primitive, we ran a microbenchmark that executes fifty thousand OpenMP barriers in a loop across both nodes. We created a thread team consisting of 16 threads on the Xeon and 96 threads on the Cavium for a total of 112 threads across the system. The average barrier time for cross-node execution using the original and optimized barrier is shown in Figure 6.1. For the single-node cases, X86 (16 cores) and ARM (96 cores), synchronization contention overhead caused by OpenMP barriers is only inside the cache hi-

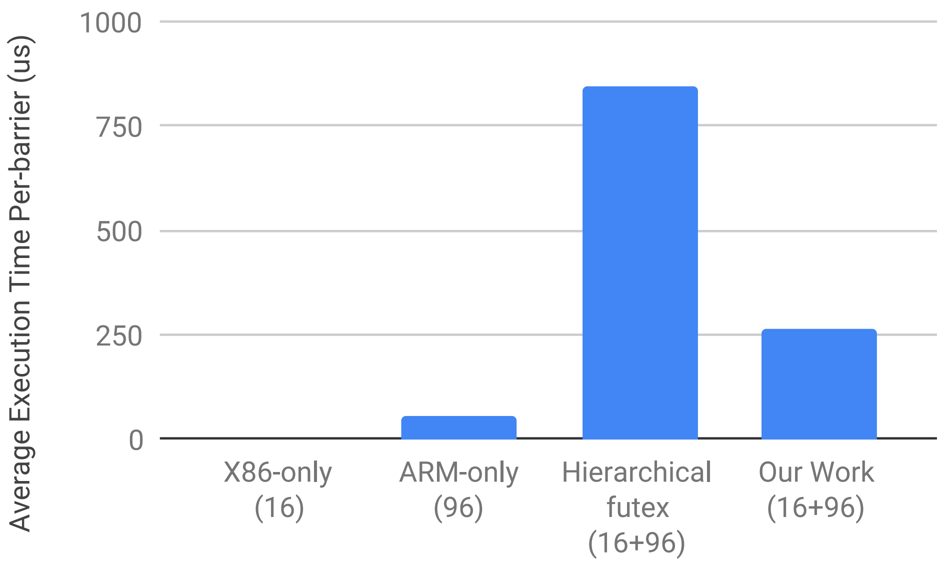
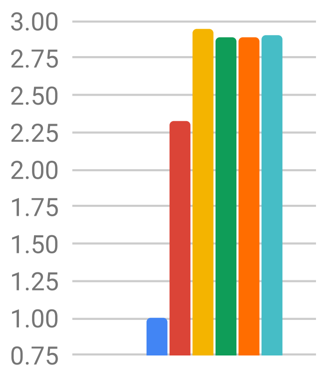


Figure 6.1: Average execution time per barrier.

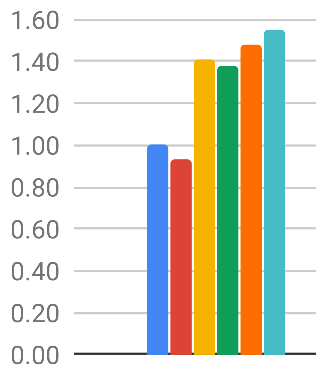
erarchy (as opposed to DSM which causes cross-node page transfers). The remaining two bars are correspond to the original futex-based barrier as implemented inside the OpenMP runtime over multiple nodes and barriers instrumented using in-kernel message passing for synchronization. With the message-passing-based synchronization design, barriers demonstrate a 3.18x speedup compared to solely relying on shared memory and traditional futexes. This shows removing redundant page transfers by using an OS-level cross-node synchronization primitive provides large speedups versus relying entirely on the DSM abstraction while providing a flexible design as well.

6.3 Real Applications

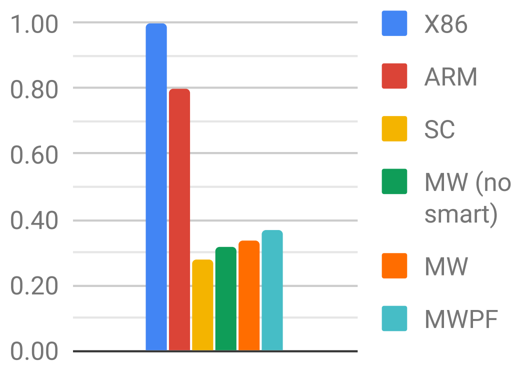
To show our performance improvement, we use an kernel-space SC DSM as our baseline [161]. First, this is the closest working prototype on real non-cache-coherent domains even though they cannot run a multithreaded process across nodes. Plus, SC is always the goal standard among DSM protocols because when using a weaker memory model, developing and debugging applications is extremely hard e.g. race condition. Indeed, there are some optimized or relaxed DSM protocols (Section 3.2.1), but they require code modification and only support heap-allocated objects. Without modification to application code, these DSM systems do not support legacy applications.



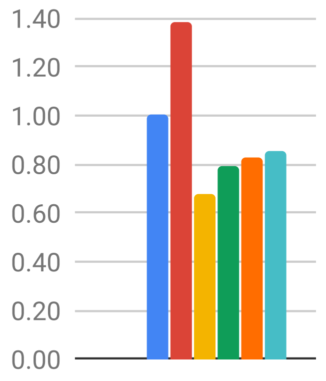
(a) EP-C



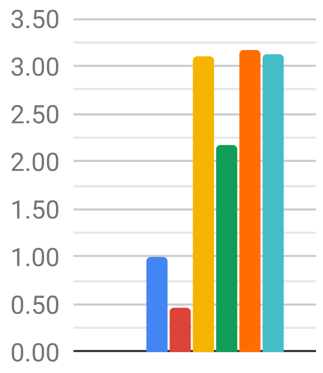
(b) CG-D



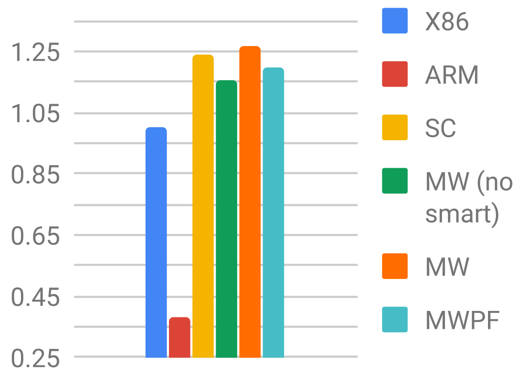
(c) SP-C



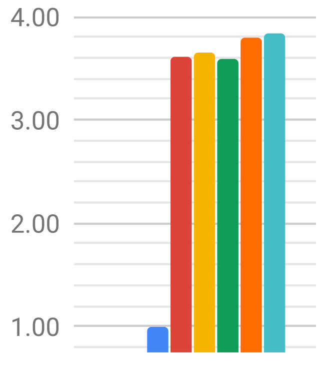
(d) BT-C



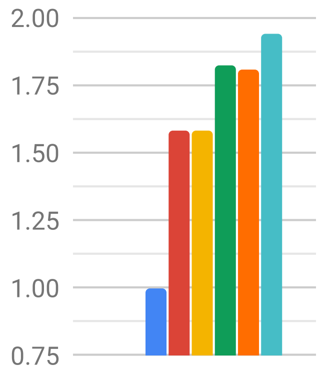
(e) BLK



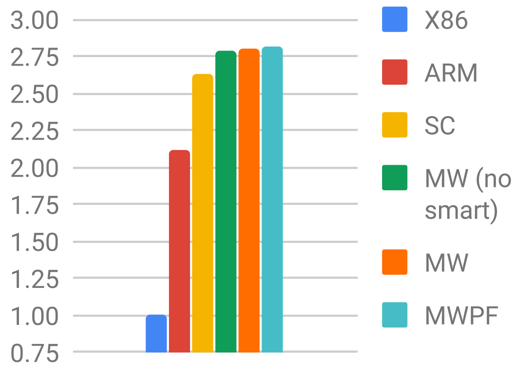
(f) HS



(g) KM



(h) LUD



(i) LAVA

Figure 6.2: Speedup of benchmarks normalized to single node execution time on the X86 for several system configurations.

6.3.1 Experimental Results and Explanation

Figure 6.2 demonstrates the speedup compared to single-node execution on X86 for several system configurations on 9 benchmarks. For each application, the different colored bars on the x-axis show speedups for single-node X86 with 16 threads (always 1), single-node ARM with 96 threads, cross-node with execution with 112 threads (16 Xeon + 96 Cavium) with the SC DSM protocol (SC), cross-node with the MW protocol and no smart regions (MW no smart), cross-node with the MW protocol and smart regions (MW), and cross-node with the MW protocol, smart regions and prefetching (MWPF). SC, MW no smart, MW and MWPF all transparently execute SMP applications on two nodes with 16 + 96 threads for parallel regions by leveraging Popcorn's thread migration ability [25]. SC does not use any of the optimizations described in Section 5.2. Conversely, except SC, all adopt the improved cross-node synchronization primitive. We present the result of each application as a smaller plot (Figures 6.2a∼6.2i).

Our final result shows 7 out of 9 applications perform better than single-node execution when using our DSM design. Among these 7 applications, 6 applications are originally scale-out across nodes when using SC [25] compared with running on a single node. These 9 applications experience improved performance when applying the DSM optimizations included in our design. Compared to SC, MW's average speedup over all 9 benchmarks is 8% and up to 22% for BT-C and SP-C. MWPF is up to 33% faster for SP-C and on average 11% faster than SC. We argue that 8%∼11% average speedup is very significant with the constraint of not modifying any line of code of existing SMP applications.

6.3.2 MW Protocol and Smart Regions

In many cases, solely using MW without applying the smart region design, MW (no smart), will make performance worse versus SC – for example, CG-D, BLK, HS, and KM all suffer from using MW without smart regions. This is because the MW protocol is not overhead-free. For those regions with only a few invalidation requests, the MW protocol's overhead will be larger than the benefits brought by the protocol. This is solved by our smart region design. Applying the MW protocol and smart region mechanisms together (MW), all benchmarks except EP, LUD, and LAVA experience performance improvements. The 6 applications that benefit from smart regions have multiple independent work-sharing regions inside the benchmark. Some of the regions are not be able to benefit from the MW protocol. Thus the smart region automatically reverts to SC for these non-benefit regions to avoid excessive MW protocol overheads. Other regions inside these benchmarks do benefit from the MW protocol. For example, BT and SP perform matrix operations, which have a variety of different computations spread across different work-sharing regions and thus some of the regions have very short computation and others have long computation. Another extreme example is BLK, which has about 500 iterations of a single work-sharing region. However, it has very few invalidation requests in each iteration. Hence, the smart region heuristic reverts the DSM protocol to SC. Contrarily, for EP and LAVA, we believe smart regions do not provide significant benefits due to the small number of work-sharing regions (less than 10) in these applications. Because of this, the smart region heuristic does not gather enough information before making a protocol determination. LUD has around a thousand regions. None of invocations of any region are determined as non-beneficial regions. This means the smart region only brings overheads such as collecting data and making decision as none of the regions should fall back to SC. Although using smart regions slightly impairs performance, LUD, LAVA, and EP still run faster using a cross-node configuration than any single-node case. This shows smart regions preserve stable performance results and provide moderately better performance in many cases.

6.3.3 MW Protocol with Smart Regions versus SC

BT, SP, LUD, LAVA, KM, and CG all benefit from the MW protocol; 98.63%∼99.9% of all invalidation messages are batched, allowing applications to enjoy smaller overheads. Among these applications, LUD is the most interesting case as it has parallel regions benefiting from running on a cross-node configuration. However, simply using SC for LUD is not faster than solely running on ARM because the computation benefit afforded by multiple machines is eliminated due to high communication costs. With the MW protocol, invalidation messages in LUD are reduced by 99.77% across nodes. And, without using our design, around 4% of the writes cause page ownerships to keep bouncing between nodes. The redesigned DSM boosts LUD's performance by up to 14% compared to using SC. Additionally, when using the MW protocol, BT and SP run 22% faster versus using SC. Conversely, BLK, HS, EP iteratively compute using well-partitioned memory access patterns that mostly do not generate many write faults. While these 3 applications do not benefit from the MW protocol, they do not have obvious performance degradation as well. This shows even for the worst case of using the MW protocol, it has very low overhead.

In summary, using the MW protocol with smart regions, BT, SP, LUD, LAVA, KM, and CG benefit from disabling MW for only a selected few regions while enabling it for others. The result shows an average speedup over all 9 benchmarks of 8% (with up to 22% for BT-C and SP-C) versus SC. This result proves that the MW protocol indeed can bring better performance by delaying and batching invalidation messages until the end of the region (or barrier) and by solving page false sharing problem.

