Received: 7 March 2022 | Revised: 8 December 2022 | Accepted: 3 March 2023

DOI: 10.1002/cpe.7702

RESEARCH ARTICLE

WILEY

Networking in next generation disaggregated datacenters

Brice Ekane[1] | Alain Tchana[2] | Daniel Hagimont[3] | Boris Teabe[3] | Noel De Palma[1]

[1]University of Grenoble, Grenoble, France

[2]École normale supérieure de Lyon, Lyon, France

[3]IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3, Toulouse, France

Correspondence
Boris Teabe, IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3, 2 Rue Charles Camichel, Toulouse, 31000, France.
Email: boris.teabedjomgwe@enseeiht.fr

Funding information
French Agence nationale de la recherche, Grant/Award Number: ANR-20-CE25-0013

Summary

Nowadays, datacenters lean on a computer-centric approach based on monolithic servers which include all necessary hardware resources (mainly CPU, RAM, network, and disks) to run applications. Such an architecture comes with two main limitations: (1) difficulty to achieve full resource utilization and (2) coarse granularity for hardware maintenance. Recently, many works investigated a resource-centric approach called disaggregated architecture where the datacenter is composed of self-content resource boards interconnected using fast interconnection technologies, each resource board including instances of one resource type. The resource-centric architecture allows each resource to be managed (maintenance, allocation) independently. LegoOS is the first work which studied the implications of disaggregation on the operating system, proposing to disaggregate the operating system itself. They demonstrated the suitability of this approach, considering mainly CPU and RAM resources. However, they did not study the implication of disaggregation on network resources. We reproduced a LegoOS infrastructure and extended it to support disaggregated networking. We show that networking can be disaggregated following the same principles, and that classical networking optimizations such as DMA, DDIO, or loopback can be reproduced in such an environment. Our evaluations show the viability of the approach and the potential of future disaggregated infrastructures.

KEYWORDS

disaggregated datacenters, network, split kernel

1 | INTRODUCTION

1.1 | Context and background

Despite the virtualization technology, data centers (DC) still face the problem of *server fragmentation*, which causes a considerable loss of money for operators. For example, Microsoft estimates in 2020 that a 1% reduction in fragmentation within its Azure cloud platform would result in hundreds of millions of dollars in savings. This problem is fundamentally due to the classical *server-centric* architecture[1] (Figure 1 top), which is massively adopted in todays DC. In this architecture, the DC is made of *monolithic servers*, each including all necessary hardware resources (mainly CPU, RAM, network, and disk) to run applications. The latter are constrained to a single server boundaries.

Among the approaches that have been proposed to reduce fragmentation, *rack disaggregation*[*1-7] appears to be the most promising approach. It consists in consolidating a large pool of hardware resources (CPUs, RAM, disk, network, etc.) into a single large *rack-scale computer*. Hence, in this model, the DC can be seen as a set of very large machines, that can each be dynamically partitioned into applications with very flexible and

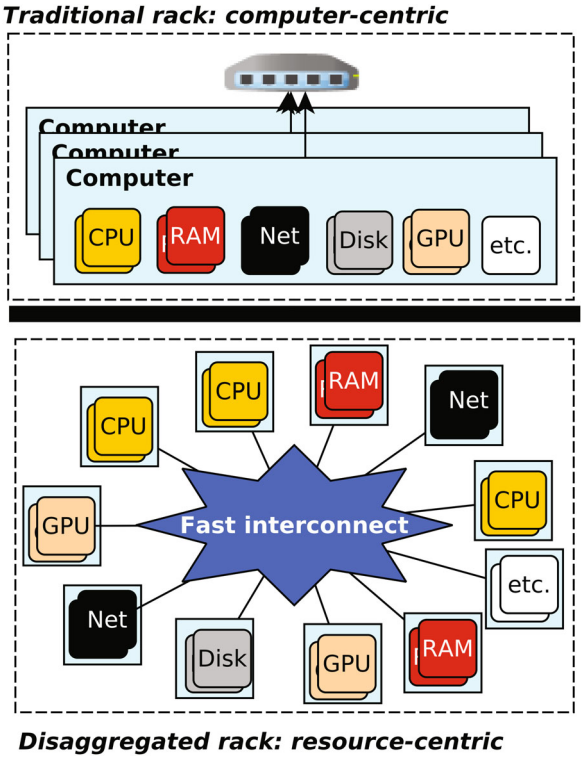

FIGURE 1 (Top) Computer- and (bottom) resource-centric architectures.

diverse resource allocations. In order to become viable in practice, disaggregation requires (i) very efficient network communication between the elements/machines within the rack and (ii) specific support from the system software (OS, hypervisor).

Two forms of disaggregation, that we name *soft disaggregation* and *hard disaggregation*, have been proposed in the recent years:

Soft disaggregation[1,3-6] consists in using traditional monolithic servers and adapting *only the software layers* to allow an application to simultaneously use resources from several machines (within a rack). The good performance offered by current DC network technologies (e.g., Infiniband) makes soft disaggregation exploitable in the current server-centric architecture.[2]

Hard disaggregation[7-10] is a more aggressive approach, as it requires a deep redesign of both hardware and software layers. In hard disaggregation, the rack is built as a cluster of specialized and independent resource boards (a *resource-centric* architecture, Figure 1 bottom) instead of a cluster of monolithic servers. Each resource board is a specialized machine that offers only one resource type (e.g., pComponent, mComponent, sComponent, and nComponent for respectively CPU, memory, storage, and network boards) to applications. Resource boards are interconnected with an ultra-fast network fabric.[11]

Our project focuses on hard disaggregation as it maximizes the potential benefits of disaggregation. First, it allows resources to scale independently. Second, it facilitates application scalability since the application can transparently use the entire rack resources. Third, it bounds the impact of a component failure on a board to that component only. In contrast, a single failure in a traditional machine would take out the full server.

Hereafter, we use the term disaggregation to refer to hard disaggregation.

1.2 | Motivations

Shan et al.[7] (best paper awards OSDI 2018) studied for the first time the implications of disaggregation on the operating system (OS). The authors argued that the OS should be disaggregated in the same way as the hardware, resulting to what they called the *split-kernel* model: kernel services are decoupled into loosely-coupled components (called monitors) following a strict separation of concerns.[7] Each component is responsible for the management of a single resource board type (e.g., CPU) and its code base does not include stuff for managing another component type. For instance, the monitor of the CPU should not include memory virtualization (e.g., paging) code. The authors released *LegoOS*, the first split-kernel-based OS. In its current state, LegoOS takes into account three resource types: CPU, memory, and storage. Network resources for intra-rack (a kind of loopback, since the rack is seen as a giant computer), inter-rack (within the same DC) and communication with the rest of the world were not considered by the authors.

