FlexChain: An Elastic Disaggregated Blockchain

Chenyuan Wu
University of Pennsylvania
wucy@seas.upenn.edu

Mohammad Javad Amiri
University of Pennsylvania
mjamiri@seas.upenn.edu

Jared Asch
University of Pennsylvania
jasch16@seas.upenn.edu

Heena Nagda
University of Pennsylvania
hnagda@seas.upenn.edu

Qizhen Zhang
University of Pennsylvania
qizhen@seas.upenn.edu

Boon Thau Loo
University of Pennsylvania
boonloo@seas.upenn.edu

ABSTRACT

While permissioned blockchains enable a family of data center applications, existing systems suffer from imbalanced loads across compute and memory, exacerbating the underutilization of cloud resources. This paper presents *FlexChain*, a novel permissioned blockchain system that addresses this challenge by physically disaggregating CPUs, DRAM, and storage devices to process different blockchain workloads efficiently. Disaggregation allows blockchain service providers to upgrade and expand hardware resources independently to support a wide range of smart contracts with diverse CPU and memory demands. Moreover, it ensures efficient resource utilization and hence prevents resource fragmentation in a data center. We have explored the design of XOV blockchain systems in a disaggregated fashion and developed a tiered key-value store that can elastically scale its memory and storage. Our design significantly speeds up the execution stage. We have also leveraged several techniques to parallelize the validation stage in FlexChain to further improve the overall blockchain performance. Our evaluation results show that FlexChain can provide independent compute and memory scalability, while incurring at most 12.8% disaggregation overhead. FlexChain achieves almost identical throughput as the state-of-the-art distributed approaches with significantly lower memory and CPU consumption for compute-intensive and memory-intensive workloads respectively.

PVLDB Reference Format:
Chenyuan Wu, Mohammad Javad Amiri, Jared Asch, Heena Nagda, Qizhen Zhang, and Boon Thau Loo. FlexChain: An Elastic Disaggregated Blockchain. PVLDB, 16(1): 23 - 36, 2022.
doi:10.14778/3561261.3561264

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/chenyuanwu/FlexChain.

1 INTRODUCTION

Blockchain systems, in particular, *permissioned* blockchain systems, have enabled a new class of data center applications, ranging from contact tracing [42], crowdworking [14], supply chain assurance [15, 53], and federated learning [43]. The popularity of these services has motivated the cloud providers, e.g., Amazon [6], IBM [7], Oracle [8] and Alibaba [5], to provide *Blockchains-as-a-Service* (BaaS) [21].

BaaS providers need to guarantee high-throughput services over various transaction workloads. While auto-scaling can improve throughput, the heterogeneity of resource requirements is still a challenge. In the blockchain domain, smart contracts have been traditionally used to perform *memory-intensive* operations such as retrieving customer profiles stored in large databases [22]. However, recent applications of blockchains in food dissemination [9, 39, 60], carbon pricing [20] and communication security [28] require smart contracts to have the ability of performing classification and regression. These tasks, usually accomplished by machine learning algorithms, demonstrate the need for supporting *compute-intensive* workloads in blockchains. Moreover, a smart contract can interchangeably be compute- and memory-intensive at different times and execution stages. Deploying smart contracts with diverse hardware demands in a BaaS setting requires rethinking existing system architectures to enable the flexibility of scaling compute and memory resources independently and elastically.

Data center resources have been traditionally arranged in monolithic servers. These servers contain limited compute, memory, and storage resources for processing independent jobs or partitions of jobs. Resource disaggregation is a recent trend in data center design [31, 46, 61, 62]. In a *disaggregated data center* (DDC), resources are reorganized from servers to physically distinct pools that are dedicated to processing, memory or storage. All nodes, regardless of type, might include a small amount of ancillary processing and memory resources to run simple control software. As a result of disaggregation, processing nodes will continually “page” memory from remote nodes into and out of its small on-board working set, write chunks to remote disks, or farm out tasks to remote CPUs. To each program running in this environment, the system provides the illusion of a near-infinite pool of any resource [29].

Resource disaggregation is an ideal solution to the diverse resource requirements of BaaS workloads. DDCs allow operators to upgrade and expand each resource independently. For instance, if a new processor technology becomes available or if compute-intensive smart contracts require additional CPUs, the operator can deploy additional compute nodes without upgrading memory nodes or worrying about compatibility between different components. Likewise, memory-intensive smart contracts can scale up by adding more memory nodes. Moreover, DDCs promote efficient resource utilization and prevent fragmentation because they can allocate different resources separately from the corresponding pools.

Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:10.14778/3561261.3561264

This feature can improve the resource consumption of BaaS workloads. For example, suppose a customer's blockchain application requires more memory. Instead of rigidly allocating or migrating to a VM with a fixed CPU count and memory, a DDC-enabled blockchain service can simply provide more memory resources to the application on the fly from the memory pool.

In this paper, we present *FlexChain*, a novel *disaggregated* permissioned blockchain service that leverages recent innovations in DDCs. FlexChain's design is based on a few key insights. We observe that the XOV architecture used in Hyperledger Fabric [16] and several other permissioned blockchain systems [33, 45, 47] lends itself naturally to disaggregation. First, the simulation (X) and most of the validation (V) stages are inherently parallelizable, leading to a design where we can auto-scale these stages to arbitrary numbers of CPU cores in the compute pool. Second, most of the blockchain state resides in key-value stores, which can be hosted in the memory pool. This design will reduce the need of storing and retrieving data to and from the disk because there is sufficient memory for buffering the working sets of transactions.

Hence, FlexChain designs a tiered key-value store and leverages auto-scaling to achieve both memory and compute elasticity. Remote memory connected by a high-speed network is used as a cache for data stored on disk. Read/write and buffer eviction protocols all need careful designs to ensure correctness. As smart contract executions are amenable to parallelization in a DDC, sequential conflict check and world state update in the validation stage of XOV would become the bottleneck. We further adopt a concurrency control mechanism to parallelize the validation stage.

We experimentally evaluate both the operational and performance benefits of disaggregation. Our comparison between FlexChain and traditional distributed architectures demonstrates that disaggregating complex distributed systems like blockchains is not only feasible (achieving comparable or even better performance), but also preferable (achieving independent elasticity for hosting diverse workloads and better resource utilization).

The key contributions of this paper are:

- **Blockchains on DDCs.** For the first time, we explore how different components in a permissioned blockchain system should be redesigned for resource disaggregation.
- **Tiered world state.** We design a *tiered* key-value store based on disaggregated memory and storage that provides elasticity for scaling the blockchain world state.
- **Parallel validation stage.** We leverage a dependency-graph-based concurrency control mechanism to fully parallelize the validation stage.
- **Prototype and evaluation.** We implement a prototype of FlexChain and evaluate its benefits with realistic benchmarks. Our results demonstrate that FlexChain achieves optimal resource utilization for diverse blockchain workloads with minimal performance overhead.

The rest of this paper is organized as follows. Section 2 briefly discusses the background and motivation behind FlexChain. The FlexChain architecture is introduced in Section 3. Sections 4 and 5 present disggregated world state and parallel validation in details. Section 6 evaluates the performance of FlexChain. Section 7 discusses related work, and Section 8 concludes the paper.

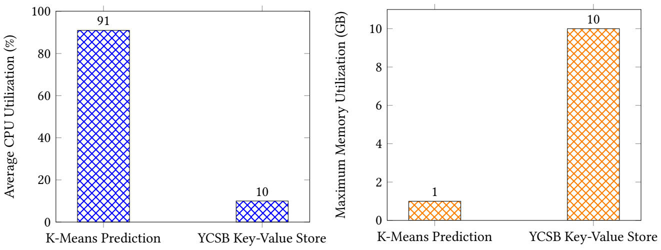


Figure 1: Profiling results of compute-intensive and memory-intensive smart contracts (on an Ubuntu server with a 20-core Intel(R) Xeon(R) Silver 4114 CPU and 64 GB of RAM).

2 MOTIVATION AND BACKGROUND

We provide a brief background on emerging smart contracts, DDCs, and blockchain's XOV architecture.

2.1 Emerging Smart Contracts

In data-intensive smart contracts, as shown in BlockBench [22], increasing the number of users/records stored on the blockchain as well as processing large records (common in government data, medical data, etc.) level up disk activities significantly. Disk paging, however, should be avoided in order to maintain performance at a high level, which can be made possible by enlarging the memory to accommodate the working sets of transactions (as shown in Figure 7). This leads to the memory-intensive characteristic of many smart contacts. This section mainly focuses on outlining an *emerging* class of compute-intensive smart contracts.

The World Food Program (WFP) has put together a pilot program in the Azraq refugee camp in Jordan where refugees have their identities stored on a blockchain [9, 39, 60]. The blockchain is used not only for food dissemination but also for detecting possible corruption and food siphoning. The blockchain defines computer vision and image recognition functions in its smart contracts for various purposes. In particular, the blockchain uses retina scans for identifying refugees, K-Means clustering for image segmentation, and k-nearest neighbor (KNN) calculations to determine food distribution strategies in a decentralized fashion.

Image segmentation (commonly accomplished with K-Means clustering and its variants) and its applications such as face detection, fingerprint detection, iris recognition [37], pedestrian detection, and locating tumors in medical imaging [57] have also been used in many smart contracts [40]. In general, machine learning and artificial intelligence in smart contracts have been proposed for a wide range of applications, including carpooling [20] (to match passengers and drivers [25]), countermeasure against DDoS attacks in 5G [28], games [20], and carbon pricing [20]. AI-driven blockchains have also received some commercial traction [10].