6.3.4 Profiling Prefetch

When using profile-guided prefetching, the results can be categorized into two types. BT, SP, CG, LUD, and KM benefit from the profiling prefetch mechanism. We recorded the number of read page faults reduced in work-sharing regions inside these applications. We discovered 84%∼99% of total cross-node page faults were eliminated for these applications after applying the profiling prefetch design. These pages are aggressively prefetched in batches before entering the work-sharing region so as to reduce run-time page fault handling overheads and the latency that the interconnect introduces. HS, however, suffers from 4% slowdown because our current prefetch implementation requires nodes to synchronize for prefetching before entering each region. The overhead of synchronizing is larger than the benefit. The concept of smart regions can also be applied to solve this problem. For, LAVA, BLK, and EP, prefetching does not provide benefit as the applications have only a small number of read faults to begin with. Pre-fetching will not bring much benefit but instead will only add overhead.

6.4 Summary

A prototype implementing operating system level primitives and a slightly modified OpenMP runtime to adopt the primitives are built for evaluating our design. We presented a micro benchmark indicating our synchronization mechanism reduces redundant DSM traffic. Nine applications are selected to represent HPC workloads in the cloud to evaluate our MWPF system.

- With MWPF, synchronization operations in particular barriers can be 3.18x faster since MWPF eliminates redundant page transfer cased by traditional futex design.
- The average speedup over all 9 benchmarks for MW is 8% faster and up to 22% faster versus the baseline SC DSM with the constraint of not changing a single line of code.
- Over the 9 benchmarks MWPF transparently increases on average by 11% and up to 33% compared with the baseline SC DSM implementation without legacy SMP program modification.

Chapter 7

The PuzzleHype Distributed Hypervisor

In this chapter, we propose PuzzleHype which aggregates fragmented resources from multiple machines to provide a single VM to the guest operating system. We provide several design principles for building such a system. PuzzleHype software design leverages DEX's DSM and thread migration abilities to achieve a single VM illusion. However, this is not enough, PuzzleHype furthermore proposes three important design principles for aggregating distributed CPUs, RAMs, devices in a virtualization context. First, interrupts such as I/O interrupt and Inter-Processor Interrupts (IPI) are used to communicate with CPUs. This means that PuzzleHype needs to manage interrupts for virtualizing distributed CPUs located on different nodes. Second, memory synchronization is also important to run application transparently. DSM layer is therefore utilized for memory synchronization across nodes. Plus, EPT violation handler should be aware of and intergrated with the DSM layer. Last but not least, since devices and computing resources may be located on different nodes, PuzzleHype uses I/O delegation mechanism to overcome such problems, making our system more applicable.

This chapter is organized as follows: Section 7.1 describes resource fragmentation problems in datacenters and provides an overview of how PuzzleHype solves the problems. The design and architecture of PuzzleHype are presented in Section 7.2. Section 7.3 explains how we implement such a system in the Linux kernel.

7.1 Overview

Computing resource fragmentation in modern datacenters is a serious problem [56, 70, 73, 136, 151, 176]. Fragmentation arises due to several factors, such as the wide variety of resource requirements for jobs, an increase in the average amount of resources required for jobs [52, 126], job placement constraints such as software/hardware dependencies [155, 165], and the fact that the main unit of resource increment (a physical server) is different from the unit used to describe job resource requirements (RAM, cores, etc.) [164].

Even with optimized placement methods or attempts to defragment resources through migration, resource fragmentation still persists [70, 147, 164, 176]. Although hardware disaggregation [66, 78, 110, 111, 164, 176] may be an elegant solution in the long term, it is not yet a viable solution, and datacenter operators are seeking solutions to reduce fragmentation given current hardware.

We propose PuzzleHype to address this problem. It transparently distributes the classical unit of job execution, the Virtual Machine (VM), over fragmented resources belonging to different physical hosts. This incurs no additional machine purchase, resource usage, or downtime due to migration. Although our system presents some unavoidable performance overhead when accessing remote resources, we demonstrate that these slowdowns are much lower compared to solutions based on overcommitting. Thus, PuzzleHype presents a novel approach to improving resource fragmentation in datacenters.

PuzzleHype focuses on letting a VM leverage physical processor cores that are scattered among different host machines interconnected via a network. Toward this aim, we develop a distributed hypervisor which creates a VM spanning several hosts. In a way that is fully transparent to the guest OS, the hypervisor implements the distribution of virtual CPUs (vCPUs), pseudo-physical memory, and I/O devices. The hypervisor runs as one instance per host, where it allows the creation of vCPUs. This hypervisor leverages a Distributed Shared Memory (DSM) system implemented in the (Linux) kernel of the hosts, transparently presenting to a guest a unified and consistent view of a continuous pseudo-physical address space. Finally, we allow a vCPU to access a remote virtual network card by modifying the *virtio*/*virtio-net* paravirtualized architecture.

By abstracting the distribution of cores, memory, and devices into a system-level hardware interface similar to that of a classical VM, PuzzleHype is *transparent*: it can execute fully unmodified guest operating systems within a distributed VM, although some slight modifications to these OSes can bring significant performance improvements. More importantly, PuzzleHype does not require any modification to legacy user-space software, which can run as-is.

By leveraging scattered resources that naturally occur in the datacenter, PuzzleHype decreases fragmentation and thus increases utilization. It can either help to increase throughput by running more jobs on a given set of hosts, or it can reduce cost and power consumption by running a given set of jobs on a smaller set of hosts.

PuzzleHype differs from existing solutions by aggregating physically separated resources into a Single System Image VM. It leverages a popular KVM-based type II hypervisor and does not require the use of custom type I virtual machine monitor that may be hard to deploy [42, 180]. Our solution also uses standard networking equipment and does not require the use of (potentially expensive) special hardware [9, 156]. Recent works also propose to distribute a VM to run scale-out, compute-intensive workloads [194]. However, PuzzleHype focuses on reducing fragmentation by distributing all types of workloads, including I/O intensive ones, such as web servers. We solve similar issues but take a different approach compared to existing works on disaggregated computing [66, 78, 110, 111, 164]. Indeed, our objective is the *aggregation* of scattered resources in existing datacenter hardware, rather

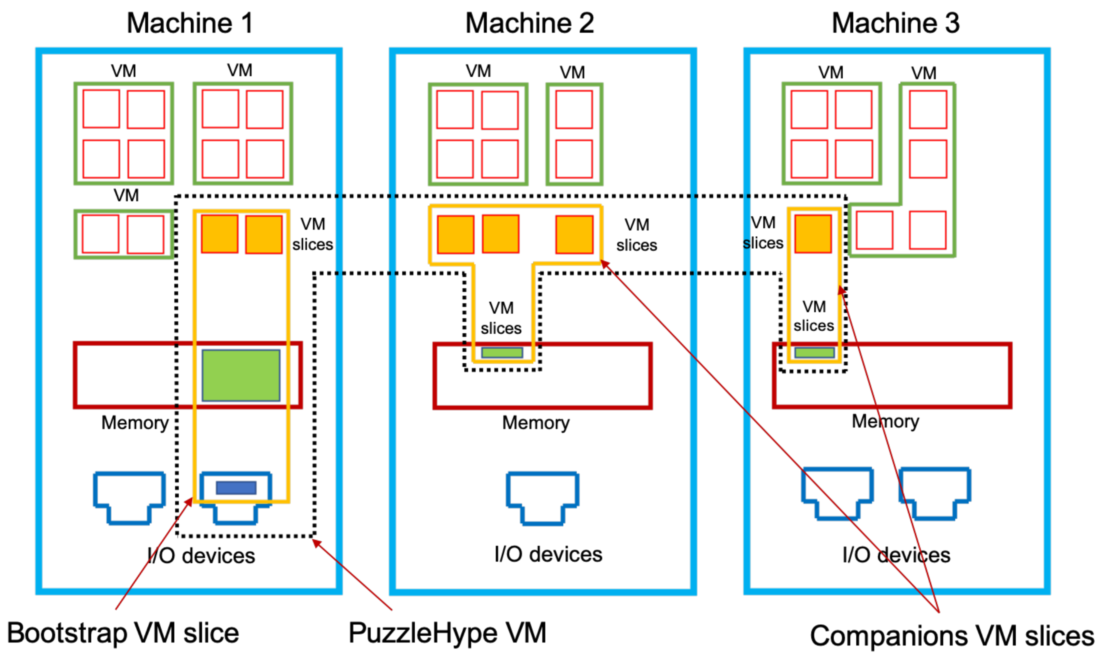


Figure 7.1: PuzzleHype (yellow) selectively exploits physical hardware resources of multiple server machines. Each traditional VM (red) employs hardware resources from a single machine.

than the abstraction of hypothetical future disaggregated hardware. In addition, contrary to these works, we target commodity servers and thus our solution is directly deployable in today's datacenters.

7.2 Design and Architecture

With the goals of (A) fast provisioning (faster than VM migration), (B) minimal overheads (lower than resource overcommitment), and (C) datacenter-wide heterogeneous resource exploitation, PuzzleHype aggregates slices of fragmented hardware resources from multiple server machines into a single VM. The proposed system is based on the following design principles: (A) compatibility with existent software bases, i.e. guest OSes and applications. (B) minimal overhead and interference to other VMs or applications on a node; (C) high performance;

Based on these principles, we designed PuzzleHype around traditional monolithic UNIX-like OSes, full-virtualization, and targeting a type-2 virtualization architecture. Moreover, we envision PuzzleHype deployed in modern datacenters where servers are connected via high-speed network technologies (e.g., 56 or 100Gbps). PuzzleHype creates VMs using hardware resources belonging to different server machines. Therefore, as depicted in Figure 7.1, a PuzzleHype VM exploits physical CPUs, RAM, and devices owned by different servers (M1, M2, and M3 in the Figure).

Operational Principles When a PuzzleHype VM is created, a PuzzleHype instance is started on each of a set of servers. Any instance is started by specifying the exact hardware resources such as CPUs, I/O devices, and memory which should contribute to the VM, including physical hardware descriptions – a *VM slice*.

One of the instances is responsible for establishing the connection among all and is responsible for starting the VM execution – thus, it should provide at least one virtualized CPU. Such an instance will be started with additional information about the other instances (hosts IP address only in most cases), as well as the disk image(s) and eventually the kernel and boot-loader to be executed. We call this instance a *bootstrap VM slice* on origin node; other instances are called *companion VM slices* on remote nodes. Despite the VM bootup phase, all VM slices are peers in PuzzleHype.

Grand Architecture PuzzleHype is a type-II multiple-hypervisor VMM [141]. Each VM slice runs on a different PuzzleHype instance. Instances communicate between each other via messages. The communication layer is implemented at the host OS kernel level, similarly to multiple-kernel OSes (Popcorn [25], Barrelfish [27], etc.), in order to reduce the user-kernel switches and improve performance. Atop the communication layer, PuzzleHype implements a set of distributed hypervisor services to provide the illusion of a single VM among multiple physical machines. First and foremost, the entire VM RAM (vRAM) is made available on each physical machine by distributed shared memory (DSM). Secondly, VM CPUs (vCPUs) may be created or migrated on different physical machines, and physical CPUs (pCPUs) do not have to be homogeneous. Finally, VM devices on any physical machine should be used by software running on any vCPU, such as interrupt controllers and timers not part of a vCPU. Note that each of these objects requires a different consistency level, and we envision their implementation in kernel-space provides higher performance.

VM Orchestration. In the datacenter, PuzzleHype does not take any decision itself about what servers will be used by a VM. A datacenter scheduler/orchestrator decides on what servers a PuzzleHype VM will run, and for each server the total amount of resources needed. Such a scheduler knows about the managed cluster's already allocated hardware resources and upcoming resource requirements of incoming tasks. Hence, we decided not to integrate placement-like functionalities in PuzzleHype. However, PuzzleHype does require that datacenter schedulers be rewritten, because current schedulers cannot exploit partial resources. Nonetheless, this is not that different from supporting heterogeneity. We left this aspect as future work.

The rest of the section is organized as follows. subsection 7.2.1 defines how PuzzleHype manages the system memory on host. subsection 7.2.2 describes how it aggregates CPU among different nodes. subsection 7.2.3 explains how I/O devices are virtualized in such a system.

7.2.1 Distributed vRAM

Modern type-2 hypervisors hold the guest pseudo-physical address space as a subset of the virtual address space of a host user-space application. Thus, PuzzleHype applies OS-level software DSM to keep that part of the address space consistent among different machines, providing the illusion of shared pseudo-physical memory for the guest OS. Parts of the address space that do not belong to the guest pseudo physical memory area usually belong to emulated devices, and thus are handled outside of the DSM protocol.

Despite heavy criticism of software DSM in the past for its overhead, we believe that the high-speed interconnect available in the datacenter makes DSM a viable option. Moreover, we make the following observations. First, the hypervisor knows a lot about the content of the guest physical address space, especially about CPU-dependent memory areas, including the memory location of the MMU, interrupt table, etc. In SMP OSes, these are shared among all CPUs, and are highly used. Therefore, it is of the utmost importance to avoid the DSM protocol from slowing down their access. Thus, we propose a contextual DSM protocol that leverages information about the memory content to reduce DSM traffic.