In this article, we fill this gap as network is of crucial importance in todays DC, most applications (e.g., machine learning, web servers, databases, etc.) being distributed and accessed through the Internet. In LegoOS, applying the split-kernel model to CPU, memory and storage components

allowed demonstrating the relevance of the disaggregation and split-kernel approaches, but applying these approaches to networking is a much tricky issue. In LegoOS, CPU capacities are managed by monitors called *pComponents* which embed a set of processors and a memory cache (an extra level cache). Memory capacities are managed by monitors called *mComponents* which embed memory banks. Logically, applying disaggregation and split-kernel leads to embedding a set of network devices in a network component (*nComponent*).

However, networking is particular as it heavily involves the CPU and memory components, thus leading to difficult issues and important design choices:

- First, it seems natural to embed device drivers in nComponents. But what about the protocol stack which can be viewed as a program running in a pComponent (using memory from a mComponent and drivers from nComponents), or embedded in nComponents?
- Second, regarding memory management, a sent or received message is read/written from/to memory (in the mComponent) by the device driver (in the nComponent). In a monolithic server, hardware features are integrated for IO memory optimization, such as DMA (direct memory access)[12] or DDIO (Data Direct IO).[13] How do such optimizations translate in a disaggregated architecture?
- In a monolithic server, local (to the server) communications are optimized and don't involve a full protocol stack traversal. Is it possible to optimize local communications between two processes within the same pComponent or between two pComponents within the same LegoOS machine?

1.3 | Contributions

In this article, we describe an extension to the LegoOS split-kernel-based system in a disaggregated architecture, aiming at integrating network support. We analyse and motivate the design and integration of networking nComponents in the LegoOS architecture and show that:

- Networking should be disaggregated as nComponents embedding the NIC device drivers as well as the protocol stacks.
- DMA like optimization can be implemented for in memory data transfers in this disaggregated network architecture.
- DDIO like optimization can be implemented for in cache data transfers in this disaggregated network architecture.
- Local (intra pComponent or inter pComponents) communications can be optimized as well.

We reproduced an experimental LegoOS disaggregated platform as described in Reference 7 (with an identical hardware and software setting), implemented in this platform network support following the previous design choices and evaluated the performance of the prototype, demonstrating the suitability of our design.

The rest of the article is structured as follows. Section 2 introduces the background information related to disaggregation, the LegoOS system and networking technologies. Section 3 presents the integration of networking in this environment. Section 4 reports on our performance evaluation of this prototype. After a review of related works in Section 5, we conclude in Section 6.

2 | BACKGROUND

This section first introduces the main concepts and the system that we use.

2.1 | Disaggregation

Disaggregation splits the monolithic computer into a number of single-resource boards that communicate using a fast interconnect (e.g., silicon photonics[11]), see Figure 1 bottom. Each resource board includes only one resource type (e.g., CPU, main memory, storage, or network). Therefore, the rack is seen as a "giant" computer. An application running in such a disaggregated rack uses several resource boards at a time (at least one CPU and one memory board). The main benefits expected from disaggregation are as follows. *(1) It facilitates the scalability of each resource type.* For instance, the memory capacity of the rack can be increased without increasing the number of CPUs. *(2) It facilitates application scalability* since the application can use the entire resources of the rack. This reduces the development burden (of implementing a distributed application) for the application owner. *(3) It allows optimal resource utilization* (thus good energy proportionality) by minimizing resource fragmentation (with monolithic servers, resource usage is often unbalanced, for example, memory is fully used and CPUs are under-used). *(4) It limits the failure impact of a given resource type.* The failure of a blade is limited to that blade and it does not impact other blades neither of the same type or of a different type. *(5) It facilitates the rapid integration of new technologies.* For instance, the integration in the rack of a new CPU generation does not require to buy a whole computer, and thus other resource types. All these advantages make us and many others[7,8,14] believe that next generation DC will be disaggregated.

Disaggregation is still in its early stages, both on the hardware and software sides. Several manufacturers have started proposing intermediate approaches for rack disaggregation based on microcomputers, for example Intel rack-scale, AMD SeaMicro[15] and HP Moonshot.[16] However, they cannot solve all the issues of monolithic computers since their hardware model is still a monolithic one. dRedBox[17] is, to the best of our knowledge, the only open project trying to roughly decouple hardware resources. At the software level, a recent breakthrough is *LegoOS*,[7] which introduces the *split-kernel* model as a way of building OSes for disaggregated racks.

2.2 | The split-kernel model and LegoOS

In a monolithic server, the OS runs on a single board and its code base is monolithic in the sense that it includes the code to manage all resource types. The OS assumes local access to all resources. Shan et al.[7] introduced an OS blueprint for disaggregated racks called *split-kernel*. The latter consists in designing the OS of a disaggregated rack as a disaggregated OS composed of several specialized and loosely-coupled specialized OSes called *monitors*. Each monitor runs at and manages a unique board. It operates for its own functionality and communicates with other monitors when there is a need to access other boards' resources. Shan et al.[7] is the first research group which proposed an OS design for real disaggregated racks. They prototyped the split-kernel model in *LegoOS*.

Figure 2 presents an overview of LegoOS' architecture. Its current design targets three resource types: processor, memory, and storage, embedded in *pComponents*, *mComponents*, and *sComponents* respectively (focus on, i.e., does not handle other resource types). LegoOS uses a two-level resource management mechanism. At the higher level, it uses global resource managers (noted *GPM*, *GMM*, and *GSM*) to perform coarse-grained global resource allocation and load balancing for processor, memory and storage respectively. Such managers run on any pComponent of the LegoOS machine. They only maintain approximate resource usage and load information. At the lower level, each resource board is locally managed by a monitor. Each monitor employs its own policies and mechanisms to manage its local resources. Shan et al.[7] mainly described the implementation of pComponent and mComponent.

LegoOS moves all hardware memory functionalities to mComponents (e.g., page tables, TLBs) and leaves only caches at the pComponent side. Therefore, pComponents only see virtual addresses, which are thus used to access caches. LegoOS assumes that each pComponent includes a larger (e.g., 1 GB) last-level-cache (called *extCache*) to entirely host both its monitor and a significant portion of application working sets, thus minimizing accesses to mComponents. Each mComponent can choose its own memory allocation technique and virtual to physical memory address mappings (e.g., segmentation).

Since there is no real disaggregated rack, Shan et al.[7] emulated disaggregated boards in LegoOS using commodity servers by limiting their internal hardware usages. To emulate mComponents, the number of usable cores of the server is limited to two cores while the emulation of pComponent uses all cores and exploits the main memory of the server as extCache. The communication between boards is emulated using Mellanox Infiniband adapters, switches and links. Each monitor embeds a customized RDMA-based RPC framework called *FIT*[7] which eases the utilization of RDMA.

We have faithfully reproduced LegoOS experimental environment in our lab in order to study the integration of network boards, enabling intra- and inter-rack communication, and communication with the rest of the world.