Finally, as an appealing application, consider a healthcare use case where the data is stored on a blockchain [12, 38] to provide stronger security properties. Machine learning is useful for classifying medical documents or other forms of healthcare data (e.g., EEG and ECG) while blockchains safeguard the integrity of the data.

2.2 Profiling Smart Contracts

To motivate a disaggregated design in the blockchain domain, we have characterized two smart contracts running on Hyperledger

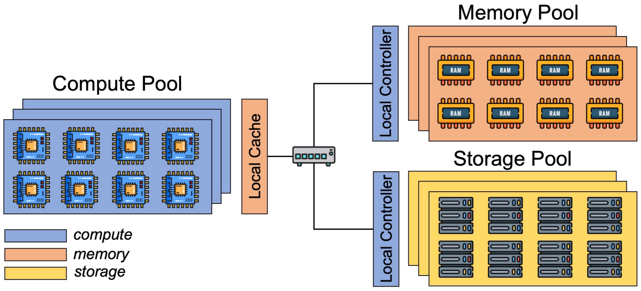


Figure 2: An illustration of resource disaggregation. Same type of resources are centralized in a *resource pool*. Resource pools are disaggregated and connected by a fast network.

Fabric [16]. Figure 1 depicts the results where the Fabric is configured with two peers (deployed on separate docker containers), one orderer, and uses LevelDB [1] as its underlying database.

- **Compute-intensive K-Means prediction contract.** The K-Means prediction contract starts off with 20 randomly seeded clusters for 10000-dimension data. Prediction queries are issued at a rate of 10 per second, and each query represents a user submitting a request with a new vector. The contract finds and returns the nearest cluster from the pre-trained clusters. The CPU utilization and memory utilization are **91%** and **1 GB** respectively. In this experiment, the K-Means training is done off-chain. If this were done on-chain, the CPU utilization would be even higher.
- **Memory-intensive key-value storage contract.** In the second smart contract, each transaction interchangeably initiates 80% reads, 10% insertions, and 10% updates on records of size 80 KB at a rate of 1500 transactions per second. CPU and memory utilization are **10%** and **10 GB** respectively.

The input parameters (i.e., the high dimensionality for K-Means and large records for KV-store) that are used in the above experiments are consistent with the emerging use cases of smart contracts described in Section 2.1. We observe extreme variation in the utilization of CPU and memory resources. These results demonstrate the need for a more efficient solution for scaling smart contracts in a BaaS environment — *one that can scale CPU and memory resources flexibly and independently.*

2.3 Resource Disaggregation

Resource disaggregation is an architectural style where the resources of a data center, traditionally spread across every server, are partitioned into physically distinct pools of resources connected with a fast network fabric such as RDMA over InfiniBand, as illustrated in Figure 2. While today's data centers commonly support storage disaggregation, a defining feature of DDCs is the full-scale disaggregation of resources including memory. The operational benefits of DDCs are vast, such as independent expansion, independent allocation, and independent failures [17, 31, 46, 48, 55, 62].

Pools hosting each type of resource typically contain a small amount of other resources, e.g., low-frequency CPUs in the memory/storage pools that manage local resources and process accesses, or a modest amount of DRAM in the compute pool that caches data. However, coordination across pools spanning different resource types is needed.

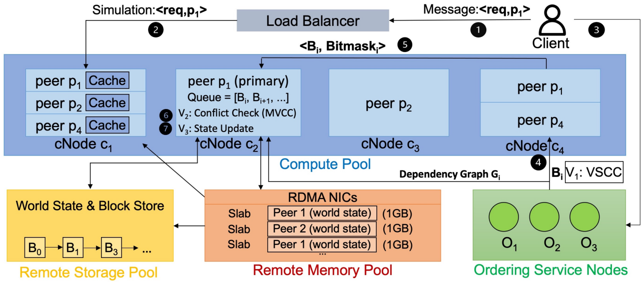


Figure 3: FlexChain architecture. For simplicity, we only present the workflow of logical peer p_1.

In exchange for those benefits, DDCs convert a subset of what used to be local memory and device accesses to remote accesses. While the latest InfiniBand networks are undoubtedly very fast (sub-600 ns latency at 200 Gb/s [11]) and some proposals have advocated for new network substrates [48], both are, nevertheless, slower than accessing resources on the same motherboard. However, these performance penalties are offset by the substantial operational benefits one can achieve using DDCs.

3 FLEXCHAIN ARCHITECTURE

Figure 3 shows the overall architecture of FlexChain, the first disaggregated permissioned blockchain system. In this section, we first introduce the key differences between FlexChain and a generic XOV blockchain system running on monolithic servers, and then describe the transaction flow of FlexChain, followed by the cross-data center deployment of FlexChain.

3.1 Comparing FlexChain with Generic XOV

Traditional XOV blockchains, e.g., HyperLedger Fabric [16], operate at the level of *peers*, where each peer runs on a single monolithic machine. XOV blockchains maintain the *world state* in a versioned local key-value store that is replicated on every peer. The world state stores all the information needed by blockchain users, e.g., medical records or bank accounts. Each peer retrieves and updates its own copy of the world state in the execution (X) and validation (V) phases without any synchronization with other peers. These interactions are the major data-intensive operations in XOV blockchains, and might become the bottleneck when memory is insufficient (Section 6.3). Data that is more transient in nature (e.g. read/write set, received blocks, intermediate data during computation) is unlikely to be reused across transactions and is maintained in the main memory instead of the key-value store. In traditional XOV blockchains, a separate set of nodes, called ordering service nodes (OSNs), are used to establish consensus on the order of transactions. These OSNs also run on monolithic servers.

Similar to generic XOV blockchains, FlexChain operates at the level of *virtual peers*, or *logical peers*[1]. However, there are two key differences between FlexChain and generic XOV:

- **Peer disaggregation.** In FlexChain, a logical peer runs in a disaggregated setup on multiple compute nodes (**cNode**), memory nodes (**mNode**), and storage nodes (**sNode**). Each hardware component (i.e., compute, memory, or storage) can also provision

[1] In the rest of the paper, we use "peer" to denote a server node in generic XOV, as well as a logical peer in FlexChain.

resources for more than one logical peer. FlexChain users interact with a peer in the XOV paradigm, but they do not know which physical components their smart contracts run on. In FlexChain, ordering service nodes continue to run on traditional monolithic server machines, since they are bottlenecked by the consensus protocol that is bandwidth-intensive. Disaggregating ordering service nodes would overburden the network.

- **World state disaggregation.** Instead of using a local key-value store, FlexChain disaggregates world state using a three-tier architecture that combines compute-local memory, remote memory, and remote storage. In particular, temporary data is kept in local memory, because such data is small and unlikely to be reused across transactions. Spilling temporary data to remote memory can only complicate memory management.

3.2 Transaction Lifecycle in FlexChain

We now describe Figure 3 in detail, by tracing through the lifecycle of a FlexChain transaction. FlexChain consists of a load balancer, a compute pool, a memory pool, a storage pool, and ordering service nodes. The load balancer is responsible for dispatching client requests to different cNodes in the compute pool. The cNodes perform smart contract execution (X), endorsement policy evaluation (V_1, i.e., the first stage in validation), conflict check (V_2), and world state update (V_3) tasks as assigned by the load balancer and OSNs. The mNodes, on the other hand, are responsible for storing each peer's own copy of world state. The sNodes store the blockchain ledger and the world state evicted from mNodes. OSNs are responsible for ordering transactions and optionally constructing a dependency graph for each block.

Step ❶: To initiate a transaction, a client sends a message <req, p_1> to the load balancer where req is the request content and p_1 is the receiver peer determined by the endorsement policy. The load balancer is a fault-tolerant component that serves as the monitoring and management plane in the BaaS. It keeps track of the utilization and status of all resource nodes, i.e., cNodes, mNodes, and sNodes, as well as the mapping from resource nodes to logical peers. The utilization and status are updated via heartbeats, and the mapping is updated when the user provisions/destroys a new logical peer and when a peer scales up/down its resources.

Step ❷: FlexChain allows multiple cNodes for a peer. In our example, cNodes c_1, c_2, c_4 are provisioned for peer p_1, so they can perform computation stages (X, V_1, V_2, and V_3) belonging to p_1. When processing compute-intensive workloads, FlexChain can allocate even more cNodes via elastic scaling. The load balancer dispatches the received request to the least loaded cNode belonging to p_1. For the transaction in the example, cNode c_1 is selected. The set of provisioned CPU cores on cNode c_1 then perform the execution phase of req for p_1, by interacting with p_1's copy of the world state that is primarily stored in the remote memory pool. At this stage, cNode c_1 only reads from the world state and computes the resulting read set and write set. It does not update its world state. When the execution is over, cNode c_1 sends the endorsement, i.e., the signed read and write sets, back to the client.

Step ❸: Upon receiving enough endorsements for req (as specified by the endorsement policy), the client assembles a transaction and submits it to the OSNs. The transaction contains the set of endorsements, the smart contract operation that includes parameters, and other metadata. The ordering service totally orders transactions and generates a transaction block. Optionally, it constructs a dependency graph for each block.

Step ❹: Once a transaction block B_i has been generated, OSNs send the block to every logical peer for validation and commitment. For example, in order to send block B_i to p_1, OSNs send block B_i to the current most spare cNode (cNode c_4 in our example) belonging to p_1, so that the cNode can perform endorsement policy evaluation (V_1) on the transactions within the block in parallel. Endorsement policy evaluation is a compute-intensive task as pointed out by [52, 56]. The complexity of the endorsement policy, which is defined by the application, affects the time and resources taken to verify signatures and evaluate the policy.

