applied sciences

MDPI

Article

QiOi: Performance Isolation for Hyperledger Fabric

Jeongsu Kim, Kyungwoon Lee, Gyeongsik Yang, Kwanhoon Lee, Jaemin Im and Chuck Yoo *

Department of Computer Science and Engineering, Korea University, 145, Anam-ro, Seongbuk-gu, Seoul 02841, Korea; jskim@os.korea.ac.kr (J.K.); kwlee@os.korea.ac.kr (K.L.); ksyang@os.korea.ac.kr (G.Y.); khlee@os.korea.ac.kr (K.L.); jaeminim95@korea.ac.kr (J.I.)

* Correspondence: chuckyoo@os.korea.ac.kr

Abstract: This paper investigates the performance interference of blockchain services that run on cloud data centers. As the data centers offer shared computing resources to multiple services, the blockchain services can experience performance interference due to the co-located services. We explore the impact of the interference on Fabric performance and develop a new technique to offer performance isolation for Hyperledger Fabric, the most popular blockchain platform. First, we analyze the characteristics of the different components in Hyperledger Fabric and show that Fabric components have different impacts on the performance of Fabric. Then, we present QiOi, component-level performance isolation technique for Hyperledger Fabric. The key idea of QiOi is to dynamically control the CPU scheduling of Fabric components to cope with the performance interference. We implement QiOi as a user-level daemon and evaluate how QiOi mitigates the performance interference of Fabric. The evaluation results demonstrate that QiOi mitigates performance degradation of Fabric by 22% and improves Fabric latency by 2.5 times without sacrificing the performance of co-located services. In addition, we show that QiOi can support different ordering services and chaincodes with negligible overhead to Fabric performance.

Keywords: blockchain; performance interference; performance isolation, cloud computing; private blockchain; Hyperledger Fabric

check for updates

Citation: Kim, J.; Lee, K.; Yang, G.; Lee, K.; Im, J.; Yoo, C. QiOi: Performance Isolation for Hyperledger Fabric. *Appl. Sci.* **2021**, *11*, 3870. https://doi.org/10.3390/app11093870

Academic Editor: Heung-No Lee

Received: 31 March 2021
Accepted: 20 April 2021
Published: 25 April 2021

1. Introduction

Blockchain technologies offer many advantages such as immutability, transparency, serializability, and cryptographic verifiability without a single point of trust [1] as they are based on the decentralized architecture without intermediary entities for cryptocurrency [2]. In order to exploit such advantages, many industrial efforts have been made to develop blockchain platforms for different service areas including trading and settlement [3], asset and finance management [4,5], and firmware verification [6]. Hyperledger Fabric [7] is one of the most popular blockchain platforms, which is managed by Linux Foundation. It is based on private blockchain, which allows only authenticated users to participate in blockchain networks. This is different from a public blockchain where any user can join the network. In addition, Fabric introduces *execute-order-validate* architecture that overcomes the limitations of the previous *execute-order* architecture [8]. This improves the scalability of throughput in blockchain networks even with a large number of peers, which enables Fabric to be adopted in many use cases such as Global Trade Digitization [9], SecureKey [10], and Everledger [11].

Based on the *execute–order–validate* architecture, Fabric executes multiple components (e.g., clients, peers, and ordering service nodes) to carry out different operations that process user requests. The Fabric components typically run as containers on a host server [12]. The containers [13] share computing resources (e.g., CPU, memory, and network) on a single operating system, called the host operating system. Accordingly, in cloud data centers, the Fabric components can run with other services concurrently and share the underlying computing resources. Thus, the performance of Fabric can be affected by co-located services that cause performance interference. For example, if Fabric components and co-located tasks contend for CPU resources concurrently, the components may not receive sufficient CPU and experience performance interference. This leads to the performance degradation of Fabric, which decreases the quality of service and the service revenue [14,15]. Therefore, it is necessary to provide performance isolation for Fabric components that run on cloud data centers.

Previous studies on Fabric mostly focus on achieving the maximum performance rather than providing performance isolation. They assume that the Fabric components receive sufficient computing resources without any other co-located services. They suggest a guideline regarding how to configure various parameters offered by the Fabric such as block sizes, batch time, and endorsement policies for achieving the maximum performance. Furthermore, they argue that some parameters such as the type of the database and consensus protocols affect the overall performance. For example, Javaid et al. [16] found that the performance of Fabric increases with large block size, while Androulaki et al. [1] showed that GoLevelDB [17] outperforms CouchDB [18]. In addition, Yusuf and Surjandari [19] demonstrated that Raft [20] is superior to Kafka [21]. Even though previous studies show that Fabric can achieve high performance by configuring the parameters, they do not consider performance interference caused by co-located services in evaluating Fabric performance.

To provide performance isolation in cloud data centers, many studies have been conducted in recent years [22–25]. They develop scheduling policies to allocate computing resources depending on the characteristics of the tasks. However, the studies are difficult to apply on Fabric because each Fabric component has different characteristics. Note that the term *component* in the Fabric equally corresponds to process, which is a unit of CPU scheduling. For example, the most popular data analytic framework, Spark [26], consists of multiple workers that perform a uniform operation only with different sets of data in a distributed manner. As the workers conduct similar operations that are mainly CPU-intensive, their processing characteristics such as the CPU time slices are homogeneous. Therefore, the existing techniques can provide performance isolation for Spark workers by applying the same scheduling policy to the workers simultaneously. On the other hand, Fabric consists of heterogeneous components performing different operations, which makes it difficult to apply the same scheduling policy to the components. For example, peers execute transaction proposals and validate transactions while orderers determine the sequence of transactions. Our profiling results reveal that peers and orderers have different processing characteristics: peers have 2.3 times higher time slice than the orderers. If the existing studies are applied to Fabric without considering the heterogeneous characteristics of the Fabric components, interference between the components could get worse or at least not improve.

This paper presents **QiOi**, component-level performance isolation technique for Hyperledger Fabric. Note that component-level means controlling heterogeneous Fabric components that have different impacts on the performance of Fabric. First, we analyze the characteristics of Fabric components using system-level metrics such as scheduling delay and time slice when the components run as containers on the host server. Then, we demonstrate that Fabric can experience performance degradation when Fabric components run with co-located tasks. By analyzing the performance interference at the component-level, we point out that each component has different impacts on performance degradation. Based on the results, we propose QiOi to control CPU scheduling policy at component-level to offer performance isolation.