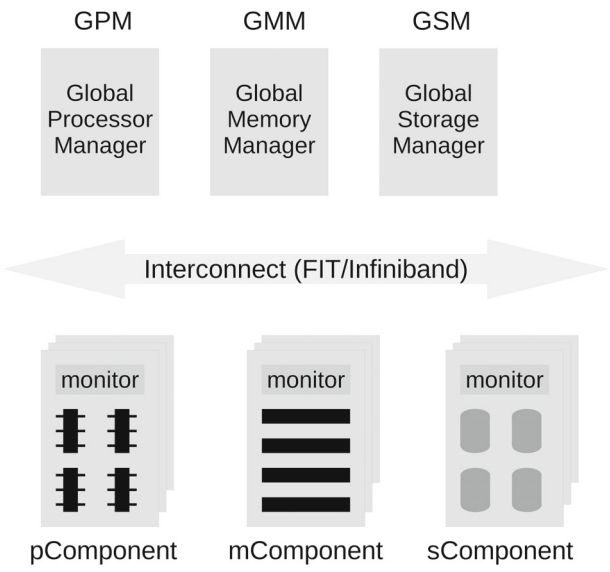

FIGURE 2 Architecture of LegoOS.

2.3 | DMA and DDIO

We will later study how traditional IO optimizations apply in such an architecture.

DMA is a mechanism which allows an I/O device to transfer data to/from main memory without the intervention of the CPU. The system software instructs the DMA engine by providing source/destination memory addresses. The main source or destination of traditional DMA transfers is main memory, although the data actually needs to be loaded into the CPU's cache for processing (for inbound packets). Outbound packets, which are initially inside the CPU's cache, need to be copied to main memory before DMA occurs. These memory trips lengthen IO latency and consume memory bandwidth.

Intel (ARM provides the same feature with Cache Stashing[18]) introduced the DDIO technology[13] with the Xeon E5 family. With DDIO, I/O devices perform DMA directly to/from the last level cache (LLC) rather than the main memory, as follows. When the NIC receives a packet, DDIO overwrites (*write update*) the LLC if the destination buffer is present in the cache (PCIe write hit). Otherwise (PCIe write miss), DDIO allocates (*write allocate*) some cache lines and write the data into the newly allocated lines. Write allocates are restricted to a limited portion (10%) of LLC.

Concerning packet sending, the NIC directly reads the data from the LLC if the source buffer is present in any LLC way (PCIe read hit). Otherwise, the NIC reads the data from main memory (with DMA), but does not fills the LLC.

3 | NETWORKING IN A DISAGGREGATED DATACENTERS

Our contribution involves adding a network component (hereafter *nComponent*) to the disaggregated rack following the split-kernel model. A nComponent allows processes within the disaggregated rack to communicate both with each other and with the rest of the world using network primitives (e.g., socket BSD API). To integrate nComponents, we need to answer four main questions: what is the composition of a nComponent from a hardware perspective? How can essential and novel hardware features such as DMA and DDIO be adapted to the disaggregated paradigm? What are the services of the nComponent monitor? How to provide best performance?

3.1 | Hardware design

Figure 3 presents the hardware architecture of a nComponent. `NICs` are either classical or sophisticated (smart NICs[19]) network adapters which include packet reception and transmission circuits, and can be referenced using MAC and IP addresses. They connect the disaggregated rack with other network devices in the data center (e.g., switches). According to the split-kernel model, a nComponent as well as other disaggregated devices includes a set of `controllers` (comparable to CPUs) to run the network monitor. The binary code of the latter resides in a local memory (`DRAM` in the figure), instead of a remote mComponent for performance purposes. The local memory also hosts incoming and outgoing packets to/from NICs. This is achieved by local DMA (noted `lDMA` in the figure) engines integrated in NICs. For all these reasons, the local memory of a nComponent must have a substantial size, in the range of giga bytes.

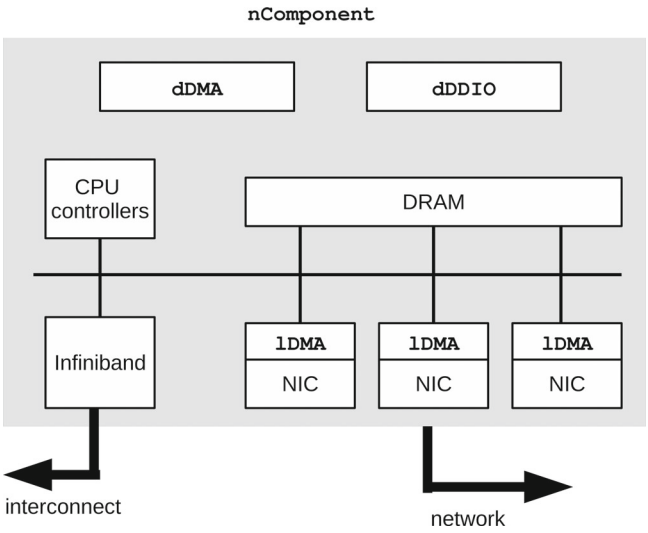

FIGURE 3 Hardware architecture of a nComponent.

The nComponent also includes a disaggregated DMA (noted `dDMA`) engine and a disaggregated DDIO (noted `dDDIO`) engine which implement packet transfer with mComponents and pComponents' extCache respectively. Although DMA and DDIO are implemented in the hardware in monolithic servers, they are implemented by the software monitor in the nComponent.

3.2 | Software design overview

Figure 4 presents the extended architecture of LegoOS which takes into account networking aspects. Our contribution in this architecture is threefold. *(1)* The global network manager (`GNM`) runs on the same machine as the other LegoOS managers (see Section 2). Once started, GNM initializes a list of available nComponents within the rack, including the IP address associated with each NIC, as it implements NIC allocation policy to processes. *(2)* The pComponent monitor is extended with a `bsdSocketStub` object which provides a complete BSD socket API interface to processes. Following the separation of concerns behind the split-kernel model, the implementation core of the socket BSD API is made inside the nComponent. It is provided by a `bsdSocketSkeleton` object. Each process in a pComponent performing a network operation is allocated an instance of `bsdSocketStub`, which manages the socket state and forwards all calls to its counterpart `bsdSocketSkeleton` instance in the nComponent, implementing a RPC like mechanism. *(3)* a nComponent runs a network monitor, an operating system dedicated to NIC management. It includes `bsdSocketSkeleton` instances, a `Proxy` (it is the entry point for the nComponent allowing to create `bsdSocketSkeleton` instances for communications), network routing rules, NIC drivers and a setup. The latter initializes the nComponent internal state (e.g., IP addresses associated with NICs) and registers the nComponent and its IPs in the GNM.

The next sections detail the implementation of the main steps of connection establishment and message exchange.