Step ❺: Upon finishing the endorsement policy evaluation, cNode c_4 sends a message <B_i, $Bitmask_i$> to the primary cNode for p_1 (cNode c_2 in our example), where $Bitmask_i$ is the consequent *validity bit mask* of block B_i. Every peer in FlexChain has a primary cNode, which is the first provisioned cNode for this peer and selected by the load balancer when the user launches the peer.

The load balancer can detect the primary failure, select a new primary for the affected peer, and immediately inform the OSNs of the update. Since FlexChain uses a write-through design for the local data cache, as described in Section 4, the new primary can seamlessly utilize the world state stored in the remote memory pool as it is guaranteed to be up-to-date.

Steps ❻ and ❼: There is a centralized *validation manager* running on the primary cNode c_2, which is responsible for performing/coordinating the last two stages of validation: conflict check (V_2) and world state update (V_3). During conflict check of a transaction, the versions of the keys recorded in the read set field are compared to those in the current world state in the remote memory pool. If the versions do not match, the transaction is marked as invalid. The world state update only happens for a transaction if it is marked as valid in both V_1 and V_2. If so, its write set is applied to the current world state. After V_1, V_2, and V_3 are performed for all transactions in block B_i, the primary cNode c_2 writes B_i with its updated validity bit mask into remote storage.

Parallel validation[2]**.** FlexChain supports both sequential and parallel validation. Sequential validation has already been used in XOV blockchains while parallel validation is an optimization we develop for FlexChain (Section 5 describes the details).

In sequential validation, the validation manager itself validates (V_2 and V_3) transactions within each block sequentially. However, in parallel validation, in addition to sending block B_i to cNode c_4 in step ❹, OSNs also send a dependency graph G_i that captures the data dependencies between the transactions in block B_i to the primary cNode c_2. The validation manager then uses the dependency graph G_i to spawn multiple validation workers. Different workers perform V_2 and V_3 for different transactions in parallel. The validation manager monitors the load and, if necessary, dispatches validation workers to other cNodes provisioned for this peer.

The disaggregated design of FlexChain enables elastic scaling when processing workloads of various resource demands. For example, when p_1 is serving a compute-intensive workload, the load

[2] We refer to the second and third stages of validation, namely conflict check and state update (V_2 and V_3 in Figure 3). The first stage of validation (V_1) is already parallelized.

balancer monitors the average utilization of all provisioned cores for p_1. If the utilization rises above a certain threshold, the load balancer launches an idle compute node for p_1. Similarly, when p_1 is serving a memory-intensive workload, the space manager on the memory controller monitors the memory pressure and allocates more slabs for p_1 in the memory pool if necessary.

3.3 Cross-Data Center Deployment

A permissioned blockchain system often involves several parties, such as enterprise users (which we call peers) typically located on multiple data center infrastructures. In that spirit, we demonstrate a cross-data center deployment of FlexChain where peers are distributed in different data centers. For instance, a logical peer p_1 can be hosted in data center DC_1, p_2 in DC_2, and p_3 in DC_3, etc., where *each* data center is still organized in the form of three RDMA-connected local resource pools. This approach is made possible by the design choice of disaggregating peers: while a single logical peer can not span multiple data centers due to the stringent network requirement between resource pools, different logical peers can be freely hosted in different data centers. This way, FlexChain still maintains its elasticity and high resource utilization. Specifically, in BaaS, there are a large number of blockchain instances running simultaneously, where peers of each instance may span across multiple data centers, e.g., b_1p_1 (representing peer p_1 in blockchain instance b_1), b_2p_1, b_3p_2 are running in DC_1 while $b_1p_2, b_2p_2, b_2p_3, b_3p_1$ are running in DC_2. Using FlexChain, each data center can still flexibly scale up/down resources for each peer according to the workload that it processes to improve performance and avoid wasting resources.

Each data center has its own load balancer in a multi-data center deployment. These load balancers periodically exchange resource utilization statistics within each data center and peer allocation information (which data center hosts which peers). The statistics help to decide where to launch a new peer/blockchain instance, as a higher level of load balancing across data centers. With the peer allocation information, clients can submit their requests to the nearest data center (or a cached target known to host the endorsing peer). The receiver data center will forward the request to whoever hosts the peer if needed.

It is also possible to deploy OSNs over multiple data centers. However, the consensus protocol that FlexChain currently uses, i.e., Raft [41], is not designed for Geo-distributed deployment. Separating the OSNs would require the implementation of high-performance global-scale consensus protocols such as GeoBFT [36], which we leave for future work.

4 DISGGREGATING WORLD STATE

This section discusses how FlexChain disaggregates its world state. The blockchain world state has been implemented using a key-value store in the XOV architecture. In FlexChain, each peer's copy of the world state might span the local cache on compute nodes, the remote memory pool, and the remote storage pool. In addition to serving as a regular application-level buffer, the local cache on a compute node is also pre-registered to the NIC as an RDMA memory region. This arrangement avoids extra memory copy when performing RDMA operations to remote memory, and thus can speed up world state accesses. The world state is primarily stored in remote memory, which is slower than the local cache but much faster than remote storage. Although users can scale up remote memory regardless of compute resources, the expense of using DRAM is significantly higher than that of using secondary storage such as disks. For that reason, FlexChain allows its users to set a limit on the remote memory size according to their own budget. Once approaching the limit, some cold world state will be evicted from remote memory to remote storage to make room for the hot state that is newly written. More importantly, it is necessary to store the world state in remote persistent storage in order to achieve fast recovery when a memory node fails.

In the remainder of this section, we first describe each component in the disaggregated world state of FlexChain in greater detail, followed by the read/write procedure, and then the buffer eviction. Our discussions are **with respect to one logical peer**, p_1 in our example. All peers follow the same steps to work with the disaggregated world state.

4.1 Overview

Figure 4 shows p_1's view of the disaggregated world state spanning remote memory, remote storage, and the local memory on its compute nodes.

4.1.1 Remote Memory. We manage remote memory by dividing it into control and data planes as follows.

Data plane. On the data plane, remote memory is managed in the unit of slab (1GB each in our implementation). To reduce the overhead of memory registration that requires pinning physical RAM, these slabs are registered to RDMA NICs when memory nodes boot up. Within a slab, different versions of a record form a linked chain. This helps compute nodes to get the latest value by walking the chain, without the need of consulting the control plane every time. Doing so avoids overburdening the memory controller that has limited CPUs.

Control plane. The control plane includes a hash table that maps a record to its latest address on the data plane (also called remote address), the address manager, the space manager, the bookkeeping agent, and the eviction manager. The latter four modules are running as different threads on the memory controller. Sections 4.2 and 4.3 describe them in detail.

4.1.2 Remote Storage. Remote persistent storage maintains the blockchain ledger and the world state.

Blockchain ledger. For the sake of stability, each peer in FlexChain appends its latest block in the blockchain ledger to the block store in remote storage.

World state. Unlike generic XOV where each peer maintains its world state on a persistent disk, FlexChain maintains the major part of the world state on volatile remote memory. Consequently, when the remote memory fails, a peer needs to replay the entire chain of blocks stored on remote storage to recover the world state. FlexChain avoids this by offloading block materialization to remote storage, using the multi-layer structure and compaction of SSTables [19] (an efficient on-disk representation of key-value pairs). Specifically, the block store keeps a *savepoint*, which is the *id* of the latest block that has been materialized into level 0 of the world state. In the materialization process, the write sets of all valid transactions in a block (as indicated by the validity bitmask) will

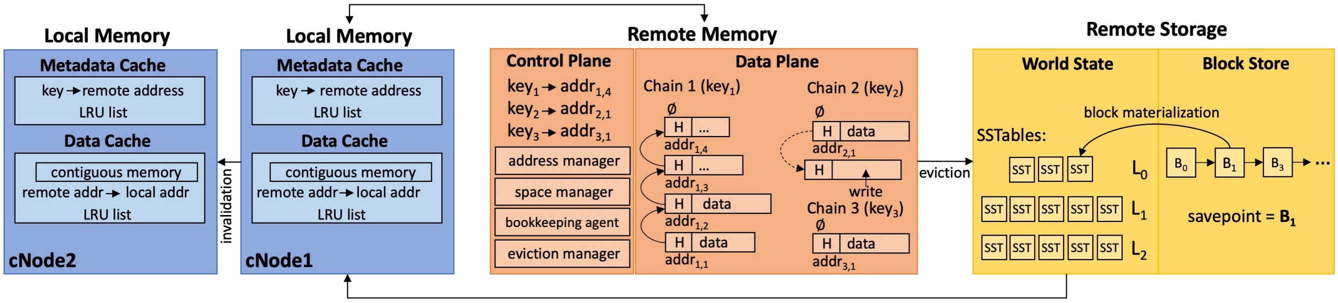


Figure 4: p_1's view of the disaggregated world state. For simplicity, we only show the data flow related to cNode1.

be added to an SSTable in sequence. With this design, the peer recovers by materializing only subsequent blocks in its blockchain ledger whose id is larger than the *savepoint* into level 0. After materializing its own ledger, the recovering peer asks other peers in its organization about the current block height and fetches missing blocks. The fetched blocks are verified, appended to the peer's own blockchain ledger, and then materialized similarly.

The materialized SSTables on level 0 have overlaps, i.e., multiple versions of a record are all kept on disk and can spread across different SSTables. As a result, when reading a record from remote storage, we need to iterate through all SSTables on level 0 in the worst case. If we can reduce data overlap, the number of SSTables on level 0 will decrease, which facilitates reading a record from remote storage after a peer recovers from memory failures. To that end, the storage nodes perform major compaction in the background. The major compaction process clears every SSTable in level 0 and combines them into new SSTables in level 1, where there are no overlaps. The multi-level structure and compaction algorithms we use here are similar to LevelDB [1]. The key difference is that when a failure occurs, our offloaded materialization process avoids replaying from the beginning of the log.

