Slashing the Disaggregation Tax in Heterogeneous Data Centers with FractOS

Lluís Vilanova [1] Lina Maudlej [2] Shai Bergman [2] Till Miemietz [3] Matthias Hille [4]
Nils Asmussen [3] Michael Roitzsch [3] Hermann Härtig [4] Mark Silberstein [2]
[1] Imperial College London [2] Technion - Israel Institute of Technology [3] Barkhausen Institut [4] TU Dresden

Abstract

Disaggregated heterogeneous data centers promise higher efficiency, lower total costs of ownership, and more flexibility for data-center operators. However, current software stacks can levy a high tax on application performance. Applications and OSes are designed for systems where local PCIe-connected devices are centrally managed by CPUs, but this centralization introduces unnecessary messages through the shared data-center network in a disaggregated system.

We present FractOS, a distributed OS that is designed to minimize the network overheads of disaggregation in heterogeneous data centers. FractOS elevates devices to be first-class citizens, enabling direct peer-to-peer data transfers and task invocations among them, without centralized application and OS control. FractOS achieves this through: (1) new abstractions to express distributed applications across services and disaggregated devices, (2) new mechanisms that enable devices to securely interact with each other and other data-center services, (3) a distributed and isolated OS layer that implements these abstractions and mechanisms, and can run on host CPUs and SmartNICs.

Our prototype shows that FractOS accelerates real-world heterogeneous applications by 47%, while reducing their network traffic by 3×.

CCS Concepts: • **Information systems** → **Data centers**; • **Security and privacy** → *Operating systems security*; *Distributed systems security*; • **Software and its engineering** → **Operating systems**.

Keywords: distributed systems, operating systems, data center, resource disaggregation, capabilities

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04…$15.00
https://doi.org/10.1145/3492321.3519569

ACM Reference Format:
Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig and Mark Silberstein. 2022. Slashing the Disaggregation Tax in Heterogeneous Data Centers with FractOS. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3492321.3519569

1 Introduction

In pursuit of higher efficiency and lower total cost of ownership (TCO), *disaggregated heterogeneous data centers* deploy various device types, such as CPUs, accelerators, storage and memory, into separate nodes interconnected over the network. Disaggregation thereby facilitates specialization together with maintenance, allocation and sharing of hardware resources [3, 10, 12, 16, 20, 23, 26, 29, 37, 40, 43, 49].

Unfortunately, applications today pay a high performance tax because the combination of heterogeneity and disaggregation introduces redundant communication over the data-center network. Frequent data and control transfers across multiple compute and storage devices are inherent to heterogeneous workloads. However, whereas in a traditional server architecture such transfers are performed over a fast dedicated local PCIe bus, in a disaggregated environment they are instead performed over a shared network with higher latency and increased performance variability (e.g., 1μs for PCIe [11] vs. average 24μs and P99 of 40μs for RoCE RDMA in Microsoft Azure [17]). While reducing inter-device communication overheads has been important even in heterogeneous servers [53], in disaggregated systems it becomes key to attaining application performance goals.

Over the years, several system architectures have been developed to support resource disaggregation, but they are largely oblivious to the changing tradeoffs associated with the transition of the device interconnect from a local PCIe bus to a shared network. We summarize them in Figure 1.

One prevalent approach (top-left) is to treat remote devices as local ones, forwarding device commands over the network. Thus, CPU servers run unmodified applications, while runtimes, drivers, or hardware expose *remote* devices via familiar *local* interfaces. Such a *centralized design* hides the distributed nature of disaggregation, preventing the optimizations that eliminate the overheads of remote device

OS \ Application	Centralized	Distributed
Centralized	PCIeOF [44, 45], Gen-Z [14], rCUDA [10], AvA [57]	(within single app.) Beam [13], Dryad [21]
Distributed	LegoOS [49], NAS, NVMeOF [40]	**FractOS**

Figure 1. Application and OS designs in the data center.

access, analogous to the known overheads of transparent distributed shared memory systems [38].

Another approach (bottom-left) is to provide a high-level RPC interface that co-locates devices with their drivers and scheduling logic (e.g., monitors in LegoOS [49]). This approach reduces the management overheads of remote devices, but keeps centralized application control. For example, in a processing pipeline where data goes first through a GPU and then an FPGA, the GPU cannot directly invoke the FPGA task without first returning control and data to the application server. This is because neither the OS nor the applications have *mechanisms to express and optimize cross-device interactions* — as a consequence, the GPU node cannot operate the FPGA directly, as it lacks the mechanisms to allocate FPGA memory or invoke its computations.

More fundamentally, both approaches suffer from a structural problem: while the system architecture became distributed, the applications remain *centralized*, and must mediate the interactions between disaggregated devices.

On the other hand, frameworks such as Apache Beam [13] (top-right) are used to develop distributed systems, but are not suitable for running heterogeneous multi-device applications. They work only across CPUs because other devices do not support the complex high-level interfaces of such frameworks. One can physically co-locate CPUs with each disaggregated device to execute such application logic, but that would defeat the purpose of resource disaggregation.

What we need is an infrastructure to enable *decentralized execution of the application logic over disaggregated heterogeneous resources*, which must allow *direct, peer-to-peer data and control transfers* among devices and services, thereby minimizing the networking costs of disaggregation.

In this paper, we realize this vision with **FractOS**, a distributed OS for disaggregated heterogeneous data centers. FractOS treats all compute and storage devices as first-class citizens like traditional CPUs, allowing them to interact directly among themselves, thus enabling fully decentralized application execution with minimal networking overheads.

At a high level, FractOS resembles distributed object systems such as CORBA [42] or Java RMI [41]. A data center is comprised of *services* accessed via RPCs. However, in addition to regular services on CPUs, data-center devices are also exposed as services via their device-specific RPC interfaces (e.g., kernel invocation for GPUs). RPC arguments may include *globally-accessible* references to memory buffers or RPCs to other services, enabling location-independent data accesses and service invocations. For example, an application can program a GPU to directly write to disk: it calls a GPU kernel invocation RPC with a reference to the `write()` storage RPC, preset with a reference to a GPU memory buffer as a data source. Thus, an application can create a graph of data transfers and service invocations, and FractOS executes it in a decentralized manner on devices and CPUs.

Building such a system poses two key challenges. First, each device should be able to use FractOS APIs, know which task to invoke next without centralized application control, and handle exceptions during the execution. Running such a logic on each device might not be possible due to hardware constraints, such as the lack of privilege separation on GPUs, or the inability to deploy user code on SSDs.

Second, allowing peer-to-peer interactions between devices without the mediation of a trusted OS is insecure. For example, references to a shared storage should not be used for unauthorized data accesses. Running a security layer on each device is not viable due to the limitations above, whereas using a central security controller would not scale.

FractOS offers systematic solutions to these challenges.

Isolated OS layer. In FractOS, *Controllers* build a distributed OS layer by implementing all trusted mechanisms for RPC, address translation, and message routing. Controllers run on CPUs or SmartNICs, possibly on top of their local OSes, and are isolated from services and applications via a message-passing interface. Each service and memory object is registered with a single Controller, either local or remote. An application requests access to FractOS objects via a resource management service. Each device is associated with an *adaptor* which implements the device's RPCs. The adaptor manages externally-accessible buffers in device memory and transforms RPCs into device operations. For example, the GPU adaptor invokes kernels, akin to a "monitor" in LegoOS [49] and an "ASM" in M^{3}X [4]. Adaptors are generic and lightweight, hence they are executed on (potentially wimpy) CPUs that run OS device drivers and are co-located with the devices, or, in the future, on a device itself. Adaptors are *untrusted* by FractOS, as any other software services in the system. This micro-kernel design makes FractOS compatible with existing devices that cannot run adaptors, provides strong isolation of the trusted FractOS Controllers, and allows offloading Controllers into SmartNICs (Section 6).

Continuation-based RPCs. To allow decentralized execution, we use a continuation-based protocol that enables adaptors to execute control flow defined by applications without themselves running application code. Each RPC call includes one or more references to other RPCs. Each reference is, effectively, a continuation object that we call *Request*. A Request encapsulates the parameters needed for the verbatim invocation of the RPC it refers to via a generic `request_invoke()`

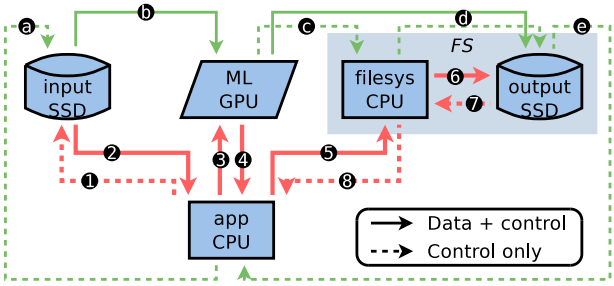

Figure 2. Cloud inference application using disaggregated devices. A centralized design (red thick lines) has 2.5× and 1.6× more data transfers and overall network messages respectively, than the distributed design (green, thin lines).

function. Thus, adaptors do not need to know *how* to invoke the next RPC, only *which* Request to invoke, as defined by their own RPC interfaces. Such an approach enables a variety of distributed execution patterns, from synchronous RPCs to complex data-flow models [52].