To provide component-level performance isolation, QiOi first receives a certain performance threshold from the Fabric service manager depending on a service. Furthermore, QiOi periodically monitors the performance of Fabric. When it becomes below the threshold, QiOi detects performance interference and dynamically controls CPU scheduling policy of specific components based on the QiOi algorithm. We implement QiOi as a user-level daemon to run on Linux kernel with a Hyperledger Fabric-based blockchain network. Note that QiOi does not require any modification in Linux kernel nor Hyperledger Fabric.

We evaluate QiOi with different ordering services, a variety of workloads of Fabric, and practical co-located tasks. Our evaluation results show that QiOi mitigates performance degradation of Fabric by 22% and improves latency of Fabric by 2.5 times without sacrificing the performance of co-located tasks when Fabric components run with co-located tasks.

The main contributions of this paper are as follows:

- We analyze the Fabric components using system-level metrics and show the different impacts of the components on performance interference.
- We design QiOi that offers component-level performance isolation for Fabric by periodically monitoring Fabric performance and adjusting CPU scheduling policy for Fabric components.
- We implement QiOi as a user-level daemon and present the evaluation results showing that QiOi mitigates performance degradation of Fabric by 22% and improves latency of Fabric by 2.5 times.

The rest of the paper is organized as follows. Section 2 explains details of Hyperledger Fabric and Linux CPU scheduler. In Section 3, we analyze the characteristics of Fabric component and demonstrate performance degradation of Fabric running with co-located tasks. Section 4 presents the design and implementation of QiOi, and Section 5 describes evaluation results. Then, Section 6 discusses the future works to further improve this paper, and Section 7 explains related work. Finally, we conclude in Section 8.

2. Background

In this section, we first describe the components of Hyperledger Fabric. Fabric components frequently run as containers on a host server. As containers run as user-level processes, the Linux CPU scheduler, completely fair scheduler (CFS) [27], determines the CPU allocation among containers. We therefore explain how CFS schedules processes including Fabric components.

2.1. Hyperledger Fabric

The operation of Fabric is divided into three phases – execute, order, and validate, with each phase conducted by different components such as clients (execute), peers (execute and validate), and ordering service nodes (order) as illustrated in Figure 1. The functionalities of each component are as follows.

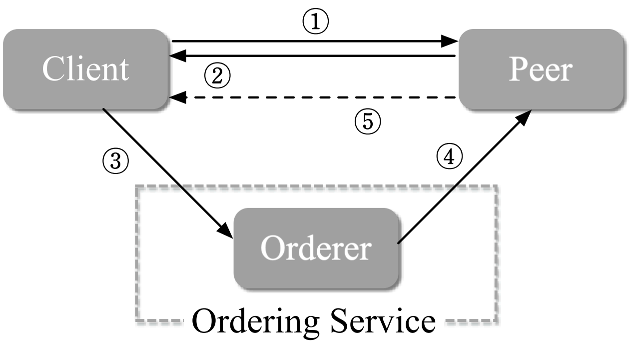


Figure 1. The Fabric component and transaction flow.

- **Clients** are the end-users of blockchain networks that submit transaction proposals to peers (① in Figure 1). The transaction proposals are executed and endorsed by the peers. When clients collect enough endorsements on a transaction proposal from peers (②), they assemble a transaction and broadcast this to ordering service nodes (③).
- **Peers** execute transaction proposals and validate transactions. There are two types of peers—endorsing peers and committing peers. The endorsing peers receive the transaction proposal from clients, execute the operation on the specified chaincode, and run the endorsement system chaincode (ESCC) to verify the transaction proposal.

A chaincode is a program code that implements and executes the Fabric service logic. The chaincode is the key to Fabric service and runs as a Docker container, isolated from the peers. ESCC is one of the special chaincodes for managing the Fabric system and maintaining parameters. Then, if the verification results satisfy the endorsement policy, the endorsing peers return the transaction proposal to the client with a digital signature using the private key of the endorsing peers (②). The committing peers validate new blocks with the endorsing peers by executing the validation system chaincode (VSCC) before the actual commit of the new blocks to the distributed database, called a ledger. When validation is completed, a new block that contains the proposed transaction is stored in the ledger. Every peer maintains the ledger locally. Then, one of the peers informs the clients that the transaction has been registered successfully (⑤).

- **Ordering service nodes (i.e., orderers)** are the nodes that compose the ordering service. The ordering service determines the sequence of transactions into blocks using a consensus protocol. Fabric offers three consensus protocols—Solo, Raft, and Kafka. Solo is intended for testing only and consists of a single orderer. Raft and Kafka utilize a "leader and follower" model. In Raft, a leader of the orderers is elected and determines the sequence of transactions. Then, it distributes the replications of ordered transactions to the followers. The distribution of the replicated transactions enables the Raft-based ordering service to provide crash fault tolerance. Raft selects one of the followers as a leader and runs it in cases where the previous leader crashes. The Kafka-based ordering service consists of orderers, Kafka brokers, and ZooKeeper. The Kafka brokers are categorized into a leader and followers. ZooKeeper manages the Kafka brokers by selecting a leader out of the brokers and monitoring that the Kafka brokers are working properly. The Kafka leader determines the sequence of transactions. Kafka brokers that are not selected as a leader become followers. The leader of the Kafka brokers transfers ordered transactions to orderers.

  The orderers (i.e., both Raft and Kafka) generate new blocks using the received transactions when one of the three conditions is satisfied: (1) the number of transactions in the block reaches the threshold, (2) the block size attains the maximum value in bytes, or (3) certain time has passed since the first transaction of the new block was received. When the block generation is finished, the orderers deliver the blocks to all peers (both endorsing and committing) for validation (④).

2.2. Linux CPU Scheduler

In this subsection, we first explain the scheduling mechanism of CFS and then the parameters of CPU scheduling as background for performance isolation.