3.3 | Connection establishment and termination (`socket, bind, connect`)

The invocation of *socket()*, which is the first call performed by a network process in a pComponent, instantiates a `bsdSocketStub` instance whose role is to handle all socket BSD calls. When *bind()* is invoked, the `bsdSocketStub` sends to GNM the (local) IP address parameter of the *bind()* operation, asking the GNM the nComponent associated with this IP address (GNM knows the IP addresses associated with each nComponents). The nComponent identifier is returned to the `bsdSocketStub` which registers it. The `bsdSocketStub` then connects with the Proxy on the target nComponent, and requests the creation of a `bsdSocketSkeleton` instance, and sends to it all information related to the created socket. These informations are: basic socket information (socket type, protocol, IP address, port), pComponent and `bsdSocketStub` instance identifiers, allowing the `bsdSocketSkeleton` instance to send messages back to the `bsdSocketStub` instance. The `bsdSocketSkeleton` instance registers these informations and creates in the nComponent a local socket on the protocol stack. From then on, the `bsdSocketStub` communicates directly with this nComponent/`bsdSocketSkeleton` using a message-based protocol (similar to a RPC, but invocations can be performed in both directions).

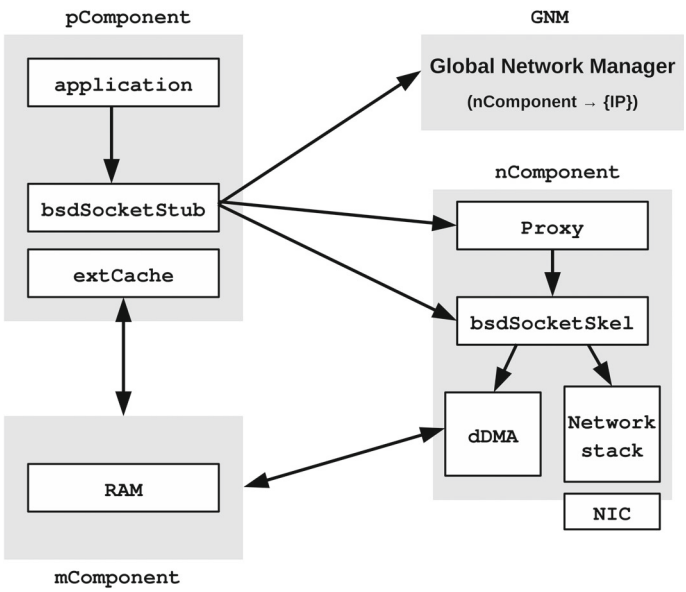

FIGURE 4 General overview of LegoOs extended with nComponent.

A call to *accept()* is forwarded by the `bsdSocketStub` to the `bsdSocketSkeleton` which reproduces it on the local socket and blocks waiting for a connection request, which (when received) is forwarded by the `bsdSocketSkeleton` back to the `bsdSocketStub`. Similarly, a call to *connect()* is forwarded by the `bsdSocketStub` to the `bsdSocketSkeleton` and reproduced on the local socket.

3.4 | Packet transmission (`send`)

We consider hereafter that the default version relies on DMA access to memory. We describe in Section 3.6 the DDIO based communication scheme. Figure 5 summarizes the implementation steps of packet sending, issued by a process on the pComponent. On `send()` invocation (step ①), the `bsdSocketStub` transfers the data to be sent from the *extCache* (in the pComponent) to the mComponent (step ②). This is a kind of data flush from extCache to memory (this step only occurs if the cached data is not synchronized with its version in the mComponent). It then informs (step ③) the `bsdSocketSkeleton` of the presence of data to be sent over the network. To this end, the `bsdSocketStub` provides to the `bsdSocketSkeleton` the size of the data to be sent and its memory address in mComponent. Upon receiving this order, the `bsdSocketSkeleton` configures dDMA in such a way that data to be sent can be loaded from the mComponent and stored in the nComponent's DRAM (step ④) for being accessed by the network stack. Subsequently, the `bsdSocketSkeleton` invokes the `send()` function on the local socket (step ⑤), walking through the kernel network stack and transferring data to the NIC driver. Notice that both dDMA and lDMA are involved when the network stack effectively handles the data.

3.5 | Packet reception (`recv`)

When a packet arrives at the NIC, the lDMA engine copies the packet to the nComponent's DRAM and the data is registered in the `bsdSocketSkeleton`. When `recv()` is invoked in the pComponent, the call is forwarded from the `bsdSocketStub` to the `bsdSocketSkeleton` (with the address of the buffer in the mComponent), which asks dDMA to copy the read data from the nComponent's DRAM to the buffer in the mComponent. Then, the `bsdSocketSkeleton` responds to the `bsdSocketStub` with the size of the read data. The reading process in the pComponent will then access the data, loading it from the mComponent to the extCache.

3.6 | Disaggregated DDIO

Data Direct I/O Technology (Intel DDIO) is a feature introduced in Intel processes that allows a NIC to directly exchange data with the CPU cache without going through the RAM. This considerably increases the network speed, reduces latency and finally reduces energy consumption. DDIO

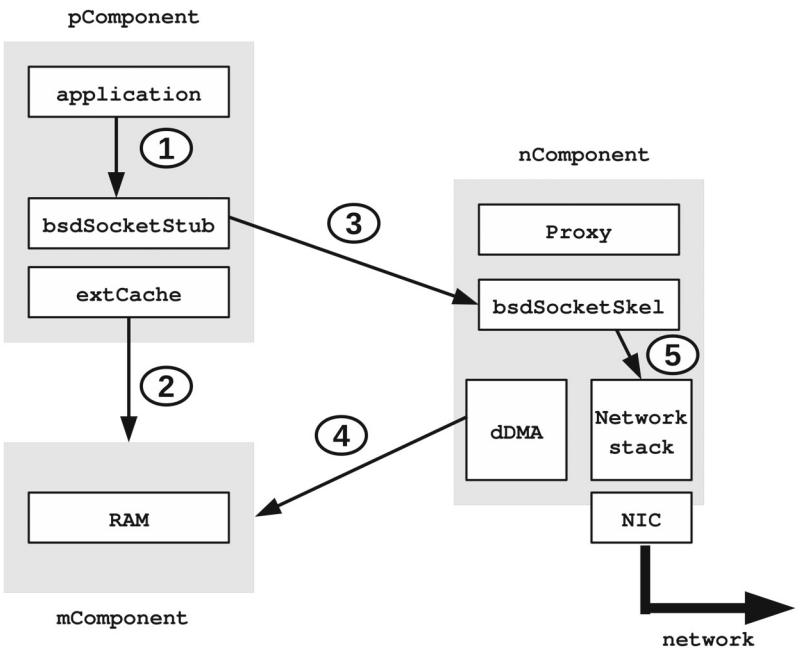

FIGURE 5 Packet transmission step.

in a disaggregated environment consists of allowing nComponents to send/receive packets directly to/from pComponents (their extCache) without going through mComponents. To make this possible in our design, the workflow of packet transmission and reception presented above slightly changes as follows. On transmission (`send()`), the `bsdSocketStub` is able to transmit the data to be sent directly from the pComponent's extCache to the nComponent's DRAM (assuming the data resides in the cache). On reception `recv()`, the `bsdSocketSkeleton` is able to transmit the received data directly to the extCache (in an entry associated with the reception buffer in memory).