Distributed capabilities. FractOS protects memory and request references by using capabilities [31], which are tightly integrated with the RPC mechanism and largely transparent to application developers. An RPC caller delegates the authority to its callee to access the references in the RPC arguments, and can later revoke that authority. For example, if an RPC argument is a memory reference to store the output, the access to that memory should be revoked after use. We design a scalable and fault-tolerant protocol for capability delegation and revocation, which in contrast to previous works [18, 25, 50], combines efficient delegation with immediate selective revocation in a distributed setting.

We implement a complete FractOS prototype and use it to build fully functional services for disaggregated GPUs and NVMe SSDs, as well as a multi-layer file-system service. We also implement a realistic heterogeneous face-verification application that uses all these services.

We comprehensively evaluate FractOS on a multi-node cluster in a variety of configurations, with FractOS Controllers running on both CPUs and Mellanox BlueField SmartNICs. FractOS enables 47% faster end-to-end application execution while reducing network traffic by 3× compared to existing disaggregation designs.

2 Motivation

In this section we highlight the structural issues of current disaggregation solutions, and motivate our FractOS design.

2.1 Motivating scenario

Figure 2 shows an inference service used in production clouds [19]. For each client request, the service reads its input from storage, processes it on a GPU-based inference engine, and writes the output to a file on a file server. We consider a disaggregated architecture with remote SSDs and GPUs. Note that the FS service also uses remote SSDs.

The figure shows both the centralized application model that we find today (steps ❶-❽ in red), and the fully distributed model we seek to achieve with FractOS (steps ⓐ-ⓔ in green). In the former, the application (app CPU) performs all the control and data transfers, acting as the center of a star-shaped topology. In the latter, the application defines how each device must be used in turn, but devices invoke the application tasks on each other directly, forming a ring topology that is *optimal* for this application: it has 2.5× fewer data transfers (thick green lines vs. red lines), and requires 1.6× fewer network messages overall.

Analysis. The *message-complexity* savings of a distributed model with peer-to-peer communications over a centralized model are apparent. For an application with N services, the distributed model reduces the number of steady-state network messages by up to 2× (from $2N$ to $N+1$), while avoiding potential communication bottlenecks at the central node.

However, in a system where services themselves are built on top of other remote services (e.g., the FS service in the example uses remote SSDs), the total message complexity improvements are much higher. In general, we can represent a system that uses a centralized model as a tree of services with the main application as its root, where the edges are inter-service interactions. The highest reduction in the number of messages is achieved if the services represented by the leaves perform the application tasks, whereas the intermediate nodes are used for control operations and can be bypassed in steady-state. In this case the total message-complexity reduction of the distributed model is as high as $2 \times \frac{N}{L}$ where N is the total number of nodes in the tree, and L is the number of leaves. However, this upper bound on the message-complexity reduction cannot be achieved if only the application is distributed, but the services themselves are not. In addition, to realize this potential, the system should allow optimizations to cut through service boundaries: note the direct interaction between ML GPU, the output SSD and the application in the example.

2.2 System requirements

Interface encapsulation for cross-service invocations. In the centralized model, devices are managed by OS drivers running on the CPU. The application thus accesses devices via convenient interfaces, whereas each device does not need to know how to access other devices. In the decentralized architecture, this is no longer true. In Figure 2, ⓑ involves direct data and control transfer from the SSD to the GPU, implying that the SSD must run a GPU driver. Similarly, the GPU must be able to use the FS server and the SSD (ⓒ and ⓓ), and the SSD should invoke a CPU service (ⓔ). Direct data transfers between devices such as GPUDirect-RDMA [39] are supported only in a single server, and must be invoked by a co-located CPU. Direct device control is sometimes possible, e.g., GPU controlling the NIC [9], but

would require porting all device drivers to all devices – an impractical proposition. Similarly, a device cannot invoke high-level services, e.g., a file system, without running the appropriate software [54].

Takeaway: we need a unified and lightweight mechanism to enable device and service invocation without knowing their low-level management protocols. It may run on the device itself, or on a wimpy co-located CPU that executes the device driver.

Application-agnostic decentralized flow. Our target model implies decentralized execution: an application invokes the SSD service (a), which invokes the GPU task (b), which invokes the FS service (c), which passes the control back to the application (e). However, deploying application tasks on devices or services is not a viable option: some devices may not allow executing user code (e.g., SSDs), and co-locating devices with CPUs to run application tasks goes against disaggregation. Fortunately, we can enable decentralized execution with a much simpler *mechanism to steer control and data flow*. For example, the SSD can invoke the GPU kernel after completing the read operation (i.e., reading data into the GPU memory). Similarly, the GPU can invoke the file system service after the kernel terminates successfully. If the invocation resulted in an application error, a GPU can invoke a different service. Thus, provided that each device knows which task to invoke next (as defined by the device's interface), we can execute applications in a fully decentralized manner.

Takeaway: we need a mechanism to specify a task graph across services and devices, and a distributed mechanism to allow decentralized execution of this graph over the participating devices, without deploying application code on them.

Composability of services and applications. Decentralized execution of applications is insufficient, since services can themselves use multiple other services and devices. Take the FS service in our example. The filesys CPU task is involved in all data transfers between the clients and the output SSD (6 and 7). In a model where the execution is fully decentralized across all services, we would transfer the data from the client (GPU) into the output SSD (d), and respond directly to the application CPU (e). More fundamentally, the decentralized model should effectively merge, or *compose*, the task graphs of the FS service and the application, thereby enabling direct interaction between the application tasks (ML GPU and app CPU) and the internal FS tasks. Note that composition relies on the interface encapsulation discussed above: the output SSD from the FS service directly invokes an application task on the CPU, but since the details of that task are *unknown* when designing the FS service, the invocation task interface must be unified.

Takeaway: we need an execution mechanism to transparently compose application and service task graphs without breaking service encapsulation.

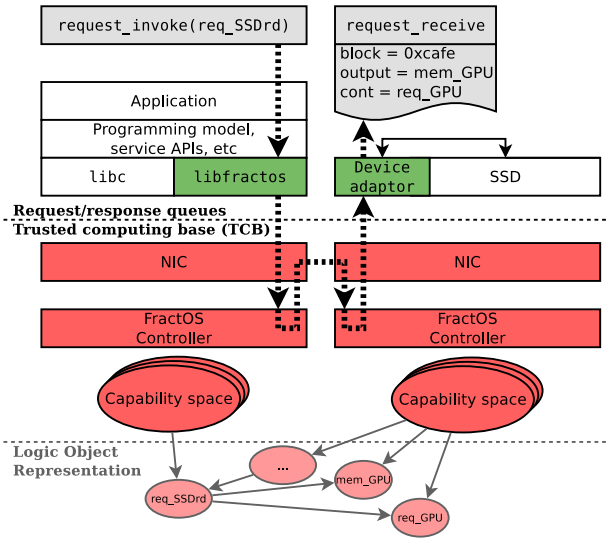

Figure 3. Architecture of FractOS on a system with a CPU (left) and a SSD node (right), and the logic representation of the corresponding capability-authorized objects (bottom).

Decentralized protection and security. Devices and services are shared among different tenants in the data center, so access to them must be protected to maintain user isolation. For example, the SSD must be temporarily granted access to a GPU memory buffer in b, which is only accessible to the main application. The access to the GPU memory buffer should be revoked right after the SSD operation is complete to enable its reuse by another client. The OS handles this authorization in the centralized model, but providing it in the decentralized model without using a single global authority is challenging. The problem is further complicated with service composability: for the ML GPU to write to the output SSD (d), the FS must grant the GPU access to specific SSD blocks. This implies that the FS serves as an additional authority limited to the disks it controls.

Takeaway: we need a decentralized mechanism to dynamically grant and revoke authority to access services and devices.

3 Design

3.1 Overview

Programming abstractions. FractOS provides two simple abstractions, *Memory* and *Request* objects, that encapsulate the details necessary to perform data transfers and send control messages between devices, respectively. Furthermore, Requests can be combined into task graphs to allow dynamically composing functionality that will execute in a distributed fashion across services and devices.

Deployment and operation. *FractOS Controllers* are deployed by the operator and run on either CPUs or SmartNICs as user-level Linux processes. Each Controller is part of the FractOS trusted computing base (TCB).

FractOS Processes (both user applications and adaptors) are also user-level Linux processes. They run on host CPUs,

and access the co-located devices via existing drivers and runtimes. FractOS does not distinguish between adaptors that expose hardware devices and regular CPU services. Both services and user applications are *FractOS Processes*. Each Process is connected to a single Controller via request/response queues, locally or via the network (Figure 3, top half). A Process can invoke FractOS operations, i.e., Request invocations and Memory object accesses, using FractOS APIs. These operations are then forwarded by the Controller to their respective destinations.