With the goal of reducing DSM traffic, PuzzleHype informs the guest software about the non-uniform access latencies by exposing a NUMA topology related to the placement of hardware resources on different server machines. In fact, each physical server machine provides to a PuzzleHype VM a chunk of physical memory. When a server provides a physical chunk, the server is the DSM home-node for that chunk. Other than a chunk of memory, each PuzzleHype VM slice is characterized by a total amount of memory that can be used by the DSM protocol to eventually cache remote memory. In this way, other processes in the system can run without PuzzleHype disturbance.

7.2.2 Distributed vCPU

A vCPU runs in a thread context in a modern type-2 hypervisor. Hence, PuzzleHype requires either the capability of spawning a thread on another machine or thread migration [25]. Threads can be spawn on one host and migrated to another machine later on. We believe an implementation should pick one or the other based on the features offered by the host OS.

Each vCPU has its own set of registers, which may include a local interrupt controller and timers (e.g., x86's Local Advanced Programmable Interrupt Controller (LAPIC) timer). There is usually no shared state among different vCPUs if not for processor-wide registers,

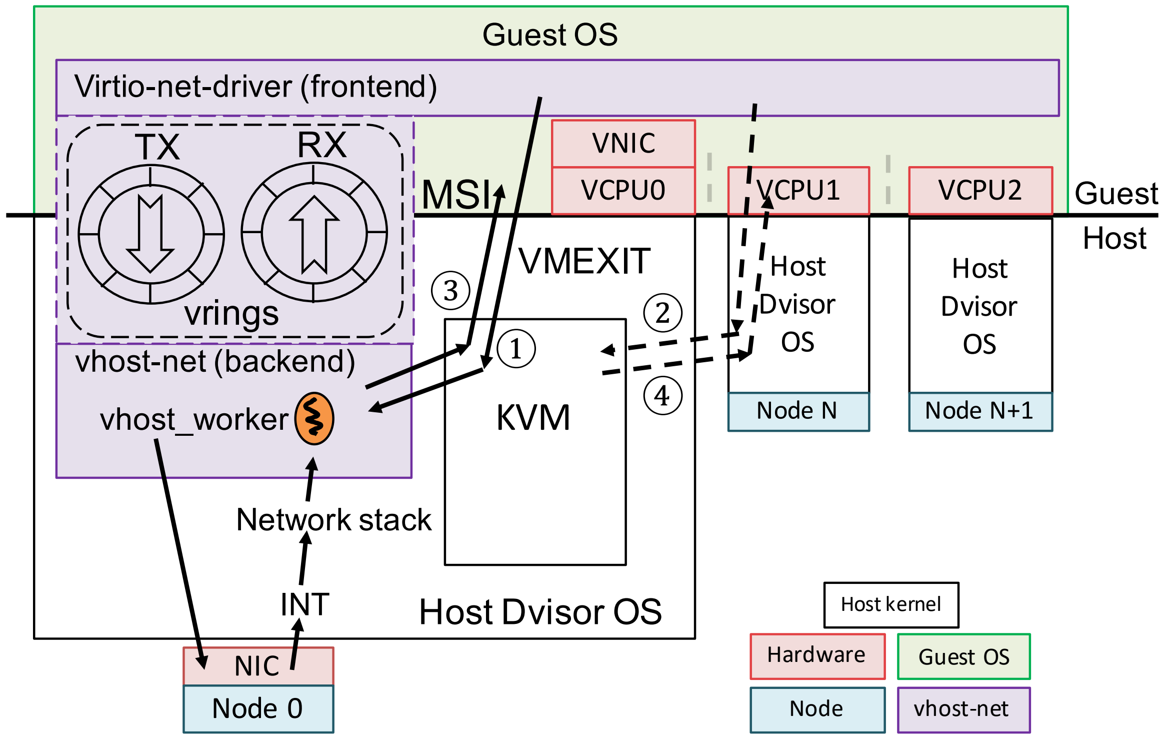


Figure 7.2: Virtual Network Devices Delegation.

such as some MSR registers in x86. PuzzleHype keeps the state of such processor-wide registers consistent among PuzzleHype VM instances. Finally, CPUs may communicate between each other via IPI, including MSI. Thus, each PuzzleHype VM instance keeps track of the node of each vCPU via a vCPU location table. IPIs to remote vCPUs are converted into messages to PuzzleHype VM instances.

Distributed Virtual Programmable Interrupt Controller (vPIC) Despite the local interrupt controller of CPUs, one or more non-local interrupt controllers may exist in a VM (e.g., x86' IO-APIC). As this non-local interrupt controller is usually an interrupt broker and interrupts are messages on our communication layer, we keep this component non-replicated and on the host that has the highest number of physical hardware devices. The alternative design is to implement this component as a distributed one.

7.2.3 Distributed Virtual Devices

Distributed virtual device access is based on the concept of delegation, i.e., guest VM software running on any PuzzleHype instance should be able to access any device exposed by the VM, but the actual communication with the physical device happens only on the PuzzleHype

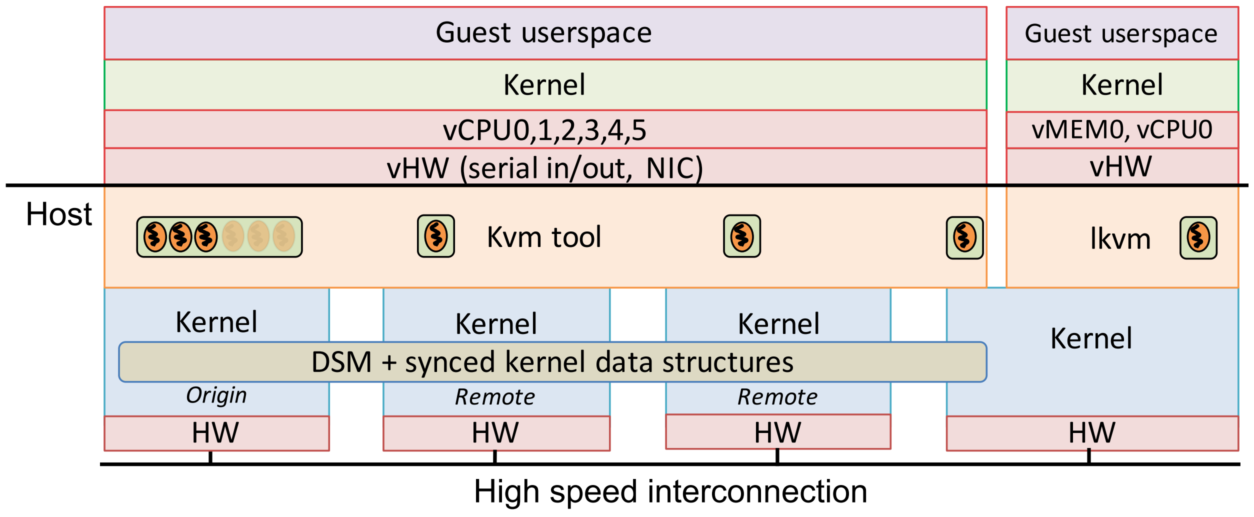


Figure 7.3: Software architecture.

instance running on the same physical server as the device.

Devices may communicate with the CPU either via memory-mapped IO or IO ports. We consider only the former, although our design can be extended to support also the latter. Specifically, we support PCIe-based devices by establishing a distributed PCIe root complex and devices emulator. Thus, virtual devices do not work via the DSM protocol. At the same time, as many I/O devices including PCIe devices work by instantiating ring buffers on RAM memory, such ring buffers are managed by the DSM, but with optimizations (explained below).

In order to bound our engineering efforts, we herein focus on paravirtualized hardware (based on *VirtIO*). We believe this is not a limitation of our design as it can be trivially extended to full-hardware virtualization. We discuss architectural details of the network interface card and a serial console as representative examples.

Networking Figure 7.2 shows how the proposed architecture leverages the host NIC. A paravirtualized network device exposes simple TX/RX ring buffers that the guest uses to enqueue or dequeue packets going to or coming from the physical network card. Because these ring buffers are on shared memory, the DSM protocol maintains them consistently across kernel instances. However, the host kernel that enqueues packets on these rings acts like an additional node for the DSM. Therefore, at any point the host kernel is writing, it needs to be the only one that owns the page of data. We observe that this can be very expensive, so we integrate the DSM and the paravitualized network device in order to release pages as soon as they are no longer used.

Moreover, this solution allows VM slices that do not own a network device to delegate the transmission of network packets to other VM slices by simply writing a packet on DSM and sending an interrupt to notify a new packet has been written.

The paravirtualized console is working similarly to the paravirtualized networking.

7.3 Implementation

We implemented a prototype of PuzzleHype (Figure 7.3) based on Linux kernel 4.4.137 using Intel processor architecture. To avoid reinventing the wheel, we based our implementation on different Linux kernel components from the Popcorn Linux project [161], including thread/process migration, kernel-level DSM, and the messaging layer. Our target *Type-2* hypervisor is Linux/KVM for x86-64 platforms, and we extend *kvmtool* (commit c57e001) as the user-level tool for creating and managing guest VMs. Our implementation accounts for a total of 9,500 LoC in the OS kernel, and 4,100 LOC in user-level tools.

7.3.1 DSM and Communication Layers

We adopt DEX's KDSM (Section 2.2) and Xfetch's messaging layer over IB/RDMA (Chapter 4) to build PuzzleHype.

A distributed virtual machine requires the host to keep data content consistent across nodes. Such a single VM address space illusion on multiple nodes can be implemented by DSM. There are two main categories of DSM, hardware DSM and software DSM (Section 2.1). For being general-purpose, flexible, and portable, our work does not consider hardware DSM. The work instead leverages software DSM.

There are two different software DSMs implementations, user-space DSM (UDSM) and kernel-space DSM (KDSM). UDSM is more commonly used and easier to develop. However, it requires redundant mode switches between host kernel and user modes. Recent VMes have been putting efforts on reducing user-kernel mode switches. Examples are virtio-net and vhost-net (Section 7.2.3). KDSM, on the other hand, does not bring redundant user-kernel mode switches. EPT violation and page fault handler naturally sit in the host kernel space. For performance reasons, there is no need to move the DSM implementation inside page fault handler to the userspace. We argue that adopting KDSM can bring better performance compared with running SDSM.

DSM is well-known for inter-node communication latency. High speed interconnections such as InfiniBand (IB) and remote direct memory access (RDMA) [181] technologies helps the usability of DSM [132, 138]. PuzzleHype's communication layer exploits remote direct memory access (RDMA) over high-speed InfiniBand (IB) to minimize inter-server messaging latency and increases throughput.

Although (k)DSM can synchronize user space data, it cannot synchronize data structures on host kernels. This will cause information asymmetry across nodes. Synchronizing host kernel information, including vCPU 7.3.3 and IO 7.3.4 metadata, is of important and necessary. Plus, memory information asymmetry happens at secondary page table and TLB as well 7.3.2. These are huge components must be addressed by the system and will be explained soon in the following subsections.

7.3.2 Distributed VM Memory

Thus, our DSM provides sequential consistency and allows multiple readers and a single writer access to memory.

DSM handles the memory consistency of the entire guest virtual RAM. Therefore, unlike traditional DSMs that handle host page table faults only, our DSM also needs to handle EPT faults. In fact, an EPT violation may trigger a DSM request for a remote page. After the DSM protocol fetches the corresponding page, the EPT mapping has to be fixed, and the VM may resume executing. Also, traditional DSM only invalidates the host page table by making the page table entry (PTE) non-present and flushing the corresponding TLB. However, with a second level of translation, our DSM needs to invalidate EPT's SPTEs and flush the secondary TLB as well.

Figure 7.4 shows our DSM in the case of two servers for simplicity, but this can be extended to any number of servers. A guest OS page fault, translating Guest Virtual Address (GVA) to Guest Physical Address (GPA), is handled by its MMU page structure as-is ①. Once fixed, the guest VM will then access the corresponding GPA. The main difference is that accessing GPA now may cause a EPT fault on the host. The GPA is emulated by Host Virtual Address (HVA) after all. EPT involves a second-level page table, the Extended Page Table, used to map from GPA to HPA. The table is maintained by VMM/hypervisor. If the EPT can find the mapping in the table ②, it directly accesses the Host Physical Address (HPA) without causing a VM exit. Otherwise, an EPT violation occurs and triggers a VM exit ③. Host kernel tries to translate the GPA to GFN and use the memory slot info to convert it back to HVA ④.

Once the HVA is determined, the host OS performs host page table walk to get the HPA ⑤. During the host page table walk, the DSM will be triggered if the page data is not located on the local node ⑥. The DSM will grab the remote page content or invalidate remote permission and finally fix the permission locally. In the end of the EPT violation fault handler, EPT saves the map from GPA to HPA to provide better performance for accesses at a later time ⑦.