DDIO makes sense when data fits in the extCache (relatively small data). For instance, on transmission (`send()`), as illustrated in Figure 6 top, DMA (right side) is a better option (than DDIO, left side) if data does not fit in the extCache (especially for large data), since data will be transferred only once (right side). But in the figure (bottom), DDIO (left side) is optimal for cached data while DMA (right side) requires modified data to be flushed to memory before sending. Notice also that DDIO involves more CPU in the pComponent than DMA which is important for large data.

3.7 | Optimizations

The implementation presented above strictly follows the split-kernel model, that is said separation of concerns: the nComponent and GNM are the only components containing network services. This has a severe consequence for intra-rack communications (e.g., which use the *loopback* interface). In fact, all network communications, including intra-rack communications, systematically involve a nComponent. This is not necessary when the two processes are within the same disaggregated rack. This limitation increases intra-rack communication latency in several use cases. It is important to note that in a disaggregated rack, the probability to see two pieces of the same application collocated atop the same rack (thus communicating) is very high compared to traditional monolithic servers. This is because the rack has a bigger size than a server. We propose an optimization which requires the distortion of the split-kernel model for tackling this limitation.

We aim at optimizing intra- and inter-pComponent communication inside the same disaggregated rack. When *connect()* is invoked on the `bsdSocketStub`, the stub asks the GNM whether the target IP address is associated with a nComponent in the rack. If so, the connection is local to the rack and optimization can take place. Then, the `bsdSocketStub` invokes the proxy on that nComponent to ask the location of the process

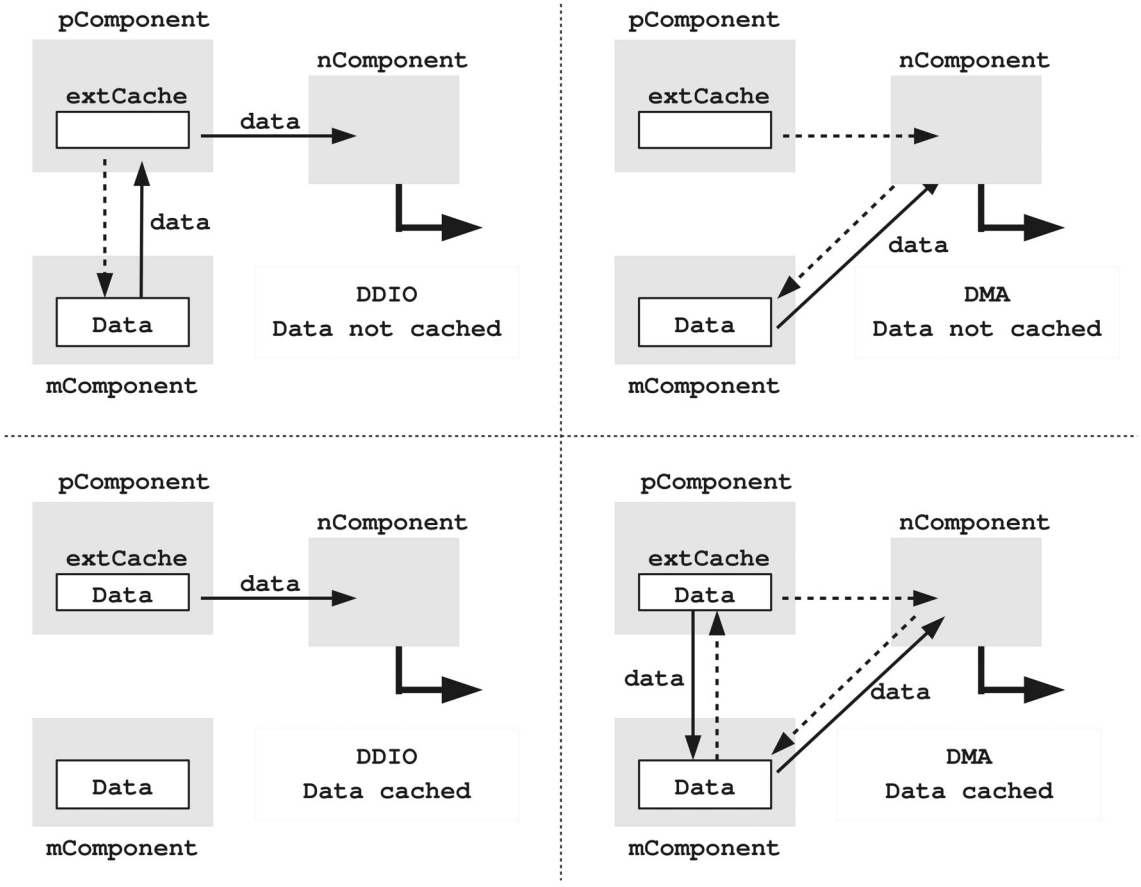

FIGURE 6 DDIO/DMA comparison.

which made a bind on the target IP/port (the target pComponent and `bsdSocketStub` instance identifiers). Thus, a direct connection between the origin and target `bsdSocketStubs` can be established. If the communication is intra-pComponent, the two `bsdSocketStubs` will communicate using Unix Pipe. If the communication is inter-pComponent, the two `bsdSocketStub` will communicate using FIT (LegoOS interconnect communication protocol). This optimization requires that the `bsdSocketStub`, which is part of the pComponent monitor, includes core networking code. The authors of LegoOS acknowledged a similar distortion to the application of the split-kernel model for achieving optimal performance in the relationship between the CPU and memory (leading to extCache extra cache level).

3.8 | Implementation details

We present here some implementation details of each of the components of our network stack.

`bsdSocketStub`. As discussed previously, this is the component that contains the BSD socket API interface. It provides to applications in the pComponent the interfaces for network syscalls: `socket, bind, connect and close`. The `bsdSocketStub` relies on a message-based protocol, **BSD_RPC** (BSD remote protocol call), in order to defer the implementation of the BSD socket API to the nComponent. All this is done over FIT which is the communication layer of LegoOS. To implement data transmission following the DMA scheme, the `bsdSocketStub` uses two functions: `send_dma_signal` which informs the nComponent that a message is available in the mComponent for transmission, and `recv_dma_signal` which is the handler of the signal sent by the mComponent indicating that a message is available in the mComponent. To optimize transfers for non blocking sockets, each of them is associated with a send and a receive buffer. These two buffers help reducing the number of communications between pComponents and nComponents.

`bsdSocketSkeleton`. It is considered as a service, that is, a Linux kernel space thread. For each new socket, an instance is created and represents the connection in the nComponent. Each `bsdSocketSkeleton` is listening on a port in the nComponent and has two processing queues: `workqueue_send` for messages that should be sent through the NIC and `workqueue_recv` for messages received by the NIC. The queues are created when `bsdSocketSkeleton` is initialized. The processing of messages in the `workqueue_send` is done using the BSD socket implementation of Linux to send packets while messages received in the `workqueue_recv` are handled by the `bsdSocketSkeleton`.