To illustrate, Figure 3 zooms into one aspect of the inference example from Figure 2, showing two FractOS Processes: a CPU application (top-left) and a SSD adaptor (top-right). The application has a `req_SSDrd` Request, which references `mem_GPU` as the GPU buffer (shown at the receiver on the top-right), and a `req_GPU` Request to invoke the GPU. When `request_invoke(req_SSDrd)` is called, its arguments are sent to the SSD adaptor, which performs the requested read operation from the SSD, and then copies the read data to `mem_GPU` and invokes `req_GPU`, without being aware that they refer to a GPU. Only the Process setting up the arguments for `req_GPU` needs to know the GPU interface, but the SSD adaptor invokes it verbatim.

Protection and isolation. FractOS has only a handful of system calls for manipulating Memory and Request objects (Table 1). Processes are isolated from each other and from FractOS Controllers, and are constrained to their own set of *capabilities*, each referencing Requests or Memory (bottom of Figure 3). The underlying mechanism for capabilities is similar to file descriptors in POSIX – the references behind the capabilities are protected by FractOS, and Processes access them via indices in their capability space (*cid* in Table 1).

3.2 Target system model

Trust model. FractOS has a layered security model, similar to other micro-kernel OSes; its TCB consists of a (pre-deployed) set of trusted Controller instances, a trusted service that FractOS Controllers use for bootstrapping and discovery, and the hardware where all these components run. We assume a single data center where FractOS Controllers are pre-deployed. Services and device adaptors are deployed by either the data center operator or its tenants, and run as FractOS Processes that are untrusted by FractOS itself but trusted by other Processes, similar to a multiserver micro-kernel architecture. For example, operator services such as resource management rely on FractOS to run, and are trusted by both tenants and device adaptors. Similarly, tenants trust the device adaptors and other tenant services that they use.

Capabilities *authorize* access to other Processes, but do not imply symmetric trust. For example, an operator's resource-management service might hold capabilities to access all resources in the system. However, if a malicious or buggy device adaptor is managed by that service, such an adaptor will not have the ability to compromise the management service or the resources it controls. Similarly, a tenant application will only be able to interact with resources it has been granted capabilities for. Naturally, the fact that one tenant uses a third-party service or device adaptor implies that this tenant trusts such third parties to fulfill their job.

We note that FractOS does not dictate a concrete security policy, but rather provides the basic mechanism to implement one with the help of capabilities.

Data-center architecture requirements. We designed FractOS to minimal requirements on the data center architecture, while envisioning a future where devices, services, and applications are tightly integrated with FractOS (see Section 7).

We run FractOS Controllers on SmartNICs. The Controllers are regular Linux processes that are developed and deployed using existing tools such as Kubernetes, and have access to all the necessary drivers and runtimes. Controllers expose a network-based request/response protocol to FractOS Processes. As a result, Controllers are fully decoupled and isolated from the Processes they manage, and can be transparently deployed on SmartNICs or traditional host CPUs (either local or remote to the Processes they manage).

As a result of this decoupling, services and device adaptor Processes are essentially indistinguishable to FractOS, and follow a model very similar to that of micro-services.

3.3 Memory and Request Objects

Memory and Request objects reside in a virtually global FractOS namespace that is effectively distributed across the many Controllers. We use the term *capability* to denote FractOS references to such objects, which are protected by the trusted FractOS Controllers and encapsulate the address of the target objects (similar to POSIX file descriptors).

Memory objects enable access to memory buffers anywhere in the system. A buffer is registered by a Process that owns the physical memory via `memory_create()`. The data can be copied between Memory objects via `memory_copy()`, regardless of the location of the buffers (i.e., third-party transfers). One can create "views" into a portion of a Memory object or reduce permissions using `memory_diminish()`; for example, the GPU in Figure 2 can send a read-only Memory capability to its memory to the `write()` storage RPC.

Request objects effectively represent service RPC endpoints. To allow access to a service, the Process that implements it (the provider Process) first creates a Request that represents the RPC to that service. The clients that obtain a reference to this Request are effectively granted the authority to invoke that RPC on the provider Process. In addition, a Request may hold arbitrary arguments, i.e., immediate values and capabilities. This allows the service provider to embed the specific details of the RPC invocation in the Request itself, thus enabling verbatim invocation of the service via `request_invoke()`. As a result, a caller does not need

FractOS primitive	*Brief description*
`cid cap_create_revtree(cid)`	Create a new revocation subtree for the given capability.
`cap_revoke(cid)`	Revoke capability.
`cid memory_create(addr, size, perms)`	Create a Memory capability to given local memory.
`cid memory_diminish(cid, offset, size, drop_perms)`	Diminish extents and/or permissions of a Memory capability.
`memory_copy(cid1, cid2)`	Copy all bytes from Memory `cid1` into `cid2`.
`cid request_create([cid], [(offset, size, addr), ...] [(cid_src, cid_dst), ...])`	Create a new Request or extend an existing Request (cid), with additional immediate and capability arguments.
`request_invoke(cid)`	Invoke the given Request.
`request_receive{imms_size, imms[], caps_size, caps[]}`	Descriptor for a received Request.

Table 1. Main syscalls in FractOS. All syscalls are fully asynchronous and posted into a message-passing channel, which later returns their result when polling the channel (not shown for brevity.)

to know the service invocation protocol. This is essential to enable third-party invocations of services as we discuss in section 3.4. The service provider uses `request_receive()` call to listen to Requests to its services.

Requests are created using `request_create()`. There are two types of Requests: (1) a new Request that is initialized to point to the calling Process as the service provider, or (2) a Request *derived* from an existing Request that points to the same service provider. A newly created Request is initialized with new arguments. A derived Request inherits the arguments from the original Request, but also allows *refining* it by adding new arguments. This feature is used for request composition as we discuss next.

3.4 Request Composition

Requests serve as the basis for composing application tasks and delegating authority; the example in Figure 3 shows how the SSD receives the `req_GPU` capability, which grants the SSD the authority to invoke a GPU kernel.

We now describe the three key properties of Requests.

Security. In Figure 3, the Request contains a block number to read from; this block number should not be modifiable by any Process that holds the Request after creation. For that reason, the Request arguments that have already been initialized cannot be changed, but Processes can *refine* Requests by adding new arguments. For example, imagine that the SSD grants a base Request `req_SSDrd_base` to the application that allows reading a single block passed in its immediate argument (`0xcafe`). The application has access to that block alone. But it can refine `req_SSDrd_base` by deriving a new Request `req_SSDrd`, in which it can set a reference to the output buffer `mem_GPU` and the continuation argument `req_GPU`. We use this mechanism to implement the storage stack (Section 5).

Distributed execution. Requests form a *generic mechanism for distributed execution* that can express a variety of distributed execution models, such as RPCs, distributed pipelines, or distributed fork/join and data-flow patterns [52]. Consider a service RPC A that expects a reference to a Request B as its argument. A invokes B after completion, i.e., B is a continuation of A. We can use this primitive to implement a synchronous RPC for A as follows. A client Process that invokes A can initialize B to contain a separate Request A' implemented by A itself (i.e., we are expressing $A \rightarrow B \rightarrow A'$). Thus, B is used to notify A of B's completion via A'. The same mechanism can be used to implement a pipeline of services, which return the results back to the client, as in the example in Figure 2. Here, the application obtains Memory and Request objects for the SSD, GPU and file system, and connects them by initializing the respective Requests to form the graph shown by the green lines in Figure 2.

Whereas the application must be familiar with the interface used by each of the services (and set Request arguments accordingly), the services do not know about each other and simply invoke the received Requests as defined by their own interfaces. As a result, the execution progresses in a decentralized fashion across devices as set by the application via the Requests and their arguments. This is, in fact, a distributed form of the continuation-passing style (CPS) model [8] that is adapted to the data center.

Dynamic Composition. The file system in Figure 2 presents one interesting case where composition happens dynamically. The output SSD is not accessible to the application, which only has capabilities to the input SSD, GPU, and FS Processes. Therefore, the application can only set up a Request graph that expresses a pipeline across the services it uses: ⓐ→ⓑ→ⓒ→ⓔ. On the other hand, we want both the application and the FS to be co-optimized by allowing the application to access the output storage device, thus cutting through the FS-service module boundaries. This should work even if the FS service is developed by a third party and is arbitrarily complex.

To achieve this, we leverage the fact that all Requests can be refined with new arguments. The FS Process will take the received input Memory buffer and continuation Request, and refine its own output SSD Request to incorporate these arguments when invoking SSD in ⓓ. The result is that the output SSD, hidden from the application, reads its input data from the GPU and invokes continuations in the application

Process. This is a pattern that can be applied recursively across Processes, giving FractOS a *general mechanism for dynamically composing services without breaking their encapsulation and isolation.*

3.5 Capability Model

FractOS transparently integrates capabilities in all its operations, using them mostly under the hood without developer involvement. Capabilities are created via `memory_create()`, `request_create()`, derived via `memory_diminish()`, `request_create()` (with an existing Request), delegated during `request_invoke()`, and revoked via `cap_revoke()`. Similar functionality is found in existing capability OSs [1, 5, 6, 18, 25, 27, 50].