Moreover, traditional DSM only invalidates host page table by making the Page Table Entry (PTE) non-present and flush the corresponding TLB. However, this is not enough in distributed hypervisor memory management. The system needs to invalidate EPT's SPTEs

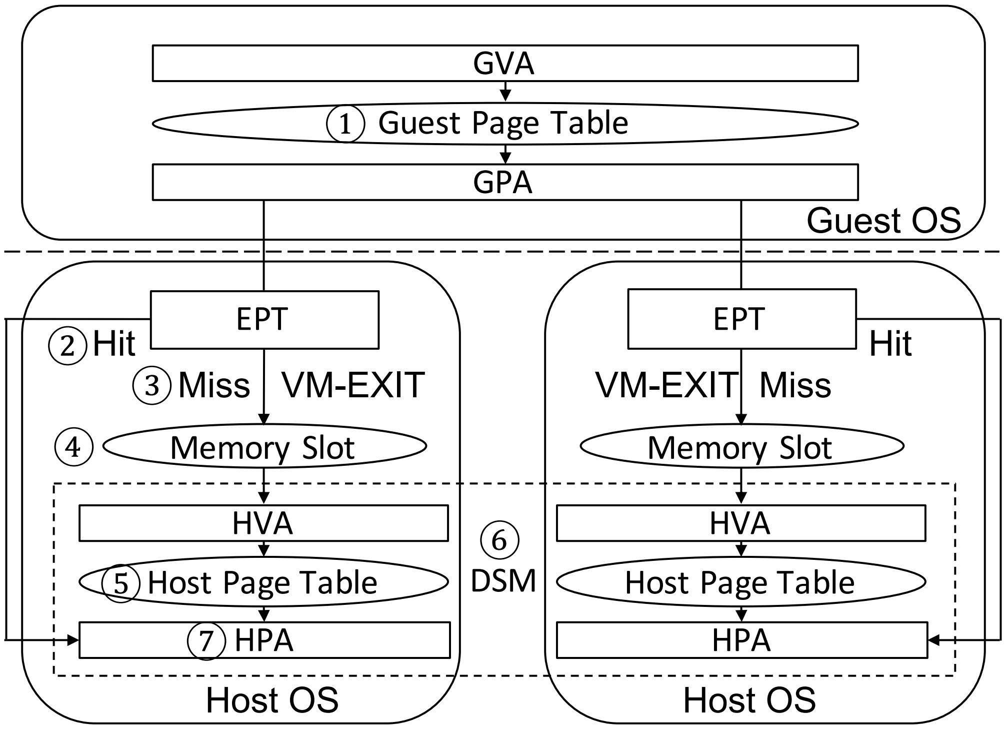


Figure 7.4: PuzzleHype DSM functioning principles.

and flush the secondary TLB as well. These SPTEs can be recreated by the EPT violation routine. Plus, it will restart any ongoing secondary page faults.

Our EPT violation routine keeps retying DSM protocol until it gets the page. While retrying, the implementation must follow the locking semantics in the page fault handler. That is, between each retry, there should be a time window without holding the VMA lock. Otherwise, this breaks memory management locking semantics and thus a deadlock will happen. After getting the host physical page, the host kernel then fixes the EPT mapping and enters the VM, resuming the guest mode.

In addition, in the EPT violation path, the host kernel may ask for multiple pages at once. A DSM implementation constraint is observed if node A tries to fetch a page which has been fetched by node B, and node B is still fixing other pages after the page. In this situation, the page permission invalidation becomes very complicated and deadlock could happen easily if there is no sophisticated protocol. So, our current implementation simply disables the prefetch mechanism in the EPT violation routine.

7.3.3 Migrating vCPUs

Instead of spawning remote vCPU threads, our prototype lets the *Bootstrap* VM slice create all vCPUs threads locally and then migrate vCPUs threads to *Companion* VM slices. We adopt Popcorn's task migration to distribute vCPUs among multiple server machines. However, migrating vCPU threads is not enough to provide distributed vCPUs. We augment the existent thread migration to account for additional vCPU-related metadata as described above. This data doesn't need to be kept consistent among all server machines. The prototype maintains on each VM slice instance a replicated array that tracks the position of every vCPU, and is updated at each migration event. Each VM slice instance is therefore aware of where each vCPU is, and this is fundamental for the functioning of the communication layer.

Interrupts vCPU threads are the recipients of interrupts. Interrupts are transformed into messages transmitted over the communication layer. When the destination of a message is a VM CPU, a message is dispatched to a vCPU thread. In the prototype, we modified the hypervisor routines for interrupt dispatching in order to check if the target vCPU is local or remote. If it is remote, a message with the description of the interrupt event is sent to another VM slice instance. Otherwise, the traditional code path is followed.

In Intel processor architecture, these interrupt messages are achieved by the Advanced Programmable Interrupt Controller (APIC). In symmetric multiprocessor (SMP) systems, the APIC consists of many local APIC (LAPIC) as an equal number of CPUs and a I/O APIC. Inter-Processor Interrupts (IPI) are triggered by having the LAPIC write to the Interrupt Command Register (ICR).

When a vCPU tries to send an IPI to another vCPU, it first traps to VMM. The VMM then injects the interrupt into the corresponding vCPU's host data structure. VMM then pauses execution of the target vCPU in the guest VM if the vCPU is running. The target vCPU injects the interrupt by writing its LAPIC ICR on VMM and then enters the guest VM again. Upon entering the guest VM, the target vCPU receives an IPI notification and then trigger the corresponding interrupt routine inside the VM. Our system forwards IPI messages when the source vCPU and target vCPU are not on the same node. The system hence has to maintain a mapping table to convert from vCPU identity (ID) to KVM vCPU kernel data structure.

7.3.4 Distributed VM Devices

As disclosed above, we rewrote most *virtio*-based and emulated `kvmtool` devices such as console, PCI, and network devices (all but not 9p, balloon, SCSI, VESA, BLK) in order to make them work atop the communication layer. Thus, a device physically located within a specific VM slice can be used by all VM slices. More details follow.

Network The prototype leverages Linux's vhost-net as its foundation for networking. As mentioned in Section 7.2.3, the vring data synchronization is transparently covered by the DSM. Remote nodes do not have to reallocate vrings on its node. Even if our distributed PCIe layer takes care of the physical PCIe address ranges, and the DSM replicates the physical memory content, we still need to properly handle the notifications between the guest and the host in a distributed environment. Thus, each node has to install the corresponding file descriptors (*ioeventfd* and MSI *irqfd*) to notify the guest and host to access the rings.

In the network sending routine, a VM exit trap happens on a host node when the guest VM has a network package in the send queue from the front-end *virtio* network driver. The front-end driver puts the package into the vring and passes the vring index to the hypervisor. The hypervisor then signals the vhost network worker via *ioeventfd* (①② in Figure 7.2). The awakened vhost network worker allocates a Linux network buffer (skb) and copies data from user space to the skb. After data is delivered to the skb, the process will be taken over by the host kernel network stack and eventually sent out the network package via the NIC.

In the network receiving routine, MSI IRQ injection support is required. Once the host NIC receives a package, it detects if the destination is a guest VM. If so, it copies the data to the RX vring buffer and notifies the guest VM to check the vNIC package. *irqfd* is used for injecting MSI interrupts to the VM from the host (③④ in Figure 7.2). The MSI interrupt signals a vCPU to check the vring which emulates a network buffer on a NIC. Eventually the vring has to be dequeued from the right vCPU. Once the data is touched, the DSM may cooperate by bring the network received data from the host. Our prototype currently always injects the interrupt to vCPU0 simply because the current implementation does not predict the network package belongs to which vCPU. Another way could be the origin host node can peek the package and directly notify the right vCPU. This may reduce redundant IPIs.

Serial Console Although this is not performance critical, without supporting console, it is almost impossible to develop or debug the system. Considering Secure Shell (SSH) requires network support to run, before things are perfectly working, it is still good to have the console working. Virtual console is achieved by virtIO mechanism as well. The main idea is to let VCPUs on host node handle tasks related to serial console.

Furthermore, to help the serial working correctly on the *Bootstrap* VM slice, the guest VM sets serial port IRQ affinity to be on a vCPU located on the *Bootstrap* VM slice. In our implementation we set it on origin node. Additionally, the system has only one terminal worker thread emulating a serial Universal Asynchronous Receiver-Transmitter (UART) chip. Only the host origin node can receive serial input and output. Serial echo needs to take user input (RX) and displays it (TX) on the console. It polls to catch the input characters and saves them into the emulated serial chip's buffer. It then injects an interrupt into the guest VM and notifies the VM to handle the interrupt. The guest VM gets an interrupt and moves the buffer data from the emulated chip. Once it is done, it will reschedules the rest of TeleTYpewriter (TTY) related works to any CPU available in the system. This causes a VM exit and eventually sets the interrupt pin level to HIGH. However, in distributed hypervisor systems, there is only one emulated chip located on origin node. If the work is completed at remote node and trapped to the VMM on a *Companion* VM slice, there is no emulated chip can help the process. That means PuzzleHype needs to make sure the work will not be handled on any remote VCPU. IRQ affinity does not help as well since the work is redistributed in the interrupt handler. Thus, the system forces such TTY handling work can be only dispatched to the VCPUs on origin node in that the thread emulating the chip is located on origin node.

Chapter 8

Evaluation of PuzzleHype

This chapter evaluates PuzzleHype by using a combination of micro and macro benchmarks. The web server stack and serverless computing frameworks are software used for the benchmarks. Since our system is distributed which incurs a little overhead to our mechanism/design, we want to understand the overhead. Two micro benchmarks are proposed; one for demonstrating the EPT faults overhead and another for presenting the network delegation overhead. Furthermore, one High Performance Computing (HPC) macro benchmark NPB suite is selected to evaluate our system performance for computation intensive workload. Last, a web page network stack and a serverless computing framework are used to evaluate how PuzzleHype can help and perform for cloud applications in the datacenter.

Section 8.1 shows our hardware configuration for evaluating PuzzleHype. Section 8.2 presents some micro benchmarks to better understand the system overhead in particular EPT faults and network delegation. Section 8.3 evaluates PuzzleHype on real network application stacks such as network application stack and serverless computing framework. Section 8.4 summarizes the performance results.

8.1 Experimental Setup

To evaluate PuzzleHype, we created distributed VMs on top of several physical hosts, whose hardware characteristics can be found at the top of Table 8.1. Concerning the VMs, while we vary the number of vCPUs and their distribution amongst hosts, we fixed the characteristics described at the bottom of Table 8.1. For all experiments, vCPUs are pinned on pCPUs. We use micro-benchmarks to evaluate the cost of accessing remote resources (memory and I/O). We also measured the overall performance of PuzzleHype by running real-world compute-/memory-intensive (NAS Parallel Benchmarks) and I/O-intensive (LEMP server, serverless framework) benchmarks representative of modern datacenter workloads. For these, we compared the performance of PuzzleHype's distributed VMs to overcommitting vCPUs, a technique that allows fitting additional jobs on a saturated (and potentially fragmented) cluster.

Table 8.1: Host/Guest setup.

Host Machines Configuration	
Machine	Intel Xeon E5-2620
ISA	X86-64
Cores	8 (16HT)
Clock (Ghz)	2.1 (3.0 boost)
LLC Cache	L3 - 16MB
RAM (Channels)	32GB (2)
Interconnection	Mellanox ConnectX-4 56Gbps
Host Kernel	Linux 4.4.137
Guest Virtual Machines Configuration	
Guest Kernel	Linux 4.4.137
RAM	20GB
Disk	Custom Ramdisk (raw format)

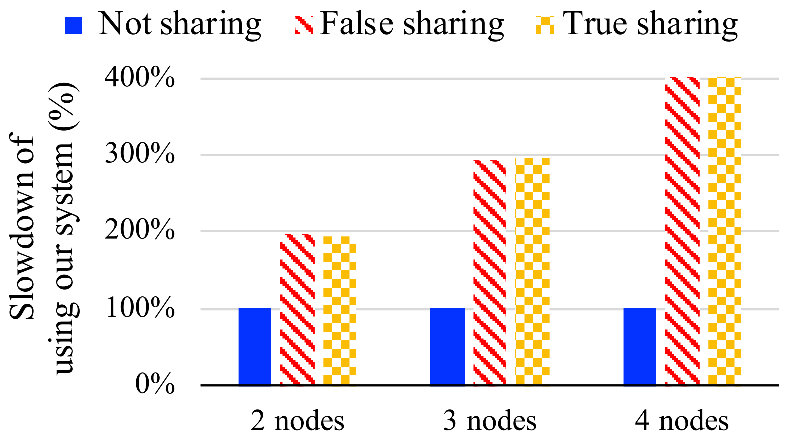


Figure 8.1: EPT traffic.

8.2 Micro Benchmarks

DSM Fault Traffic. The DSM overhead in PuzzleHype takes the form of relatively costly EPT faults due to the implemented consistency protocol. We designed a micro-benchmark to understand that overhead. We created a program to read and write in a loop at a configurable location in a virtual page. By tuning this location and running one instance of this program on several vCPUs of a distributed VM, we created 3 scenarios involving: (1) true sharing, (2) false sharing, and (3) no sharing between remote vCPUs. Note that the no sharing case is similar to running on a non-distributed VM. We ran the experiment on VMs having 2 to 4 vCPUs, each from a separate physical host, and measured the loop execution time.

