LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation

Yizhou Shan, Yutong Huang, Yilun Chen, Yiying Zhang
Purdue University

Abstract

The monolithic server model where a server is the unit of deployment, operation, and failure is meeting its limits in the face of several recent hardware and application trends. To improve resource utilization, elasticity, heterogeneity, and failure handling in datacenters, we believe that datacenters should break monolithic servers into *disaggregated, network-attached hardware components*. Despite the promising benefits of hardware resource disaggregation, no existing OSes or software systems can properly manage it.

We propose a new OS model called the *splitkernel* to manage disaggregated systems. Splitkernel disseminates traditional OS functionalities into loosely-coupled *monitors*, each of which runs on and manages a hardware component. A splitkernel also performs resource allocation and failure handling of a distributed set of hardware components. Using the splitkernel model, we built *LegoOS*, a new OS designed for hardware resource disaggregation. LegoOS appears to users as a set of distributed servers. Internally, a user application can span multiple processor, memory, and storage hardware components. We implemented LegoOS on x86-64 and evaluated it by emulating hardware components using commodity servers. Our evaluation results show that LegoOS' performance is comparable to monolithic Linux servers, while largely improving resource packing and reducing failure rate over monolithic clusters.

1 Introduction

For many years, the unit of deployment, operation, and failure in datacenters has been a *monolithic server*, one that contains all the hardware resources that are needed to run a user program (typically a processor, some main memory, and a disk or an SSD). This monolithic architecture is meeting its limitations in the face of several issues and recent trends in datacenters.

First, datacenters face a difficult bin-packing problem of fitting applications to physical machines. Since a process can only use processor and memory in the same machine, it is hard to achieve full memory and CPU resource utilization [18, 33, 65]. Second, after packaging hardware devices in a server, it is difficult to add, remove, or change hardware components in datacenters [39]. Moreover, when a hardware component like a memory controller fails, the entire server is unusable. Finally, modern datacenters host increasingly heterogeneous hardware [5, 55, 84, 94]. However, designing new hardware that can fit into monolithic servers and deploying them in datacenters is a painful and cost-ineffective process that often limits the speed of new hardware adoption.

We believe that datacenters should break monolithic servers and organize hardware devices like CPU, DRAM, and disks as *independent, failure-isolated, network-attached components*, each having its own controller to manage its hardware. This *hardware resource disaggregation* architecture is enabled by recent advances in network technologies [24, 42, 52, 66, 81, 88] and the trend towards increasing processing power in hardware controller [9, 23, 92]. Hardware resource disaggregation greatly improves resource utilization, elasticity, heterogeneity, and failure isolation, since each hardware component can operate or fail on its own and its resource allocation is independent from other components. With these benefits, this new architecture has already attracted early attention from academia and industry [1, 15, 48, 56, 63, 77].

Hardware resource disaggregation completely shifts the paradigm of computing and presents a key challenge to system builders: *How to manage and virtualize the distributed, disaggregated hardware components?*

Unfortunately, existing kernel designs cannot address the new challenges hardware resource disaggregation brings, such as network communication overhead across disaggregated hardware components, fault tolerance of hardware components, and the resource management of distributed components. Monolithic kernels, microkernels [36], and exokernels [37] run one OS on a monolithic machine, and the OS assumes local accesses to shared main memory, storage devices, network interfaces, and other hardware resources in the machine. After disaggregating hardware resources, it may be viable to run the OS at a processor and remotely manage all other hardware components. However, remote management requires significant amount of network traffic, and when processors fail, other components are unusable. Multi-kernel OSes [21, 26, 76, 106] run a kernel at each processor (or core) in a monolithic computer and these per-processor kernels communicate with each other through message passing. Multi-kernels still assume local accesses to hardware resources in a monolithic machine and their message passing is over local buses instead of a general network. While existing OSes could be retrofitted to support hardware resource disaggregation, such retrofitting will be invasive to the central subsystems of an OS, such as memory and I/O management.

We propose *splitkernel*, a new OS architecture for

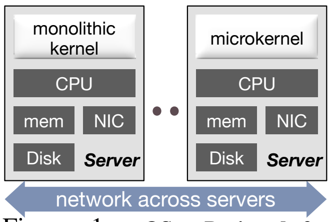


Figure 1: **OSes Designed for Monolithic Servers.**

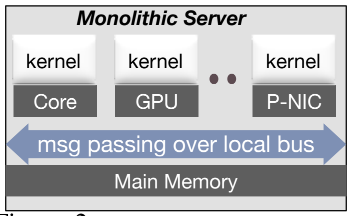


Figure 2: **Multi-kernel Architecture.** *P-NIC: programmable NIC.*

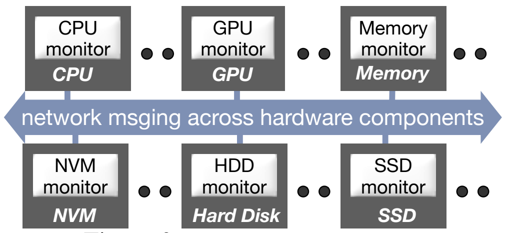


Figure 3: **Splitkernel Architecture.**

hardware resource disaggregation (Figure 3). The basic idea is simple: *When hardware is disaggregated, the OS should be also*. A splitkernel breaks traditional operating system functionalities into loosely-coupled *monitors*, each running at and managing a hardware component. Monitors in a splitkernel can be heterogeneous and can be added, removed, and restarted dynamically without affecting the rest of the system. Each splitkernel monitor operates locally for its own functionality and only communicates with other monitors when there is a need to access resources there. There are only two global tasks in a splitkernel: orchestrating resource allocation across components and handling component failure.

We choose not to support coherence across different components in a splitkernel. A splitkernel can use any general network to connect its hardware components. All monitors in a splitkernel communicate with each other via *network messaging* only. With our targeted scale, explicit message passing is much more efficient in network bandwidth consumption than the alternative of implicitly maintaining cross-component coherence.

Following the splitkernel model, we built LegoOS, the *first* OS designed for hardware resource disaggregation. LegoOS is a distributed OS that appears to applications as a set of virtual servers (called *vNodes*). A vNode can run on multiple processor, memory, and storage components and one component can host resources for multiple vNodes. LegoOS cleanly separates OS functionalities into three types of *monitors*, process monitor, memory monitor, and storage monitor. LegoOS monitors share no or minimal states and use a customized RDMA-based network stack to communicate with each other.

The biggest challenge and our focus in building LegoOS is the separation of processor and memory and their management. Modern processors and OSes assume all hardware memory units including main memory, page tables, and TLB are local. Simply moving all memory hardware and memory management software to across the network will not work.

Based on application properties and hardware trends, we propose a hardware plus software solution that cleanly separates processor and memory functionalities, while meeting application performance requirements. LegoOS moves all memory hardware units to the disaggregated memory components and organizes all levels of processor caches as virtual caches that are accessed using virtual memory addresses. To improve performance, LegoOS uses a small amount (*e.g.*, 4 GB) of DRAM organized as a virtual cache below current last-level cache.

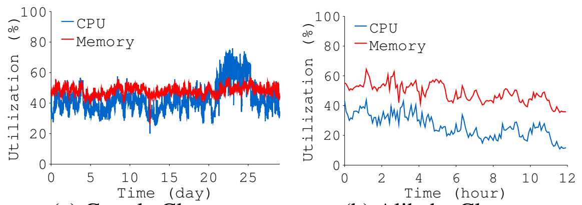
LegoOS process monitor manages application processes and the extended DRAM-cache. Memory monitor manages all virtual and physical memory space allocation and address mappings. LegoOS uses a novel two-level distributed virtual memory space management mechanism, which ensures efficient foreground memory accesses and balances load and space utilization at allocation time. Finally, LegoOS uses a space- and performance-efficient memory replication scheme to handle memory failure.

We implemented LegoOS on the x86-64 architecture. LegoOS is fully backward compatible with Linux ABIs by supporting common Linux system call APIs. To evaluate LegoOS, we emulate disaggregated hardware components using commodity servers. We evaluated LegoOS with microbenchmarks, the PARSEC benchmarks [22], and two unmodified datacenter applications, Phoenix [85] and TensorFlow [4]. Our evaluation results show that compared to monolithic Linux servers that can hold all the working sets of these applications, LegoOS is only 1.3× to 1.7× slower with 25% of application working set available as DRAM cache at processor components. Compared to monolithic Linux servers whose main memory size is the same as LegoOS' DRAM cache size and which use local SSD/DRAM swapping or network swapping, LegoOS' performance is 0.8× to 3.2×. At the same time, LegoOS largely improves resource packing and reduces system mean time to failure.

Overall, this work makes the following contributions:

- We propose the concept of splitkernel, a new OS architecture that fits the hardware resource disaggregation architecture.
- We built LegoOS, the first OS that runs on and manages a disaggregated hardware cluster.
- We propose a new hardware architecture to cleanly separate processor and memory hardware functionalities, while preserving most of the performance of monolithic server architecture.

LegoOS is publicly available at *http://LegoOS.io*.



(a) Google Cluster (b) Alibaba Cluster

Figure 4: **Datacenter Resource Utilization.**

2 Disaggregate Hardware Resource

This section motivates the hardware resource disaggregation architecture and discusses the challenges in managing disaggregated hardware.

2.1 Limitations of Monolithic Servers

A monolithic server has been the unit of deployment and operation in datacenters for decades. This long-standing *server-centric* architecture has several key limitations.