Each capability in a Process points to a single FractOS object, i.e., a Request and Memory object. *Delegation* simply creates a new capability in a delegatee Process that points to the same object as the delegated capability. The delegatee Process may be managed by a different FractOS Controller, leading to a distributed capability system. Internally, a capability holds the address of the Controller it is registered with, and the respective object ID. *Creation* adds a new object in the system, whereas *derivation* creates a new object from an existing one; a derived Memory object contains the same or lesser permissions / smaller address range, whereas a derived Request object contains the same or additional arguments as its source. Both creation and derivation create a new capability for the new object.

Besides the ability to create and derive capabilities, a *selective and immediate revocation* is critical. Imagine that our input SSD in Figure 2 delegated Requests, each granting access to different SSD blocks. If at some point a user wants to free one of their blocks, the SSD Process must *selectively* revoke all capabilities granting access to the freed block, and must do so *as fast as possible* so that the block can be reused. Failing to do so would violate protection (i.e., use-after-free attack) or cause resource leakage in the system. Therefore, *revocation* must invalidate all the capabilities that point to the revoked object.

Optimized Delegation and Revocation. Whereas maintaining capability trees is effective in existing capability OSs, it is not feasible in a distributed system like FractOS, as it would require tracking every delegation using a fault-tolerant protocol, or performing a system-wide sweep of all capabilities on every revocation (similar to distributed garbage collection). Instead, FractOS introduces a *distributed capability management protocol* that is specifically designed for disaggregated data centers in order to eliminate capability-related network overheads in the critical path.

Our approach is based on the observation that the objects referenced by capabilities can only be used by contacting the owner of the object – the Controller with which it is registered. Therefore, capabilities to that object can be revoked immediately and globally by invalidating that object at the owner. This in turn obviates the need to perform expensive tracking of delegations [18]. Thus, we split the revocation in two steps: the object invalidation at its owner, and a cleanup step to remove capabilities that reference invalidated objects.

Since delegations are not tracked, capabilities cannot be organized as a tree like in other capability systems. Without further measures, however, this implies that revoking any capability would also revoke *all* other capabilities that reference the same object. To overcome this limitation, we added the ability to create a separately invalidatable object from an existing object, and organize these objects as a tree, similar to *indirection objects* [50]. The call `cap_create_revtree()` in FractOS creates a *revocation tree*: it derives a new (child) capability from an existing (parent) one, while creating a new object that is recorded as child of the parent capability's object. This "child" capability can be delegated to other Processes and later revoked independently from the capabilities referencing the "parent" object. These objects are organized as a tree, and revoking any capability invalidates the object it references as well as all its children objects, recursively. This solution allows us to avoid delegation tracking, and we found it to be acceptable because typically only resource owners implementing a service create new entries on a revocation tree. Thus, we replace the standard capability delegation trees with a much smaller hierarchy of individually-revocable objects; essentially an adaptation of Redell's caretaker pattern [31]. Note also that this design causes implicit revocation of objects whose owner Controller fails, because they cannot be accessed without that Controller being alive anyway. The implicit revocation can be used as a failure detection mechanism for a redundancy service built on top of the FractOS primitives (see Section 3.6).

Creating or revoking capabilities requires a single message to the owning Controller. To make revocation effective immediately, the Controller invalidates the selected object. At this point, the underlying resources held by the owning Process can be freed (if any). Only the small revoked objects remain in the Controller (a few bytes each), which are eventually cleaned up after ensuring no other Controllers have capabilities referencing it. This cleanup step happens outside the critical path and is neither security nor performance critical. It can be performed using existing distributed garbage collection algorithms [7, 15, 28, 51], which ensure that no capabilities remain at any Controller or in-flight.

3.6 Resource Management and Fault Tolerance

FractOS concerns itself with fault tolerance insofar a Process needs a mechanism to either: detect when a resource it is serving through one or more capabilities is no longer needed (e.g., a GPU service must free some physical resources after one of its clients fails); or to detect when a resource it is using is no longer available (e.g., a GPU service client must abort or request new GPU resources after a GPU it uses fails).

Since interactions are possible only through capabilities, FractOS provides two new capability monitoring primitives that serve to implement fault tolerance and resource management, by detecting when specific capabilities become invalid - either by explicit revocation or because of a failure. These operations piggyback on the existing capability-revocation machinery, minimizing their complexity.

Capability monitor operations. The monitor operations register a "callback" on a capability using `monitor_delegate` or `monitor_receive` (not shown in Table 1). This callback is a user-specified value that the Controller sends back to the user when that capability has no additional derived capabilities or when its parent capability is invalidated, respectively.

`monitor_delegate(cid, callback_id)` tells FractOS to monitor the destruction of all new immediate children of `cid`, and to send message `monitor_delegate_cb{callback_id}` back to the requesting Process when these children are invalidated[1]. Internally, FractOS makes `cid` revocable, marks it with the "monitor_delegator" flag, and creates a counter for its immediate children. Every time a "monitor_delegator" capability such as `cid` is delegated, the target capability is marked as revocable with the "monitor_delegatee" flag on the revocation tree. When a capability is revoked and has the "monitor_delegatee" flag, the corresponding counter is decreased and, in reaching zero, FractOS sends the corresponding `monitor_delegate_cb` message.

`monitor_receive(cid, callback_id)` tells FractOS to send `monitor_receive_cb{callback_id}` back to the requesting Process when `cid` is revoked. Internally, FractOS makes `cid` revocable and records the requesting Process and `callback_id` to send the appropriate message when revocation is detected.

Resource management model. By using `monitor_delegate`, a typical FractOS application such as the GPU service will keep track of its clients to free any resources that it held on their behalf. For example, the GPU service will create one Request capability for each client, call `monitor_delegate` on it, and then delegate that Request. If the client stops using the SSD service and revokes that capability, the service will notice it via `monitor_delegate_cb` and act accordingly.

Similarly, the GPU client can delegate such Request further to a third party, who can then use `monitor_receive` on the received Request to realize if the GPU client has revoked the access it gave to this third party.

Failure translation model. FractOS handles failures in a similar way to how it performs resource management, by translating failures into capability revocations. The underlying idea is that both resource and failure management need to react to the same type of capability-revocation events.

For example, the GPU service will detect the failure of a client Process via `monitor_delegate`, whereas the GPU client will detect a GPU-service failure via `monitor_receive`. Note that this also indirectly signals errors on in-flight operations; if the GPU client receives `monitor_receive_cb` while it has in-flight synchronous RPCs to that GPU service, it knows those RPCs have failed. The combination of `monitor_receive` and `monitor_delegate` also works in more complex and asynchronous cases, since a single monitor callback can result in a transitive cascade of additional monitor callbacks as Processes start freeing resources (and revoking capabilities). Importantly, in-flight Request cancellation and explicit resource deallocation must be handled by Processes themselves (such as cancellation tokens in C# gRPC or a GPU deallocation Request, respectively), and FractOS only provides the necessary basic building blocks.

FractOS handles four types of failures. A Process failure is detected by the owner Controller when their channel is severed, and causes a revocation of all the capabilities held by the failed Process (which in turn might trigger monitoring callbacks). A node or Controller failure is detected by an external monitoring service such as Zookeeper. After a node failure, we inform the corresponding Controller to fail all Processes running in it. After a Controller failure, we consider all its Processes failed and their capabilities revoked[2], and rely on the asynchronous capability-revocation cleanup described in section 3.5 to handle such revocations with all relevant Controllers. Note that we can handle network failures as node or Controller failures, accordingly, but do not explore how to handle transient network failures (we deploy FractOS within a single data center).

We want this failure-to-revocation protocol to happen asynchronously, while at the same time allowing us to immediately reintegrate failed nodes and Controllers into the system. To this end, we store a simple form of Lamport timestamps on capabilities, and eagerly detect Controller failure-triggered revocations when capabilities are used (recall that "normal" revocations are detected upon use via revocation trees). Each capability contains the Controller's address (which is unique) and a Controller reboot counter (which increases monotonically on every reboot). By comparing the reboot counter in the capability and the target Controller, we can immediately detect a stale capability and signal it as revoked.

4 Implementation

FractOS has 17.5K LoC of C++, not including user applications and device adaptors. The kernel is 6K LoC, and utility libraries used by Processes are 5k LoC. FractOS also contains other trusted services to implement authentication, node management, and a key/value store to bootstrap capabilities on new Processes (2.5K LoC, this would typically be replaced by a resource manager).

[1] For simplicity, `cid` cannot have children when calling `monitor_delegate`.

[2] FractOS does not yet handle forceful Process termination, and we instead rely on the host OS to kill it.