4.1.3 Local Memory. Each compute node has a metadata cache and a local data cache.

Metadata cache. In the compute-local memory, the metadata cache maps a key, e.g., key_1, to its remote address on the chain. If key_1 has a cache hit, this prevents the compute node from contacting the address manager on the control plane of remote memory when reading the record. The mapping for key_1 is updated when the compute node either writes a new record for key_1, or observes an update made by other compute nodes (when it performs a chainwalk to read the latest version). The metadata cache also keeps an LRU list to keep track of LRU keys on this compute node.

Local data cache. The local data cache contains a contiguous memory region that is registered to the RDMA NIC in advance, in order to perform RDMA transfers. It also serves as a cache for smart contracts to accelerate world state lookups. FlexChain uses an address hash table and an LRU list to manage the local data cache. The address hash table maps a remote address to an iterator that points to an entry in the LRU list. The entry contains the remote address, the corresponding local address in the registered memory region, and other control information. With this design, when FlexChain wants to read a record stored in remote memory, it first looks up its local data cache to see if that buffer has already been cached locally.

4.2 Read/Write Procedure

RDMA supports *one-sided* and *two-sided* communication patterns. One-sided RDMA operations allow one node to directly access the memory on another node without involving the latter's CPUs. Two-sided RDMA operations involve both sender's and receiver's CPUs, where both sides need to either poll the completion queue or use the slower event-driven notifications. Thus, one-sided RDMA usually has better performance and consumes fewer CPU resources on the receiver node. In our read/write procedure, we use one-sided RDMA (to a feasible extent) to optimize performance and minimize the burden of CPUs on remote memory controllers.

Read procedure. The compute node first looks up the metadata cache to see if the remote address of the key it wants to read is already known. If not, it issues a two-sided RDMA request to ask the *address manager* on the control plane. If the remote address is not found on the control plane either, the compute node asks the storage node to search for this key on SSTables. If the record is in remote memory and hence its remote address is known, the compute node looks it up in the local data cache. If there is no local cache of this record, the compute node will allocate a buffer in the local data cache and perform a one-sided RDMA read to fetch the record from the remote buffer. The compute node next checks the validity flag stored in the header field (which is set to `False` during buffer eviction). If it is invalid, the compute node will retry the read procedure after a backoff time. Once a valid record has been cached in the local data cache, the compute node checks if the header is `NULL` (which means this is the latest version); if not, it performs a chainwalk by repeatedly fetching the next remote address specified by the header, until finding the latest version. Finally, both local metadata cache and data cache are updated.

Write procedure. FlexChain performs *out-of-space* updates as follows. When updating a record, FlexChain writes it into a newly allocated remote buffer, and then links it to the old records on the chain of this record. This design allows FlexChain to deal with variable-length key-value records easily and ensures writes do not block reads on the data path. Concurrent reads and writes to the same key are common in both generic XOV blockchains and FlexChain, due to their optimistic concurrency control mechanism.

The compute node first issues a two-sided RDMA request to the *space manager* on the control plane, asking for a free buffer to write the new record in remote memory. It then allocates a buffer in the local data cache in order to perform the RDMA transfer. After writing the record in the local data cache and updating the metadata cache, the compute node issues a one-sided RDMA write request (with immediate value) to write the record into remote memory.

The *bookkeeping agent* on the control plane will be notified with the immediate value, which embeds the buffer offset and length. The agent then locks the written key in the control plane and checks if this key already exists (as it might have been evicted) in the control plane. If so, the record is linked to the previous version, and the hash table on the control plane is updated. If this key does not exist, a new entry will be added to the hash table. The agent then releases the lock and sends a reply to the compute node. Once the compute node gets a reply from the bookkeeping agent, it invalidates other compute nodes provisioned for the same peer, by notifying them of the updated key and new remote address. Upon receiving an invalidation request, a compute node checks if its local data cache has a cached record for this key. If it has and the header is `NULL`, the compute node will update the header in the cached record to point to the new remote address. A record is considered as committed only if the compute node has received all invalidation responses. This mechanism ensures that every committed record will be seen by other compute nodes, preventing them from reading outdated cached records. Optionally, under an update-heavy and skewed workload, in-memory garbage collection can be enabled to recycle old-version records in the background.

4.3 Buffer Eviction

The buffer eviction for remote memory is an expensive operation because it needs to write dirty records to SSTables in the remote storage pool, which involves both disk flushing and network round trips. However, this is an inevitable operation when the amount of memory used has reached the upper bound specified by the user. To alleviate its negative performance impact on the critical read/write path, FlexChain appoints a background *eviction manager* to periodically perform buffer eviction. The eviction manager is triggered if the number of free addresses drops below a certain threshold.

Due to the use of one-sided RDMA operations, the control plane is not able to keep an LRU list of keys to evict. To resolve this, once the eviction manager is triggered, it first asks the compute nodes provisioned for the peer for the locally recorded cold keys, and then unions them to derive the keys to evict. The eviction manager next locks the keys to evict on the control plane and sets the validity flag in the record headers to `False`. This ensures atomicity during the eviction process by preventing the evicted records from being updated or read. Otherwise, there might be inconsistencies among compute nodes, memory nodes, and storage nodes, e.g., a newly written record is considered as evicted on the control plane, but it has not been flushed to remote storage. The eviction manager then sends the latest version of the evicted records to remote storage and writes them into SSTables on level 0. Once the persistence to SSTables completes, the eviction manager deletes these keys from the control plane and invalidates them on the compute nodes provisioned for the peer. Specifically, the manager notifies the compute nodes to mark their locally cached remote addresses and records as outdated. The evicted buffers are now marked as free and can be re-allocated by the space manager. Finally, the locks are released, and the validity flags are set to `True`.

The eviction procedure implicitly interacts with the block materialization process, since both of them write to SSTables in remote storage. To save disk space and improve eviction performance, when adding a record to SSTables during eviction, FlexChain compares the version of the record (represented by the block id and transaction id) with the *savepoint*. If the block is older than the *savepoint*, FlexChain skips the record since it has already been persisted. More importantly, to avoid stale versions overwriting the latest version, FlexChain keeps track of the evicted keys along with their versions. During block materialization, FlexChain checks the current block id and its keys. If there is a newer version of the key that has already been evicted, FlexChain skips the materialization of the key. This is necessary since during the compaction, records newly added to SSTables will overwrite previous records of the same key.

5 PARALLELIZING VALIDATION

As discussed in Section 3, the validation phase consists of three main steps: endorsement policy evaluation (V_1), conflict check (V_2), and world state update (V_3). FlexChain can dynamically scale up CPU cores and remote RAM according to the workload. As a result, the execution phase (X) and endorsement policy evaluation (V_1) are no longer the bottleneck. However, unlike the execution phase that processes transaction proposals in parallel, the traditional design of the validation manager processes transactions within a block sequentially: it picks transaction t_i, performs conflict check (V_2) and, if valid, updates the world state (V_3); then picks the next transaction t_{i+1} and repeats until all transactions are validated.

This sequential approach bottlenecks the performance of FlexChain due to the high latency in fetching/committing to the world state, as incurred by the read/write procedure. Hence, to maximize the end-to-end throughput of FlexChain, we leverage dependency graph-based techniques to fully parallelize validation: for transactions with no data dependencies, FlexChain will validate them (V_2 and V_3) in parallel. The next sections detail our techniques.

5.1 Dependency Graph Construction

Dependency graph-based concurrency control has been shown to be effective in database management systems [26, 27, 59] and blockchains [13]. In FlexChain, by capturing dependencies between transactions, conflict check (V_2) and world state update (V_3) steps can be performed for a transaction as soon as all its predecessors in the dependency graph have been committed or aborted. Given a transaction block B, dependency graph $G(B) = (T, E)$ is a directed acyclic graph where T denotes the set of transactions within B that are marked as valid in the Validation System Chaincode (VSCC) step (V_1), and $(t_i, t_j) \in E$ if and only if t_i is ordered before t_j, t_i and t_j are conflicted, and there is no transaction t_k between t_i and t_j in B such that t_k conflicts with both t_i and t_j. The dependency graph, therefore, captures write-read (WR), read-write (RW) and write-write (WW) conflicts.

We illustrate three types of conflicts using an example. Suppose the validation manager on peer p_1 receives a block $B = [t_1, t_2, ..., t_n]$ from the ordering service nodes. First, if there is **WR** conflict between t_1 and t_2, i.e., t_1 updated key_A to v_i and t_2 reads key_A with a version older than v_i (this is because t_2 is simulated before t_1 enters

Algorithm 1 Parallel Validation (V_2 and V_3)

```
Input: A block $B$ and its dependency graph $G(B) = (T, E)$
Initialize Set $W \leftarrow T$ and Set $C \leftarrow empty$
while $W$ is not empty do
    for Transaction $t$ in $W$ do
        if all Predecessor($t$) in $C$ then
            Remove $t$ from $W$
            Trigger ValidationWorker($t$)
        end if
    end for
end while

Thread ValidationWorker($t$):
if CheckConflict($t$) is false then
    UpdateState($t$)
end if
Add $t$ to $C$
```

the validation phase), then t_2 must be marked as invalid during validation since it reads a stale version. Thus, we need an edge (t_1, t_2) to ensure t_2 will be validated after t_1. Otherwise, some peer p_2 may validate t_2 before t_1 due to scheduling, possibly marking t_2 as valid, which is inconsistent with peer p_1. If there is **RW** conflict between t_1 and t_2, similarly an edge (t_1, t_2) is needed. Finally, if there is **WW** conflict between t_1 and t_2, i.e., t_1 updates key_A to v_i while t_2 updates key_A to $v_{i'}$, an edge (t_1, t_2) is needed to ensure eventual consistency of key_A across all peers. Note that since the dependency graph is acyclic, i.e., edges are added from older transactions to newer ones, the resulting schedule is conflict serializable where an equivalent serial schedule can be given by the topological sort of the dependency graph.