`Proxy`. This is the entry/output point of the nComponent, which manages all interactions with other components (pComponents, mComponents). It is composed of a reception buffer (`recv_ring`) and a `service registry`. Each time a request for sending a message is received in the `recv_ring` from a pComponent, the `Proxy` selects the target `bsdSocketSkeleton` thanks to the `service registry` and adds the message to the `bsdSocketSkeleton` `workqueue_send`. If the request is for the creation of a new socket, it creates a new instance of `bsdSocketSkeleton` and adds it in the `service registry`. The `service registry` records all TCP services, and is implemented in the form of a hash table with the socket port as key. All new socket details are stored in the `service registry` which allows to access these information knowing the socket port. The `Proxy` also interacts with the mComponent through the `dDMA` to store and retrieve data from mComponents. `dDMA` is an extension added to the memory controller of mComponents which allows accessing data from its address in the mComponent.

4 | EVALUATIONS

This section presents the performance evaluations. We aim at answering the following questions:

- What is the network latency and throughput with our network stack implementation in a disaggregated rack?
- What is the overhead with a disaggregated rack compared to a monolithic Linux system? In a disaggregated environment, the overhead is unavoidable due to the communication between components. The objective here is to estimate this overhead.
- What performance can we expect with a real world benchmark?

The goal of the comparison with a monolithic Linux system is to show that a disaggregated server can perform in the same order of magnitude as a monolithic server (and not perform better), and that a real disaggregated hardware (especially a dedicated interconnect instead of Infiniband) would lead to a reasonable overhead while benefiting from the flexibility of disaggregation.

To this end, we rely on two benchmarks for evaluation: a micro-benchmark which is a TCP client/server application and a macro-benchmark which is a spark streaming like benchmark (representing a real world benchmark). The micro-benchmark is used to evaluate the internal mechanisms of our implementation while the macro-benchmark provides a global performance view.

We carried out the evaluations with three configurations:

- **DMA (default configuration)**. This configuration corresponds to the basic implementation (with dDMA) of our network stack on LegoOS with no optimization. Like the traditional DMA feature, it limits the intervention of CPU (here pComponent) in data transfers, the dDMA (in the

nComponent) being responsible for fetching data from the mComponent. This is better suited for large data transfers (where throughput is important) and it offloads processing from the pComponent to the nComponent.

- **DDIO**. This corresponds to the optimized implementation similar to the intel CPU DDIO feature. This mechanism involves CPU (pComponent) for each data transfer and is better suited for small data transfers (where latency is important), in the extCache and involving limited CPU.
- **Linux**. This corresponds to evaluation with a standard Linux system (without LegoOS).

We did not compare to vanilla LegoOS because in his actual state, the network is not handled as stated earlier.

4.1 | Hardware setup and LegoOS deployment

Since there is no real resource disaggregation hardware, we emulated our disaggregated environment using 5 standard servers, 5 Infiniband network adapters and an Infiniband switch. Table 1 presents our server characteristics. Each server has an InfiniBand network adapter (*Mellanox Technologies MT27500 Family [ConnectX-3]*) and there are all connected through a Mellanox Infiniband switch (*Mellanox IS5022 Infiniscale-IV 8-ports*). Notice that the heterogeneity of the environment is typical to a disaggregated rack which will have pComponents, mComponents, and nComponents from different vendors. In our emulated environment, servers are used as components by limiting their internal hardware usages, that is, a server considered as a pComponent will see its CPU used but not its network adapter, while a server considered as a nComponent will have its network adapter used but only few CPU for its internal monitor. We used 2 servers as pComponents with an extCache of size 1 GB each, 1 server as nComponent with 1 GB of DRAM, 1 server as mComponent with 8 GB of RAM and finally 1 server as sComponent.

The Mellanox cards used in our infrastructure are identical to those used by the designers of LegoOS.[7] Thus, the latency is similar to that contained in the article which is of the order of 8 μs for message round-trip.

We downloaded the current available version of LegoOS from git[20] and deployed it in our infrastructure.

4.2 | Micro-benchmark evaluation

Basic evaluation. We use a simple TCP client/server application as micro-benchmark. The experimental procedure is as follow: the server is deployed on the system to be evaluated while the client is on a remote machine running a standard Linux. The client sends and receives a flow of fixed size messages to/from the server. This procedure is repeated with various message sizes from 128 to 2048 Bytes. Figure 7 presents the results with the latency and the throughput metrics. The first figure (a) presents latency results while the last figure plots the throughput (b).

TABLE 1 Servers characteristics.

CPU	Memory size	Ethernet card	Infiniband card
Intel(R) Core(TM) i7-3770	4 GB	Intel Corporation 82574L Gigabit	Mellanox MT27500
Intel(R) Xeon(R) CPU E5-1603 v3	8 GB	Intel Corporation 82574L Gigabit	Mellanox MT27500
Intel Core Processor (Haswell, no TSX)	3 GB	Intel Corporation 82574L Gigabit	Mellanox MT27500

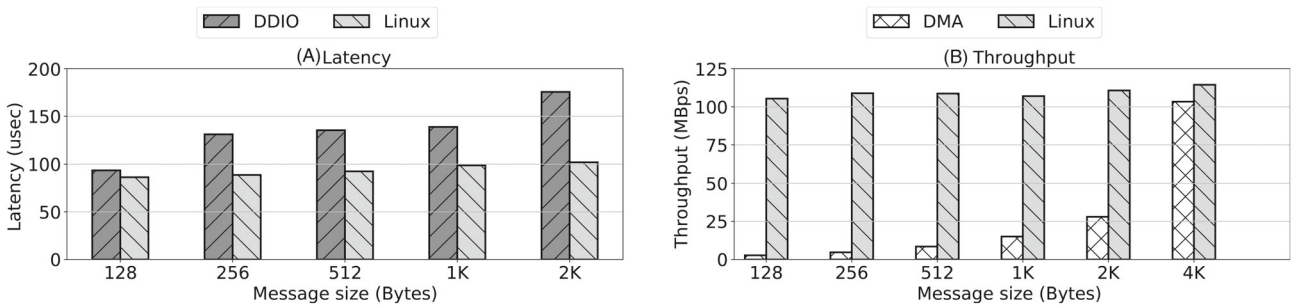

FIGURE 7 High level metrics for micro-benchmark evaluation (latency and throughput) on LegoOS.

For the latency evaluation, the sent messages are cached (in the extCache) and DDIO is used. With DDIO and cached data, there is a direct communication between the pComponent and nComponent, relying on the FIT protocol from LegoOS. We observe that overall, the latency is in the same order of magnitude as Linux. We also observe a degradation of latency when the message size increases, which is due to the internal buffer management of FIT (over Infiniband).

For the throughput evaluation, the sent message are located in memory (in the mComponent) and not cached (in the extCache) and DMA is used (as DDIO is not best suited for large data transfer as it heavily involves the pComponent). We observe a very low throughput for small messages as the high number of messages incurs many FIT based interactions with the mComponent. But for larger messages, the obtained throughput is very close to that of Linux (we extended the experiment with 4K messages) as less interactions with mComponent are needed. Notice that messages are fetched on demand from the mComponent and we did not implement message packing which would limit interactions with the mComponent (this is a track of improvement).