Inefficient resource utilization. With a server being the physical boundary of resource allocation, it is difficult to fully utilize all resources in a datacenter [18, 33, 65]. We analyzed two production cluster traces: a 29-day Google one [45] and a 12-hour Alibaba one [10]. Figure 4 plots the aggregated CPU and memory utilization in the two clusters. For both clusters, only around half of the CPU and memory are utilized. Interestingly, a significant amount of jobs are being evicted at the same time in these traces (*e.g.*, evicting low-priority jobs to make room for high-priority ones [102]). One of the main reasons for resource underutilization in these production clusters is the constraint that CPU and memory for a job have to be allocated from the same physical machine.

Poor hardware elasticity. It is difficult to add, move, remove, or reconfigure hardware components after they have been installed in a monolithic server [39]. Because of this rigidity, datacenter owners have to plan out server configurations in advance. However, with today's speed of change in application requirements, such plans have to be adjusted frequently, and when changes happen, it often comes with waste in existing server hardware.

Coarse failure domain. The failure unit of monolithic servers is coarse. When a hardware component within a server fails, the whole server is often unusable and applications running on it can all crash. Previous analysis [90] found that motherboard, memory, CPU, power supply failures account for 50% to 82% of hardware failures in a server. Unfortunately, monolithic servers cannot continue to operate when any of these devices fail.

Bad support for heterogeneity. Driven by application needs, new hardware technologies are finding their ways into modern datacenters [94]. Datacenters no longer host only commodity servers with CPU, DRAM, and hard disks. They include non-traditional and specialized hardware like GPGPU [11, 46], TPU [55], DPU [5], FPGA [12, 84], non-volatile memory [49], and NVMe-based SSDs [98]. The monolithic server model tightly couples hardware devices with each other and with a motherboard. As a result, making new hardware devices work with existing servers is a painful and lengthy process [84]. Mover, datacenters often need to purchase new servers to host certain hardware. Other parts of the new servers can go underutilized and old servers need to retire to make room for new ones.

2.2 Hardware Resource Disaggregation

The server-centric architecture is a bad fit for the fast-changing datacenter hardware, software, and cost needs. There is an emerging interest in utilizing resources beyond a local machine [41], such as distributed memory [7, 34, 74, 79] and network swapping [47]. These solutions improve resource utilization over traditional systems. However, they cannot solve all the issues of monolithic servers (*e.g.*, the last three issues in §2.1), since their hardware model is still a monolithic one. To fully support the growing heterogeneity in hardware and to provide elasticity and flexibility at the hardware level, we should *break the monolithic server model.*

We envision a *hardware resource disaggregation* architecture where hardware resources in traditional servers are disseminated into network-attached *hardware components*. Each component has a controller and a network interface, can operate on its own, and is an *independent, failure-isolated* entity.

The disaggregated approach largely increases the flexibility of a datacenter. Applications can freely use resources from any hardware component, which makes resource allocation easy and efficient. Different types of hardware resources can *scale independently*. It is easy to add, remove, or reconfigure components. New types of hardware components can easily be deployed in a datacenter — by simply enabling the hardware to talk to the network and adding a new network link to connect it. Finally, hardware resource disaggregation enables fine-grain failure isolation, since one component failure will not affect the rest of a cluster.

Three hardware trends are making resource disaggregation feasible in datacenters. First, network speed has grown by more than an order of magnitude and has become more scalable in the past decade with new technologies like Remote Direct Memory Access (*RDMA*) [69] and new topologies and switches [15, 30, 31], enabling fast accesses of hardware components that are disaggregated across the network. InfiniBand will soon reach 200Gbps and sub-600 nanosecond speed [66], being only 2× to 4× slower than main memory bus in bandwidth. With main memory bus facing a bandwidth wall [87], future network bandwidth (at line rate) is even projected to exceed local DRAM bandwidth [99].

Second, network interfaces are moving closer to hardware components, with technologies like Intel Omni-Path [50], RDMA [69], and NVMe over Fabrics [29, 71]. As a result, hardware devices will be able to access network directly without the need to attach any processors.

Finally, hardware devices are incorporating more processing power [8, 9, 23, 67, 68, 75], allowing application and OS logics to be offloaded to hardware [57, 92]. On-device processing power will enable system software to manage disaggregated hardware components locally.

With these hardware trends and the limitations of monolithic servers, we believe that future datacenters will be able to largely benefit from hardware resource disaggregation. In fact, there have already been several initial hardware proposals in resource disaggregation [1], including disaggregated memory [63, 77, 78], disaggregated flash [59, 60], Intel Rack-Scale System [51], HP "The Machine" [40, 48], IBM Composable System [28], and Berkeley Firebox [15].

2.3 OSes for Resource Disaggregation

Despite various benefits hardware resource disaggregation promises, it is still unclear how to manage or utilize disaggregated hardware in a datacenter. Unfortunately, existing OSes and distributed systems cannot work well with this new architecture. Single-node OSes like Linux view a server as the unit of management and assume all hardware components are local (Figure 1). A potential approach is to run these OSes on processors and access memory, storage, and other hardware resources remotely. Recent disaggregated systems like soNUMA [78] take this approach. However, this approach incurs high network latency and bandwidth consumption with remote device management, misses the opportunity of exploiting device-local computation power, and makes processors the single point of failure.

Multi-kernel solutions [21, 26, 76, 106, 107] (Figure 2) view different cores, processors, or programmable devices within a server separately by running a kernel on each core/device and using message passing to communicate across kernels. These kernels still run in a single server and all access some common hardware resources in the server like memory and the network interface. Moreover, they do not manage distributed resources or handle failures in a disaggregated cluster.

There have been various distributed OS proposals, most of which date decades back [16, 82, 97]. Most of these distributed OSes manage a set of monolithic servers instead of hardware components.

Hardware resource disaggregation is fundamentally different from the traditional monolithic server model. A complete disaggregation of processor, memory, and storage means that when managing one of them, there will be no local accesses to the other two. For example, processors will have no local memory or storage to store user or kernel data. An OS also needs to manage distributed hardware resource and handle hardware component failure. We summarize the following key challenges in building an OS for resource disaggregation, some of which have previously been identified [40].

- How to deliver good performance when application execution involves the access of network-partitioned disaggregated hardware and current network is still slower than local buses?
- How to locally manage individual hardware components with limited hardware resources?
- How to manage distributed hardware resources?
- How to handle a component failure without affecting other components or running applications?
- What abstraction should be exposed to users and how to support existing datacenter applications?

Instead of retrofitting existing OSes to confront these challenges, we take the approach of designing a new OS architecture from the ground up for hardware resource disaggregation.

3 The Splitkernel OS Architecture

We propose *splitkernel*, a new OS architecture for resource disaggregation. Figure 3 illustrates splitkernel's overall architecture. The splitkernel disseminates an OS into pieces of different functionalities, each running at and managing a hardware component. All components communicate by message passing over a common network, and splitkernel globally manages resources and component failures. Splitkernel is a general OS architecture we propose for hardware resource disaggregation. There can be many types of implementation of splitkernel. Further, we make no assumption on the specific hardware or network type in a disaggregated cluster a splitkernel runs on. Below, we describe four key concepts of the splitkernel architecture.

Split OS functionalities. Splitkernel breaks traditional OS functionalities into *monitors*. Each monitor manages a hardware component, virtualizes and protects its physical resources. Monitors in a splitkernel are loosely-coupled and they communicate with other monitors to access remote resources. For each monitor to operate on its own with minimal dependence on other monitors, we use a stateless design by sharing no or minimal *states*, or metadata, across monitors.

Run monitors at hardware components. We expect each non-processor hardware component in a disaggregated cluster to have a controller that can run a monitor. A hardware controller can be a low-power general-purpose core, an ASIC, or an FPGA. Each monitor in a splitkernel can use its own implementation to manage the hardware component it runs on. This design makes it easy to integrate heterogeneous hardware in datacenters — to deploy a new hardware device, its developers only need to build the device, implement a monitor to manage it, and attach the device to the network. Similarly, it is easy to reconfigure, restart, and remove hardware components.

Message passing across non-coherent components. Unlike other proposals of disaggregated systems [48] that rely on coherent interconnects [24, 42, 81], a splitkernel runs on general-purpose network layer like Ethernet and neither underlying hardware nor the splitkernel provides cache coherence across components. We made this design choice mainly because maintaining coherence for our targeted cluster scale would cause high network bandwidth consumption. Instead, all communication across components in a splitkernel is through *network messaging*. A splitkernel still retains the coherence guarantee that hardware already provides within a component (*e.g.*, cache coherence across cores in a CPU), and applications running on top of a splitkernel can use message passing to implement their desired level of coherence for their data across components.

Global resource management and failure handling. One hardware component can host resources for multiple applications and its failure can affect all these applications. In addition to managing individual components, the splitkernel also needs to globally manage resources and failure. To minimize performance and scalability bottleneck, the splitkernel only involves global resource management occasionally for coarse-grained decisions, while individual monitors make their own fine-grained decisions. The splitkernel handles component failure by adding redundancy for recovery.

4 LegoOS Design

Based on the splitkernel architecture, we built *LegoOS*, the first OS designed for hardware resource disaggregation. LegoOS is a research prototype that demonstrates the feasibility of the splitkernel design, but it is not the only way to build a splitkernel. LegoOS' design targets three types of hardware components: processor, memory, and storage, and we call them *pComponent, mComponent*, and *sComponent*.