FlexChain constructs the dependency graph in the ordering stage. The ordering service nodes of FlexChain, similar to Hyperledger Fabric, use the crash fault-tolerant protocol Raft [41] to establish consensus on the order of transactions. Since we assume ordering service nodes are honest, i.e., they might crash but they do not behave maliciously, it is sufficient to construct the graph only on a single node, e.g., the leader. Specifically, apart from the currently running Raft instances, the leader embodies a *block formation* thread. Once a transaction t_n is committed (i.e., the leader receives confirmation from the majority that the entry has been replicated and applies the entry to its log), the block formation thread checks conflicts between t_n and any transaction $t_j (1 \leq j < n)$, resulting in incremental graph construction. This approach minimizes the sequential validation bottleneck because (1) the block formation thread operates on a monolithic server and uses local data, and (2) the graph construction, which is sequential, is integrated into the block formation, which is inevitably sequential.

5.2 Concurrency Control

The primary cNode of each peer has a centralized validation manager that launches validation workers from a local thread pool. The validation manager assigns transactions within a block to validation workers based on the dependency graph of the block. Algorithm 1 demonstrates the parallel validation of transactions. Specifically, the validation manager maintains a set W as transactions to be validated and a set C as transactions completed by validation workers. The validation manager repeatedly checks each transaction t in W. If all t's predecessors have been validated, the manager assigns t to the least-loaded validation worker, which adds t to C upon finishing the validation. FlexChain also allows the validation manager to dispatch validation workers to other cNodes to further benefit from the elastic compute resources.

6 EVALUATION

Our evaluation aims to answer the following questions:

(1) How well does the elasticity of FlexChain handle different blockchain workloads? What operational benefits does it provide for the users? (Section 6.2)
(2) How well does FlexChain perform when compared to existing XOV blockchains that are deployed as traditional distributed systems? (Section 6.3)
(3) How does the extent of disaggregation (e.g., the cache size in the compute pool and the CPU clock speed in the memory pool) affect the performance of FlexChain? (Section 6.4)
(4) Does FlexChain retain its performance improvements when it is deployed across multiple data centers? (Section 6.5)

6.1 Experimental Setup

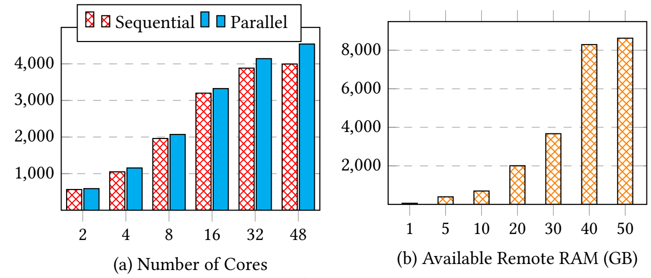
We have implemented a prototype of FlexChain in C++. We use `libibverbs` [2] for all RDMA communications that are specific to a disaggregated system. `libibverbs` enables user-space processes to utilize RDMA and provides fine-grained control over hardware to achieve optimal performance. In FlexChain, RDMA is used for communications between compute and memory nodes, and between different compute nodes. In addition to RDMA for disaggregation, we use gRPC over regular Ethernet for other communications in an XOV blockchain, such as messages between orderers and peers. For the disaggregated storage in FlexChain, we use LevelDB [1] for its highly optimized compaction performance.

Although we cannot directly use Hyperledger Fabric [3] due to the lack of RDMA support in Go (the language that implements Fabric), we have implemented an XOV blockchain that is consistent with the design of Fabric in C++, which achieves even better performance than Fabric as we show shortly.

Testbed. Our testbed consists of 20 c6220 bare-metal machines on CloudLab [24], each with two Xeon E5-2650v2 processors (8 cores each, 2.6Ghz) and two 1TB SATA 3.5" 7.2K rpm hard drives. These machines are connected by two networks, each with one interface: (1) an Infiniband network that achieves 56 Gbps bandwidth, using Mellanox FDR CX3 NIC and Mellanox SX6036G switches; (2) a 10 Gbps Ethernet commodity network. This testbed allows us to emulate a compute pool with 192 cores, a memory pool with 192 GB RAM, and a storage pool with 4TB HDDs.

In our experiments, unless otherwise specified, we run a single blockchain channel that consists of 3 peers and uses Raft [41] as the consensus protocol, with 3 orderers and a batch size of 200. To minimize the end-to-end latency, we use the smallest block size that maximizes throughput for each experiment. To measure the highest throughput, we increase the number of clients in the system until the rate of committed transactions per second (tps) reaches its maximum value. All numbers are the average over three runs.

Workloads. In our experiments, we use three types of workloads: YCSB, Smallbank, and a machine learning workload. In the YCSB benchmark, we preload the blockchain with a number of records and issue frequent requests to the key-value store with different ratios of read and write operations. In Smallbank, we implement a smart contract that transfers money between bank accounts. The



(a) Number of Cores (b) Available Remote RAM (GB)

Figure 5: (a) Compute and (b) Memory scaling throughput.

smart contract preloads the blockchain with a number of users, each with a checking account and a savings account. In each run, we randomly pick one of the five modifying transactions with probability P_w and the reading transactions with probability $1 - P_w$.

Both YCSB and Smallbank mirror the macro benchmarks described in Blockbench[22]. These benchmarks are memory-intensive and do not truly stretch compute resources. In our third benchmark, we utilize a smart contract that performs the same compute-intensive operations as in the World Food Program [9, 39, 60]: here, we interchangeably run two transactions: (1) `updateProfile()`, which is write-only and emulates refugee profile updates to the world state, and (2) `getFood()`, which performs K-Means computation to conduct image classification and determine food distribution. The `getFood()` transaction requires reading data points from the blockchain world state, performing K-Means computation, and finally, writing the result back to the world state.

We describe the specific parameters we use for each workload in each experiment in the following sections.

6.2 The Elasticity of Disaggregation

Compute scaling. Our first set of experiments aims to demonstrate that *given a compute-intensive smart contract, FlexChain can scale up compute resources to achieve better performance while leaving the amount of provisioned RAM unchanged.* This is a key operational benefit of DDCs that is not easily achievable in a traditional setup. We use the food distribution smart contract as a representative workload. The smart contract runs the `updateProfile()` transaction with 5% probability, and the `getFood()` transaction with 95% probability. In each transaction, we access the world state based on a uniform distribution. We set $K = 20$ in K-Means, epoch= 2000, data dimensions = 100, and the convergence threshold $\tau = 0.01$. For each peer, we fix the local cache size on the compute nodes of this peer as 200 MB and its remote memory size as 8 GB. We vary the number of CPU cores each peer uses in the compute pool and measure the throughput.

Figure 5(a) shows that when we raise the CPU cores per peer from 2 to 16 (within a compute node), the throughput of FlexChain with sequential validation increases from 566 tps to 3196 tps. Scaling a FlexChain peer out from one compute node (16 cores) to three nodes (48 cores) further boosts its throughput to 3995 tps. Our results demonstrate the capability of FlexChain at handling compute-intensive smart contracts—when the workload requires higher throughput, FlexChain simply provisions more CPU cores within a single compute node first and, if needed, scales out by including additional compute nodes. Moreover, due to resource pooling, FlexChain can utilize an arbitrary amount of compute resources, an elastic benefit that is hard to achieve in the traditional generic XOV architecture.

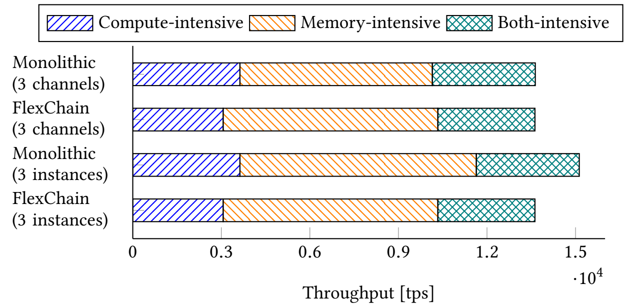


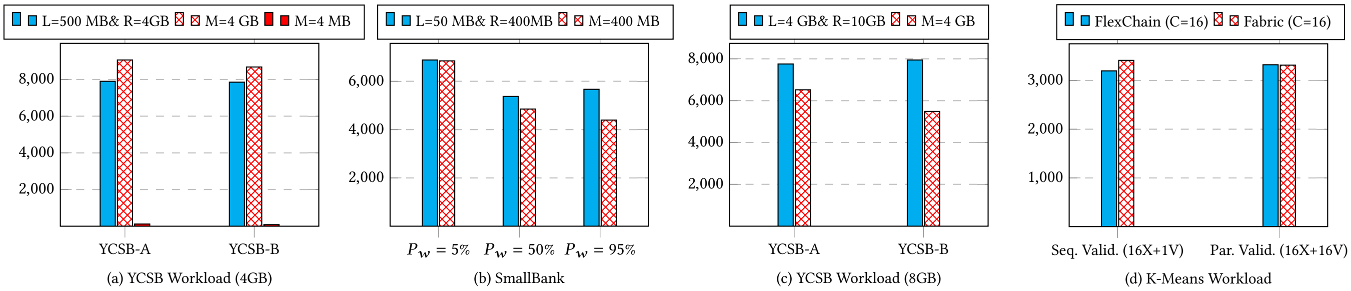
Figure 6: Running multiple contracts simultaneously.