Processes are decoupled from their Controller via an RoCE queue pair, as well as Controllers between themselves – this is similar to FlexSC [55], but we take this decoupling further to explore various Controller deployment models (e.g., CPUs, SmartNICs, and remote nodes). The Controller deployments evaluated in section 6 use two cores per instance, using polling to reduce latency (a dynamic poll/interrupt model is the next step), consume 64 MB of RoCE buffers per managed Process, 64 MB for each other Controller it connects to (we are exploring shared receive queues to reduce memory costs), a set amount of memory for the capability space as set at Process creation time (can be capped via quotas), and 24 B per revocation tree object. To put these numbers into perspective, the SmartNIC we use in the evaluation has 16 GB of memory.

FractOS implements congestion control by limiting the number of outstanding FractOS responses in a Process, such that Processes can apply back-pressure to Request invocations from other Processes.

We pervasively use C++ promises and futures to develop asynchronous code, and build our own promise/future library to optimize per-thread concurrency (4.8K LoC). Together with minimizing uses of `shared_ptr`, these optimizations reduced the latency of FractOS operations by an order of magnitude (`shared_ptr` has a particularly large impact on SmartNIC deployments).

Limitations. RoCE has no third-party RDMA support, which would allow Controllers to perform direct Process-to-Process RDMA. Instead, Controllers can only copy data to/from Processes using intermediate RDMA bounce buffers in the Controller (increasing the cost of `memory_copy`).

For simplicity, the cleanup step of capability revocation (section 3.5) is based on a broadcast algorithm that invalidates all capabilities referencing a revoked object. We do not yet implement `monitor_*` callbacks.

We do not implement a resource allocation and scheduling layer. These layers are orthogonal to the key novel mechanisms in FractOS, and can be easily integrated into FractOS. We leave it for future work.

5 Applications and Services

Here we describe our implementation of an end-to-end application similar to that in Figure 2, as well as the constituting accelerator, file system, and storage services it uses.

Accelerator Service: GPU. We build a GPU adaptor to expose a disaggregated GPU. The GPU adaptor runs on the host CPU, using the OS GPU driver, and offers several RPCs exposed through Requests: GPU context initialization, memory de/allocation, kernel loading, kernel invocation, and cleanup.

An application obtains a capability to the GPU initialization Request. When invoked, the adaptor creates and returns the Request objects for memory allocation and kernel loading RPCs, which implicitly refer to the respective GPU context. A Process can use these Requests to create a Memory object pointing to a buffer in GPU memory, or load a kernel and obtain a Request for the specific kernel invocation. It can further delegate these Memory and kernel invocation Request to other Processes. Those can refine them further, but the GPU kernel, the Memory buffer and the context stay fixed.

To allow composition, the GPU-kernel invocation Requests expect two Request arguments used to signal success/error of the kernel invocation; all other immediate arguments are forwarded to the GPU kernel itself.

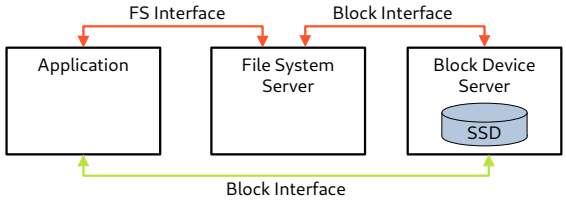

Figure 4. Overview of the FractOS storage stack. The red and green arrows denote the data flow for the normal file system and the DAX optimization, respectively.

Storage Stack: File System and Block Device. We implement a storage stack that is internally composed of independent Processes for the FS and a block-device adaptor for an NVMe SSD. Storage clients only see the capabilities returned by the FS Process. We use this multi-tier architecture to demonstrate composition across multiple layers.

The stack works in two different modes: FS and DAX (Figure 4). In the FS mode, we implement a simple FS layer that stores data from clients. It follows the centralized execution model, where all operations are mediated by the FS Process. The FS Process exposes Requests to open extent-based files [30, 46]. A successful completion returns Requests to read/write the file contents. Internally, the FS uses one logical volume in the block device for each file extent. The block-device adaptor exposes Requests that read/write the contents of logical volumes (managed through separate Requests), and delegates them to the FS Process. Thus, for each read/write Request, the FS copies the contents from/to the provided Memory capabilities and invokes and block-device Requests as synchronous RPCs.

In DAX mode (for "direct-access"), applications bypass the FS Process to directly read/write into a device after successfully opening a file in the FS. In this case, the FS returns to the clients the Requests delegated by the block device to access the corresponding volumes (one per file extent), after setting their read/write access permissions according to the file's open mode. This way, client applications can directly set an offset and size to access inside the extent, and the Request is served by the underlying block device.

Application: Face Verification. We develop an end-to-end application that composes the storage and GPU services in a way similar to the example in Figure 2. The application is a *face-verification* service used to verify the identity of a person

Host CPU	Intel Xeon E5-2620 v2
Host memory	64 GB, DDR3 @ 1333 MHz
SSD	Samsung 970evo Plus (500GB)
GPU	NVIDIA Tesla K80
Smart NIC	Mellanox BlueField MT416842, RoCEv2
Network	10 Gbps fabric and switch (split cables)

Table 2. Evaluation environment.

	Latency (usec)
Raw loopback w/ server @ CPU	2.42
Raw loopback w/ server @ sNIC	3.68
FractOS @ CPU	3.00 ± 0.02
FractOS @ sNIC	4.50 ± 0.13

Table 3. Latency of a null FractOS operation, compared to raw loopback latency. The serving side (ping-pong server or FractOS Controller) executes on either a CPU or sNIC.

by matching the photo and the ID in the input with the photo corresponding to that ID from a secure database [24].

Our service receives a batch of photos and their associated IDs. The images from the database are read from storage; the face matching algorithm runs on the GPU.

The application creates and builds a pipeline of Requests to (1) open and read the corresponding files from storage into the GPU (it uses a small pool of pre-allocated GPU memory buffers), (2) execute the face-verification GPU kernel, (3) copy the results from the GPU into the application memory, and (4) send a response to the client.

6 Evaluation

We seek to answer the following questions: (1) What is the performance of the core Memory and Request operations in FractOS? (2) What is the cost of deploying FractOS Controllers outside the node's local CPU? (3) Can FractOS efficiently manage accelerators and storage? (4) What are the benefits of distributed data transfer and control flow optimizations in higher-level services, such as an FS server? (5) What is the end-to-end application performance?

Experiments are performed on a 3-node cluster with the characteristics listed in Table 2. All measurements have a standard deviation below 3% of the mean, with 2σ confidence.

6.1 FractOS primitives

Null-operation latency. We first evaluate the latency of invoking a simple null syscall in FractOS. The results shown in Table 3 compare the latency seen by a CPU Process when deploying its Controller either on the same CPU package or a co-located SmartNIC (sNIC), and the lower-bound latency using the `ibv_rc_pingpong` benchmark.

The results indicate that FractOS's decoupled architecture is quite efficient and adds only 0.6 usec in CPUs. Running the controller on an sNIC adds 1.3 usec due to remote access via

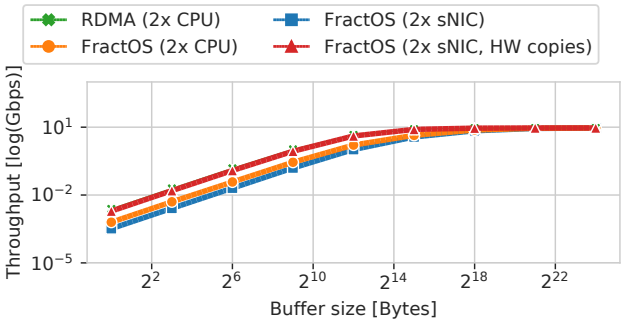

Figure 5. Throughput of a single data transfer across nodes (Higher is better).

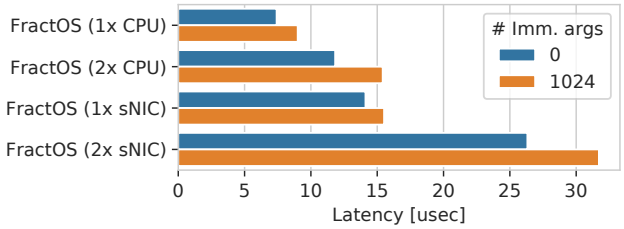

Figure 6. Latency of invoking a two-way Request (i.e., RPC) between Processes placed on one (1×) or two (2×) nodes (Shorter is better).

PCIe, and an additional 0.7 usec for slower Controller execution (the sNIC has an ARM core running at 800 Mhz, which is slow, in particular when performing atomic operations).

Memory copy. We compare `memory_copy()` and raw RDMA – the best possible baseline. Source and destination Memory objects are on separate nodes, and Controllers on either the node's CPU or sNIC.