This section first introduces the abstraction LegoOS exposes to users and then describes the hardware architecture of components LegoOS runs on. Next, we explain the design of LegoOS' process, memory, and storage monitors. Finally, we discuss LegoOS' global resource management and failure handling mechanisms.

Overall, LegoOS achieves the following design goals:

- Clean separation of process, memory, and storage functionalities.
- Monitors run at hardware components and fit device constraints.
- Comparable performance to monolithic Linux servers.
- Efficient resource management and memory failure handling, both in space and in performance.
- Easy-to-use, backward compatible user interface.
- Supports common Linux system call interfaces.

4.1 Abstraction and Usage Model

LegoOS exposes a distributed set of *virtual nodes*, or *vNode*, to users. From users' point of view, a vNode is like a virtual machine. Multiple users can run in a vNode and each user can run multiple processes. Each vNode has a unique ID, a unique virtual IP address, and its own storage mount point. LegoOS protects and isolates the resources given to each vNode from others. Internally, one vNode can run on multiple pComponents, multiple mComponents, and multiple sComponents. At the same time, each hardware component can host resources for more than one vNode. The internal execution status is transparent to LegoOS users; they do not know which physical components their applications run on.

With splitkernel's design principle of components not being coherent, LegoOS does not support writable shared memory across processors. LegoOS assumes that threads within the same process access shared memory and threads belonging to different processes do not share writable memory, and LegoOS makes scheduling decision based on this assumption (§4.3.1). Applications that use shared writable memory across processes (*e.g.*, with MAP_SHARED) will need to be adapted to use message passing across processes. We made this decision because writable shared memory across processes is rare (we have not seen a single instance in the datacenter applications we studied), and supporting it makes both hardware and software more complex (in fact, we have implemented this support but later decided not to include it because of its complexity).

One of the initial decisions we made when building LegoOS is to support the Linux system call interface and unmodified Linux ABI, because doing so can greatly ease the adoption of LegoOS. Distributed applications that run on Linux can seamlessly run on a LegoOS cluster by running on a set of vNodes.

4.2 Hardware Architecture

LegoOS pComponent, mComponent, and sComponent are independent devices, each having their own hardware controller and network interface (for pComponent, the hardware controller is the processor itself). Our current hardware model uses CPU in pComponent, DRAM in mComponent, and SSD or HDD in sComponent. We leave exploring other hardware devices for future work.

To demonstrate the feasibility of hardware resource disaggregation, we propose a pComponent and an

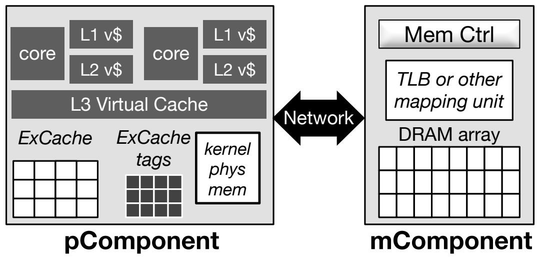


Figure 5: **LegoOS pComponent and mComponent Architecture.**

mComponent architecture designed within today's network, processor, and memory performance and hardware constraints (Figure 5).

Separating process and memory functionalities. LegoOS moves all hardware memory functionalities to mComponents (e.g., page tables, TLBs) and leaves *only* caches at the pComponent side. With a clean separation of process and memory hardware units, the allocation and management of memory can be completely transparent to pComponents. Each mComponent can choose its own memory allocation technique and virtual to physical memory address mappings (*e.g.*, segmentation).

Processor virtual caches. After moving all memory functionalities to mComponents, pComponents will only see virtual addresses and have to use virtual memory addresses to access its caches. Because of this, LegoOS organizes all levels of pComponent caches as *virtual caches* [44, 104], *i.e.*, virtually-indexed and virtually-tagged caches.

A virtual cache has two potential problems, commonly known as synonyms and homonyms [95]. Synonyms happens when a physical address maps to multiple virtual addresses (and thus multiple virtual cache lines) as a result of memory sharing across processes, and the update of one virtual cache line will not reflect to other lines that share the data. Since LegoOS does not allow writable inter-process memory sharing, it will not have the synonym problem. The homonym problem happens when two address spaces use the same virtual address for their own different data. Similar to previous solutions [20], we solve homonyms by storing an address space ID (ASID) with each cache line, and differentiate a virtual address in different address spaces using ASIDs.

Separating memory for performance and for capacity. Previous studies [41, 47] and our own show that today's network speed cannot meet application performance requirements if all memory accesses are across the network. Fortunately, many modern datacenter applications exhibit strong memory access temporal locality. For example, we found 90% of memory accesses in PowerGraph [43] go to just 0.06% of total memory and 95% go to 3.1% of memory (22% and 36% for TensorFlow [4] respectively, 5.1% and 6.6% for Phoenix [85]).

With good memory-access locality, we propose to leave a small amount of memory (*e.g.*, 4 GB) at each pComponent and move most memory across the network (*e.g.*, few TBs per mComponent). pComponents' local memory can be regular DRAM or the on-die HBM [53, 72], and mComponents use DRAM or NVM.

Different from previous proposals [63], we propose to organize pComponents' DRAM/HBM as cache rather than main memory for a clean separation of process and memory functionalities. We place this cache under the current processor Last-Level Cache (LLC) and call it an extended cache, or *ExCache*. ExCache serves as another layer in the memory hierarchy between LLC and memory across the network. With this design, ExCache can serve hot memory accesses fast, while mComponents can provide the capacity applications desire.

ExCache is a virtual, inclusive cache, and we use a combination of hardware and software to manage ExCache. Each ExCache line has a (virtual-address) tag and two access permission bits (one for read/write and one for valid). These bits are set by software when a line is inserted to ExCache and checked by hardware at access time. For best hit performance, the hit path of ExCache is handled purely by hardware — the hardware cache controller maps a virtual address to an ExCache set, fetches and compares tags in the set, and on a hit, fetches the hit ExCache line. Handling misses of ExCache is more complex than with traditional CPU caches, and thus we use LegoOS to handle the miss path of ExCache (see §4.3.2).

Finally, we use a small amount of DRAM/HBM at pComponent for LegoOS' own kernel data usages, accessed directly with physical memory addresses and managed by LegoOS. LegoOS ensures that all its own data fits in this space to avoid going to mComponents.

With our design, pComponents do not need any address mappings: LegoOS accesses all pComponent-side DRAM/HBM using physical memory addresses and does simple calculations to locate the ExCache set for a memory access. Another benefit of not handling address mapping at pComponents and moving TLBs to mComponents is that pComponents do not need to access TLB or suffer from TLB misses, potentially making pComponent cache accesses faster [58].

4.3 Process Management

The LegoOS *process monitor* runs in the kernel space of a pComponent and manages the pComponent's CPU cores and ExCache. pComponents run user programs in the user space.

4.3.1 Process Management and Scheduling

At every pComponent, LegoOS uses a simple local thread scheduling model that targets datacenter applications (we will discuss global scheduling in § 4.6). LegoOS dedicates a small amount of cores for kernel background threads (currently two to four) and uses the rest of the cores for application threads. When a new process starts, LegoOS uses a global policy to choose a pComponent for it (§ 4.6). Afterwards, LegoOS schedules new threads the process spawns on the same pComponent by choosing the cores that host fewest threads. After assigning a thread to a core, we let it run to the end with no scheduling or kernel preemption under common scenarios. For example, we do not use any network interrupts and let threads busy wait on the completion of outstanding network requests, since a network request in LegoOS is fast (*e.g.*, fetching an ExCache line from an mComponent takes around 6.5 μs). LegoOS improves the overall processor utilization in a disaggregated cluster, since it can freely schedule processes on any pComponents without considering memory allocation. Thus, we do not push for perfect core utilization when scheduling individual threads and instead aim to minimize scheduling and context switch performance overheads. Only when a pComponent has to schedule more threads than its cores will LegoOS start preempting threads on a core.

4.3.2 ExCache Management

LegoOS process monitor configures and manages ExCache. During the pComponent's boot time, LegoOS configures the set associativity of ExCache and its cache replacement policy. While ExCache hit is handled completely in hardware, LegoOS handles misses in software. When an ExCache miss happens, the process monitor fetches the corresponding line from an mComponent and inserts it to ExCache. If the ExCache set is full, the process monitor first evicts a line in the set. It throws away the evicted line if it is clean and writes it back to an mComponent if it is dirty. LegoOS currently supports two eviction policies: FIFO and LRU. For each ExCache set, LegoOS maintains a FIFO queue (or an approximate LRU list) and chooses ExCache lines to evict based on the corresponding policy (see §5.3 for details).

4.3.3 Supporting Linux Syscall Interface

One of our early decisions is to support Linux ABIs for backward compatibility and easy adoption of LegoOS. A challenge in supporting the Linux system call interface is that many Linux syscalls are associated with *states*, information about different Linux subsystems that is stored with each process and can be accessed by user programs across syscalls. For example, Linux records the states of a running process' open files, socket connections, and several other entities, and it associates these states with file descriptors (*fd*s) that are exposed to users. In contrast, LegoOS aims at the clean separation of OS functionalities. With LegoOS' stateless design principle, each component only stores information about its own resource and each request across components contains all the information that the destination component needs to handle the request. To solve this discrepancy between the Linux syscall interface and LegoOS' design, we add a layer on top of LegoOS' core process monitor at each pComponent to store Linux states and translate these states and the Linux syscall interface to LegoOS' internal interface.