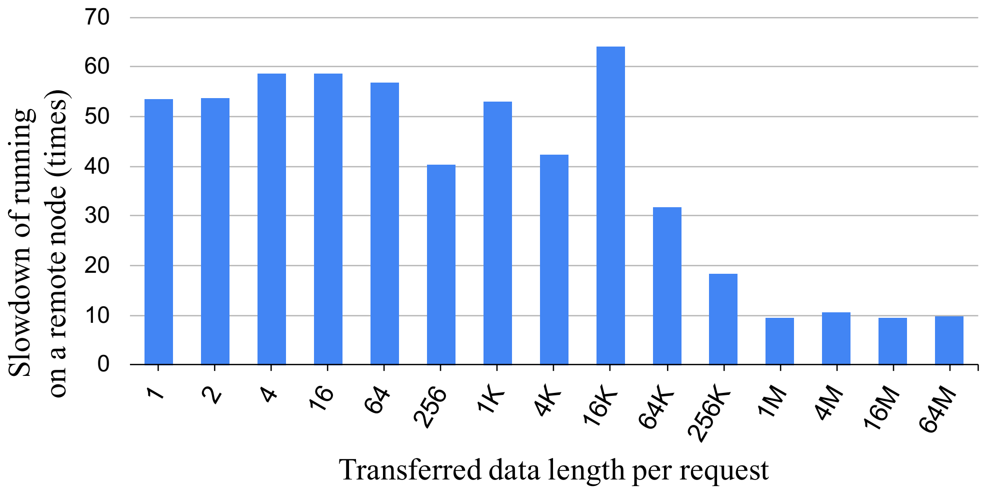


Figure 8.2: Network delegation overhead.

Results are presented on Figure 8.1, with the loop execution time on the Y-axis normalized to the "no sharing" execution time. When remote memory is accessed, the loop execution time increases linearly with the number of nodes involved, i.e., it doubles for 2 nodes, triples for 3 nodes, etc. False and true sharing cases result in the same behavior; the same physical page is accessed.

Network Delegation Overhead Another overhead from the system is I/O delegation, specifically network delegation. To evaluate that overhead, we compare the network throughput of a NGINX web server where the worker runs (1) on a vCPU (vCPU0) that is local to the host's virtual switch used to communicate with the client and (2) on a vCPU (vCPU1) that is on a remote node. The client runs ApacheBench (AB) [2] on a node on the same 1Gb Local Area Network (LAN). It sends 1000 requests with 10 concurrent connections to the web server. We vary the served page size and measure throughput.

Results are in Figure 8.2, where throughput on the Y-axis is normalized to the throughput when running on vCPU0. Although the slowdown is very high on small requests (40-50x below 16KB), it decreases to 10x when serving 1 MB or more. Network delegation is thus more efficient on throughput-oriented workloads and should probably not be used for latency-sensitive applications.

8.3 Real-World Applications

With PuzzleHype, a datacenter operator can pack more jobs on a saturated but fragmented cluster. A common solution to achieve this goal without PuzzleHype is to overcommit re-

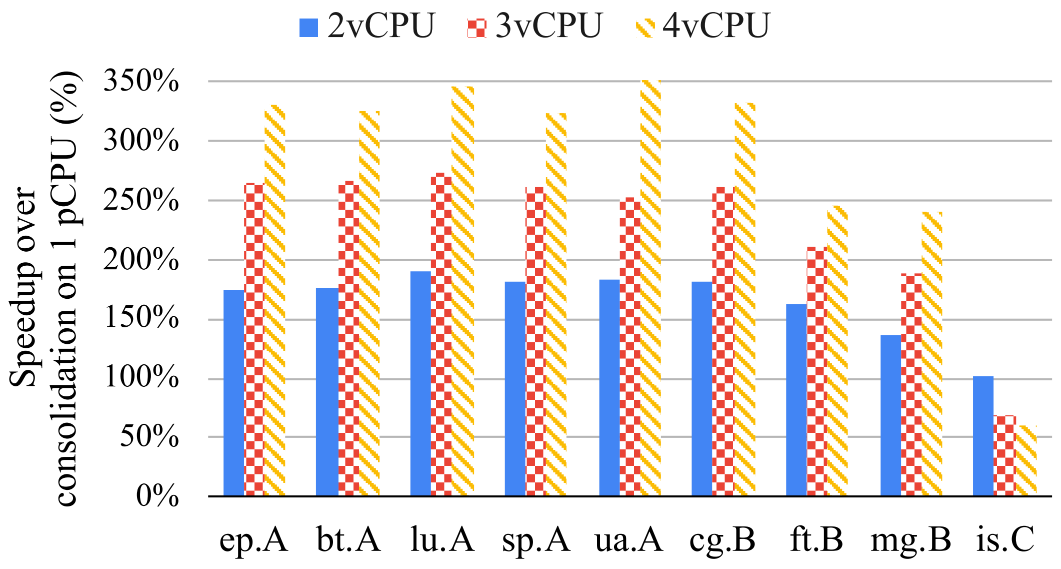


Figure 8.3: NAS Parallel Benchmark results.

sources. We evaluated PuzzleHype on real-world applications by comparing the performance of distributed VMs versus traditional VMs with overcommitted vCPUs. We varied the number of vCPUs to be 2, 3, and 4. In the case of the distributed VM, each vCPU runs on a different host.

High Performance Computing (HPC) Applications. We ran the NPB suite, selecting for each benchmark a data set size which would result in an execution time of at least 10 seconds. For each benchmark and VM setup, we run in parallel one instance of the serial version of the benchmark on each vCPU and measure the total execution time of this set of instances.

The results are shown on Figure 8.3. The y-axis represents PuzzleHype's distributed VM speedup normalized to the overcommitting case where a traditional VM's vCPUs are consolidated on a single on pCPU. Speedup ranges from 60% to 355% and the average is 285%. EP, BT, LU, SP, UA, and CG have significant performance improvement. On the other hand, IS sees a slowdown. We notice that IS includes a memory allocation phase that is longer than the subsequent computation phase and we estimate that the slowdown is due to DSM contention that results from kernel data structure synchronization during that allocation phase. FT and MG also share the same problem but unlike IS, their computation phase is long enough to amortize the overhead.

To conclude, for most of the applications requiring high computation power, PuzzleHype can efficiently increase performance compared to overcommitted scenarios. OS-intensive tasks, such as memory allocation, bring more overhead due to kernel-level DSM contention.

Network Application:

LEMP Stack. LEMP [3] is a open-source web service stack consisting of NGINX [5], PHP [8], and MySQL [4]. In this setup, NGINX acts as a front-end HTTP server receiving requests from the client. PHP is invoked in response to requests, fetching back-end data and performing various processing operations to create the web page that will be sent back to the client. MySQL is a database management system for storing data and providing interfaces for services such as PHP. In this experiment, we do not use the database.

We run the LEMP stack on the distributed VM as follows. First, NGINX is configured with a single worker thread running on the vCPU that is local to the virtual switch used by the VM to access the network (VCPU0), so as to avoid network delegation overheads as demonstrated in Section 8.2. Upon receiving a client request, the NGINX worker thread invokes PHP which runs a worker thread on each vCPU except the one running the NGINX thread. In other words, the 2 vCPUs configuration has 1 NGINX worker on the first vCPU, and 2 PHP worker on the other. Similarly, the 3 vCPUs configuration has 1 NGINX worker on the first vCPU and 3 PHP workers on the others.

We set the served page size to 2MB, the average page size on the web according to recent statistics [19]. We vary the amount of computation realized by PHP by adding a busy loop of configurable length in the PHP code invoked upon reception of a client's request that leads to the building of the served page. We vary these PHP computations to last from 10ms to 400ms, which is relatively representative of modern servers' response times [58], averaging from 200 to 500ms [134]. AB is used as the client, running on the host node that runs the pCPU where the vCPU running the NGINX worker is mapped. AB is configured to make 100 requests with 10 concurrent connections. We collect the reported throughput and compared PuzzleHype' results to an overcommitment scenario where all vCPUs run on a single pCPU.

The results are presented on Figure 8.4, where the throughput speedup on the Y-axis is normalized to the throughput of the overcommitment case. When the PHP processing time is short, our system does not perform well due to expensive communication between NGINX and PHP workers. Within the guest, this communication goes through a local socket between the NGINX worker and the PHP workers. This is obviously slower in the case of the distributed VMs, as the two communicating processes are running on two separate physical machines.

Starting from 100ms of PHP processing per request, the communication overheads start to become lower compared to the pure computing time, and are amortized by the benefits of leveraging remote computation resources in parallel. The speedup increases with both the demand for computation time (PHP processing) and the number of vCPUs. For example with 4 vCPUs and a 400ms PHP processing time, the speedup is 225%.

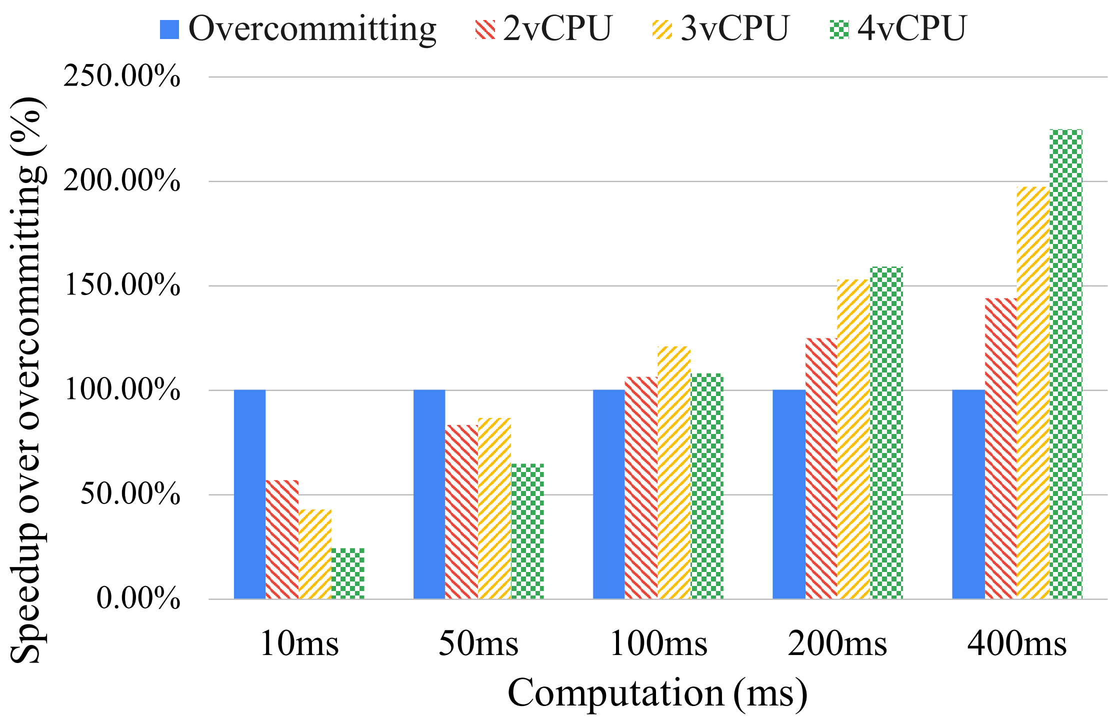


Figure 8.4: Customized LEMP - NGINX and PHP with an adjustable busy loop.

We also relaunched these experiments with a different page size of 150 bytes and obtained similar results.

Serverless Computing Framework. Serverless computing, also known as Function-as-a-Service, is an emerging programming paradigm where users upload and execute small pieces of code, or named emphfunctions, in the cloud. Due to many benefits such as scalability by design and truly on-demand pricing, the use of serverless computing is expected to skyrocket in the years to come [93]. All main cloud providers are already offering FaaS solutions [15, 72, 125].

For this experiment, we used the open-source OpenLambda [82] serverless computing framework. In a PuzzleHype distributed VM, we run the OpenLambda server configured to spawn functions upon reception of a HTTP request. Functions can be spawned on each vCPU of the VM. All functions run the same python code, that (1) fetches a series of pictures as a compressed file from a database on the same LAN via the network, then (2) runs a face detection algorithm [68] on these images and returns the number of detected faces to the client. This behavior is typical of serverless computing applications.