To allocate CPU resources between processes fairly, CFS first calculates the *period* depending on the number of processes in the run queue. Then, CFS determines a *time slice* for each process depending on its *load weight*. When a specific process is scheduled, it runs the *time slice* of the process. The process may fully use the *time slice* or it may not use all of the *time slice* by preemption. As the process is preempted, CFS measures the execution time of the process. When the process is dequeued from the run queue, the virtual runtime (*vruntime*) is updated based on the measured execution time. Depending on its *vruntime*, the process is sorted and stored in the red-black tree, which is a self-balancing binary search tree [28]. A process with the least *vruntime* will be the leftmost node in the tree. When CFS schedules the next process to run, it picks the leftmost node. Details of the main parameters, *load weight* and *vruntime*, are as follows.

First, the *load weight* is given to all processes, and CFS provides the CPU time (*time slice*) for processes in proportion to their *load weight* [29]. *time slice* is calculated as in Equation (1). Note that the default *load weight* is 1024 in Linux kernel. The *period* is 24 ms by default if the number of processes in the run queue is less than eight. Otherwise, the *period* is calculated by multiplying the number of processes in the run queue by 3 ms. The value of 3 ms is set to avoid preempting processes too frequently in Linux kernel. $se \rightarrow load.weight$ indicates the *load weight* of a process, and *cfs_rq* $\rightarrow$ *load.weight* indicates the sum of *load weights* of all processes in the run queue.

$$time\ slice = period \times \frac{se \rightarrow load.weight}{cfs\_rq \rightarrow load.weight} \quad (1)$$

Second, the *vruntime* is inversely proportional to the amount of CPU time that a process has already used. The process with the lowest *vruntime* has the highest scheduling priority. Equation (2) shows how CFS calculates *vruntime*, where *delta_exec* is CPU time actually used. If a process has a high *load weight*, it receives small *vruntime* compared to other processes consuming similar amounts of CPU time. This enables the process with the high *load weight* to be scheduled frequently.

$$vruntime = delta\_exec \times \frac{default\ load.weight}{se \rightarrow load.weight} \quad (2)$$

In order to control the *load weight*, CFS offers an adjustable parameter called *CPU share*. Since Linux cgroups feature calls *load weight* as *CPU share*, we use the term *CPU share* for the rest of this paper [30].

3. Motivation

Now, we analyze the characteristics of Fabric components and demonstrate that Fabric performance can be affected by the co-located tasks on a host server. Then, we explore the cause of performance degradation at the component level and show that the components have different impacts on the performance degradation of Fabric.

3.1. Characteristics of Fabric Components

First, we measure system-level metrics such as scheduling delay to understand the different characteristics of Fabric components. Since Fabric consists of multiple components that perform different operations, it is essential to characterize them component by component. Furthermore, such characterization is necessary to develop a performance isolation techniques for Fabric.

For experiments, we utilized a server equipped with two Intel Xeon processors (E5-2650 v4@2.2 GHz, 12 cores) running Ubuntu 16.04 LTS (Linux v4.15.0) with 128 GB of RAM and 447 GB HDD as a local disk. In addition, we used Fabric v1.4.1 with the Raft-based ordering service and Hyperledger Caliper v0.2.0. Hyperledger Caliper (Caliper) [31] is a widely-used performance benchmark framework for blockchain networks. We utilized Caliper to construct the blockchain network based on Fabric and to measure Fabric performance in terms of throughput and latency by executing various chaincodes provided by Caliper. In our experiments, Fabric components ran as containers on Docker v19.03.13. Figure 2 shows the setup of Fabric components used in our experiments, which consisted of four peers, three orderers using GoLevelDB. We configured Caliper to run 10 clients that generated and transmit 10,000 transaction proposals in total. As chaincode, we ran the smallbank [32] which is one of the most widely used benchmarks and performs account opening, deposit, and money transfer [16]. Note that Table 1 shows the detailed configuration of Fabric.

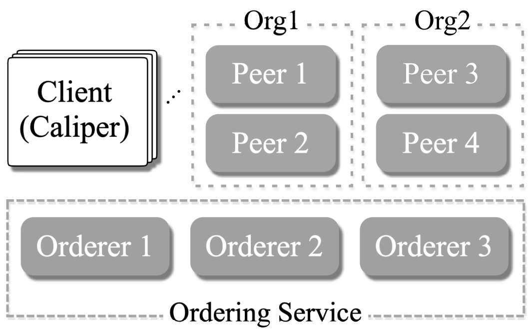


Figure 2. Experimental setup of Fabric components.

Table 1. Fabric configuration.

Parameters	Values
Ordering service	Raft
StateDB	GoLevelDB
Endorsement policy	OR(Org1, Org2)
Block size/timeout	100 MB/2 s
Total transactions	10000
Transaction send rate	1000

We utilized a system-level profiling tool, *perf sched*, and measure scheduling metrics, such as average time slices, scheduling delay, and the number of context switches. The average time slice is the average CPU time occupied by a process while Fabric is running, and the scheduling delay refers to the length of time the process waits for it to be scheduled. The number of context switches indicates the number of stopping a process to starting another one.

Table 2 presents the results of the average time slices, scheduling delay, and the number of context switches per second for Fabric components. We find that all scheduling metrics are in the order of peers, orderers, and chaincodes. This means that peers consume the largest CPU time among the components and are more scheduled than orderers and chaincodes. We find that this is because peers perform verification of the client's transaction proposals and validate new blocks. On the other hand, orderers and chaincodes consume relatively less CPU time. In addition, the average time slice and scheduling delay of all components do not exceed 0.53 ms, making them quite short periods of time. This indicates that Fabric components finish their operations in a short period of time while being scheduled rather frequently. Note that Fabric performance may vary depending on the Fabric version. However, this paper focuses on demonstrating that Fabric experiences performance interference because of the co-located tasks. Because peers and orderers perform similar operations regardless of the version after v1.4.1, we believe that we will have similar results with the latest Fabric version. Note that the latest versions after v2.0 mostly focus on supporting functionalities for chaincode and channel [33].