Parallel validation. As we increase the number of CPU cores available for the simulation (X) phase, we observe that the bottleneck on overall performance shifts to the sequential validation phase, i.e., conflict check (V_2) and commit (V_3), the overhead of which comes from accessing remote memory. To further optimize FlexChain's performance, we apply the parallel validation techniques described in Section 5 and run the validation workers on 16 additional cores. Again, this elasticity is made possible by FlexChain's disaggregated architecture. Figure 5(a) shows that parallel validation indeed results in better performance: compared to sequential validation on three compute nodes (48 cores), the throughput of FlexChain now increases from 3995 tps to 4541 tps, thereby achieving better scalability. This evaluation shows that parallelizing the validation phase is necessary when FlexChain executes the simulation phase at scale.

Memory scaling. Our next experiment focuses on another operational benefit of DDCs: *by scaling memory nodes in the presence of data-intensive workloads, remote memory can be used in lieu of more expensive disk swapping operations.* We created a data-intensive workload using YCSB workload A (50% write) with a total of 4, 000, 000 records and 10 KB record size, resulting in 40 GB of prepopulated data store. Note that the record number and size are common in blockchain-enabled healthcare applications. In each transaction, we choose a record according to the uniform distribution. We fix the local data cache as 500 MB and scale up the amount of remote memory available for each peer from 1 GB to 50 GB. The remote storage pool has 2 TB capacity. Each peer has a fully provisioned compute node with 16 cores.

Figure 5(b) depicts the results of memory scaling. As remote memory increases from 1 GB to 50 GB, FlexChain achieves a 140× speedup, saturating at 8623 tps. Data-intensive smart contracts perform poorly when remote memory is as limited as 1 GB, The root cause is that the remote storage nodes must perform expensive disk-intensive operations for compaction and random reads. As remote memory increases, most requests that access the world state are served purely in memory, and thus, there are fewer evictions to remote storage, resulting in higher throughput. This also justifies our tiered key-value store design: disaggregating only the storage is not sufficient to achieve high performance.

Benefits of decoupling resources. One of the key operational benefits of DDC is decoupling compute and memory resources, thereby promoting efficient resource utilization. We conducted an



(a) YCSB Workload (4GB)

(b) SmallBank

(c) YCSB Workload (8GB)

(d) K-Means Workload

Figure 7: Comparing FlexChain (●) throughput [tps] with generic XOV (⊗). Here L stands for local data cache per peer in FlexChain, R stands for available remote memory per peer in FlexChain, and M stands for memtable in generic XOV. In each transaction, we choose a record/user according to the uniform distribution. (a): prepopulated with $400,000$ x 10 KB = 4 GB records. (b): prepopulated with $2,000,000$ users, each with 100 B for checking/savings account. (c): prepopulated with $800,000$ x 10 KB = 8 GB records. (d): C stands for CPU cores per peer, X for simulation thread count, and V for validation thread count.

experiment to show when serving a mixture of workloads, FlexChain saves a significant amount of resources while achieving similar performance to that of monolithic BaaS. We run three smart contracts simultaneously in three blockchain channels/instances: one compute-intensive contract (the same as in Figure 5(a)), one memory-intensive contract (the same as in Figure 5(b)), and one contract that is both compute-intensive and memory-intensive, generated by increasing the size of the prepopulated data store in the K-Means workload. Each channel/instance consists of three peers and runs one distinct type of contract.

Figure 6 shows when running three contracts simultaneously in three channels, FlexChain saves 48 cores and 192GB RAM (1/3 resources) compared to monolithic BaaS, and achieves approximately the same aggregated throughput. We observe that when running multiple contracts together sharing the same OSNs, the bottleneck shifts to the consensus protocol and the Ethernet network between orderers, which limits the throughput of memory-intensive peers in monolithic BaaS (6517 tps). To show FlexChain's performance without OSNs being the bottleneck, we run three contracts simultaneously in three blockchain instances, where each instance has its own OSNs. As a result, the throughput of a memory-intensive peer in monolithic BaaS increases to 8007 tps. In total, FlexChain incurs only 9.93% performance degradation compared to monolithic BaaS, but still saves 48 cores and 192GB RAM. FlexChain saves even more resources when the number of peers in a blockchain increases, which is usually the case in real-world large-scale BaaS.

6.3 Comparisons with Generic XOV

Our next set of experiments compares FlexChain with an XOV blockchain deployed on three monolithic servers (peers) communicating with three OSNs. Each peer uses local LevelDB for its world state. In the rest of this section, we refer to this distributed strawman as *generic XOV*. This is in contrast to FlexChain, where each peer node is disaggregated across compute and memory nodes rather than on a single monolithic server.

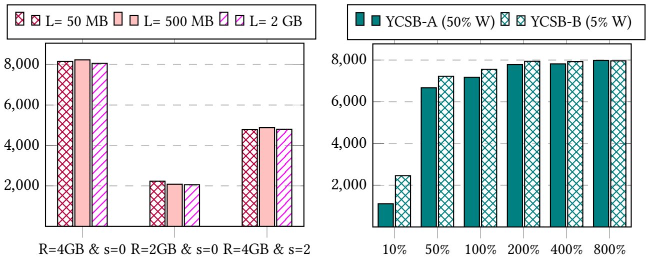
YCSB at 4GB. We first run YCSB benchmark on a modest dataset of 4 GB preloaded records. Our performance results are summarized in Figure 7(a). The left bar shows FlexChain running with a 500 MB local data cache (L) and 4 GB remote memory (R) per peer. The middle and right bars show generic XOV with memtable (M) sizes of 4 GB and 4 MB respectively. The latter 4 MB is the default setting in Fabric. The memtable serves as a temporary in-memory database for serving read requests and storing write updates before being evicted to the local disk. We repeat our experiments for YCSB-A (50% reads/50% writes) and YCSB-B (95% reads/5% writes).

We observe that FlexChain has 12.8% performance degradation on YSCB-A and 9.5% degradation on YCSB-B compared to generic XOV (M=4GB). This is an acceptable and expected performance overhead of disaggregation, causes by remote memory accesses. However, generic XOV (M=4 MB) performs poorly (only 120 tps) due to excessive disk paging. Our results suggest that even when running blockchains in a traditional non-DDC setting, maintaining sufficient local memory that caches the working set of updates is critical. This is more important when deploying blockchain services in the cloud, where dynamic workloads are typical. Unlike FlexChain, in order to keep good performance in serving dynamic workloads, the traditional BaaS requires error-prone cross-node live migrations, which result in resource fragmentation.

Given the poor performance of generic XOV with the default memtable setting (M=4 MB), in the next two experiments, we focus on comparing FlexChain to generic XOV with a large memtable.

SmallBank. We next run Smallbank workload with three different P_w ratios. The results are demonstrated in Figure 7(b). Interestingly, FlexChain consistently outperforms generic XOV, and its advantage is more significant in write-intensive workloads: +0.5% when P_w = 5%, +11% when P_w = 50%, and +29% when P_w = 95%. The reason is, although accessing remote memory in FlexChain incurs some overhead, LevelDB adopted in generic XOV has its own inefficiencies: it waits until the entire memtable is full before minor compaction takes place. For large memtables, this compaction operation blocks all future write requests and prevents further updates. FlexChain addresses this problem by evicting buffers in remote memory periodically in the background without blocking the critical data path (as described in Section 4). Consequently, the overhead of disaggregation in FlexChain is outweighed by LevelDB's limitations in generic XOV.

YCSB at 8GB. We next revisit YCSB, but this time with doubled number of records (8 GB), while increasing remote memory to 10 GB and keeping local memory at 4 GB. Figure 7(c) shows that FlexChain outperforms generic XOV: +18.9% for YCSB-A and +44.9%



(a) Configurations

(b) CPU Time

Figure 8: Impact of local cache/remote CPU on throughput.

<table>
<tr><th>Workloads</th><th>Single-DC</th><th>Multi-DC</th><th>50ms</th><th>100ms</th><th>250ms</th></tr>
<tr><td rowspan="2">Compute-intensive
Seq. Valid. | Par. Valid.</td><td rowspan="2">3995 | 4541</td><td>1Gbps</td><td>4000 | 4530</td><td>3997 | 4529</td><td>3136 | 3130</td></tr>
<tr><td>500Mbps</td><td>3998 | 4521</td><td>3990 | 4540</td><td>3127 | 3131</td></tr>
<tr><td rowspan="2">Memory-intensive</td><td rowspan="2">8623</td><td>1Gbps</td><td>8649</td><td>5155</td><td>1687</td></tr>
<tr><td>500Mbps</td><td>6106</td><td>2993</td><td>290</td></tr>
</table>

Table 1: Throughput [tps] over multiple data centers. We also report the corresponding performance in a single data center deployment for comparison.

for YCSB-B. These results suggest that with extra remote memory (10 GB remote memory vs. 4 GB local RAM), FlexChain significantly outperforms generic XOV. Due to memory pooling, allocating extra remote memory for a peer is straightforward in FlexChain.

K-Means. We further run the same compute-intensive workload as Figure 5(a) in generic XOV. We set the amount of resources, i.e., both CPUs and memory, to be the same as the "16 cores" bar (one compute node) in Figure 5(a). As Figure 7(d) shows, using sequential validation, FlexChain incurs only a 6.7% performance degradation compared to generic XOV, and with parallel validation, FlexChain achieves almost the same performance as generic XOV. This result shows that FlexChain minimizes the performance penalty caused by fetching data from the remote memory. An interesting finding in Figure 7(d) is that parallel validation improves the performance of FlexChain by 126 tps, but *reduces* the performance of generic XOV by 98 tps. This is because validation phase is not a bottleneck in generic XOV, so applying Algorithm 1 backfires as it requires synchronization. The improvement of parallel validation on FlexChain is more obvious when the simulation phase further scales up as in Figure 5(a). This is also consistent with the XOX fabric [32] where parallel validation is only beneficial when validation is expensive: in XOX, it is due to the transaction re-execution embedded in the validation phase; in FlexChain, it is due to memory disaggregation and the large number of keys accessed in the K-Means workload.