Figure 5 shows the results. FractOS uses double buffering for buffers larger than 16 KB, achieving the full throughput at 256 KB. Smaller buffers under-perform due to intermediate bounce buffer copies – 1-Byte RDMA takes 3,3 usec, whereas Controllers take 12.7 usec and 24.5 usec on a CPU and sNIC, respectively. The red line ("HW copies") shows the throughput after replacing bounce buffers with 3rd-party RDMAs in the NIC (modeled as: null + raw RDMA copy - raw loopback latency). Concurrent copies (not shown for brevity) quickly saturate throughput at 4 KB and 32 KB for CPU and sNIC Controllers, respectively.

We conclude that FractOS is slower than RDMA, but would attain its latency and throughput even on a sNIC deployment, if it could use existing hardware acceleration on the NIC.

Request invocation. We evaluate the latency of an RPC using Request invocation (Processes exchange Requests ahead of time to avoid delegations; measured later). We compare invocation inside a node and across two nodes with different Controller placement and argument sizes.

The results in Figure 6 show that the CPU deployment adds 1.41 usec for Request handling both ways (including

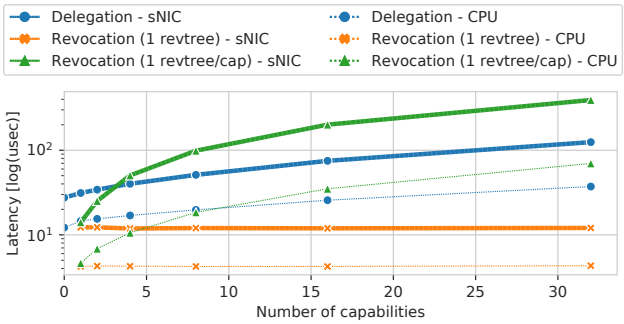

Figure 7. Latency of capability delegation and revocation (Lower is better).

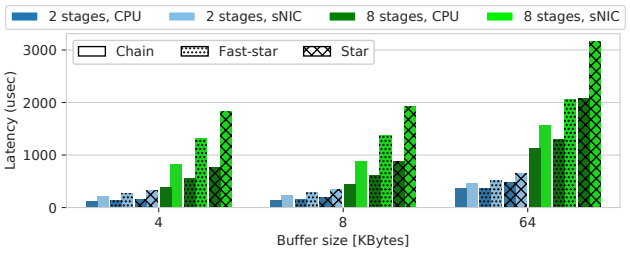

Figure 8. Request latency for processing pipelines with centralized application model (`star`), centralized control flow but distributed data flow (`fast-star`), and a fully distributed model (`chain`) (Lower is better).

user and FractOS code) and that (de)serializing Requests across the network adds additional 4.41 usec. The sNIC deployment adds 5.11 and 12.21 usec for Request handling and (de)serialization, respectively, whereas the cost of immediate arguments is in line with memory-copy throughput.

Our cross-node latency with FractOS on CPUs is in line with existing, latency-optimized RPC frameworks [22] (considering the 3 hops over a NIC each way, which hardware offload would eliminate), whereas the sNIC is less competitive (more than 30% of the time is spent on atomic `shared_ptr` operations related to capability and object lookups, which software engineering or hardware offload would eliminate).

Capability management. To evaluate the cost of common capability operations we measure the latency of an RPC with capability delegation (i.e., capability arguments), as well as capability revocation across two Processes on different nodes. For revocations, we explicitly revoke capabilities while comparing revocable (traditional) capabilities, (implemented by creating one revocation tree per capability) with the FractOS-optimized ones (which point to the same indirection object) that disallow individual revocations (one revocation tree for all). For the experiment we vary the number of capabilities on the revocation tree in the latter case.

The delegation results in Figure 7 show the RPC roundtrip, which indicates that (de)serializing a single capability during delegation takes about 2.4 usec and 3.8 usec for the CPU and sNIC deployments, respectively. For the revocation, the traditional capabilities (1 revtree/cap in the Figure) incur high linear overhead as we increase the number of capabilities, whereas the optimized ones incur a small cost as they all belong to the same revocation tree.

We conclude that our revocation optimizations are essential, and capability delegation has an acceptable cost that could be reduced through additional optimizations, e.g., by caching serialized Requests.

6.2 Service composition

Now we measure the performance gains of service composition in FractOS. We use a simple multi-stage pipeline where data is streamed across stages. Consecutive stages are deployed on different nodes. We compare the traditional centralized model (`star`), a centralized control with direct data transfers across stages (`fast-star`), and a fully distributed model with direct data transfer and direct control flow across stages (`chain`). In the context of the related works, these configurations cover the design space represented in Figure 1. The first one corresponds to the centralized designs (top left quadrant in Figure 1, e.g., rCUDA [10]), the second corresponds to the centralised application but distributed OS (e.g., LegoOS [49]), and the last one corresponds to fully distributed designs (e.g., Apache Beam [13]).

The results in Figure 8 confirm the cumulative gains of optimized service composition with FractOS. As we increase the memory-copy size or number of stages, optimizing data transfers is crucial for performance (`star` vs. `fast-star`: 1.6 x for 64 KB on CPU). Optimizing distributed control flow dominates when data transfers are of 4 KB or less (`fast-star` vs. `chain`: 1.45 x; `star` vs. `fast-star`: 1.4 x for 4 KB on CPU). Note that the workload is I/O bound, and thus shows the highest speedups due to reduced message complexity.

6.3 Accelerator service: GPU

We now compare a remote GPU service on FractOS with the same service implemented with the rCUDA generic GPU-remoting framework [10]. We run the face-verification kernel (section 5).

The left part of Figure 9, shows the latency of executing the GPU kernel on a single image, with a breakdown for data transfer and FractOS overheads (not for rCUDA as it is closed source). FractOS is substantially faster; rCUDA accesses remote GPUs transparently by interposing CUDA driver calls, whereas FractOS GPU service uses a single roundtrip Request invocation per kernel invocation. We can also see that our simple GPU-service prototype has relatively small FractOS-related overheads, and even the sNIC deployment of the FractOS Controller is still faster than rCUDA.

The right part of Figure 9, shows that FractOS achieves near-optimal throughput (on par with "Local GPU") with

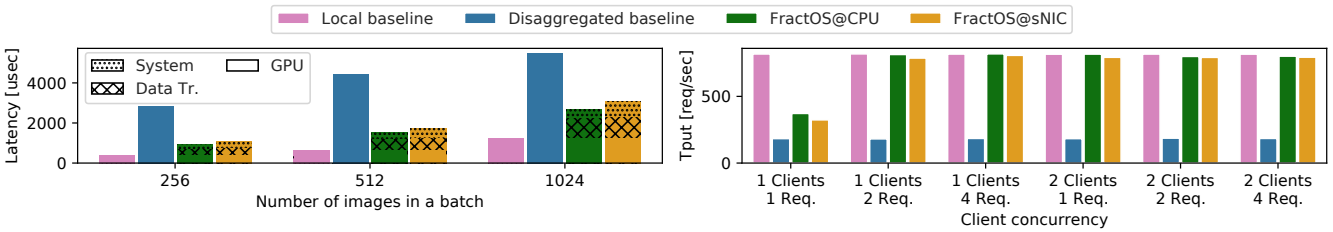

Figure 9. Left: Latency of executing the Face Verification kernel on a remote GPU vs. the image batch size (Lower is better); Right: Throughput with a fixed batch size of 1024 images vs. the number of clients and in-flight requests (Higher is better).

more than one in-flight request, even when running on sNICs. This result demonstrates the ability to serve multiple clients with FractOS.

6.4 Composition: storage stack

We now evaluate the storage stack described in Section 5. We compare the results of a traditional storage stack implemented with FractOS (`FS`), its DAX-style optimizations for data transfers (`DAX`), a baseline that uses the same FractOS FS service with a remote NVMe-oF device (`Disaggregated Baseline`), and a local block device (`Local Baseline`).

Figure 10 shows the latency for both random read and random write storage operations. The breakdown shows raw device times, data transfers, and FractOS software overheads. For latency, `FS` is competitive compared to the `Disaggregated Baseline` for random reads, as the Linux cache on the FS-service node is ineffective in this case (two network transfers per read). However the random writes are slower because the NVMe-oF device in `Disaggregated Baseline` absorbs writes through the cache, whereas we did not implement caching in `FS` for simplicity. More interestingly, the client-block service composition in `DAX` optimizes network data transfers by 2×. `DAX` speedup goes from 1.1× for 4 KB reads, where the NVMe latency dominates (70 usec), to 1.3× for larger sizes where network transfer makes up for most of the service time. Write operations in `Disaggregated Baseline` go through Linux' block cache, which makes its write latency almost as good as `DAX`. With sequential operations, `DAX` latency is equivalent to the `Disaggregated Baseline` due to its effective read-ahead caching (results not shown for brevity). When running FractOS on sNICs, the system overheads grows, leading to a higher overall latency. We believe that it is mostly caused by the lower clock rate of the ARM chips on the BlueField cards.

We also evaluate the throughput of storage operations with FractOS. For brevity, we only show a comparison between random and sequential read operations with a block size of 1024 KiB and four requests in flight (Figure 11). For this configuration, the storage stack of FractOS is capable of saturating the network line rate when running the `DAX` optimization. The traditional storage stack as well as the `Disaggregated Baseline` yield roughly 20% less throughput, demonstrating that the `DAX` optimization is also beneficial for achieving higher storage bandwidths.