4.4 Memory Management

We use mComponents for three types of data: anonymous memory (*i.e.*, heaps, stacks), memory-mapped files, and storage buffer caches. The LegoOS *memory monitor* manages both the virtual and physical memory address spaces, their allocation, deallocation, and memory address mappings. It also performs the actual memory read and write. No user processes run on mComponents and they run completely in the kernel mode (same is true for sComponents).

LegoOS lets a process address space span multiple mComponents to achieve efficient memory space utilization and high parallelism. Each application process uses one or more mComponents to host its data and a *home mComponent*, an mComponent that initially loads the process, accepts and oversees all system calls related to virtual memory space management (*e.g.*, `brk`, `mmap`, `munmap`, and `mremap`). LegoOS uses a global memory resource manager (*GMM*) to assign a home mComponent to each new process at its creation time. A home mComponent can also host process data.

4.4.1 Memory Space Management

Virtual memory space management. We propose a two-level approach to manage distributed virtual memory spaces, where the home mComponent of a process makes coarse-grained, high-level virtual memory allocation decisions and other mComponents perform fine-grained virtual memory allocation. This approach minimizes network communication during both normal memory accesses and virtual memory operations, while ensuring good load balancing and memory utilization. Figure 6 demonstrates the data structures used.

At the higher level, we split each virtual memory address space into coarse-grained, fix-sized *virtual regions*, or *vRegions* (*e.g.*, of 1 GB). Each vRegion that contains allocated virtual memory addresses (an active vRegion) is *owned* by an mComponent. The owner of a vRegion handles all memory accesses and virtual memory requests within the vRegion.

The lower level stores user process virtual memory area (*vma*) information, such as virtual address ranges and permissions, in *vma trees*. The owner of an active vRegion stores a vma tree for the vRegion, with each node in the tree being one vma. A user-perceived virtual memory range can split across multiple mComponents, but only one mComponent owns a vRegion.

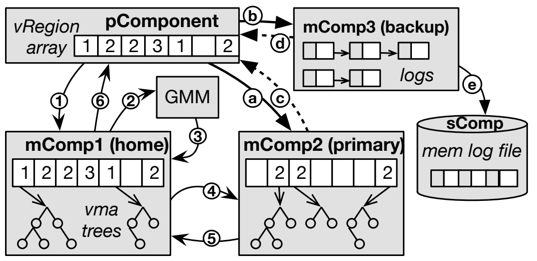


Figure 6: **Distributed Memory Management.**

vRegion owners perform the actual virtual memory allocation and vma tree set up. A home mComponent can also be the owner of vRegions, but the home mComponent does not maintain any information about memory that belongs to vRegions owned by other mComponents. It only keeps the information of which mComponent owns a vRegion (in a *vRegion array*) and how much free virtual memory space is left in each vRegion. These metadata can be easily reconstructed if a home mComponent fails.

When an application process wants to allocate a virtual memory space, the pComponent forwards the allocation request to its home mComponent (① in Figure 6). The home mComponent uses its stored information of available virtual memory space in vRegions to find one or more vRegions that best fit the requested amount of virtual memory space. If no active vRegion can fit the allocation request, the home mComponent makes a new vRegion active and contacts the GMM (② and ③) to find a candidate mComponent to own the new vRegion. GMM makes this decision based on available physical memory space and access load on different mComponents (§ 4.6). If the candidate mComponent is not the home mComponent, the home mComponent next forwards the request to that mComponent (④), which then performs local virtual memory area allocation and sets up the proper vma tree. Afterwards, the pComponent directly sends memory access requests to the owner of the vRegion where the memory access falls into (*e.g.*, ⓐ and ⓒ in Figure 6).

LegoOS' mechanism of distributed virtual memory management is efficient and it cleanly separates memory operations from pComponents. pComponents hand over all memory-related system call requests to mComponents and only cache a copy of the vRegion array for fast memory accesses. To fill a cache miss or to flush a dirty cache line, a pComponent looks up the cached vRegion array to find its owner mComponent and sends the request to it.

Physical memory space management. Each mComponent manages the physical memory allocation for data that falls into the vRegion that it owns. Each mComponent can choose their own way of physical memory allocation and own mechanism of virtual-to-physical memory address mapping.

4.4.2 Optimization on Memory Accesses

With our strawman memory management design, all ExCache misses will go to mComponents. We soon found that a large performance overhead in running real applications is caused by filling empty ExCache, *i.e.*, *cold misses*. To reduce the performance overhead of cold misses, we propose a technique to avoid accessing mComponent on first memory accesses.

The basic idea is simple: since the initial content of anonymous memory (non-file-backed memory) is zero, LegoOS can directly allocate a cache line with empty content in ExCache for the first access to anonymous memory instead of going to mComponent (we call such cache lines *p-local lines*). When an application creates a new anonymous memory region, the process monitor records its address range and permission. The application's first access to this region will be an ExCache miss and it will trap to LegoOS. LegoOS process monitor then allocates an ExCache line, fills it with zeros, and sets its R/W bit according to the recorded memory region's permission. Before this p-local line is evicted, it only lives in the ExCache. No mComponents are aware of it or will allocate physical memory or a virtual-to-physical memory mapping for it. When a p-local cache line becomes dirty and needs to be flushed, the process monitor sends it to its owner mComponent, which then allocates physical memory space and establishes a virtual-to-physical memory mapping. Essentially, LegoOS *delays physical memory allocation until write time*. Notice that it is safe to only maintain p-local lines at a pComponent ExCache without any other pComponents knowing them, since pComponents in LegoOS do not share any memory and other pComponents will not access this data.

4.5 Storage Management

LegoOS supports a hierarchical file interface that is backward compatible with POSIX through its vNode abstraction. Users can store their directories and files under their vNodes' mount points and perform normal read, write, and other accesses to them.

LegoOS implements core storage functionalities at sComponents. To cleanly separate storage functionalities, LegoOS uses a stateless storage server design, where each I/O request to the storage server contains all the information needed to fulfill this request, *e.g.*, full path name, absolute file offset, similar to the server design in NFS v2 [89].

While LegoOS supports a hierarchical file use interface, internally, LegoOS storage monitor treats (full) directory and file paths just as unique names of a file and place all files of a vNode under one internal directory at the sComponent. To locate a file, LegoOS storage monitor maintains a simple hash table with the full paths of files (and directories) as keys. From our observation, most datacenter applications only have a few hundred files or less. Thus, a simple hash table for a whole vNode is sufficient to achieve good lookup performance. Using a non-hierarchical file system implementation largely reduces the complexity of LegoOS' file system, making it possible for a storage monitor to fit in storage devices controllers that have limited processing power [92].

LegoOS places the storage buffer cache at mComponents rather than at sComponents, because sComponents can only host a limited amount of internal memory. LegoOS memory monitor manages the storage buffer cache by simply performing insertion, lookup, and deletion of buffer cache entries. For simplicity and to avoid coherence traffic, we currently place the buffer cache of one file under one mComponent. When receiving a file read system call, the LegoOS process monitor first uses its extended Linux state layer to look up the full path name, then passes it with the requested offset and size to the mComponent that holds the file's buffer cache. This mComponent will look up the buffer cache and returns the data to pComponent on a hit. On a miss, mComponent will forward the request to the sComponent that stores the file, which will fetch the data from storage device and return it to the mComponent. The mComponent will then insert it into the buffer cache and returns it to the pComponent. Write and fsync requests work in a similar fashion.

4.6 Global Resource Management

LegoOS uses a two-level resource management mechanism. At the higher level, LegoOS uses three global resource managers for process, memory, and storage resources, *GPM, GMM*, and *GSM*. These global managers perform coarse-grained global resource allocation and load balancing, and they can run on one normal Linux machine. Global managers only maintain approximate resource usage and load information. They update their information either when they make allocation decisions or by periodically asking monitors in the cluster. At the lower level, each monitor can employ its own policies and mechanisms to manage its local resources.

For example, process monitors allocate new threads locally and only ask GPM when they need to create a new process. GPM chooses the pComponent that has the least amount of threads based on its maintained approximate information. Memory monitors allocate virtual and physical memory space on their own. Only home mComponent asks GMM when it needs to allocate a new vRegion. GMM maintains approximate physical memory space usages and memory access load by periodically asking mComponents and chooses the memory with least load among all the ones that have at least vRegion size of free physical memory.

LegoOS decouples the allocation of different resources and can freely allocate each type of resource from a pool of components. Doing so largely improves resource packing compared to a monolithic server cluster that packs all type of resources a job requires within one physical machine. Also note that LegoOS allocates hardware resources only *on demand*, *i.e.*, when applications actually create threads or access physical memory. This on-demand allocation strategy further improves LegoOS' resource packing efficiency and allows more aggressive over-subscription in a cluster.

4.7 Reliability and Failure Handling

After disaggregation, pComponents, mComponents, and sComponents can all fail independently. Our goal is to build a reliable disaggregated cluster that has the same or lower application failure rate than a monolithic cluster. As a first (and important) step towards achieving this goal, we focus on providing memory reliability by handling mComponent failure in the current version of LegoOS because of three observations. First, when distributing an application's memory to multiple mComponents, the probability of memory failure increases and not handling mComponent failure will cause applications to fail more often on a disaggregated cluster than on monolithic servers. Second, since most modern datacenter applications already provide reliability to their distributed storage data and the current version of LegoOS does not split a file across sComponent, we leave providing storage reliability to applications. Finally, since LegoOS does not split a process across pComponents, the chance of a running application process being affected by the failure of a pComponent is similar to one affected by the failure of a processor in a monolithic server. Thus, we currently do not deal with pComponent failure and leave it for future work.