The main takeaways of our evaluation results are as follows. Despite expensive remote memory accesses, the overhead of disaggregation in FlexChain is kept as low as 6.7%. In addition to the operational benefits of DDCs on scaling up compute and memory separately, there are scenarios where FlexChain actually outperforms generic XOV, due to better eviction mechanism and more elastic remote memory. In some cases, these benefits can increase throughput by more than 2000 tps (44.9%).

6.4 Varying the Extent of Disaggregation

A disaggregated architecture comes with some limitations. For example, there is less memory on compute nodes compared to monolithic servers. The CPU clock speed of remote memory controllers may also be lower. Thus, it is crucial to study how sensitive FlexChain is to the extent of disaggregation, i.e., the size of the data cache left on compute nodes and the CPU clock speed of memory controllers.

Impact of local data cache size. Figure 8(a) shows the experimental results for running YCSB-A (50% reads/50% writes) on FlexChain. In this experiment, we initialize FlexChain with 4 GB records. We vary the amount of local data cache (L) under different remote memory size (R) and Zipfian s-value (s). The Zipfian distribution is used to choose a key for a certain transaction. Note that an s-value of 0 corresponds to the uniform distribution and hence less contention, while a larger s-value reflects skewness in accessed keys and thus more contention. We observe that when FlexChain has abundant remote memory and little contention (R = 4 GB, s = 0), as we increase local data cache from 50 MB to 2 GB, the cache hit ratio increases from 50% to 73% for read operations. FlexChain achieves a high cache hit ratio even with only 50 MB local memory, due to the fact that for each key it accessed and cached in the simulation phase, FlexChain reads the key again in the validation phase. For write operations, since we use a write-through design, the performance of FlexChain is not affected by the amount of local data cache but is limited by RDMA communications and the write agents on remote memory controllers instead. Thus, this slight increase in the cache hit ratio does not boost the performance.

When FlexChain has insufficient remote memory and little contention (R = 2 GB, s = 0), we observe similar insensitivity across different local cache sizes. However, remote storage becomes the bottleneck in this case: 50.1% of read requests are served by searching SSTables on the persistent storage. This results in poor performance due to increased disk activities: the throughput drops from 8000 tps to 2000 tps, a 75% reduction. A larger remote memory would prevent such frequent disk activities from happening and hence improve the performance.

When FlexChain has sufficient remote memory and increased contention (R = 4 GB, s = 2), we again observe similar insensitivity in local cache sizes. In this case, no matter we have a 50 MB local cache or a 2 GB one, FlexChain consistently achieves 99.9% cache hit ratio and only 0.002% of requests go to SSTables, because of the small hot record set. However, more than 50% of transactions are aborted due to conflicts, which limits the end-to-end throughput severely. Introducing reordering and early aborts [45, 47] can boost XOV-style blockchains' performance under contention.

We repeated our experiments on SmallBank and K-Means workloads and found similar observations.

Impact of remote CPU clock speed. In FlexChain, some tasks residing on the remote memory controllers need CPU involvement, e.g., the main dispatcher thread, address manager, space allocator, bookkeeping agent, eviction manager and garbage collector. We now study the impact of weak memory controller CPUs on FlexChain's performance.

Figure 8(b) depicts the performance of FlexChain under YCSB workloads (400K records with 10KB size). Here, we use `cgroups` to limit the CPU time the remote memory controller can use per period (0.1 ms). This has the same effect as limiting the CPU clock speed. Our experiments vary the available CPU time from 10% (0.1 cores) to 800% (8 cores). In both YCSB-A and YCSB-B benchmarks, FlexChain achieves good performance with only two 2.6 Ghz cores, and there are diminishing returns as more CPU cycles are made available. If the remote memory controller only has one 1.3 Ghz core (a reasonable configuration for disaggregation), FlexChain experiences 1308 tps (16.4%) and 741 tps (10.3%) throughput degradation for YCSB-A and YCSB-B respectively, relative to the 8 cores (800% CPU time) setting. Performance degradation is larger for write-heavy workloads like YCSB-A, since every write involves more operations on the remote memory controller than reads. In contrast, reads benefit more from local metadata and data caches.

6.5 Cross-Data Center Deployment

Our final set of experiments evaluates FlexChain in a cross-data center deployment to investigate the impact of wide area network performance. We use both the compute- and memory-intensive workloads in Figure 5(a) and 5(b) in an emulated multi-data center environment. Specifically, we scale up resources to the largest extent as the rightmost bars in Figure 5(a) and 5(b), and measure FlexChain's throughput under different network environments. Our setup consists of four data centers connected by a wide area network. We place all OSNs in DC_1 as well as 3 logical peers in DC_2 to DC_4 respectively, and vary the network bandwidth and RTT latency between different data centers using Linux `netem` [30].

Table 1 summarizes the results. As reported in [4], 1 Gbps bandwidth with 50 ms RTT is a common network condition between two data centers. Under this condition, the cross-data center deployment achieves comparable throughput with the single-data center deployment on both compute- and memory-intensive workloads. FlexChain retains much of its performance improvements derived via elastic scaling even when the latency (bandwidth) of the network doubles (halves) to 100 ms (500 Mbps). Like any other distributed system, the performance of FlexChain inevitably drops under extreme network conditions, e.g., with 250 ms latency, bottlenecked by the transaction transfer between peers and orderers. Such conditions only affect memory-intensive workloads, as the amount of data transferred across data centers is insignificant in compute-intensive workloads. These conditions are uncommon as the Internet speed is getting increasingly higher [4].

7 RELATED WORK

In this section, we briefly survey several related research lines.

OS abstractions for disaggregated memory. Building operating systems suitable for DDCs has been proposed in LegoOS [46] that maintains a POSIX API for system calls, and in Infiniswap [35] that provides a page-oriented abstraction for paging to remote memory. LegoOS is designed for general purpose applications, but by itself cannot be a drop-in replacement for database systems given its poor performance [61–63]. LegoOS does not satisfy our needs either since it does not support writable shared memory across different processors. Infiniswap cannot be directly used by FlexChain to support the remote key-value store, since it forces every access to go through a time-consuming kernel data path. Further, FlexChain is optimized at a record level, and a page-level eviction approach backfires with poor performance [64]. It may be possible to use middleware such as Infiniswap for paging transient smart contract data currently maintained in local memory.

Disaggregated databases. Given the poor performance of running databases on LegoOS and Infiniswap, there are also new proposals on rethinking the design of relational databases [18, 63, 64] that are optimized for the DDC setting. PolarDB Serverless [18] and Legobase [64] also study how independent hardware failures in DDCs accelerate the failure recovery. There is a significant architectural difference between relational databases and XOV blockchains. As a result, the design principles and performance implications of these studies are not applicable to disaggregated XOV blockchains. Apart from relational databases, disaggregating key-value store using RDMA and remote memory has been studied [23, 54]. None of these studies utilize a tiered design that includes remote stable storage and local cache. Consequently, they are not suitable to be used in disaggregated blockchains that require data durability and affordable budgets for users.

Optimizing XOV blockchains. Several studies have improved the performance of Hyperledger Fabric [16] while still following its XOV architecture [32, 34, 44, 45, 47, 49–51, 58]. FastFabric [34] targets conflict-free transaction workloads and improves Fabric's throughput by using different data structures and caching techniques for gRPC, and pipelining the transaction validation. However, FastFabric assumes the entire blockchain world state fits in its in-memory hashtable. Fabric++ [47] employs concurrency control techniques to early abort transactions or reorder them after the order phase to reconcile the potential conflicts. FabricSharp [45] goes one step further and presents an algorithm to early filter out transactions that can never be reordered and also presents a reordering technique that eliminates unnecessary aborts. All such conflict resolution techniques can be employed in the ordering phase of FlexChain.

8 CONCLUSION

This paper presents FlexChain, the first XOV blockchain system that is designed following the disaggregation paradigm. FlexChain uses a tiered key-value store to scale its memory and storage independently and elastically. FlexChain also presents a parallel validation process that removes a bottleneck in disaggregated XOV systems. Our experimental results demonstrate that FlexChain provides independent compute and memory scalability while incurring at most 12.8% performance overhead of disaggregation.

Our work is the first step towards a longer-term research agenda on distributed systems in an exciting new hardware paradigm based on DDCs. As the next steps, we are exploring optimizing other aspects of blockchain systems, e.g., the consensus component to take advantage of the new DDC hardware. Handling heterogeneous and correlated failures induced by DDCs gracefully is also a challenging next step. We also plan to explore a mix of hardware resources, e.g., combining CPUs, GPUs and FPGAs in compute pools. Finally, FlexChain is a demonstration that a complex distributed system can be made to run efficiently in DDCs. In the long term, we plan to work towards an automated tool that seamlessly migrates legacy distributed systems to run efficiently in DDCs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback and suggestions. This work is funded by NSF grants CNS-2104882, and CNS-2107147 and by ONR grant N00014-18-1-2021.

REFERENCES

[1] [n.d.]. https://github.com/google/leveldb

[2] [n.d.]. https://github.com/linux-rdma/rdma-core

[3] [n.d.]. https://github.com/hyperledger/fabric

[4] [n.d.]. https://www.cloudping.co/grid

[5] [n.d.]. Alibaba Cloud Blockchain as a Service. https://www.alibabacloud.com/product/baas.