In summary, these results show that a traditional storage stack using FractOS is competitive with existing hardware-accelerated NVMe-oF, while FractOS-based compositions can enable DAX-style interface without compromising service encapsulation or isolation.

6.5 Multi-service application: face verification

This application combines the storage stack and the GPU service to demonstrate FractOS' ability to optimize both the control and data paths, which should manifest in lower latency and higher throughput results compared to conventional solutions. For the baseline, we use existing disaggregation technologies that support device disaggregation. Specifically, we use a frontend node that fetches files from a remote ext4 file system via NFS. The file system is backed by NVMe-over-Fabrics storage. Both NFS and NVMe-oF use in-kernel drivers in Linux. The image data is then copied to and processed by a remote GPU via rCUDA. This setup uses the same level of disaggregation as our FractOS application: remote GPU and two-tiered remote storage. Storage and GPU configurations are otherwise the same as above.

We show latency results for different image batch sizes (Figure 12) and throughput results for multiple in-flight requests (Figure 13) of a single client. In the baseline, data is transferred over network three times: over NVMe-oF, NFS, and rCUDA. Since FractOS replaces these disparate disaggregation technologies with a unified one, it can optimize the data path down to a single transfer: from NVMe directly to GPU. This benefit manifests in lower per-request latencies for FractOS for both CPU and sNIC deployments. Baseline throughput is bottlenecked by rCUDA as discussed above. With four requests in-flight, the GPU itself becomes the bottleneck for FractOS.

FractOS achieves further benefits on the control path due to request composition. In contrast to FractOS, the baseline operates in a star topology: After opening the file, the frontend service node requests the data from NFS, which in turn calls to the NVMe storage, replies travel back to the frontend. The frontend then actively calls the GPU to invoke computation. A total of eight control messages (two for open, four

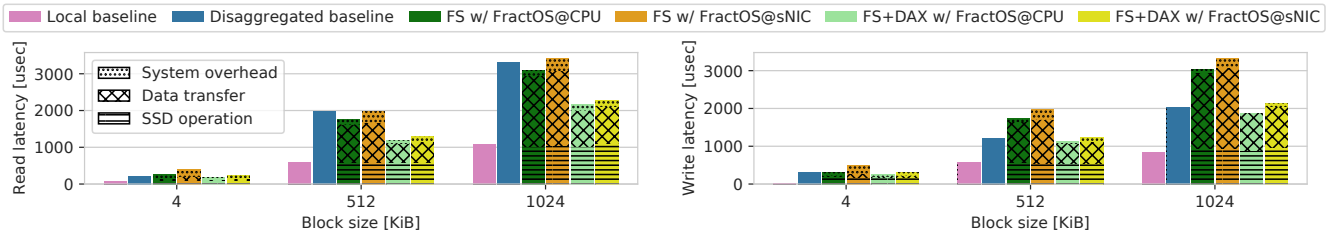

Figure 10. Left: latency of random reads; Right: latency of random writes vs. I/O size (Lower is better).

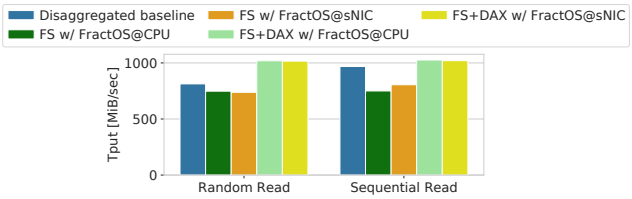

Figure 11. Throughput of random and sequential reads with 1024 KiB block size and 4 requests in flight (Higher is better).

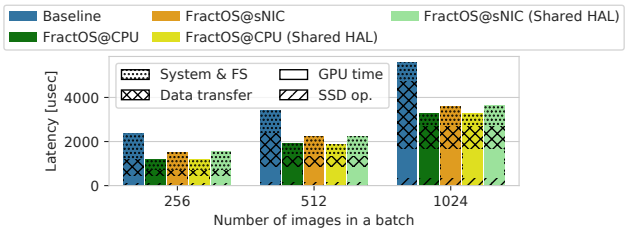

Figure 12. End-to-end latency of a face verification request for different image batch sizes (Lower is better).

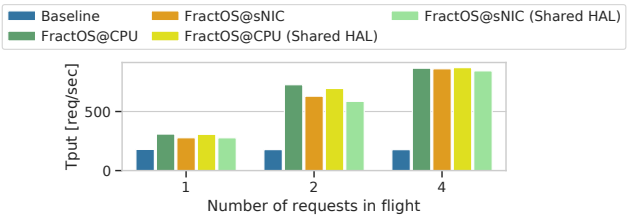

Figure 13. End-to-end throughput of the face verification application (Higher is better).

for reading, two for GPU). With FractOS' distributed control plane, this is reduced to five messages (two for open, call to storage to GPU to frontend is chained).

We also evaluate the configuration where all the Processes connect to a single FractOS Controller running on a CPU (Shared HAL). Such a deployment offers a middle ground between the per-node and sNIC configurations that frees some per-node CPUs for application logic.

7 Discussion

Early prototype with salient results. Micro-benchmarks show promising results, but highlight that sNIC deployments suffer from lower performance. Future sNICs, however, will likely perform better due to more powerful cores and new hardware offloads on the sNIC. Importantly, even on existing sNICs, the end-to-end benchmarks perform with FractOS better than the baseline in all configurations.

Decoupled architecture. FractOS follows a micro-kernel design that allows us to decouple devices, Processes and Controllers from each other. This decoupling makes them agnostic to each other's physical location, and allows us to take various isolation, performance, and cost trade-offs. For example, device adaptor Processes can choose a number of strategies, from traditional programs in co-located host CPUs (e.g., NVIDIA GPU drivers are sufficiently complex that other options are not yet realistic) to compute logic embedded in the device itself (e.g., in a programmable SSD [48]). Furthermore, we demonstrated that the same is true for Controllers, and we envision future systems where the Controller placement is dynamically chosen between host CPUs (isolated by the host OS or hypervisor), co-located sNICs (which can offer isolation from untrusted devices and the host OS or hypervisor), and remote-aggregated Controllers (e.g., by defining a rack-scale Controller that can be particularly cost effective when combined with hardware acceleration).

Present-proof. FractOS is readily deployable and compatible with existing hardware: Controllers and device adaptor Processes are regular user-level Linux programs, and device adaptors use existing device drivers from the host OS.

Future data-center architecture. We envision a FractOS-native data center with separate racks for storage, CPUs, GPUs, etc., all of them accessible using RDMA offloads. In this vision, we have FractOS-capable sNICs to run Controllers, possibly a few per rack, and each rack, enclosure or device is FractOS-native by having some modest CPUs or embedded controllers to execute device drivers and FractOS adaptors, reducing their cost as there is no client application logic deployed on them.

Optimizing Controllers and device adaptors. We have identified a number of optimizations to speed up FractOS Controllers on sNICs; a lot still remains to be optimized on our code (e.g., lock contention and memory coherency traffic). The same lessons can be used when deploying in-device adaptor Processes as they are will execute on wimpy in-device CPUs. Additionally, we will explore how to accelerate critical Controller and device-adaptor operations via hardware offloads. For example, small-size `memory_copy()` calls would substantially benefit from hardware support for third-party RDMAs in the sNIC (eliminating the bounce buffers

we use). Further, Controllers perform frequent lookups for capabilities and objects. Such lookups can be accelerated by sNICs. Additionally, request/response queues over RoCE lead to redundant memory transfers, which can be eliminated with inline receives that currently are not supported by standard Infiniband verbs. Furthermore, in-device acceleration of adaptor Processes would also improve their integration and offloading the critical path in a similar way we would offload the Controllers.

High-level programming models. FractOS's user-level API (`libfractos`) is only slightly more higher-level than the equivalent `libc` on a Linux system, but one could easily use it as a backend for existing RPC-based systems. We plan to explore how well-known programming models and frameworks such as streaming and dataflow [13, 21] can leverage FractOS to build more efficient applications while composing them with other FractOS-aware services and devices.

8 Related Work

FractOS touches on various research areas that study similar challenges but either do not focus on the data center or, in isolation, lead to different designs and trade-offs.

Distributed computations. Programming frameworks such as Apache Beam [13], Dryad [21], Naiad [34], PTask [47] or Ray [32] express distributed computations at a high-level, similar to FractOS, but cannot perform the same optimizations across services, and are impervious to disaggregation.

Service orchestration frameworks. Frameworks such as Apache Airflow [2] and Netflix Conductor [35] address service composition via external orchestration frameworks, but cannot enable optimizations across services (e.g., ⓓ and ⓔ in Figure 2). FaaS storage hooks (e.g., AWS S3 with Lambda hooks) behave similarly: storage acts as an intermediate data and control exchange point. In contrast, FractOS allows optimizations across services and devices.