A naive approach to handle memory failure is to perform a full replication of memory content over two or more mComponents. This method would require at least $2\times$ memory space, making the monetary and energy cost of providing reliability prohibitively high (the same reason why RAMCloud [80] does not replicate in memory). Instead, we propose a space- and performance-efficient approach to provide in-memory data reliability in a best-effort way. Further, since losing in-memory data will not affect user persistent data, we propose to provide memory reliability in a best-effort manner.

We use one primary mComponent, one secondary mComponent, and a backup file in sComponent for each vma. A mComponent can serve as the primary for some vma and the secondary for others. The primary stores all memory data and metadata. LegoOS maintains a small append-only log at the secondary mComponent and also replicates the vma tree there. When pComponent flushes a dirty ExCache line, LegoOS sends the data to both primary and secondary in parallel (step ⓐ and ⓑ in Figure 6) and waits for both to reply (ⓒ and ⓓ). In the background, the secondary mComponent flushes the backup log to a sComponent, which writes it to an append-only file.

If the flushing of a backup log to sComponent is slow and the log is full, we will skip replicating application memory. If the primary fails during this time, LegoOS simply reports an error to application. Otherwise when a primary mComponent fails, we can recover memory content by replaying the backup logs on sComponent and in the secondary mComponent. When a secondary mComponent fails, we do not reconstruct anything and start replicating to a new backup log on another mComponent.

5 LegoOS Implementation

We implemented LegoOS in C on the x86-64 architecture. LegoOS can run on commodity, off-the-shelf machines and support most commonly-used Linux system call APIs. Apart from being a proof-of-concept of the splitkernel OS architecture, our current LegoOS implementation can also be used on existing datacenter servers to reduce the energy cost, with the help of techniques like Zombieland [77]. Currently, LegoOS has 206K SLOC, with 56K SLOC for drivers. LegoOS supports 113 syscalls, 15 pseudo-files, and 10 vectored syscall opcodes. Similar to the findings in [100], we found that implementing these Linux interfaces are sufficient to run many unmodified datacenter applications.

5.1 Hardware Emulation

Since there is no real resource disaggregation hardware, we emulate disaggregated hardware components using commodity servers by limiting their internal hardware usages. For example, to emulate controllers for mComponents and sComponents, we limit the usable cores of a server to two. To emulate pComponents, we limit the amount of usable main memory of a server and configure it as LegoOS software-managed ExCache.

5.2 Network Stack

We implemented three network stacks in LegoOS. The first is a customized RDMA-based RPC framework we implemented based on LITE [101] on top of the Mellanox mlx4 InfiniBand driver we ported from Linux. Our RDMA RPC implementation registers physical memory addresses with RDMA NICs and thus eliminates the need for NICs to cache physical-to-virtual memory address mappings [101]. The resulting smaller NIC SRAM can largely reduce the monetary cost of NICs, further saving the total cost of a LegoOS cluster. All LegoOS internal communications use this RPC framework. For best latency, we use one dedicated polling thread at RPC server side to keep polling incoming requests. Other thread(s) (which we call worker threads) execute the actual RPC functions. For each pair of components, we use one physically consecutive memory region at a component to serve as the receive buffer for RPC requests. The RPC client component uses RDMA write with immediate value to directly write into the memory region and the polling thread polls for the immediate value to get the metadata information about the RPC request (*e.g.*, where the request is written to in the memory region). Immediately after getting an incoming request, the polling thread passes it along to a work queue and continues to poll for the next incoming request. Each worker thread checks if the work queue is not empty and if so, gets an RPC request to process. Once it finishes the RPC function, it sends the return value back to the RPC client with an RDMA write to a memory address at the RPC client. The RPC client allocates this memory address for the return value before sending the RPC request and piggy-backs the memory address with the RPC request.

The second network stack is our own implementation of the socket interface directly on RDMA. The final stack is a traditional socket TCP/IP stack we adapted from lwip [35] on our ported e1000 Ethernet driver. Applications can choose between these two socket implementations and use virtual IPs for their socket communication.

5.3 Processor Monitor

We reserve a contiguous physical memory region during kernel boot time and use fixed ranges of memory in this region as ExCache, tags and metadata for these caches, and kernel physical memory. We organize ExCache into virtually indexed sets with a configurable set associativity. Since x86 (and most other architectures) uses hardware-managed TLB and walks page table directly after TLB misses, we have to use paging and the only chance we can trap to OS is at page fault time. We thus use paged memory to emulate ExCache, with each ExCache line being a 4 KB page. A smaller ExCache line size would improve the performance of fetching lines from mComponents but increase the size of ExCache tag array and the overhead of tag comparison.

An ExCache miss causes a page fault and traps to LegoOS. To minimize the overhead of context switches, we use the application thread that faults on a ExCache miss to perform ExCache replacement. Specifically, this thread will identify the set to insert the missing page using its virtual memory address, evict a page in this set if it is full, and if needed, flush a dirty page to mComponent (via a LegoOS RPC call to the owner mComponent of the vRegion this page is in). To minimize the network round trip needed to complete a ExCache miss, we piggy-back the request of dirty page flush and new page fetching in one RPC call when the mComponent to be flushed to and the mComponent to fetch the missing page are the same.

LegoOS maintains an approximate LRU list for each ExCache set and uses a background thread to sweep all entries in ExCache and adjust LRU lists. LegoOS supports two ExCache replacement policies: FIFO and LRU. For FIFO replacement, we simply maintain a FIFO queue for each ExCache set and insert a corresponding entry to the tail of the FIFO queue when an ExCache page is inserted into the set. Eviction victim is chosen as the head of the FIFO queue. For LRU, we use one background thread to sweep all sets of ExCache to adjust their LRU lists. For both policies, we use a per-set lock and lock the FIFO queue (or the LRU list) when making changes to them.

5.4 Memory Monitor

We use regular machines to emulate mComponents by limiting usable cores to a small number (2 to 5 depending on configuration). We dedicate one core to busy poll network requests and the rest for performing memory functionalities. The LegoOS memory monitor performs all its functionalities as handlers of RPC requests from pComponents. The memory monitor handles most of these functionalities locally and sends another RPC request to a sComponent for storage-related functionalities (*e.g.*, when a buffer cache miss happens). LegoOS stores application data, application memory address mappings, vma trees, and vRegion arrays all in the main memory of the emulating machine.

The memory monitor loads an application executable from sComponents to the mComponent, handles application virtual memory address allocation requests, allocates physical memory at the mComponent, and reads/writes data to the mComponent. Our current implementation of memory monitor is purely in software, and we use hash tables to implement the virtual-to-physical address mappings. While we envision future mComponents to implement memory monitors in hardware and to have specialized hardware parts to store address mappings, our current software implementation can still be useful for users that want to build software-managed mComponents.

5.5 Storage Monitor

Since storage is not the focus of the current version of LegoOS, we chose a simple implementation of building storage monitor on top of the Linux *vfs* layer as a loadable Linux kernel module. LegoOS creates a normal file over vfs as the mount partition for each vNode and issues vfs file operations to perform LegoOS storage I/Os. Doing so is sufficient to evaluate LegoOS, while largely saving our implementation efforts on storage device drivers and layered storage protocols. We leave exploring other options of building LegoOS storage monitor to future work.

5.6 Experience and Discussion

We started our implementation of LegoOS from scratch to have a clean design and implementation that can fit the splitkernel model and to evaluate the efforts needed in building different monitors. However, with the vast amount and the complexity of drivers, we decided to port Linux drivers instead of writing our own. We then spent our engineering efforts on an "as needed" base and took shortcuts by porting some of the Linux code. For example, we re-used common algorithms and data structures in Linux to easily port Linux drivers. We believe that being able to support largely unmodified Linux drivers will assist the adoption of LegoOS.

When we started building LegoOS, we had a clear goal of sticking to the principle of "clean separation of functionalities". However, we later found several places where performance could be improved if this principle is relaxed. For example, for the optimization in §4.4.2 to work correctly, pComponent needs to store the address range and permission for anonymous virtual memory regions — memory-related information that otherwise only mComponents need to know. Another example is the implementation of `mremap`. With LegoOS' principle of mComponents handling all memory address allocations, memory monitors will allocate new virtual memory address ranges for `mremap` requests. However, when data in the `mremap` region is in ExCache, LegoOS needs to move it to another set if the new virtual address does not fall into the current set. If mComponents are ExCache-aware, they can choose the new virtual memory address to fall into the same set as the current one. Our strategy is to relax the clean-separation principle only by giving "hints", and only for frequently-accessed, performance-critical operations.

6 Evaluation

This section presents the performance evaluation of LegoOS using micro- and macro-benchmarks and two unmodified real applications. We also quantitatively analyze the failure rate of LegoOS. We ran all experiments on a cluster of 10 machines, each with two Intel Xeon CPU E5-2620 2.40GHz processors, 128 GB DRAM, and one 40 Gbps Mellanox ConnectX-3 InfiniBand network adapter; a Mellanox 40 Gbps InfiniBand switch connects all of the machines. The Linux version we used for comparison is v4.9.47.