[6] [n.d.]. Blockchain on AWS Enterprise blockchain made real. https://aws.amazon.com/blockchain/.

[7] [n.d.]. IBM Blockchain Platform. https://www.ibm.com/cloud/blockchain-platform.

[8] [n.d.]. Oracle Blockchain. https://www.oracle.com/blockchain/.

[9] 2017. Blockchain Against Hunger: Harnessing Technology In Support Of Syrian Refugees | World Food Programme — wfp.org. https://www.wfp.org/news/blockchain-against-hunger-harnessing-technology-support-syrian-refugees. [Accessed 27-Mar-2022].

[10] 2018. Cortex Labs Pte. Ltd., Singapore. https://www.cortexlabs.ai/

[11] 2020. ConnectX-6 Single/Dual-Port Adapter supporting 200Gb/s with VPI. https://www.mellanox.com/products/infiniband-adapters/connectx-6.

[12] Cornelius C. Agbo, Qusay H. Mahmoud, and J. Mikael Eklund. 2019. Blockchain Technology in Healthcare: A Systematic Review. *Healthcare* 7, 2 (2019). https://doi.org/10.3390/healthcare7020056

[13] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. Par-Blockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems. In *Int. Conf. on Distributed Computing Systems (ICDCS)*. IEEE, 1337–1347.

[14] Mohammad Javad Amiri, Joris Duguépéroux, Tristan Allard, Divyakant Agrawal, and Amr El Abbadi. 2021. SEPAR: Separ: Towards Regulating Future of Work Multi-Platform Crowdworking Environments with Privacy Guarantees. In *Proceedings of The Web Conf. (WWW)*. 1891–1903.

[15] Mohammad Javad Amiri, Boon Thau Loo, Divyakant Agrawal, and Amr El Abbadi. 2022. Qanaat: A Scalable Multi-Enterprise Permissioned Blockchain System with Confidentiality Guarantees. *Proc. of the VLDB Endowment* 15, 11 (2022), 1.

[16] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, et al. 2018. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *European Conf. on Computer Systems (EuroSys)*. ACM, 30.

[17] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. 2020. Disaggregation and the Application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[18] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021.* ACM, 2477–2489. https://doi.org/10.1145/3448016.3457560

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[20] CortexFoundation. 2019. tech-doc/cortex-details.md at master · CortexFoundation/tech-doc — github.com. https://github.com/CortexFoundation/tech-doc/blob/master/cortex-details.md. [Accessed 27-Mar-2022].

[21] Sam Daley. 2021. 18 Blockchain-as-a-Service Companies Making the DLT More Accessible. https://builtin.com/blockchain/blockchain-as-a-service-companies.

[22] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. Blockbench: A framework for analyzing private blockchains. In *SIGMOD Int. Conf. on Management of Data*. ACM, 1085–1100.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. {FaRM}: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 401–414.

[24] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[25] Subhieh El Salhi, Fairouz Farouq, Randa Obeidallah, Yousef Kilani, et al. [n.d.]. Real-Time Carpooling Application based on k-NN Algorithm: A Case Study in Hashemite University. ([n. d.]).

[26] Jose M Faleiro and Daniel J Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1190–1201.

[27] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proc. of the VLDB Endowment* 10, 5 (2017), 613–624.

[28] Liming Fang, Bo Zhao, Yang Li, Zhe Liu, Chunpeng Ge, and Weizhi Meng. 2020. Countermeasure Based on Smart Contracts and AI against DoS/DDoS Attack in 5G Circumstances. *IEEE Network* 34, 6 (2020), 54–61. https://doi.org/10.1109/MNET.021.1900614

[29] Ethan Frey and Christopher Goes. [n.d.]. Cosmos Inter-Blockchain Communication (IBC) Protocol. https://cosmos.network. 2018.

[30] The Linux Fundation. 2021. netem. Retrieved July 4, 2022 from https://wiki.linuxfoundation.org/networking/netem

[31] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation.

[32] Christian Gorenflo, Lukasz Golab, and Srinivasan Keshav. 2020. XOX Fabric: A hybrid approach to transaction execution. In *Int. Conf. on Blockchain and Cryptocurrency (ICBC)*. IEEE, 1–9.

[33] Christian Gorenflo, Stephen Lee, Lukasz Golab, and S. Keshav. 2019. FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second. *arXiv preprint arXiv:1901.00910* (2019).

[34] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. 2019. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. In *Int. Conf. on Blockchain and Cryptocurrency (ICBC)*. IEEE, 455–463.

[35] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with INFINISWAP.

[36] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proc. VLDB Endow.* 13, 6 (feb 2020), 868–883. https://doi.org/10.14778/3380750.3380757

[37] Zhaofeng He, Tieniu Tan, Zhenan Sun, and Xianchao Qiu. 2008. Toward accurate and fast iris segmentation for iris biometrics. *IEEE transactions on pattern analysis and machine intelligence* 31, 9 (2008), 1670–1684.

[38] Marco D. Huesch and Timothy J. Mosher. 2017. Using It or Losing It? The Case for Data Scientists Inside Health Care. https://catalyst.nejm.org/doi/full/10.1056/CAT.17.0493 [Accessed 27-Mar-2022].

[39] Nir Kshetri and Jeffrey Voas. 2018. Blockchain in developing countries. *It Professional* 20, 2 (2018), 11–14.

[40] Rajesh Kumar, WenYong Wang, Jay Kumar, Ting Yang, Abdullah Khan, Wazir Ali, and Ikram Ali. 2021. An integration of blockchain and AI for secure data sharing and detection of CT images for the hospitals. *Computerized Medical Imaging and Graphics* 87 (2021), 101812.

[41] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[42] Zhe Peng, Cheng Xu, Haixin Wang, Jinbin Huang, Jianliang Xu, and Xiaowen Chu. 2021. P2B-Trace: Privacy-Preserving Blockchain-based Contact Tracing to Combat Pandemics. In *SIGMOD Int. Conf. on Management of Data*. 2389–2393.

[43] Zhe Peng, Jianliang Xu, Xiaowen Chu, Shang Gao, Yuan Yao, Rong Gu, and Yuzhe Tang. 2021. Vfchain: Enabling verifiable and auditable federated learning via blockchain systems. *IEEE Transactions on Network Science and Engineering* (2021).

[44] Ravi Kiran Raman, Roman Vaculin, Michael Hind, Sekou L Remy, Eleftheria K Pissadaki, Nelson Kibichii Bore, Roozbeh Daneshvar, Biplav Srivastava, and Kush R Varshney. 2018. Trusted Multi-Party Computation and Verifiable Simulations: A Scalable Blockchain Approach. *arXiv preprint arXiv:1809.08438* (2018).

[45] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2020. A Transactional Perspective on Execute-order-validate Blockchains. In *SIGMOD Int. Conf. on Management of Data*. ACM, 543–557.

[46] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.).

[47] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. 2019. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *SIGMOD Int. Conf. on Management of Data*. ACM, 105–122.

[48] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A network architecture for disaggregated racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 255–270.

[49] Joao Sousa, Alysson Bessani, and Marko Vukolic. 2018. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE, 51–58.

[50] Parth Thakkar and Senthil Nathan. 2020. Scaling Hyperledger Fabric Using Pipelined Execution and Sparse Peers. *arXiv preprint arXiv:2003.05113* (2020).

[51] Parth Thakkar, Senthil Nathan, and Balaji Vishwanathan. 2018. Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform. *arXiv preprint arXiv:1805.11390* (2018).

[52] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. 2018. Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 264–276. https://doi.org/10.1109/MASCOTS.2018.00034

[53] Feng Tian. 2017. A supply chain traceability system for food safety based on HACCP, blockchain & Internet of things. In *Int. Conf. on service systems and service management (ICSSSM)*. IEEE, 1–6.

[54] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated {Key-Value} Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 33–48.

[55] Amin Vahdat. 2020. Coming Of Age In The Fifth Epoch Of Distributed Computing: The Power Of Sustained Exponential Growth. SIGCOMM 2020 Keynote.

[56] Canhui Wang and Xiaowen Chu. 2020. Performance Characterization and Bottleneck Analysis of Hyperledger Fabric. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 1281–1286. https://doi.org/10.1109/ICDCS47774.2020.00165

[57] Wikipedia. 2022. Image segmentation — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Image%20segmentation&oldid=1078773313. [Online; accessed 27-March-2022].

[58] Lu Xu, Wei Chen, Zhixu Li, Jiajie Xu, An Liu, and Lei Zhao. 2020. Locking Mechanism for Concurrency Conflicts on Hyperledger Fabric. In *Int. Conf. on Web Information Systems Engineering*. Springer, 32–47.

[59] Chang Yao, Divyakant Agrawal, Pengfei Chang, Gang Chen, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. 2015. Dgcc: A new dependency graph based concurrency control protocol for multicore database systems. *arXiv preprint arXiv:1503.03642* (2015).

[60] Raul Zambrano, Andrew Young, and Stefaan Verhulst. 2018. Connecting refugees to aid through blockchain-enabled ID management: world food programme's building blocks. *GovLab October* (2018).

[61] Qizhen Zhang, Yifan Cai, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Rethinking Data Management Systems for Disaggregated Data Centers.

[62] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. *Proceedings of the VLDB Endowment* 13, 9 (May 2020), 1568–1581.

[63] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2022. Optimizing Data-intensive Systems in Disaggregated Data Centers with TELEPORT. In *SIGMOD*.

[64] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. 2021. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. *Proc. VLDB Endow.* 14, 10 (2021), 1900–1912. http://www.vldb.org/pvldb/vol14/p1900-zhang.pdf