Device-oriented disaggregation. Existing disaggregation solutions [3, 10, 12, 14, 16, 20, 23, 40, 43–45, 49, 57] focus on transparency, but fail to meet many of our requirements (e.g., composition, delegation, or encapsulation), thus failing to eliminate the associated redundant network traffic.

Accelerator-centric OSes. Several prior works highlight the limitations of a centralized OS architecture in heterogeneous systems, such as lack of network and storage abstractions for GPUs [24, 39, 54]. Lynx [56] focused on a server architecture that uses SmartNICs to manage accelerators in accelerator-heavy servers. OmniX [53] and Helios [36] are accelerator-centric OS architectures that enable decentralized application execution via peer-to-peer interaction among heterogeneous devices in a single server. Similarly, M³x [4] showed how accelerators can get direct access to each other and OS services based on a new inter-device communication hardware. FractOS generalizes these ideas to a disaggregated environment, by introducing a novel RPC mechanism that supports decentralized execution and composition, and is fully integrated with distributed capabilities for dynamic authorization.

Distributed capabilities. Distributed capabilities have received a lot of attention across many OSes [6, 18, 33] and languages [31]. FractOS presents a unique design point in this space that removes costly capability-tracking network messages from the critical path without compromising fault tolerance, by specializing revocation and reliability for disaggregated devices (via its owner-centric capability indirection objects). While SemperOS [18] and Barrelfish [6] are technically distributed capability OSes, they are designed for low-latency many-core interconnects, disregarding reliability, which makes it hard to scale them out over the network.

9 Conclusions

We describe FractOS, a new distributed OS for heterogeneous disaggregated data centers. FractOS has the goal of decentralizing application execution across multiple services and devices, and therefore eliminating redundant network messages that are intrinsic to existing centralized systems.

FractOS offers a comprehensive solution by reconciling disaggregation and heterogeneous devices with the way we build applications and the OSes that manage them. It elevates devices to the status of first-class citizens in the system, and enables direct device-to-device interaction by providing mechanisms to encapsulate device-specific protocols, to compose functionality across devices, and to manage access to them in a decentralized and dynamic way.

Our results show that FractOS achieves higher performance than the existing solutions, enabling speedups of 47% and a 3× reduction in network traffic for a realistic application using multiple disaggregated devices and services. We demonstrate that FractOS can outperform existing accelerator and storage disaggregation solutions, while reducing network traffic by up to 2×.

We believe that the core concepts in FractOS provide a path to substantially reduce the network tax of disaggregation in heterogeneous data centers, and envision that our work will motivate the development of new disaggregation-friendly hardware management interfaces.

10 Acknowledgments

We would like to thank our shepherd, Gustavo Alonso, for his helpful feedback and the anonymous reviewers for their insightful comments. This research was possible thanks to funding from the state of Saxony/Germany, the European Union's Horizon2020 programme (grant 957216), the German Research Council DFG (through CFAED, grant HA 2461/11-1, and grant 912 HAEC), the Technion Hiroshi Fujiwara Cyber Security Research Center, the Israel National Cyber Directorate, the Israel Science Foundation (grant 1027/18), and a generous donation from Intel Corporation.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Summer Usenix*, July 1986.

[2] Apache Foundation. Apache airflow. https://airflow.apache.org. Last accessed: April 2021.

[3] Krste Asanović. Firebox: A hardware building block for 2020 warehouse-scale computers. In *USENIX Conf. on File and Storage Technologies (FAST)*, February 2014.

[4] Nils Asmussen, Michael Roitzsch, and Hermann Härtig. M³x: Autonomous accelerators via context-enabled fast-path communication. In *2019 USENIX Annual Technical Conference*, USENIX ATC'19, pages 617–632, Renton, WA, 2019. USENIX Association.

[5] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, pages 189–203. ACM, 2016.

[6] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Symposium on Operating Systems Principles (SOSP)*, October 2009.

[7] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. *Distributed garbage collection for network objects*. Citeseer, 1993.

[8] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2, June 1992.

[9] Feras Daoud, Amir Wated, and Mark Silberstein. Gpurdma: Gpu-side library for high performance networking from GPU kernels. In Kamil Iskra and Torsten Hoefler, editors, *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers, Kyoto, Japan, June 1, 2016*, pages 6:1–6:8. ACM, 2016.

[10] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *Intl. Conf. on High Performance Computing and Simulation (HPCS)*, June 2010.

[11] Keith G Erickson, M Dan Boyer, and David Higgins. Nstx-u advances in real-time deterministic pcie-based internode communication. *Fusion Engineering and Design*, 133:104–109, 2018.

[12] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond processor-centric operating systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2015.

[13] Apache Foundation. Apache beam. https://beam.apache.org/, 2016.

[14] Gen-Z Consortium. *Gen-Z Core Specification*. Version 1.1.

[15] Benjamin Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. *ACM SIGPLAN Notices*, 24(7):313–321, 1989.

[16] Google. Cloud tensor processing units (TPUs). https://cloud.google.com/tpu/docs/tpus. Last accessed: Nov 2020.

[17] Chuanxiong Guo. RDMA in data centers: Looking back and looking forward. https://conferences.sigcomm.org/events/apnet2017/slides/cx.pdf. Last accessed: October 2021.

[18] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. Semperos: A distributed capability system. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 709–722, 2019.

[19] Huawei Technologies. *Huawei Atlas 300I Inference Card, Technical White Paper (Model 3000)*, September 2020.

[20] Intel. Intel rack scale design architecture, 2014.

[21] Michael Isard, Mihai Budiu, Yuan Yu, Andrew D. Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, March 2007.

[22] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, February 2019.

[23] 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 *Design, Automation Test in Europe Conf. Exhibition (DATE)*, March 2016.

[24] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. Gpunet: Networking abstractions for GPU programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 201–216, Broomfield, CO, October 2014. USENIX Association.

[25] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Symposium on Operating Systems Principles (SOSP)*, October 2009.

[26] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *European Conference on Computer Systems (EuroSys)*, April 2016.

[27] Adam Lackorzynski and Alexander Warg. Taming subsystems: Capabilities as universal resource access control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, IIES'09, pages 25–30, New York, NY, USA, 2009. ACM.

[28] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *1992 12th International Conference on Distributed Computing System*, pages 708–709. IEEE Computer Society, 1992.

[29] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2009.

[30] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, volume 2, pages 21–33, 2007.

[31] Mark S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, May 2006.

[32] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications. In *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, October 2018.

[33] Sape J. Mullender and Andrew S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29(4):289–299, 1986.

[34] Derek Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A timely dataflow system. In *Symposium on Operating Systems Principles (SOSP)*, November 2013.

[35] Netflix. Netflix conductor. https://netflix.github.io/conductor. Last accessed: April 2021.

[36] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Symposium on Operating Systems Principles (SOSP)*, page 221–234, 2009.

[37] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *European Conference on Computer Systems (EuroSys)*, April 2018.

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

[39] Nvidia. GPUDirect RDMA – CUDA toolkit documentation. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html. Last accessed: October 2021.

[40] NVM Express. *NVM Express over Fabrics*, July 2018. Revision 1.0a.

[41] Oracle. Java remote method invocation. https://www.oracle.com/java/technologies/javase/remote-method-invocation-home.html. Last accessed: October 2021.

[42] OMG Standard Development Organization. Common object request broker architecture. https://www.omg.org/spec/CORBA. Last accessed: October 2021.

[43] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Prashanth Gopi Gopal, and Simon Pope. A reconfigurable fabric for accelerating large-scale datacenter services. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2014.

[44] Murali Ravindran. Extending cabled PCI Express to connect devices with independent PCI domains. In *IEEE Systems Conference (SysCon)*, April 2008.

[45] Jack Regula. Using non-transparent bridging in PCI Express systems. Technical report, PLX Technology, 2004.

[46] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9:1–9:32, Aug 2013.

[47] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *Symposium on Operating Systems Principles (SOSP)*, page 233–248, October 2011.

[48] Samsung. *Samsung SmartSSD Computational Storage Drive*.

[49] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed os for hardware resource disaggregation. In *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, October 2018.

[50] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. Eros: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, 1999.

[51] Marc Shapiro, David Plainfossé, and Olivier Gruber. A garbage detection protocol for a realistic distributed object-support system. Technical report, INRIA, November 1990.

[52] R. M. Shapiro and H. Saint. The representation of algorithms as cyclic partial orderings. Technical report, New York: Meta Information Applications, Inc., July 1971.

[53] Mark Silberstein. OmniX: An accelerator-centric OS for omni-programmable systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*, page 69–75, May 2017.

[54] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. Gpufs: integrating a file system with gpus. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, pages 485–498. ACM, 2013.

[55] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, October 2010.

[56] Maroun Tork, Lina Maudlej, and Mark Silberstein. *Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers*, page 117–131. March 2020.

[57] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. AvA: Accelerated virtualization of accelerators. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2020.