A client, running as a separate machine on the same LAN as the host, sends requests to trigger function execution. We vary the number of parallel requests to be equal to the number

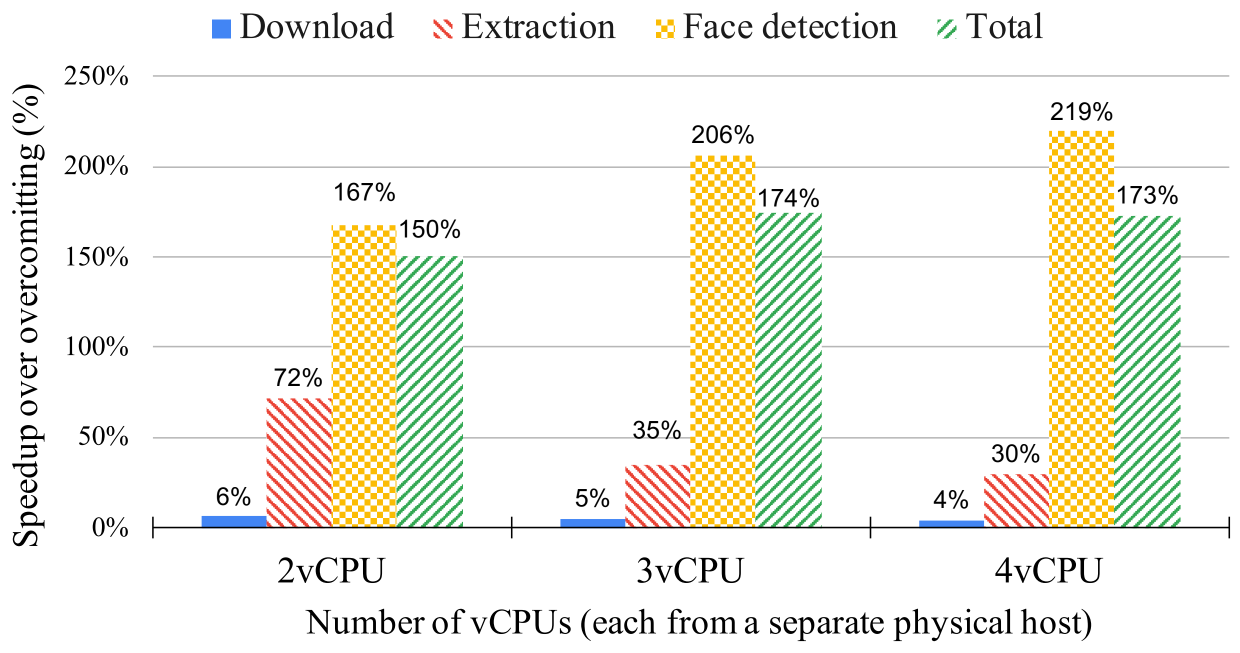


Figure 8.5: OpenLambda breakdown.

of vCPUs: 2, 3 and 4. We report a breakdown of the server-side execution times average speedups including downloading the pictures, compressed file extraction, face detection, and total time. The results are presented in Figure 8.5.

All results are normalized to the overcommitting cases where 2, 3 and 4 vCPUs are consolidated on the same pCPU. The speedup increases with the number of vCPUs because the overcommitted VM sees an increases in its execution time as it needs to run more processing on the same number of computing resources (1 physical core). Regarding the extraction time, even if there are write operations to different pages, the first write to a new region on a remote node always causes write-exclusive invalidation messages by DSM, contributing to the slowdown of the extraction as the number of vCPU increases.

The overcommitting cases do not suffer from network delegation overhead. For this reason, our download time is prolonged and the total improvement less notable.

Although the I/O delegation mechanism can be optimized, aggregating remote CPU resources brings performance improvement in general. The face detection phase is considerably sped up (up to 219% for 4vCPU) with PuzzleHype. Each vCPU processes on its own images parallelly. It also dominates the most time of the lambda function, and the overall performance still outperforms than the overcommitting cases (∼174% for 3/4vCPU). This indicates our system can be beneficial in datacenters or serverless computing systems.

8.4 Summary

We evaluated PuzzleHype's performance at the scale of a rack, over a set of micro- and macro-benchmarks.

- Compared with running on a single machine, in PuzzleHype, accessing shared memory regions in guest VM is no longer close to free overhead. Whereas, in PuzzleHype if more than one pCPUs are on different machines, extra DSM overhead may be generated. According to our micro benchmark, the overhead almost linearly increases with the number of nodes concurrently accessing shared data increases.
- Network delegation provides a mechanism for a remote vCPU which does not have a host physical NIC to send a network package as long as there is one physical NIC in the system. Our micro benchmarks show there is IO delegation overhead. When transferred data length per request is larger, the more efficient it is. We also found when
- The results show that with 4vCPUs, PuzzleHype versus overcommitment:
  - SNU NPB suite representing HPC workload demonstrates a speedup up to 355% and on average by 285%,
  - Customized LEMP, a web service stack, presents a speedup of 225% with 400ms PHP processing time,
  - OpenLambda, a serverless computing framework, experiment provides a 173% speedup.

Chapter 9

Conclusions

This chapter summarizes the main contributions of the dissertation and draws conclusions. Additionally, it proposes directions for post-preliminary research.

Section 9.1 revisits the dissertation's contributions. Section 9.2 presents post-preliminary examination work.

9.1 Contributions Revisited

With single-threaded CPU performance reaching a limit, chip vendors are designing a vast array of novel computing architectures to accelerate performance. Among many design points in the architecture space, an interesting point that is being studied by the academic research community and one that is also gaining increasing traction in the commodity setting is heterogeneous-ISA multiprocessors with multiple cache-coherency domains and no inter-domain coherency. This dissertation investigates the abstraction of software distributed shared memory for such architectures in order to improve their programmability. DSM gives the illusion of an unified, globally consistent single address space for such architectures, enabling them to be programmed as a symmetric multiprocessing machine.

The dissertation presents three contributions. First, the dissertation presents the Xfetch mechanism for the DEX DSM system, leveraging high performance network interconnects such as the InfiniBand network infrastructure. Xfetch exploits spatial locality, and aggressively and sequentially prefetches pages before they are accessed, improving network throughput and DSM performance. Our experimental evaluations using an eight-node, 56 Gbps InfiniBand-based rack-scale system reveal that, with a maximum 32KB payload size per message, Xfetch achieves up to 142.74% speedup and up to 58.56% lesser remote page faults over the baseline DEX that does not prefetch page data.

Second, the dissertation presents the MWPF DSM protocol that leverages OpenMP's fork-join regions where weaker consistency semantics are allowed by delaying synchronization updates, which permit greater parallelism and higher performance. Additionally, MWPF includes a number of optimizations including delaying and batching invalidation messages and performing cross-domain synchronization with low overhead. Experimental evaluations using a two-node, Intel Xeon x86-64/Cavium ThunderX ARMv8 servers interconnected using a 56 Gbps InfiniBand network reveal MWPF's effectiveness: average speedup of up to 11% over the baseline DSM implementation.

The dissertation's third contribution is a type-2 distributed hypervisor called PuzzleHype that leverages DSM of host operating systems to provide a single address space to guest operating systems in a distributed virtualized setting. This allows a single virtual machine to use fragmented resources such as CPU cores, memory, and I/O devices of different physical machines, improving resource utilization. To transparently utilize CPU and I/O resources, PuzzleHype integrates multiple physical CPUs into a single VM by adopting thread migration and supporting interrupt forwarding. In addition, PuzzleHype allows a vCPU to access remote I/O resources (e.g., NIC) by supporting I/O delegations. Our experimental evaluations reveal PuzzleHype's effectiveness over a baseline over-provisioning scenario (which is otherwise necessary due to resource fragmentation): a speedup of up to 355% (average is 285%) for the SNU NPB suite, 225% for a customized LEMP stack, and 173% for the OpenLambda benchmark.

At its core, the dissertation's contributions demonstrate that DSM performance can be significantly improved by leveraging state-of-the-art, high-speed network infrastructures and by an array of software innovations including reducing the number of DSM messages, minimizing false sharing, aggressively prefetching data, and redesigning mechanisms to move memory-intensive operations out from DSM subsystems such as synchronization operations and shared data structures.

9.2 Proposed Post-Preliminary Examination Research

We propose three directions for post-preliminary exam research. These are discussed as follows.

9.2.1 Mitigating PuzzleHype DSM Overhead

In Chapter 8, we found current PuzzleHype design suffers a huge slowdown caused by the DSM layer. The DSM overhead can be very high in some cases and thus there is no benefit of using such a DSM system.

We propose multiple directions to mitigate PuzzleHype DSM's overhead. First, guest operating system designs that share as minimal information as possible can mitigate the overhead due to DSM's false sharing. Microkernel operating systems that follow a share-nothing model are good guest OS candidates. We propose to experiment with microkernel OSes (e.g., L4Linux [103] and MINIX3 [84]) or multikernel OSes (e.g., Barrelfish [27]) or replicated-kernel OSes (e.g., early version of Popcorn Linux [97]) as guest OSes for PuzzleHype.

Another direction to optimize DSM performance is to modify monolithic OSes. Popular monolithic OSes such as Linux, BSD, and the like, follow a share-everything model. It may be possible to slightly modify them to optimize their DSM traffic when they are used as a guest OS. To do this, we must first identify the root causes of significant DSM traffic. For example, a single page in the (guest OS) kernel memory that is concurrently accessed by multiple (guest) OS/node instances can cause large amounts of DSM traffic. To understand such causes, a "tracing tool" at the host OS-level that tracks page accesses of guest OSes in a distributed virtualization setting that uses DSM is necessary. This tool must be implemented in the host OS as only the host OS has a full view of the memory layout. Furthermore, knowing which line of code generates what traffic is not enough. To better understand whether those traffic is due to true or false page sharing, the tool must walk the guest OS stack to identify the call chain in the guest OS. This gives more information to characterize the run-time behavior and thereby pinpoint the sources of DSM traffic and the nature of page accesses. We propose to develop such a tracing tool, use it to debug the DSM traffic, and optimize the DSM overhead.

9.2.2 Improving PuzzleHype's Para-Virtualized I/O Device Support

PuzzleHype's current implementation leverages KVM's paravirtualized network drivers and DSM to synchronize data located in *vrings* such as skb (Section 7.3.4). This is a very intuitive design and indeed reduces significant programming effort. However, it also harms performance to some degree. For instance, the size of skb is less than the minimal synchronization unit of the DSM layer (4Kb page in PuzzleHype). To deal with this, we can create a page-aligned *vring* data structure.

Another solution is to move *vrings* out from the DSM layer. Each host maintains its own individual *vrings*. Since the *vring* is shared with the guest VM, this requires significant modification of the front-end drivers in the guest OS. The guest OS will see duplicated *vrings* for a TX/RX buffer instead of just one.

PuzzleHype also needs to explicitly copy data from one *vring* to another asynchronously along with interrupts and signals (Section 7.2.3). This approach can eliminate DSM false sharing and reduce EPT violation and the DSM traffic, compared with the current approach. Additionally, similar to the current approach, it still preserves the ability to run a network application on a vCPU which does not have a physical NIC on the host machine.

Another possible approach is to aggregate multiple physical NICs together from different hosts and present it as a single vNIC to the guest VM. A possible solution to do this is to utilize HAProxy [11] which can redistribute the I/O workload to balance the I/O load. PuzzleHype can leverage it to redirect network requests and provide a single NIC illusion to the guest VM users. We anticipate that this approach will transparently improve the network throughput by leveraging multiple physical NICs.

These ideas can also be applied to other I/O devices (e.g., storage) for improving PuzzleHype's performance. We propose to investigate these directions.

9.2.3 Extending PuzzleHype to Heterogeneous-ISA Systems

Many works indicate heterogeneous-ISA platforms are pushing into the datacenter [25, 141, 176]. However, it is still not clear how the current commodity datecenters can run a single virtual machine transparently merging these platforms on type-II VMMs. We propose to extend PuzzleHype to provide a single VM illusion on these heterogeneous-ISA platforms. This approach leads to a more elastic datacenter where a user can lend a VM with different heterogeneous-ISA cores. And also it will transparently provides users better performance, power efficiency, and programming experience [25].

To support heterogeneous-ISA systems, PuzzleHype needs to maintain the host OS's data structures across diverse ISAs (e.g., x86, ARM) as their methodologies (e.g., bootup process) and hardware are completely different (e.g., interrupt chips). For instance, PuzzleHype has to consider architecture-independent OS code which currently is not considered at all. Moreover, a new guest OS must be designed. None of the existing OSes can fit into such a heterogeneous-ISA hypervisor model without modification.

A possible direction is to explore the replicated OS kernel model, as exemplified in Popcorn Linux's early design [97]. In this model, an OS kernel is replicated, one for each ISA. The replicated kernel instances share a global, pseudo-physical memory region for communication among the guest OSes. We propose to investigate this direction, leveraging Popcorn Linux's early design.

Bibliography

[1] Ampere™ Altra™ 64-Bit Multi-Core Arm® Processor. https://amperecomputing.com/altra/.

[2] Apache HTTP Server Benchmarking Tool, Jan 2020. http://httpd.apache.org/docs/2.4/programs/ab.html.

[3] LEMP Stack, Jan 2020. https://lempstack.com/.

[4] MySQL, Jan 2020. https://www.mysql.com/.

[5] Nginx, Jan 2020. https://nginx.org/en/.