Overall, this evaluation shows that networking in a disaggregated architecture can achieve acceptable performance (close to Linux) and that the tuning of the rack interconnection system (here LegoOS FIT over Infiniband) is key to performance.

Connection establishment. The results presented above do not include connection establishment times which is the time spend when the client invoke the `connect` syscall. The connection times are very close (255 us for LegoOS and 252 us for Linux) as LegoOS only adds a RPC interaction between the `bsdSocketStub` in the pComponent and the `bsdSocketSkeleton` in the nComponent, which in turn runs the same connection opening as Linux.

Scalability evaluation. Regarding scalability, we evaluated the ability of the nComponent to handle an important communication load. We measured the amount of CPU (cores) needed to saturate our 1 GB Ethernet card, on emission and on reception (the other side being provided enough CPU). This is a means to dimension the maximum number of cores in the nComponent. We observed that 15% and 25% of a core were required respectively for emission and reception, reaching an effective bandwidth of 880 Mb/s. We also observed that additional NICs (and cores accordingly) in the nComponent allowed to scale up the nComponent and absorbed a more important load from pComponents.

Inter and intra pComponent communication. Inter and intra pComponent communication has been optimized as described in Section 3. Intra pComponent communication relies on Unix pipes while inter pComponent communication relies on direct communication with FIT (between pComponents). We evaluated the benefits of these optimizations (Figure 8). In these measurements, the exchanged data is cached in the extCache of the source/destination pComponent. We observe that intra pComponent latency and throughput are close to those of Linux loopback (with a monolithic Linux system). Regarding inter pComponent communication performance, the latency is good, but the throughput is disappointing, considering that FIT relies on Infiniband. We relied on a standard configuration of FIT and a fine tuning of FIT would allow to improve it. Here again, the rack interconnection system (here LegoOS FIT over Infiniband) is key to performance.

4.3 | Macro-benchmark

For the macro benchmark evaluation, we used the word count application on a spark-streaming like platform following the map-reduce paradigm. The benchmark consists of a single mapper and a single reducer: the mapper runs on the evaluated system (Legos or standard Linux) while the reducer runs on a remote machine running a standard Linux. Therefore, there is intense communication between the evaluated system and the outside because each pair <word, count> must be transmitted between the mapper and the reducer. The processing was done on 1.5 GB of data. The results obtained are shown in Figure 9. We can observe that the DDIO version performs in the same time as Linux, since each pair (which is very small) is sent over the network as soon as it is produced. The DMA version cannot reach the same performance level, because its interactions with the mComponent are too penalizing for small data exchanges. Sending larger blocks which don't fit in the extCache would be more favorable for DMA.

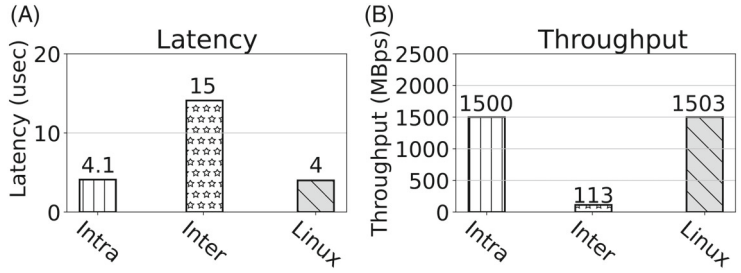

FIGURE 8 Intra and inter-pComponent evaluation. The figures present the latency (lower is better) and the throughput (higher is better).

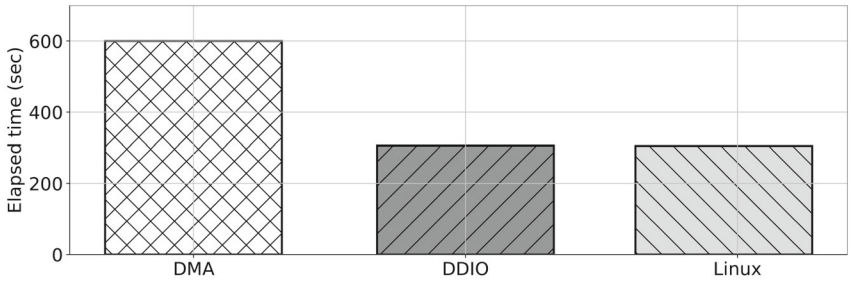

FIGURE 9 Streaming benchmark evaluation.

5 | RELATED WORK

In this section, we first review previous works on disaggregation in general, before focusing on OS level works in such environments.

5.1 | Disaggregation

Disaggregation splits the monolithic computer into a number of resource boards which communicate using a fast interconnect. This is inspired by various trends from the past such as diskless machines using network storage such as network block devices. Reference 8 is the first work on memory disaggregation in datacenters, extending a host memory with remote servers memory, thus decoupling CPU from memory allocations. In the same vein, several other works have studied memory disaggregation, such as Reference 1 which proposed enabling remote access to the memory of a suspended server. Not so long ago were studied fully disaggregated infrastructures, notably with storage rack disaggregation[21] and network requirements for resource disaggregation.[2,22,23] More precisely, Reference 24 carried out a study to estimate the latency and bandwidth requirements that the interconnect in disaggregated datacenters must meet to avoid degrading application-level performance with existing network designs. References 22 and 23 explored new architecture designs to integrate 1 Tbps silicon photonic, remote nonvolatile memory, and System-on-Chips (SoCs) to datacenter for disaggregation. Although these works are related to networking, they only deal with the hardware level unlike our work which includes a network stack implementation for a disaggregated OS.

Several manufacturers have proposed intermediate approaches for rack disaggregation based on microcomputers. As example we can cite Intel Rack-Scale,[25] AMD SeaMicro[15] and HP Moonshot.[16] All these works correspond to an intermediate step towards full disaggregation. With the dRedBox European project,[17] IBM and several academic institutions are currently trying to build a real disaggregated rack prototype, but only focus on hardware aspects.

5.2 | OSes in disaggregation and the splitkernel model

Several OS architectures try to respond to huge computing capacity environments and scalability issues: Multi-kernelOSes like Barrelfish[26] and Helios,[27] multi-instance OSes like Popcorn[28] and Pisces,[29] and finally distributed OSes.[30,31] These OS architectures, even though they look interesting and are designed to handle large infrastructures, are not suitable for disaggregated environments as they do not assume a resource centric infrastructure composed of single resource type boards. To the best of our knowledge, the only attempt to design an OS for a fully disaggregated infrastructure is LegoOS[7] which introduces the split-kernel model as a way of building OSes for disaggregated racks. The splitkernel model consists in designing the OS of a disaggregated rack as a disaggregated OS composed of several specialized and loosely-coupled OSes. The authors prototyped their model with *LegoOS* to demonstrate its feasibility. In its current state, LegoOS targets three types of hardware components: CPU, memory, and storage, but does not address network support. In this work we proposed an approach to handle network components in the splitkernel model and we carry out an implementation in LegoOS.