6.1 Micro- and Macro-benchmark Results

Network performance. Network communication is at the core of LegoOS' performance. Thus, we evaluate LegoOS' network performance first before evaluating LegoOS as a whole. Figure 7 plots the average latency of sending messages with socket-over-InfiniBand (Linux-

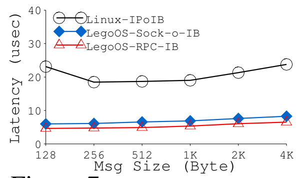


Figure 7: **Network Latency.**

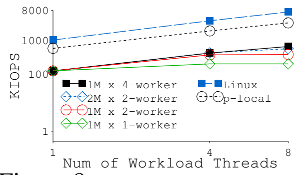


Figure 8: **Memory Throughput.**

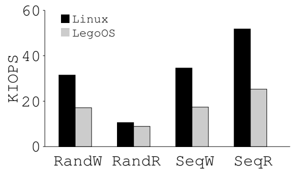


Figure 9: **Storage Throughput.**

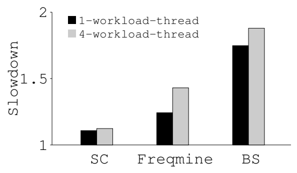


Figure 10: **PARSEC Results.** *SC: StreamClsuter. BS: BlackScholes.*

IPoIB) in Linux, LegoOS' implementation of socket on top of InfiniBand (LegoOS-Sock-o-IB), and LegoOS' implementation of RPC over InfiniBand (LegoOS-RPC-IB). LegoOS uses LegoOS-RPC-IB for all its internal network communication across components and uses LegoOS-Sock-o-IB for all application-initiated socket network requests. Both LegoOS' networking stacks significantly outperform Linux's.

Memory performance. Next, we measure the performance of mComponent using a multi-threaded user-level micro-benchmark. In this micro-benchmark, each thread performs one million sequential 4 KB memory loads in a heap. We use a huge, empty ExCache (32 GB) to run this test, so that each memory access can generate an ExCache (cold) miss and go to the mComponent.

Figure 8 compares LegoOS' mComponent performance with Linux's single-node memory performance using this workload. We vary the number of per-mComponent worker threads from 1 to 8 with one and two mComponents (only showing representative configurations in Figure 8). In general, using more worker threads per mComponent and using more mComponents both improve throughput when an application has high parallelism, but the improvement largely diminishes after the total number of worker threads reaches four. We also evaluated the optimization technique in § 4.4.2 (*p-local* in Figure 8). As expected, bypassing mComponent accesses with p-local lines significantly improves memory access performance. The difference between p-local and Linux demonstrates the overhead of trapping to LegoOS kernel and setting up ExCache.

Storage performance. To measure the performance of LegoOS' storage system, we ran a single-thread micro-benchmark that performs sequential and random 4 KB read/write to a 25 GB file on a Samsung PM1725s NVMe SSD (the total amount of data accessed is 1 GB). For write workloads, we issue an *fsync* after each *write* to test the performance of writing all the way to the SSD.

Figure 9 presents the throughput of this workload on LegoOS and on single-node Linux. For LegoOS, we use one mComponent to store the buffer cache of this file and initialize the buffer cache to empty so that file I/Os can go to the sComponent (Linux also uses an empty buffer cache). Our results show that Linux's performance is determined by the SSD's read/write bandwidth. LegoOS' random read performance is close to Linux, since network cost is relatively low compared to the SSD's random read performance. With better SSD sequential read performance, network cost has a higher impact. LegoOS' write-and-fsync performance is worse than Linux because LegoOS requires one RTT between pComponent and mComponent to perform write and two RTTs (pComponent to mComponent, mComponent to sComponent) for fsync.

PARSEC results. We evaluated LegoOS with a set of workloads from the PARSEC benchmark suite [22], including BlackScholes, Freqmine, and StreamCluster. These workloads are a good representative of compute-intensive datacenter applications, ranging from machine-learning algorithms to streaming processing ones. Figure 10 presents the slowdown of LegoOS over single-node Linux with enough memory for the entire application working sets. LegoOS uses one pComponent with 128 MB ExCache, one mComponent with one worker thread, and one sComponent for all the PARSEC tests. For each workload, we tested one and four workload threads. StreamCluster, a streaming workload, performs the best because of its batching memory access pattern (each batch is around 110 MB). BlackScholes and Freqmine perform worse because of their larger working sets (630 MB to 785 MB). LegoOS performs worse with higher workload threads, because the single worker thread at the mComponent becomes the bottleneck to achieving higher throughput.

6.2 Application Performance

We evaluated LegoOS' performance with two real, unmodified applications, TensorFlow [4] and Phoenix [85], a single-node multi-threaded implementation of MapReduce [32]. TensorFlow's experiments use the Cifar-10 dataset [2] and Phoenix's use a Wikipedia dataset [3]. Unless otherwise stated, the base configuration used for all TensorFlow experiments is 256 MB 64-way ExCache, one pComponent, one mComponent, and one sComponent. The base configuration for Phoenix is the same as TensorFlow's with the exception that the base ExCache size is 512 MB. The total amount of virtual memory addresses touched in TensorFlow is 4.4 GB (1.75 GB for Phoenix). The total working sets of the TensorFlow and Phoenix execution are 0.9 GB and 1.7 GB. Our default

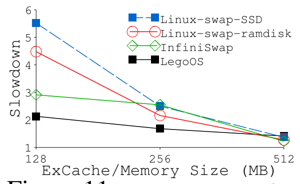


Figure 11: **TensorFlow Perf.**

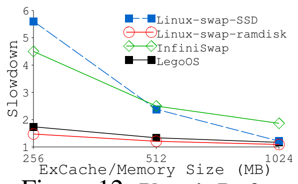


Figure 12: **Phoenix Perf.**

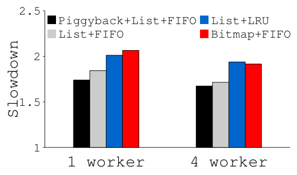


Figure 13: **ExCache Mgmt.**

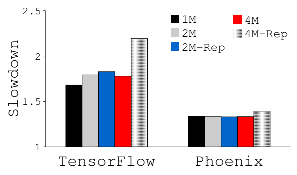


Figure 14: **Memory Config.**

ExCache sizes are set as roughly 25% of total working sets. We ran both applications with four threads.

Impact of ExCache size on application performance. Figures 11 and 12 plot the TensorFlow and Phoenix run time comparison across LegoOS, a remote swapping system (InfiniSwap [47]), a Linux server with a swap file in a local high-end NVMe SSD, and a Linux server with a swap file in local ramdisk. All values are calculated as a slowdown to running the applications on a Linux server that have enough local resources (main memory, CPU cores, and SSD). For systems other than LegoOS, we change the main memory size to the same size of ExCache in LegoOS, with rest of the memory on swap file. With around 25% working set, LegoOS only has a slowdown of 1.68× and 1.34× for TensorFlow and Phoenix compared to a monolithic Linux server that can fit all working sets in its main memory.

LegoOS' performance is significantly better than swapping to SSD and to remote memory largely because of our efficiently-implemented network stack, simplified code path compared with Linux paging subsystem, and the optimization technique proposed in §4.4.2. Surprisingly, it is similar or even better than swapping to local memory, even when LegoOS' memory accesses are across network. This is mainly because ramdisk goes through buffer cache and incurs memory copies between the buffer cache and the in-memory swap file.

LegoOS' performance results are not easy to achieve and we went through rounds of design and implementation refinement. Our network stack and RPC optimizations yield a total improvement of up to 50%. For example, we made all RPC server (mComponent's) replies *unsignaled* to save mComponent' processing time and to increase its request handling throughput. Another optimization we did is to piggy-back dirty cache line flush and cache miss fill into one RPC. The optimization of the first anonymous memory access (§4.4.2) improves LegoOS' performance further by up to 5%.

ExCache Management. Apart from its size, how an ExCache is managed can also largely affect application performance. We first evaluated factors that could affect ExCache hit rate and found that higher associativity improves hit rate but the effect diminishes when going beyond 512-way. We then focused on evaluating the miss cost of ExCache, since the miss path is handled by LegoOS in our design. We compare the two eviction policies LegoOS supports (FIFO and LRU), two implementations of finding an empty line in an ExCache set (linearly scan a free bitmap and fetching the head of a free list), and one network optimization (piggyback flushing a dirty line with fetching the missing line).

Figure 13 presents these comparisons with one and four mComponent worker threads. All tests run the Cifar-10 workload on TensorFlow with 256 MB 64-way ExCache, one mComponent, and one sComponent. Using bitmaps for this ExCache configuration is always worse than using free lists because of the cost to linearly scan a whole bitmap, and bitmaps perform even worse with higher associativity. Surprisingly, FIFO performs better than LRU in our tests, even when LRU utilizes access locality pattern. We attributed LRU's worse performance to the lock contention it incurs; the kernel background thread sweeping the ExCache locks an LRU list when adjusting the position of an entry in it, while ExCache miss handler thread also needs to lock the LRU list to grab its head. Finally, the piggyback optimization works well and the combination of FIFO, free list, and piggyback yields the best performance.

Number of mComponents and replication. Finally, we study the effect of the number of mComponents and memory replication. Figure 14 plots the performance slowdown as the number of mComponents increases from one to four. Surprisingly, using more mComponents lowers application performance by up to 6%. This performance drop is due to the effect of ExCache piggyback optimization. When there is only one mComponent, flushes and misses are all between the pComponent and this mComponent, thus enabling piggyback on every flush. However, when there are multiple mComponents, LegoOS can only perform piggyback when flushes and misses are to the same mComponent.