[6] OpenNebula, Jan 2020. http://opennebula.org/.

[7] OpenStack Neat, Jan 2020. http://openstack-neat.org/.

[8] PHP, Jan 2020. https://www.php.net/.

[9] ScaleMP vSMP, Jan 2020. https://www.scalemp.com/.

[10] Distributed Resource Scheduler, Distributed Power Management., Jan 2020. http://www.vmware.com/products/vsphere/enhanced-app-performance.html.

[11] HAProxy, Mar 2020. http://www.haproxy.org/.

[12] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices*, 41(11):2–13, 2006.

[13] Sudhir Ahuja, N Curriero, and David Gelernter. Linda and friends. *Computer*, 19(8), 1986.

[14] Amazon. Introducing Amazon EC2 A1 instances powered by new ARM-based AWS Graviton processors. https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-ec2-a1-instances/.

[15] Amazon. AWS Lambda, 2020. https://aws.amazon.com/lambda.

[16] Ampere Computing. Ampere Processors, 2018. https://amperecomputing.com/product/.

[17] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, February 1996.

[18] Gabriel Antoniu and Luc Bougé. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. In *International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 55–70. Springer, 2001.

[19] HTTP Archive. Report: Page Weight, Jan 2020. https://httparchive.org/reports/page-weight.

[20] Krste Asanovic. FireBox: A hardware building block for 2020 warehouse-scale computers. In *Proceedings of 12th USENIX Conference on File and Storage Technologies (FAST)*, February 2014.

[21] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS parallel benchmark results. In *Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 386–393, Nov 1992.

[22] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE transactions on software engineering*, (3):190–205, 1992.

[23] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13: 361–372, March 1998.

[24] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the programmability gap in heterogeneous-ISA platforms. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 29:1–29:16, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741962. URL http://doi.acm.org/10.1145/2741948.2741962.

[25] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-ISA datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 645–659, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037738. URL http://doi.acm.org/10.1145/3037697.3037738.

[26] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *OSR*, 37(5):164–177, 2003.

[27] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The

Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629579. URL http://doi.acm.org/10.1145/1629575.1629579.

[28] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd PPoPP*, pages 168–176, Seattle, WA, USA, March 1990.

[29] Brian N Bershad, Matthew J Zekauskas, and Wayne A Sawdon. The midway distributed shared memory system. In *Compcon Spring '93, Digest of Papers*, feb 1993.

[30] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 26–35, 2008.

[31] Nikhil Bhatia. Performance evaluation of intel ept hardware assist. *VMware, Inc*, 2009.

[32] Ricardo Bianchini, Raquel Pinto, and Claudio L Amorim. Data prefetching for software DSMs. In *Proceedings of the 12th international conference on Supercomputing*, pages 385–392, 1998.

[33] Roberto Bisiani and Mosur Ravishankar. PLUS: A distributed shared-memory system. *ACM SIGARCH Computer Architecture News*, 18(2SI):115–124, 1990.

[34] OpenMP Architecture Review Board. OpenMP application program interface v4.5. *Technical Report. https://tinyurl.com/yxzbx5cn*, 2015.

[35] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.

[36] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.", 2005.

[37] Rajkumar Buyya, Toni Cortes, and Hai Jin. Single system image. *The International Journal of High Performance Computing Applications*, 15(2):124–135, 2001.

[38] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proceedings of the VLDB Endowment*, 11(11): 1604–1617, 2018.

[39] Francois Cantonnet, Yiyi Yao, Mohamed Zahran, and Tarek El-Ghazawi. Productivity analysis of the UPC language. In *Proceedings of the 18th IPDPS*, Phoenix, AZ, USA, April 2004.

[40] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13rd SOSP*, pages 152–164, Pacific Grove, CA, October 1991.

[41] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. *ACM SIGARCH Computer Architecture News*, 19 (2):224–234, 1991.

[42] Matthew Chapman and Gernot Heiser. vnuma: A virtual shared-memory multiprocessor. In *USENIX Annual Technical Conference*, pages 349–362, 2009.

[43] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM, 2005.

[44] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[45] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium*, pages 1–11. IEEE, 2010.

[46] David R Cheriton, Hendrik A. Goosen, and Patrick D. Boyle. Paradigm: A highly scalable shared-memory multicomputer architecture. *Computer*, 24(2):33–46, 1991.

[47] George Chrysos. Intel Xeon Phi coprocessor–the architecture. Technical report, 2014. Intel Whitepaper 176.

[48] Hongsuk Chung, Munsik Kang, and Hyun-Duk Cho. Heterogeneous multi-processing solution of Exynos 5 Octa with ARM® big. little technology. *Samsung White Paper*, 2012.

[49] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd NSDI*, pages 273–286, Boston, MA, May 2005.

[50] Alpha Architecture Committee et al. *Alpha architecture reference manual*. Digital Press, 2014.

[51] The Standard Performance Evaluation Corporation. SPEC OMP2012. spec.org/omp2012, 2012.

[52] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[53] CRIU Community. Checkpoint/restore in userspace, 2011. https://criu.org/Main_Page,.

[54] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[55] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. ARM virtualization: performance and architectural implications. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 304–316. IEEE, 2016.

[56] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[57] Gary S Delp, David J Farber, Ronald G Minnich, Jonathan M Smith, and M-C Tam. Memory as a network abstraction. *IEEE Network*, 5(4):34–41, 1991.

[58] Google Developers. Improve Server Response Time, Dec 2018. https://developers.google.com/speed/docs/insights/Server#overview.

[59] Zhuocheng Ding. vDSM: Distributed shared memory in virtualized environments. In *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, pages 1112–1115. IEEE, 2018.

[60] Docker Inc. Docker - build, ship, and run any app, anywhere. Online: http://www.docker.com, accessed 2017-08-08.

[61] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

[62] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *Proceedings of the 11st NSDI*, pages 401–414, Seattle, WA, March 2014.

[63] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[64] Michael Feldman. Intel ships Xeon Skylake processor with integrated FPGA, May 2018. https://www.top500.org/news/intel-ships-xeon-skylake-processor-with-integrated-fpga/.

[65] Brett Fleisch and Gerald Popek. *Mirage: A coherent distributed shared memory design*, volume 23. ACM, 1989.

[66] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Proceedings of the 12th OSDI*, Savannah, GA, November 2016.

[67] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[68] Adam Geitgey. Open-source face detection algorithm, 2020. https://github.com/ageitgey/face_recognition.

[69] Isaac Gelado, John E Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the 15th ASPLOS*, pages 347–358, New York, NY, March 2010.

[70] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi*, volume 11, pages 24–24, 2011.

[71] Peter N Glaskowsky. NVIDIA's Fermi: the first complete GPU computing architecture. *White paper*, 18, 2009.

[72] Google. Google Cloud Functions, 2020. https://cloud.google.com/functions.

[73] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2014.

[74] P. Greenhalgh. big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. Technical report, 2011. Technical report, ARM.

[75] Matthias Gries, Ulrich Hoffmann, Michael Konow, and Michael Riepen. SCC: A flexible architecture for many-core platform research. *Computing in Science & Engineering*, 13(6):79–83, 2011.

[76] William D Gropp, William Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[77] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 153–165, Piscataway, NJ, USA, 2016. IEEE Press. ISBN 978-1-4673-8947-1. doi: 10.1109/ISCA.2016.23. URL https://doi.org/10.1109/ISCA.2016.23.

[78] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network Support for Resource Disaggregation in Next-Generation Datacenters. In *Proceedings of the 14th HotNets*, Philadelphia, PA, November 2015.

[79] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, 2011.

[80] Tim Harris. Hardware trends: Challenges and opportunities in distributed computing. *SIGACT News*, 46(2):89–95, June 2015. ISSN 0163-5700. doi: 10.1145/2789149.2789165. URL http://doi.acm.org/10.1145/2789149.2789165.

[81] Nicole Hemsoth. Cray ARMs Highest End Supercomputer with ThunderX2, November 2017. https://tinyurl.com/y95ljwd4.

[82] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[83] Sutter Herb. Welcome to the jungle, 2012. http://herbsutter.com/welcome-to-the-jungle/.

[84] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[85] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 VEE*, pages 51–60, 2009.

[86] Yang Hong, Yang Zheng, Fan Yang, Bin-Yu Zang, Hai-Bing Guan, and Hai-Bo Chen. Scaling out numa-aware applications with rdma-based distributed shared memory. *Journal of Computer Science and Technology*, 34(1):94–112, 2019.

[87] HPC Advisory Council. Introduction to high-speed infiniband interconnect. https://tinyurl.com/y7xl2df7. [Online; accessed 11-January-2018].

[88] Joel Hruska. Intel uses new Foveros 3D chip-stacking to build core, Atom on same silicon. *ExtremeTech*, Dec 2018. URL https://bit.ly/2SSQ62R.

[89] Weiwu Hu, Fuxin Zhang, and Haiming Liu. Dynamic data prefetching in home-based software DSMs. *Journal of Computer Science and Technology*, 16(3):231–241, 2001.

[90] Intel. Intel Xeon processor scalable family, 2018. https://intel.ly/2t1apTH.

[91] Vadim Iosevich and Assaf Schuster. Software distributed shared memory: a via-based implementation and comparison of sequential consistency with home-based lazy release consistency. *Software: Practice and Experience*, 35(8):755–786, 2005.

[92] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013. ISBN 9780124104143, 9780124104945.

[93] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[94] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided RDMA datagram RPCs. In *Proceedings of the 12th OSDI*, pages 185–201, Savannah, GA, November 2016.

[95] Shubham Kamdar and Neha Kamdar. big.LITTLE architecture: Heterogeneous multicore processing. *International Journal of Computer Applications*, 119(1), 2015.

[96] Mohamed L. Karaoui, Anthony Carno, Robert Lyerlyand Sang-Hoon Kim, Pierre Olivier, Changwoo Min, and Binoy Ravindran. Poster: Scheduling HPC workloads on heterogeneous-ISA architectures. In *Proceedings of the 24nd PPoPP*, Washington, DC, February 2019.

[97] David Katz, Antonio Barbalace, Saif Ansary, Akshay Ravichandran, and Binoy Ravindran. Thread migration in a replicated-kernel os. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 278–287. IEEE, 2015.

[98] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th ISCA*, pages 13–21, Queensland, Australia, May 1992.

[99] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *In Proc. Linux Symposium*, pages 225–230, July 2007.

[100] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. HPVM: Heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 68–80, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4982-6. doi: 10.1145/3178487.3178493. URL `http://doi.acm.org/10.1145/3178487.3178493`.

[101] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, et al. The

stanford flash multiprocessor. In *Proceedings of 21 International Symposium on Computer Architecture*, pages 302–313. IEEE, 1994.

[102] KVM Contributors. KVM Website - Virtio, 2008. https://www.linux-kvm.org/-page/Virtio.

[103] Adam Lackorzynski et al. L4Linux porting optimizations. *Master's thesis, TU Dresden*, 2004.

[104] AI-C Lai and Chin-Laung Lei. Data prefetching for distributed shared memory systems. In *Proceedings of HICSS-29: 29th Hawaii International Conference on System Sciences*, volume 1, pages 102–110. IEEE, 1996.

[105] Gwangmu Lee, Hyunjoon Park, Seonyeong Heo, Kyung-Ah Chang, Hyogun Lee, and Hanjun Kim. Architecture-aware automatic computation offload for native applications. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 521–532, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4034-2. doi: 10.1145/2830772.2830833. URL http://doi.acm.org/10.1145/2830772.2830833.

[106] Juchang Lee, Kihong Kim, and Sang Kyun Cha. Differential logging: A commutative and associative logging scheme for highly parallel main memory database. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 173–182. IEEE, 2001.

[107] Daniel Lenoski, Kourosh Gharachorloo, James Laudon, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. Design of scalable shared-memory multiprocessors: The dash approach. In *Compcon Spring '90. Intellectual Leverage. Digest of Papers. 35th IEEE Computer Society International Conference.*, pages 62–67. IEEE, 1990.

[108] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *TOCS*, 7(4):321–359, 1989.

[109] Wen-Yew Liang, Chun-Ta King, and Feipei Lai. Adsmith: An efficient object-based distributed shared memory system on PVM. In *Proceedings Second International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'96)*, pages 173–179. IEEE, 1996.

[110] Kevin Lim, Jichuan Chang, Trevor Muge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 35th ISCA*, Austin, TX, USA, 2008.

[111] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parhasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *Proceedings of the 18th HPCA*, New Orleans, Louisiana, USA, February 2012.

[112] Felix Xiaozhu Lin, Zhen Wang, Robert LiKamWa, and Lin Zhong. Reflex: using low-power processors in smartphones without knowing them. In *Proceedings of the 17th ASPLOS*, London, UK, March 2012.

[113] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: A mobile operating system for heterogeneous coherence domains. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 285–300, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5.

[114] Liran Liss, Yitzhak Birk, and Assaf Schuster. Efficient exploitation of kernel access to InfiniBand: a software DSM example. In *Proceedings of the 11st HOTI*, San Francisco, CA, USA, August 2013.