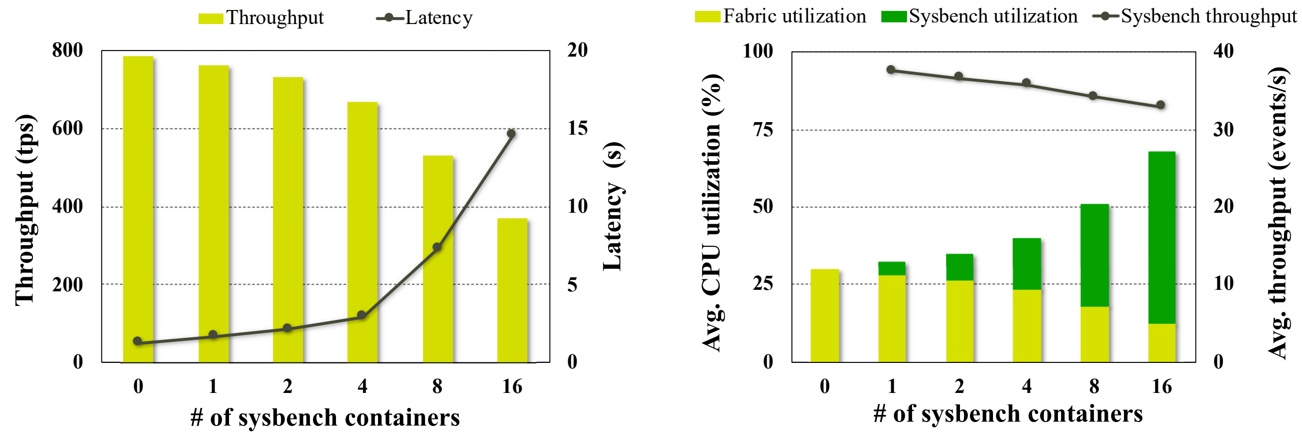
Table 2. System-level profiling of Fabric components.

Component	Avg. Time Slice (ms)	Scheduling Delay (ms)	The Number of Context Switch/s
Peer	0.22	0.53	13051
Orderer	0.10	0.41	7947
Chaincode	0.05	0.10	6241

3.2. Motivating Example

This subsection demonstrates that the performance of Fabric can be affected by co-located tasks. In the same experimental setting in Section 3.1, we run the sysbench CPU benchmark [34] on containers (i.e., sysbench containers). The sysbench CPU benchmark is one of the most commonly used benchmarks in the related works and performs computation-intensive operation similar to the map phase in Spark [22,35]. We therefore believe believe that sysbench represents a realistic interference case in multi-tenant environments. In addition, we measured the performance and CPU utilization of Fabric and sysbench containers, respectively. Furthermore, we increased the number of sysbench containers from 0 to 16 and compared the results. Note that when Fabric components run with 16 sysbench containers concurrently, the maximum CPU utilization of the entire system is 2286% including Caliper and other system daemons. This means that Fabric components and sysbench containers have sufficient computing resources on the experiment server with 24 CPU cores.

Figure 3a shows Fabric performance in terms of throughput (left y-axis as bars) and latency (right y-axis as a solid line) depending on the number of sysbench containers (x-axis). We found that the Fabric throughput decreased, while latency increased as the number of sysbench containers increased. Note that the value of zero indicates that there are only Fabric components without co-located tasks on the host server, which results in the maximum baseline Fabric performance. When Fabric components run with eight sysbench containers concurrently, throughput decreases by 32% and latency increases by 5.7 times compared to the maximum performance. When there are 16 sysbench containers, the Fabric throughput decreases by 53% and latency increases by a factor of 11.5.



(**a**) Fabric performance comparison in terms of increasing number of sysbench containers.

(**b**) Average CPU utilization and sysbench throughput depending on increasing number of sysbench containers.

Figure 3. Impact of co-located tasks on Fabric.

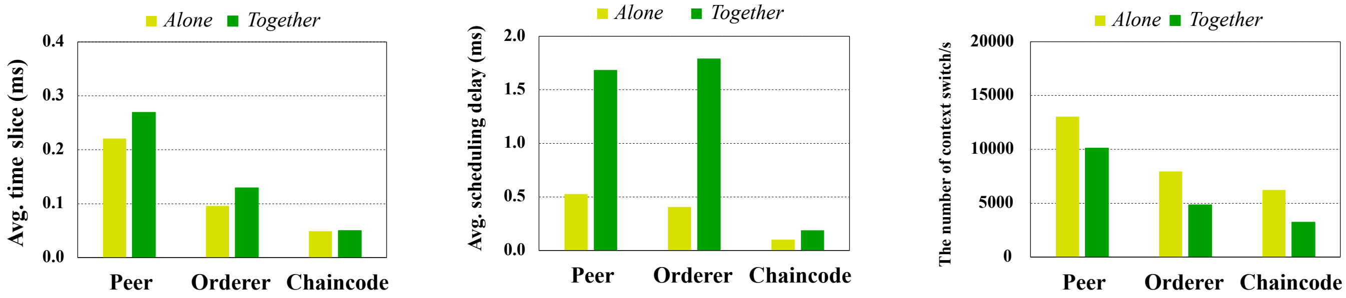
Figure 3b depicts the average CPU utilization of the Fabric and sysbench containers (left y-axis as bars) and the average throughput of sysbench containers (right y-axis as a solid line) depending on the number of sysbench containers (x-axis). As the number of sysbench containers increased, the CPU utilization of sysbench containers increased while that of Fabric components decreased. For example, when eight sysbench containers ran concurrently with Fabric components, the sysbench containers utilized 33% of CPU, while Fabric components consumed 18% of CPU, which is a 40% reduction compared to the case without sysbench containers. Note that throughput of sysbench containers does not decrease even though the sysbench containers run with Fabric components.

In summary, we show that the CPU utilization of Fabric components decreases when we increase the number of sysbench containers. This results in performance degradation of Fabric up to 53% compared to the maximum performance. Therefore, it is necessary to develop the performance isolation technique for Fabric to prevent the performance interference by co-located tasks.

3.3. Performance Interference Analysis

Next, we explore the cause of performance degradation in Figure 3 at the component-level and show that the components have different impacts on performance degradation. We measure the system-level metrics in Table 2 for each component and compare the results when Fabric components run without the sysbench containers (i.e., *Alone*) and with eight sysbench containers (i.e., *Together*).