We also evaluated LegoOS' memory replication performance in Figure 14. Replication has a performance overhead of 2% to 23% (there is a constant 1 MB space overhead to store the backup log). LegoOS uses the same application thread to send the replica data to the backup mComponent and then to the primary mComponent, resulting in the performance lost.

Running multiple applications together. All our experiments so far run only one application at a time. Now we evaluate how multiple applications perform when running them together on a LegoOS cluster. We use a simple scenario of running one TensorFlow instance and one Phoenix instance together in two settings: 1) two pCom-

	Processor	Disk	Memory	NIC	Power	Other	Monolithic	LegoOS
MTTF (year)	204.3	33.1	289.9	538.8	100.5	27.4	5.8	6.8 - 8.7

Table 1: **Mean Time To Failure Analysis.** *MTTF numbers of devices (columns 2 to 7) are obtained from [90] and MTTF values of monolithic server and LegoOS are calculated using the per-device MTTF numbers.*

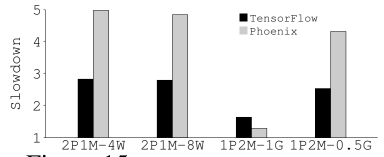


Figure 15: **Multiple Applications.**

ponents each running one instance, both accessing one mComponent(2P1M), and 2) one pComponent running two instances and accessing two mComponents (1P2M). Both settings use one sComponent. Figure 15 presents the runtime slowdown results. We also vary the number of mComponent worker threads for the 2P1M setting and the amount of ExCache for the 1P2M setting. With 2P1M, both applications suffer from a performance drop because their memory access requests saturate the single mComponent. Using more worker threads at the mComponent improves the performance slightly. For 1P2M, application performance largely depends on ExCache size, similar to our findings with single-application experiments.

6.3 Failure Analysis

Finally, we provide a qualitative analysis on the failure rate of a LegoOS cluster compared to a monolithic server cluster. Table 1 summarizes our analysis. To measure the failure rate of a cluster, we use the metric Mean Time To (hardware) Failure (MTTF), the mean time to the failure of a server in a monolithic cluster or a component in a LegoOS cluster. Since the only real per-device failure statistics we can find are the mean time to hardware replacement in a cluster [90], the MTTF we refer to in this study indicates the mean time to the type of hardware failures that require replacement. Unlike traditional MTTF analysis, we are not able to include transient failures.

To calculate MTTF of a monolithic server, we first obtain the replacement frequency of different hardware devices in a server (CPU, memory, disk, NIC, motherboard, case, power supply, fan, CPU heat sink, and other cables and connections) from the real world (the COM1 and COM2 clusters in [90]). For LegoOS, we envision every component to have a NIC and a power supply, and in addition, a pComponent to have CPU, fan, and heat sink, an mComponent to have memory, and an sComponent to have a disk. We further assume both a monolithic server and a LegoOS component to fail when any hardware devices in them fails and the devices in them fail independently. Thus, the MTTF can be calculated using the harmonic mean (*HM*) of the MTTF of each device.

$$MTTF = \frac{HM_{i=0}^{n}(MTTF_i)}{n} \tag{1}$$

where n includes all devices in a machine/component.

Further, when calculating MTTF of LegoOS, we estimate the amount of components needed in LegoOS to run the same applications as a monolithic cluster. Our estimated worst case for LegoOS is to use the same amount of hardware devices (*i.e.*, assuming same resource utilization as monolithic cluster). LegoOS' best case is to achieve full resource utilization and thus requiring only about half of CPU and memory resources (since average CPU and memory resource utilization in monolithic server clusters is around 50% [10, 45]).

With better resource utilization and simplified hardware components (*e.g.*, no motherboard), LegoOS improves MTTF by 17% to 49% compared to an equivalent monolithic server cluster.

7 Related Work

Hardware Resource Disaggregation. There have been a few hardware disaggregation proposals from academia and industry, including Firebox [15], HP "The Machine" [40, 48], dRedBox [56], and IBM Composable System [28]. Among them, dRedBox and IBM Composable System package hardware resources in one big case and connect them with buses like PCIe. The Machine's scale is a rack and it connects SoCs with NVMs with a specialized coherent network. FireBox is an early-stage project and is likely to use high-radix switches to connect customized devices. The disaggregated cluster we envision to run LegoOS on is one that hosts hundreds to thousands of non-coherent, heterogeneous hardware devices, connected with a commodity network.

Memory Disaggregation and Remote memory. Lim *et al.* first proposed the concept of hardware disaggregated memory with two models of disaggregated memory: using it as network swap device and transparently accessing it through memory instructions [63, 64]. Their hardware models still use a monolithic server at the local side. LegoOS' hardware model separates processor and memory completely.

Another set of recent projects utilize remote memory without changing monolithic servers [6, 34, 47, 74, 79, 93]. For example, InfiniSwap [47] transparently swaps local memory to remote memory via RDMA. These remote memory systems help improve the memory resource packing in a cluster. However, as discussed in §2, unlike LegoOS, these solutions cannot solve other limitations of monolithic servers like the lack of hardware heterogeneity and elasticity.

Storage Disaggregation. Cloud vendors usually provision storage at different physical machines [13, 103, 108]. Remote access to hard disks is a common practice, because their high latency and low throughput can easily hide network overhead [61, 62, 70, 105]. While disaggregating high-performance flash is a more challenging task [38, 59]. Systems such as ReFlex [60], Decibel [73], and PolarFS [25], tightly integrate network and storage layers to minimize software overhead in the face of fast hardware. Although storage disaggregation is not our main focus now, we believe those techniques can be realized in future LegoOS easily.

Multi-Kernel and Multi-Instance OSes. Multi-kernel OSes like Barrelfish [21, 107], Helios [76], Hive [26], and fos [106] run a small kernel on each core or programmable device in a monolithic server, and they use message passing to communicate across their internal kernels. Similarly, multi-instance OSes like Popcorn [17] and Pisces [83] run multiple Linux kernel instances on different cores in a machine. Different from these OSes, LegoOS runs on and manages a distributed set of hardware devices; it manages distributed hardware resources using a two-level approach and handles device failures (currently only mComponent). In addition, LegoOS differs from these OSes in how it splits OS functionalities, where it executes the split kernels, and how it performs message passing across components. Different from multi-kernels' message passing mechanisms which are performed over buses or using shared memory in a server, LegoOS' message passing is performed using a customized RDMA-based RPC stack over InfiniBand or RoCE network. Like LegoOS, fos [106] separates OS functionalities and run them on different processor cores that share main memory. Helios [76] runs *satellite kernels* on heterogeneous cores and programmable NICs that are not cache-coherent. We took a step further by disseminating OS functionalities to run on individual, network-attached hardware devices. Moreover, LegoOS is the first OS that separates memory and process management and runs virtual memory system completely at network-attached memory devices.

Distributed OSes. There have been several distributed OSes built in late 80s and early 90s [14, 16, 19, 27, 82, 86, 96, 97]. Many of them aim to appear as a single machine to users and focus on improving inter-node IPCs. Among them, the most closely related one is Amoeba [96, 97]. It organizes a cluster into a shared process pool and disaggregated specialized servers. Unlike Amoeba, LegoOS further separates memory from processors and different hardware components are loosely coupled and can be heterogeneous instead of as a homogeneous pool. There are also few emerging proposals to build distributed OSes in datacenters [54, 91], *e.g.*, to reduce the performance overhead of middleware. LegoOS achieves the same benefits of minimal middleware layers by only having LegoOS as the system management software for a disaggregated cluster and using the lightweight vNode mechanism.

8 Discussion and Conclusion

We presented LegoOS, the first OS designed for hardware resource disaggregation. LegoOS demonstrated the feasibility of resource disaggregation and its advantages in better resource packing, failure isolation, and elasticity, all without changing Linux ABIs. LegoOS and resource disaggregation in general can help the adoption of new hardware and thus encourage more hardware and system software innovations.

LegoOS is a research prototype and has a lot of room for improvement. For example, we found that the amount of parallel threads an mComponent can use to process memory requests largely affect application throughput. Thus, future developers of real mComponents can consider use large amount of cheap cores or FPGA to implement memory monitors in hardware.

We also performed an initial investigation in load balancing and found that memory allocation policies across mComponents can largely affect application performance. However, since we do not support memory data migration yet, the benefit of our load-balancing mechanism is small. We leave memory migration for future work. In general, large-scale resource management of a disaggregated cluster is an interesting and important topic, but is outside of the scope of this paper.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Michael Swift for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We are also thankful to Sumukh H. Ravindra for his contribution in the early stage of this work.

This material is based upon work supported by the National Science Foundation under the following grant: NSF 1719215. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] Open Compute Project (OCP). http://www.opencompute.org/.

[2] The CIFAR-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html.

[3] Wikipedia dump. https://dumps.wikimedia.org/.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

[5] S. R. Agrawal, S. Idicula, A. Raghavan, E. Vlachos, V. Govindaraju, V. Varadarajan, C. Balkesen, G. Giannikis, C. Roth, N. Agarwal, and E. Sedlar. A Many-core Architecture for In-memory Data Processing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*.

[6] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novaković, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (ATC '18)*.

[7] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*.

[8] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[9] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[10] Alibaba. Alibaba Production Cluster Trace Data. https://github.com/alibaba/clusterdata.

[11] Amazon. Amazon EC2 Elastic GPUs. https://aws.amazon.com/ec2/elastic-gpus/.