[115] Haiming Liu and Weiwu Hu. A comparison of two strategies of dynamic data prefetching in software DSM. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 6–pp. IEEE, 2000.

[116] Liang Liu, Hao Wang, Xue Liu, Xing Jin, Wen Bo He, Qing Bo Wang, and Ying Chen. Greencloud: a new architecture for green data center. In *Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session*, pages 29–38, 2009.

[117] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *Proceedings of the 2017 ATC*, Santa Clara, CA, July 2017.

[118] Hongtu Ma, Rongcai Zhao, Xiang Gao, and Youwei Zhang. Barrier optimization for OpenMP program. In *2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, pages 495–500. IEEE, 2009.

[119] Marvell Technology. LiquidIO ii 10/25gbe Adapter family. https://www.marvell.com/ethernet-adapters-and-controllers/liquidio-smart-nics/liquidio-ii-smart-nics/, Online, accessed 2019/02/11.

[120] Marvell Technology. ThunderX ARM-based processors, 2013. https://www.marvell.com/server-processors/thunderx-arm-processors/.

[121] Marvell Technology. ThunderX2 ARM-based processors, 2018. https://www.marvell.com/server-processors/thunderx2-arm-processors/.

[122] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System v application binary interface amd64 architecture processor supplement. November 2014.

[123] Babar Naveed Memon, Xiayue Charles Lin, Arshia Mufti, Arthur Scott Wesley, Tim Brecht, Kenneth Salem, Bernard Wong, and Benjamin Cassell. RaMP: A lightweight RDMA abstraction for loosely coupled applications. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, 2018.

[124] Ke Meng and Guangming Tan. Ring: Numa-aware message-batching runtime for data-intensive applications. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 368–375. IEEE, 2017.

[125] Microsoft. Microsoft Azure Functions, 2020. https://azure.microsoft.com/en-us/services/functions.

[126] Microsoft Azure. Microsoft azure public dataset repository, 2020. https://github.com/Azure/AzurePublicDataset.

[127] Dejan S Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[128] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the OLS*, volume 2, pages 85–90, Ottawa, Canada, July 2008.

[129] Timothy P. Morgan. Tilera rescues CPU cycles with network coprocessors, 2013. https://bit.ly/2DfM53R, Online, accessed 01/05/2019.

[130] Sape J. Mullender, Guido Van Rossum, AS Tananbaum, Robbert Van Renesse, and Hans Van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[131] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), 2006.

[132] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 ATC*, pages 291–305, Santa Clara, CA, July 2015.

[133] Netronome. Agilio SmartNICs, 2019. https://www.netronome.com/products/smartnic/overview/.

[134] Yahoo Developer Network. Best Practices for Speeding Up Your Web Site, Dec 2006. https://developer.yahoo.com/performance/rules.html.

[135] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234, 2009.

[136] Vlad Nitu, Boris Teabe, Leon Fopa, Alain Tchana, and Daniel Hagimont. StopGap: Elastic VMs to enhance server consolidation. *Software: Practice and Experience*, 47 (11):1501–1519, 2017.

[137] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[138] Ranjit Noronha and Dhabaleswar K Panda. Can high performance software DSM systems designed with InfiniBand features benefit from PCI-Express? In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, volume 2, pages 945–952. IEEE, 2005.

[139] Nvidia. Variable SMP – a multi-core CPU architecture for low power and high performance, 2011. https://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911b.pdf.

[140] Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. OS support for thread migration and distribution in the fully heterogeneous datacenter. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 174–179, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5068-6. doi: 10.1145/3102980.3103009. URL http://doi.acm.org/10.1145/3102980.3103009.

[141] Pierre Olivier, Binoy Ravindran, and Antonio Barbalace. The Multihype: Virtualizing heterogeneous-ISA architectures. In *9th Workshop on Systems for Multi-core and Heterogeneous Architectures (SFMA)*, 2019.

[142] Stephen L. Olivier, Allan K. Porterfield, Kyle B. Wheeler, and Jan F. Prins. Scheduling task parallelism on multi-socket multicore systems. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '11, pages 49–56, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0761-1. doi: 10.1145/1988796.1988804. URL http://doi.acm.org/10.1145/1988796.1988804.

[143] RedHat OpenShift. VHost-net, Jan 2020. http://www.linux-kvm.org/page/UsingVhost.

[144] OpenVZ. OpenVZ Virtuozzo Containers. Online: http://openvz.org, accseed 08-08-2017.

[145] Nir Oren. A survey of prefetching techniques. *Technical report, July 2000*, 2000.

[146] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. Dynamos: dynamic schedule migration for heterogeneous cores. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 322–333. ACM, 2015.

[147] Loïc Perennou, Mar Callau-Zori, Sylvain Lefebvre, and Raka Chiky. Workload characterization for a non-hyperscale public cloud platform. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 409–413. IEEE, 2019.

[148] Ian Pratt, Andrew Warfield, P Barham, and R Neugebauer. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 113–1118, 2003.

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

[150] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: concepts and systems. *IEEE Parallel Distributed Technology: Systems Applications*, 4 (2):63–71, 1996.

[151] Md Golam Rabbani, Rafael Pereira Esteves, Maxim Podlesny, Gwendal Simon, Lisandro Zambenedetti Granville, and Raouf Boutaba. On tackling virtual data center embedding problem. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 177–184. IEEE, 2013.

[152] Carl Ramey. Tile-gx100 manycore processor: Acceleration interfaces and architecture. In *Hot Chips 23 Symposium (HCS), 2011 IEEE*, pages 1–21. IEEE, 2011.

[153] Keith Harold Randall. *Cilk: Efficient multithreaded computing*. PhD thesis, Massachusetts Institute of Technology, 1998.

[154] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[155] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–13, 2012.

[156] Einar Rustad. Numascale numaconnect. White paper, online: https://www.numascale.com/numa_pdfs/numaconnect-white-paper.pdf, accessed 2017-08-08.

[157] Stefan Rusu, Simon Tam, Harry Muljono, Jason Stinson, David Ayers, Jonathan Chang, Raj Varada, Matt Ratta, Sailesh Kottapalli, and Sujal Vora. A 45 nm 8-core enterprise Xeon processor. *IEEE Journal of Solid-State Circuits*, 45(1):7–14, 2010.

[158] Marina Sadini, Antonio Barbalace, Binoy Ravindran, and Francesco Quaglia. A page coherency protocol for Popcorn replicated-kernel operating system. In *Proceedings of the 2013 Many-Core Architecture Research Community Symposium (MARC)*, October 2013.

[159] Samsung. Exynos 4210 application processor. Available http://www.samsung.com/global/business/semiconductor/productInfo.do?fmly_id=844&partnum=Exynos%204210.

[160] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[161] Robert Lyerly Pierre Olivier Changwoo Min Sang-Hoon Kim, Ho-Ren Chuang and Binoy Ravindran. DEX: Scaling applications beyond machine boundaries. In *2020 IEEE 40th International Conference on Distributed Computing Systems*. IEEE, 2020.

[162] Seoul National University Centers for Manycore Programming. SNU NPB suite. https://tinyurl.com/y3jrfrqg.

[163] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337, 2017.

[164] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proceedings of the 13rd OSDI*, pages 69–87, Carlsbad, CA, October 2018.

[165] Bikash Sharma, Victor Chudnovsky, Joseph L Hellerstein, Rasekh Rifaat, and Chita R Das. Modeling and synthesizing task placement constraints in Google compute clusters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011.

[166] Mark Silberstein. GPUs: High-performance accelerators for parallel applications: The multicore transformation (ubiquity symposium). *Ubiquity*, 2014(August):1:1–1:13, August 2014. ISSN 1530-2180. doi: 10.1145/2618401. URL http://doi.acm.org/10.1145/2618401.

[167] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 485–498, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451169. URL http://doi.acm.org/10.1145/2451116.2451169.

[168] Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. GPUnet: Networking abstractions for gpu programs. *ACM Trans. Comput. Syst.*, 34(3):9:1–9:31, September 2016. ISSN 0734-2071. doi: 10.1145/2963098. URL http://doi.acm.org/10.1145/2963098.

[169] Jackie Silcock. *Distributed shared memory: A survey*. Deakin University, School of Computing and Mathematics, 1995.

[170] Luís Moura Silva, João Gabriel Silva, and Simon Chapple. Implementing distributed shared memory on top of MPI: The DSMPI library. In *Proceedings of 4th Euromicro Workshop on Parallel and Distributed Processing*, pages 50–57. IEEE, 1996.

[171] Balaram Sinharoy, Ron Kalla, William J Starke, Hung Q Le, Robert Cargnoni, James A Van Norstrand, Bruce J Ronchetti, Jeffrey Stuecheli, Jens Leenstra, Guy L Guthrie, et al. IBM POWER7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1–1, 2011.

[172] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation Intel Xeon Phi. *IEEE micro*, 36(2):34–46, 2016.

[173] Evan Speight and John K Bennett. Brazos: A third generation DSM system. In *Proceedings of the 1997 USENIX Windows/NT Workshop*, pages 95–106. USENIX Association Berkeley, CA, 1997.

[174] William Evan Speight and Martin Burtscher. Delphi: Predition-based page prefetching to improve the performance of shared virtual memory systems. In *PDPTA*, volume 2, pages 49–55, 2002.

[175] Justin Talbot, Richard M Yoo, and Christos Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*, pages 9–16. ACM, 2011.

[176] Alain Tchana and Renaud Lachaize. Rebooting virtualization research (again). In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 99–106, 2019.

[177] Texas Instruments. OMAP5912 multimedia processor device overview and architecture reference guide, 2004. http://www.ti.com/lit/ug/spru748c/spru748c.pdf.

[178] Texas Instruments. OMAP4 applications processor: Technical reference manual. Technical report, 2010.

[179] Texas Instruments. OMAP543x: Technical reference manual. Technical report, 2010.

[180] TidaScale. Tidascale website, 2020. https://www.tidalscale.com/.

[181] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th SOSP*, pages 306–324, Shanghai, China, October 2017.

[182] Steve VanderWiel and David J Lilja. A survey of data prefetching techniques. In *Proceedings of the 23rd International Symposium on Computer Architecture*. Citeseer, 1996.

[183] Ashish Venkat and Dean M. Tullsen. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 121–132, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-4394-4. URL http://dl.acm.org/citation.cfm?id=2665671.2665692.

[184] Ashish Venkat and Dean M. Tullsen. Harnessing isa diversity: Design of a heterogeneous-ISA chip multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 121–132, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-4394-4. URL `http://dl.acm.org/citation.cfm?id=2665671.2665692`.

[185] Ashish Venkat, Harsha Basavaraj, and Dean M. Tullsen. Composite-ISA cores: Enabling multi-ISA heterogeneity using a single ISA. In *The 25th International Symposium on High-Performance Computer Architecture (HPCA'19)*, February 2019.

[186] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 205–218, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-839-1. doi: 10.1145/1736020.1736044.

[187] VMware, Inc. Live migration for virtual machines without service interruption, 2009.

[188] David G Von Bank, Charles M Shub, and Robert W Sebesta. A unified model of pointwise equivalence of procedural computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1842–1874, 1994.

[189] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The locus distributed operating system. *ACM SIGOPS Operating Systems Review*, 17(5): 49–70, 1983.

[190] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and Steven Hand. Parallax: managing storage for a million machines. In *Proceedings of the 10th HOTOS (HotOS X)*, Santa Fe, NM, June 2005.

[191] David Wentzlaff and Anant Agarwal. Factored operating systems (fos) the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.

[192] Andrew Wilson, Marc Teller, Thomas Probert, Dyung Le, and R LaRowe. Lynx/-Galactica Net: A distributed, cache coherent multiprocessing system. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, volume 1, pages 416–426. IEEE, 1992.

[193] Fengguang Wu. Sequential file prefetching in linux. In *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*, pages 218–261. IGI Global, 2010.

[194] Jin Zhang, Zhuocheng Ding, Yubin Chen, Xingguo Jia, Boshi Yu, Zhengwei Qi, and Haibing Guan. GiantVM: a type-II hypervisor implementing many-to-one virtualization. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 30–44, 2020.

[195] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware graph-structured analytics. In *Proceedings of the 20th PPoPP*, pages 183–193, San Francisco, CA, February 2015.

[196] S. Zhou, M. Stumm, and T. McInerney. Extending distributed shared memory to heterogeneous environments. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 30–37, 1990.

[197] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the 2nd OSDI*, pages 75–88, Seattle, WA, October 1996.

[198] Yuanyuan Zhou, Liviu Iftode, Jaswinder Pal Sing, Kai Li, Brian R. Toonen, Ioannis Schoinas, Mark D. Hill, and David A. Wood. Relaxed consistency and coherence granularity in DSM systems: A performance evaluation. In *Proceedings of the 6th PPoPP*, pages 193–205, Las Vegas, Nevada, USA, June 1997.