First, Figure 4a depicts that the average time slice of peers, orderers, and chaincodes in *Together* increases by 22%, 35%, and 3%, respectively, compared to those of *Alone*. This indicates that the Fabric components in *Together* run on CPUs longer than in *Alone*. In addition, Figure 4b shows that the average scheduling delay for each component increases by 3.2 times, 4.4 times, and 1.9 times, respectively. This is because the Fabric components need to wait until the sysbench containers consume their time slice, which is quite a long time, i.e., 58 ms as illustrated in Table 3. As a result, the Fabric components experience severe scheduling delay, which leads to performance degradation of Fabric. The number of context switches in *Together* in Figure 4c decreases by 22%, 39%, and 48%, respectively, compared to that of *Alone*. This means that the total runtime of the Fabric components decreases, which results in the decrease in Fabric CPU utilization.



(**a**) Average time slice. (**b**) Average scheduling delay. (**c**) Context switch.

Figure 4. System-level profiling of Fabric components in terms of scheduling metrics.

Table 3. System-level profiling of sysbench containers.

	Avg. Time Slice (ms)	Scheduling Delay (ms)	The Number of Context Switch/s
Alone	328	0.30	24
Together	58	0.53	138

Second, Table 3 shows the system metrics of the eight sysbench containers when the containers run without the Fabric components (i.e., *Alone*) and with the Fabric components (i.e., *Together*). Although the time slice of sysbench containers in *Together* decreases by 5.7 times, the number of context switches increases by 5.65 times. Therefore, the average CPU utilization of sysbench containers slightly decreases from 34% to 33%, and throughput of sysbench containers does not decrease.

From these results, we identify that the scheduling delay is a key bottleneck in performance interference. Therefore, a design goal of QiOi is to control the scheduling delay via CPU allocation of Fabric components. We focus on the orderers and the peers because their impacts have been known on determining Fabric performance [16,36–38].

4. Design and Implementation

This section presents QiOi, component-level performance isolation technique for Fabric. First, we explain that *CPU share* can be controlled to mitigate performance degradation of Fabric based on the analysis results in Section 3.3. Then, we introduce the design of QiOi and describe the overall architecture, implementation, and algorithm in detail.

4.1. Our Approach

In Section 3.3, we observe that the cause of performance degradation in Fabric is the increased scheduling delay of orderers and peers when the Fabric components run with sysbench containers. To mitigate performance degradation of Fabric, we propose to adjust one of the CPU scheduling parameters, *CPU share*, to minimize the scheduling delay of orderers and peers. In other words, we control *load weight* and *vruntime* by utilizing *CPU share*.

For example, when there are two processes (e.g., A and B) in the run queue and different values of *CPU share* are given to each process (e.g., A = 1024, B = 2048), process B receives twice as much CPU time as process A. In addition, if the processes with a high *CPU share* occupy a short period of time on CPU that is similar to that of orderers and peers in Fabric, they will be given higher scheduling priorities by receiving small *vruntime*. As a result, they get scheduled to CPU more frequently than other processes with a low *CPU share*.

Therefore, if we provide a high *CPU share* for orderers and peers, they will receive more CPU time and get scheduled to CPU more frequently, which can decrease their scheduling delay. In short, we propose dynamically controlling the value of *CPU share* for orderers and peers to reduce the scheduling delay of Fabric.

4.2. QiOi Architecture

QiOi is designed as follows: (1) monitor the performance of Fabric to detect performance interference, (2) dynamically control the *CPU share* of orderers and peers to minimize scheduling delay. The architecture of QiOi consists of performance monitor and QiOi controller as in Figure 5. QiOi is implemented as a user-level daemon on the host Linux with Hyperledger Fabric-based blockchain network. It adjusts *CPU share* of the Fabric components while monitoring Fabric performance.

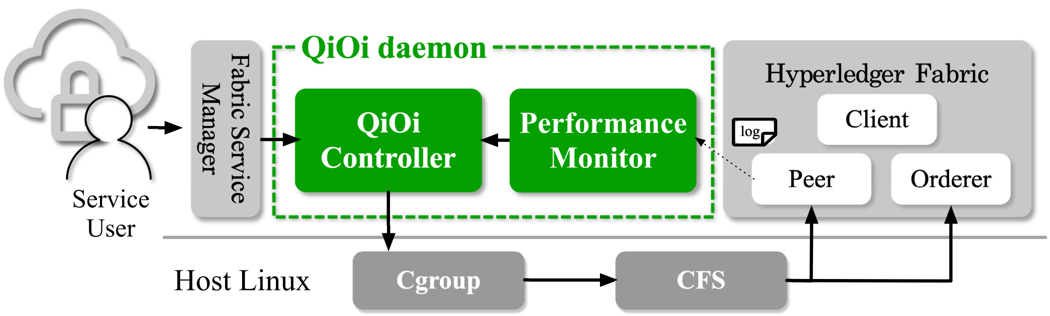


Figure 5. QiOi architecture.

We assume that the isolation threshold is given via Fabric service manager in Figure 5 because determining the isolation threshold is an orthogonal problem to the performance interference. The isolation threshold is a baseline for detecting performance interference and indicates the maximum throughput of Fabric, which varies depending on Fabric network configuration. Then, the Fabric service manager delivers the isolation threshold to the QiOi controller. The performance monitor periodically collects the actual throughput of Fabric and delivers it to the QiOi controller. The QiOi controller detects performance interference by comparing the isolation threshold with the monitored throughput. Then, QiOi adjusts *CPU share* of orderers and peers using cgroups. A detailed explanation of the architecture and implementation is as follows:

- **Performance monitor** periodically collects the Fabric throughput by analyzing Fabric network information in the container log of the peers. The container log is stored in the file *[container id]-json.log* under the directory */var/lib/docker/ containers/[container id]*. The Fabric network information contains transaction proposal, chaincode verification,

and block commit with the timestamp. The performance monitor parses the Fabric network information every second and calculates throughput using the block commit information. Note that the monitoring period in QiOi is also adjustable.
Throughput is the rate at which transactions are committed by the Fabric network in a time period. This rate is expressed as transactions per second (i.e., tps) [12]. During a period of time from t_i to t_j, throughput is calculated as in Equation (3). $N(t)$ is the total number of committed transactions at t.

$$throughput = \frac{the\ number\ of\ committed\ transactions}{time\ period} = \frac{N(t_j) - N(t_i)}{t_j - t_i} \quad (3)$$

- **QiOi controller** adjusts *CPU share* of orderers and peers when performance interference occurs. The QiOi controller regards a situation when the actual throughput is less than the isolation threshold received from the Fabric service manager as performance interference. Then the QiOi controller adjusts the *CPU share* of orderers and peers with the algorithm explained in Section 4.3. The adjusted *CPU share* is applied to CFS.