6 | CONCLUSION

Shan et al.[7] proposed, for the first time, the split-kernel model as an appropriate operating system design model for disaggregated datacenters. The authors validated their claim by prototyping the CPU, memory and storage parts of a split-kernel-based operating system. In the present paper, we studied the support of networking in a disaggregated rack in respect with the split-kernel model. We showed that this is not a straightforward task

because of the strong relationship between network devices and the CPU-memory couple. In addition, we studied how classical networking features such as DMA, DDIO or loopback can be efficiently taken into account in such an environment. We extended the original LegoOS prototype and evaluated our implementation using both micro- and macro-benchmarks. Our evaluations show that with the emulated disaggregated platform defined by LegoOS (that we reproduced), our network support implementation can perform in the same order of magnitude as a monolithic Linux server for latency and very close for bandwidth, leading to very close results for a big data benchmark. Therefore, a real disaggregated hardware (especially a dedicated interconnect instead of Inifiniband) would lead to a reasonable overhead while benefiting from the flexibility of disaggregation.

ACKNOWLEDGMENTS

This work was supported by the French *Agence nationale de la recherche* under the project ANR PicNic (ANR-20-CE25-0013 https://anr.fr/Projet-ANR-20-CE25-0013).

CONFLICT OF INTEREST STATEMENT

All authors declare that they have no conflicts of interest.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ENDNOTE

*For practicability purposes, DC disaggregation is always studied at the rack scale.

ORCID

Boris Teabe https://orcid.org/0000-0001-6528-8904

REFERENCES

1. Nitu V, Teabe B, Tchana A, Isci C, Hagimont D. Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. Proceedings of the 13th European Conference on Computer Systems; 2018:1–12.
2. Gao PX, Narayan A, Karandikar S, et al. Network requirements for resource disaggregation. Proceedings of the USENIX OSDI; 2016:249–264.
3. Amaro E, Branner-Augmon C, Luo Z, et al. Can far memory improve job throughput? Proceedings of the Fifteenth European Conference on Computer Systems; 2020:1–16.
4. Gu J, Lee Y, Zhang Y, Chowdhury M, Shinothers KG. Efficient memory disaggregation with INFINISWAP. Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation; 2017:649–667.
5. Ruan Z, Schwarzkopf M, Aguilera MK, Belay A. AIFM: high-performance, application-integrated far memory. Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation; 2020:315–332.
6. Zhang J, Ding Z, Chen Y, et al. GiantVM: a Type-II hypervisor implementing many-to-one virtualization. Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments; 2020:30–44.
7. Shan Y, Huang Y, Chen Y, Zhang Y. LegoOS: a disseminated, distributed OS for hardware resource disaggregation. Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation; 2018:69–87.
8. Lim K, Chang J, Mudge T, Ranganathan P, Reinhardt SK, Wenisch TF. Disaggregated memory for expansion and sharing in blade servers. Proceedings of the 36th Annual International Symposium on Computer Architecture; 2009:267–278.
9. Wang C, Ma H, Liu S, et al. Semeru: a memory-disaggregated managed runtime. Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation; 2020.
10. Tchana A, Lachaize R. Rebooting virtualization research (Again). Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems; 2019:99–106.
11. Glick M, Rumley S, Bergman K. Silicon photonics enabling the disaggregated data center. Proceedings of the Advanced Photonics 2018; 2018.
12. Network direct memory access. Accessed September 13, 2021. https://patents.google.com/patent/US7836220B2/en
13. Intel data direct I/O technology (Intel DDIO). Accessed September 13, 2021. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf
14. Lim K, Turner Y, Santos JR, et al. System-level implications of disaggregated memory. IEEE International Symposium on High-Performance Comp Architecture, Washington, DC; 2012:1–12.
15. EAMICRO, A. AMD SeaMicro SM15000 fabric compute systems. Accessed August 5, 2020. http://www.seamicro.com/
16. HP Moonshot. Accessed September 13, 2021. https://files.vogel.de/vogelonline/vogelonline/files/6284.pdf
17. Katrinis K, Syrivelis D, Pnevmatikatos D, et al. Rack-scale disaggregated cloud data centers: the dReDBox project vision. Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE); 2016. doi:10.3850/9783981537079_1014
18. Arm DynamIQ Shared Unit technical reference manual. Accessed September 13, 2021. https://developer.arm.com/documentation/100453/0300/functional-description/l3-cache/cache-stashing
19. What Is a SmartNIC. Accessed August 5, 2020. https://blog.mellanox.com/2018/08/defining-smartnic/
20. LegoOS git. Accessed September 13, 2021. https://github.com/WukLab/LegoOS
21. Legtchenko S, Williams H, Razavi K, et al. Understanding rack-scale disaggregated storage. Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems, Berkeley, CA; 2017; 2017: 2.
22. Asanović K. FireBox: a hardware building block for 2020 warehouse-scale computers. Proceedings of the USENIX Association, Santa Clara, CA; 2014.
23. Walraed-Sullivan M, Padhye J, Maltz D. Theia: simple and cheap networking for ultra-dense data centers. Proceedings of the 13th ACM Workshop on Hot Topics in Networks; 2014.

24. Gao PX, Narayan A, Karandikar S, et al. Network requirements for resource disaggregation. Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation; 2016: 249–264.
25. Intel rack scale. Accessed September 13, 2021. https://www.intel.com/content/dam/www/public/us/en/documents/guides/architecture-spec-v2-4-guide.pdf
26. Baumann A, Barham P, Dagand PE, et al. The Multikernel: a new OS architecture for scalable multicore systems. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles; 2009:29–44.
27. Nightingale EB, Hodson O, McIlroy R, Hawblitzel C, Hunt G. Helios: heterogeneous multiprocessing with satellite kernels. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. New York, NY: Association for Computing Machinery; 2009:221-234.
28. Barbalace A, Sadini M, Ansary S, et al. Popcorn: bridging the programmability gap in heterogeneous-ISA platforms. Proceedings of the Tenth European Conference on Computer Systems; 2015:1–16.
29. Ouyang J, Kocoloski B, Lange JR, Pedretti K. Achieving performance isolation with lightweight Co-kernels. Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing. New York, NY: Association for Computing Machinery; 2015:149-160.
30. Govil K, Teodosiu D, Huang Y, Rosenblum M. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Trans Comput Syst*. 2000;18(3):229-262. doi:10.1145/354871.354873
31. Baskett F, Howard JH, Montague JT. Task communication in DEMOS. *SIGOPS Oper Syst Rev*. 1977;11(5):23-31. doi:10.1145/800214.806544

How to cite this article: Ekane B, Tchana A, Hagimont D, Teabe B, De Palma N. Networking in next generation disaggregated datacenters. *Concurrency Computat Pract Exper*. 2023;35(21):e7702. doi: 10.1002/cpe.7702