[12] Amazon. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

[13] Amazon. Amazon EC2 Root Device Volume. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/RootDeviceStorage.html#RootDeviceStorageConcepts.

[14] Y. Artsy, H.-Y. Chang, and R. Finkel. Interprocess communication in charlotte. *IEEE Software*, Jan 1987.

[15] K. Asanovi. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers, February 2014. Keynote talk at the 12th USENIX Conference on File and Storage Technologies (FAST '14).

[16] A. Barak and R. Wheeler. *MOSIX: An integrated unix for multiprocessor workstations*. International Computer Science Institute, 1988.

[17] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran. Popcorn: Bridging the programmability gap in heterogeneous-isa platforms. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*.

[18] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12), December 2007.

[19] F. Baskett, J. H. Howard, and J. T. Montague. Task Communication in DEMOS. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles (SOSP '77)*.

[20] A. Basu, M. D. Hill, and M. M. Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*.

[21] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. 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)*.

[22] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*.

[23] M. N. Bojnordi and E. Ipek. PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*.

[24] Cache Coherent Interconnect for Accelerators, 2018. https://www.ccixconsortium.com/.

[25] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment (VLDB '18)*.

[26] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-memory Multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*.

[27] D. Cheriton. The V Distributed System. *Commun. ACM*, 31(3), March 1988.

[28] I.-H. Chung, B. Abali, and P. Crumley. Towards a Composable Computer System. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia '18)*.

[29] Cisco, EMC, and Intel. The Performance Impact of NVMe and NVMe over Fabrics. http://www.snia.org/sites/default/files/NVMe_Webcast_Slides_Final.1.pdf.

[30] P. Costa. Towards rack-scale computing: Challenges and opportunities. In *First International Workshop on Rack-scale Computing (WRSC '14)*.

[31] P. Costa, H. Ballani, K. Razavi, and I. Kash. R2C2: A Network Stack for Rack-scale Computers. In *Proceedings of the ACM SIGCOMM 2015 Conference on SIGCOMM (SIGCOMM '15)*.

[32] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation (OSDI '04)*.

[33] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*.

[34] Dragojević, Aleksandar and Narayanan, Dushyanth and Hodson, Orion and Castro, Miguel. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*.

[35] A. Dunkels. Design and Implementation of the lwIP TCP/IP Stack. *Swedish Institute of Computer Science*, 2001.

[36] K. Elphinstone and G. Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*.

[37] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Princi-*

ples (SOSP '95).

[38] Facebook. Introducing Lightning: A flexible NVMe JBOF. https://code.fb.com/data-center-engineering/introducing-lightning-a-flexible-nvme-jbof/.

[39] Facebook. Wedge 100: More open and versatile than ever. https://code.fb.com/networking-traffic/wedge-100-more-open-and-versatile-than-ever/.

[40] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic. Beyond Processor-centric Operating Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS '15)*.

[41] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

[42] GenZ Consortium. http://genzconsortium.org/.

[43] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI '12)*.

[44] J. R. Goodman. Coherency for Multiprocessor Virtual Address Caches. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS '87)*.

[45] Google. Google Production Cluster Trace Data. https://github.com/google/cluster-data.

[46] Google. GPUs are now available for Google Compute Engine and Cloud Machine Learning. https://cloudplatform.googleblog.com/2017/02/GPUs-are-now-available-for-Google-Compute-Engine-and-Cloud-Machine-Learning.html.

[47] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. Shin. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*.

[48] Hewlett-Packard. The Machine: A New Kind of Computer. http://www.hpl.hp.com/research/systems-research/themachine/.

[49] Intel. Intel Non-Volatile Memory 3D XPoint. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html?wapkw=3d+xpoint.

[50] Intel. Intel Omni-Path Architecture. https://tinyurl.com/ya3x4ktd.

[51] Intel. Intel Rack Scale Architecture: Faster Service Delivery and Lower TCO. http://www.intel.com/content/www/us/en/architecture-and-technology/intel-rack-scale-architecture.html.

[52] Intel, 2018. https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/.

[53] JEDEC. HIGH BANDWIDTH MEMORY (HBM) DRAM JESD235A. https://www.jedec.org/standards-documents/docs/jesd235a.

[54] W. John, J. Halen, X. Cai, C. Fu, T. Holmberg, V. Katardjiev, M. Sedaghat, P. Sköldström, D. Turull, V. Yadhav, and J. Kempf. Making Cloud Easy: Design Considerations and First Components of a Distributed Operating System for Cloud. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '18)*.

[55] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. D. Mike Daley, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, H. K. Alexander Kaplan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, M. R. Jonathan Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, A. S. Dan Steinberg, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*.

[56] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE '16)*.

[57] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*.

[58] S. Kaxiras and A. Ros. A New Perspective for Efficient Virtual-cache Coherence. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*.

[59] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*.

[60] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*.

[61] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*.

[62] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown, A. Klimovic, A. Slowey, and A. Rowstron. Understanding Rack-Scale Disaggregated Storage. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17)*.

[63] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*.

[64] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level Implications of Disaggregated Memory. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*.

[65] D. Meisner, B. T. Gold, and T. F. Wenisch. The powernap server architecture. *ACM Trans. Comput. Syst.*, February 2011.

[66] Mellanox. ConnectX-6 Single/Dual-Port Adapter supporting 200Gb/s with VPI. http://www.mellanox.com/page/products_dyn?product_family=265&mtag=connectx_6_vpi_card.

[67] Mellanox. Mellanox BuleField SmartNIC. http://www.mellanox.com/page/products_dyn?product_family=275&mtag=bluefield_smart_nic.

[68] Mellanox. Mellanox Innova Adapters. http://www.mellanox.com/page/programmable_network_adapters?mtag=&mtag=programmable_adapter_cards.

[69] Mellanox. Rdma aware networks programming user manual. http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[70] J. Mickens, E. B. Nightingale, J. Elson, D. Gehring,

B. Fan, A. Kadav, V. Chidambaram, O. Khan, and K. Nareddy. Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*.

[71] D. Minturn. NVM Express Over Fabrics. 11th Annual OpenFabrics International OFS Developers' Workshop, March 2015.

[72] T. P. Morgan. Enterprises Get On The Xeon Phi Roadmap. https://www.enterprisetech.com/2014/11/17/enterprises-get-xeon-phi-roadmap/.

[73] M. Nanavati, J. Wires, and A. Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*.

[74] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-Tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC '15)*.

[75] Netronome. Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/.

[76] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*.

[77] V. Nitu, B. Teabe, A. Tchana, C. Isci, and D. Hagimont. Welcome to Zombieland: Practical and Energy-efficient Memory Disaggregation in a Datacenter. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*.

[78] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*.

[79] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*.

[80] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*.

[81] Open Coherent Accelerator Processor Interface, 2018. https://opencapi.org/.

[82] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite Network Operating System. *Computer*, 21(2), February 1988.

[83] J. Ouyang, B. Kocoloski, J. R. Lange, and K. Pedretti. Achieving Performance Isolation with Lightweight Co-Kernels. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*.

[84] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*.

[85] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA '07)*.

[86] R. F. Rashid and G. G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles (SOSP '81)*.

[87] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*.

[88] S. M. Rumble. Infiniband Verbs Performance. https://ramcloud.atlassian.net/wiki/display/RAM/Infiniband+Verbs+Performance.

[89] R. Sandberg. The Design and Implementation of the Sun Network File System. In *Proceedings of the 1985 USENIX Summer Technical Conference*, 1985.

[90] B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*.

[91] M. Schwarzkopf, M. P. Grosvenor, and S. Hand. New Wine in Old Skins: The Case for Distributed Operating Systems in the Data Center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSys '13)*.

[92] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A User-Programmable SSD. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*.

[93] Y. Shan, S.-Y. Tsai, and Y. Zhang. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*.

[94] M. Silberstein. Accelerators in data centers: the systems perspective. https://www.sigarch.org/accelerators-in-data-centers-the-systems-perspective/.

[95] A. J. Smith. Cache Memories. *ACM Comput. Surv.*, 14(3), September 1982.

[96] A. S. Tanenbaum, M. F. Kaashoek, R. Van Renesse, and H. E. Bal. The Amoeba Distributed Operating System&Mdash;a Status Report. *Comput. Commun.*, 14(6), August 1991.

[97] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, and S. J. Mullender. Experiences with the Amoeba Distributed Operating System. *Commun. ACM*, 33(12), December 1990.

[98] E. Technologies. NVMe Storage Accelerator Series. https://www.everspin.com/nvme-storage-accelerator-series.

[99] S. Thomas, G. M. Voelker, and G. Porter. CacheCloud: Towards Speed-of-light Datacenter Communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '18)*.

[100] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter. A Study of Modern Linux API Usage and Compatibility: What to Support when You'Re Supporting. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*.

[101] S.-Y. Tsai and Y. Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[102] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys '15)*.

[103] VMware. Virtual SAN. https://www.vmware.com/products/vsan.html.

[104] W. H. Wang, J.-L. Baer, and H. M. Levy. Organization and Performance of a Two-level Virtual-real Cache Hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA '89)*.

[105] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing Storage for a Million Machines. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems (HotOS '05)*.

[106] D. Wentzlaff, C. Gruenwald, III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and

A. Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*.

[107] G. Zellweger, S. Gerber, K. Kourtis, and T. Roscoe. Decoupling cores, kernels, and operating systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*.

[108] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, M. Chintalapati, A. Krishnamurthy, and T. Anderson. Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*.