4.3. QiOi Algorithm

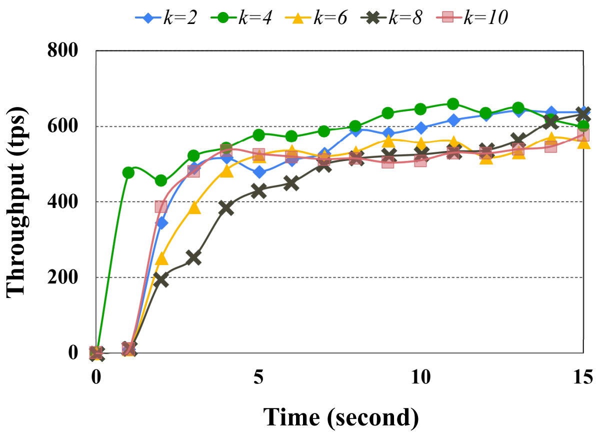
As shown in Algorithm 1, the QiOi algorithm is based on proportional control. In other words, the *CPU share* (S) is proportionally calculated depending on the difference between the isolation threshold and actual throughput (CPU_{diff}). Note that $S_{default}$ is default *CPU share*. If the actual throughput is lower than the isolation threshold, the *CPU share* of orderers and peers is increased to improve throughput to the isolation threshold. Otherwise, the value of the previous *CPU share* is sustained. For example, if the isolation threshold and the actual throughput are 500 tps and 400 tps, respectively, the QiOi controller detects performance interference because the actual throughput is below the isolation threshold. The QiOi controller calculates the value of the next *CPU share* (S_{next}) to be 1024 + 1024 $\times\ k \times \frac{100}{500}$. Note that the QiOi algorithm works on both orderers and peers.

Algorithm 1: QiOi algorithm.

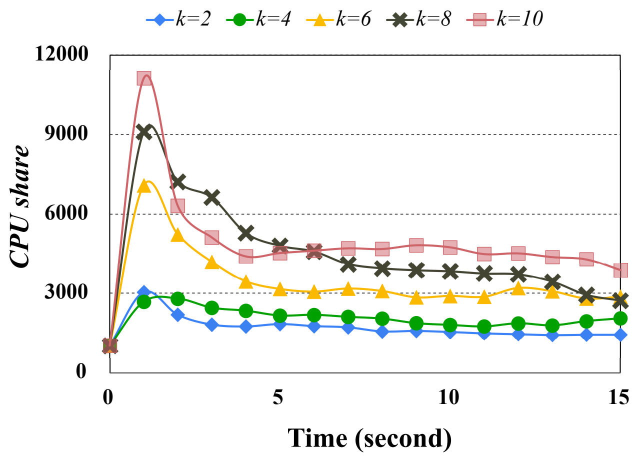
T: Isolation threshold
A: Actual throughput
S: *CPU share* of orderers and peers
$diff = |T - A|$
if $diff > 0$ **then**
  $CPU_{diff} = S_{default} \times k \times \frac{diff}{T}$
  $S_{next} = S_{default} + CPU_{diff}$;
else
  $S_{next} = S_{prev}$;
return S_{next}

The value of k is an adjustable parameter that affects the rate of the *CPU share* adjustment. As the k value increases, the *CPU share* changes rapidly. For a small k value, the scheduling delay of the orderers and peers may not minimize effectively. For a large k value, the *CPU share* of orderers and peers can easily become significantly larger than that of other Fabric components, which may increase the scheduling delay of other components. Therefore, it is necessary to find a proper k value providing performance isolation more effectively.

To investigate the impact of the value of k, we conduct experiments with the same setting used in Section 3. The number of sysbench containers is eight and smallbank chaincode takes about 15 seconds to execute transaction proposals. Figure 6a shows Fabric throughput (y-axis) over time (x-axis) when the value of k is changed. When the value of k is 4, Fabric throughput increases by 14% (from 533 tps to 607 tps) compared to throughput without QiOi. Figure 6b depicts the *CPU share* of orderers and peers (y-axis) over time (x-axis) when the value of k is changed. By the QiOi algorithm, the *CPU share* of orderers and peers becomes high compared with other processes that have default *CPU share* (1024), so CFS allocates a larger amount of *time slice* to them and schedules more frequently. Based on these results, we empirically determined the value of k and set it to 4. Figure 7 compares the average scheduling delay of Fabric components among without sysbench containers and QiOi (*Alone*), without QiOi (*Together*), and with QiOi (*Together w/ QiOi*). As a result, their scheduling delay in *Together w/ QiOi* is decreases by 31% (peers) and 58% (orderers), respectively, compared to those of *Together*, which means that performance interference is mitigated. On the other hand, the *CPU share* of chaincodes keeps the default value, which is lower than that of orderers and peers. Therefore, the scheduling frequency and the amount of time slice decrease. For this reason, the scheduling delay of chaincodes increases from 0.19 ms (*Together*) to 0.24 ms (*Together w/ QiOi*). Note that Table 4 depicts throughput of sysbench containers depending on the value of k. *Baseline* indicates the throughput of sysbench containers before applying QiOi. The throughput of sysbench containers drops by 1.2% on average, and QiOi does not sacrifice throughput of sysbench containers.



(**a**) Fabric throughput over time.



(**b**) *CPU share* of orderers and peers over time.

Figure 6. Impact of QiOi algorithm depending on the value of k.

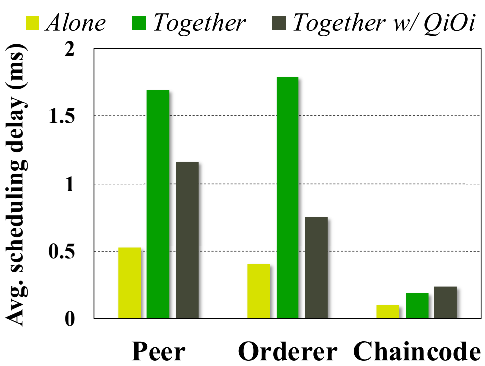


Figure 7. Comparison of scheduling delay for Fabric components.

Table 4. Sysbench throughput depending on the value of k.

The Value of k	*Baseline*	2	4	6	8	10
Sysbench throughput (events/s)	34.2	33.9	33.7	33.9	33.6	33.9

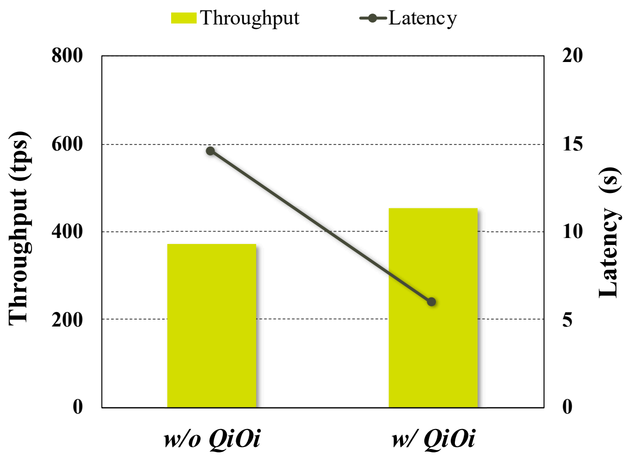
5. Evaluation

In this section, we comprehensively evaluate QiOi through four sets of experiments. Note that we utilized the same experiment setting as in Section 3. First, we show that QiOi can offer performance isolation when Fabric network is composed of different ordering services (Section 5.1) and chaincodes (Section 5.2). Then, we demonstrate the impact of QiOi when Fabric runs with practical co-located tasks such as Spark on a host server (Section 5.3). Finally, we investigate the overhead of QiOi in terms of CPU utilization (Section 5.4).

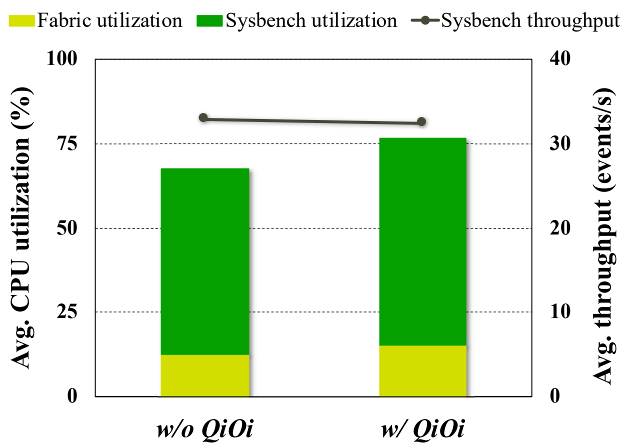
5.1. Support for Different Ordering Services

First, we evaluate QiOi when the Fabric network runs different types of ordering services. Fabric offers three ordering services, i.e., Solo, Raft, and Kafka, and users can select one of the ordering services depending on their purposes. Therefore, it is important to support the different ordering services in providing performance isolation. For experiments, we run Raft and Kafka, respectively, for the ordering services, as they are considered representative ordering services in Fabric. We run the smallbank chaincode of Caliper and measure the performance and CPU utilization of Fabric and sysbench containers, respectively, when the Fabric components run with the sysbench containers. Note that the number of sysbench containers is 16, and the Kafka-based ordering service consists of three orderers, four Kafka brokers, and three ZooKeepers.

Figure 8 illustrates the impact of QiOi on the Raft-based Fabric network. In the x-axis, *w/o QiOi* indicates before applying QiOi, and *w/ QiOi* indicates after applying QiOi. Figure 8a shows that Fabric throughput (left y-axis as bars) increases by 22% while latency (right *y*-axis as a solid line) decreases by 2.5 times when we apply QiOi. This indicates that QiOi can mitigate performance degradation of Fabric running with sysbench containers. In addition, Figure 8b depicts that the average CPU utilization of Fabric (left *y*-axis as bars) increases by 24% with QiOi. This is because QiOi controls the *CPU share* of orderers and peers when Fabric performance decreases because of the sysbench containers. Although QiOi improves Fabric performance by increasing the CPU utilization of Fabric components, the average throughput of sysbench containers (right *y*-axis as a solid line) only decreases by 2%, which means it is negligible. Note that the average CPU utilization of sysbench containers (left *y*-axis as bars) increases by 11%.



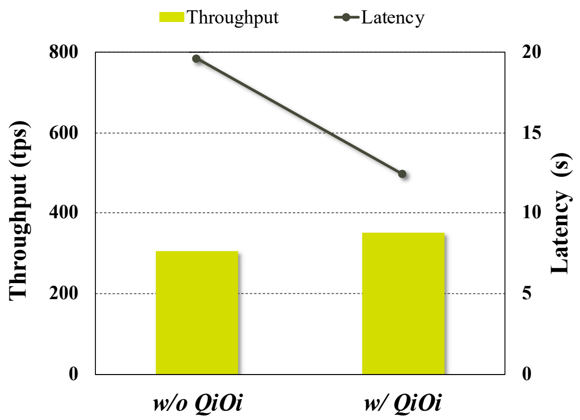
(**a**) Fabric throughput and latency.



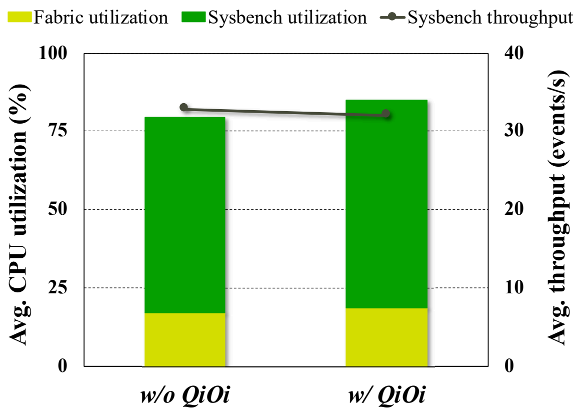
(**b**) CPU utilization and sysbench throughput.

Figure 8. Impact of QiOi on Raft ordering service-based Fabric network.

Figure 9 shows that QiOi is also effective for a Kafka-based Fabric network. Similar to the result of a Raft-based Fabric network, Fabric throughput increases by 15%, and latency decreases by 37% when we apply QiOi as presented in Figure 9a. In addition, Figure 9b illustrates that performance degradation of sysbench containers is negligible in Kafka as well (decreasing by 2%) and CPU utilization of Fabric and sysbench containers increases by 9% and 6%, respectively. Although Raft and Kafka have some differences in architecture and mechanism, we show that QiOi can offer performance isolation for both of ordering services. This is because QiOi adjusts the *CPU share* of orderers and peers independent of the type of ordering services. Note that from the experiments in Section 5.2, we focus on the Raft-based ordering service because it is highly recommended by Hyperledger and known for easier management than Kafka [39].



(**a**) Fabric throughput and latency.



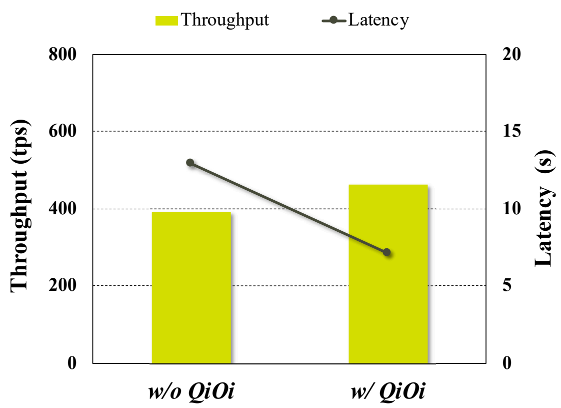
(**b**) CPU utilization and sysbench throughput.

Figure 9. Impact of QiOi on Kafka ordering service-based Fabric network.

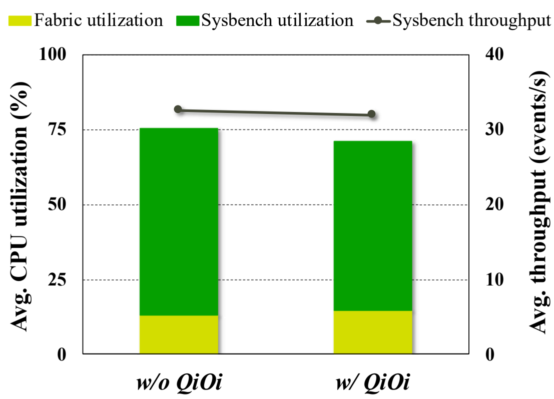
5.2. Support for Different Chaincodes

Now, because Fabric performance and resource usage are different depending on the chaincodes of Fabric, we investigate the effectiveness of QiOi when Fabric runs different chaincodes. For experiments, we run simple and marbles chaincodes [32]. Simple chaincode executes simple queries such as checking and storing values. Marbles chaincode consists of a series of transactions transferring marbles between multiple marble owners and stores data as various data structures such as JSON format. We measured the performance and CPU utilization of Fabric and sysbench containers, respectively, when the Fabric components ran with 16 sysbench containers.

Figure 10 illustrates the impact of QiOi when simple chaincode is executed in Fabric. Figure 10a shows Fabric throughput (left *y*-axis as bars) and latency (right *y*-axis as a solid line) compared between *w/o QiOi* and *w/ QiOi*. When QiOi runs with Fabric, Fabric throughput increases by 18% and latency decreases by 1.8 times. In addition, Figure 10b shows that the CPU utilization of Fabric (left y-axis as bars) increases by 12% and throughput of sysbench containers (right y-axis as a solid line) only decreases by 2%. This is because QiOi minimizes scheduling delay by increasing the *CPU share* of orderers and peers. As a result, QiOi lets the CPU utilization of Fabric increase and then improves Fabric performance. Note that CPU utilization of sysbench containers (left *y*-axis as bars) decreases by 9%.



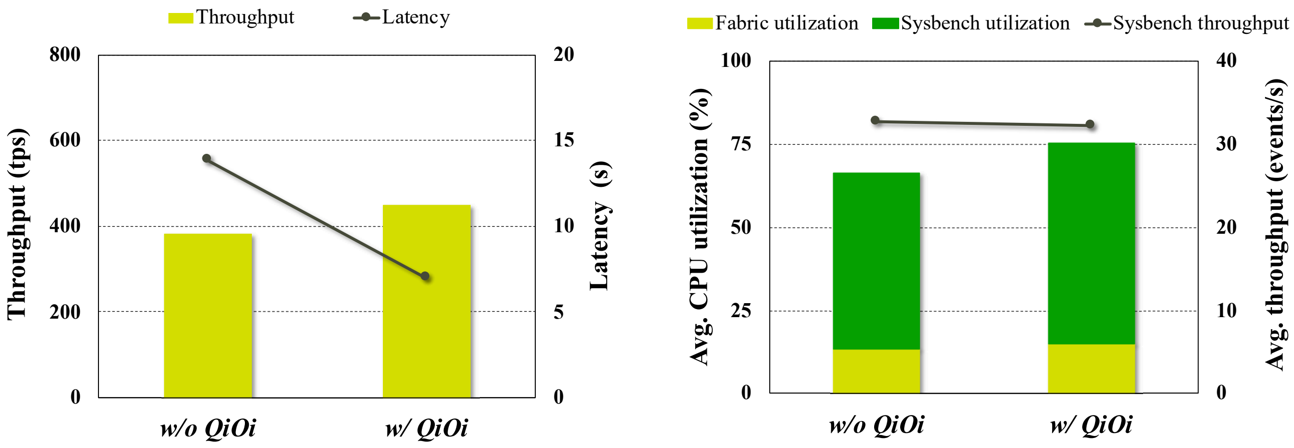
(**a**) Fabric throughput and latency.



(**b**) CPU utilization and sysbench throughput.

Figure 10. Impact of QiOi on Fabric using simple chaincode.

Figure 11 shows the result of marbles chaincode. Fabric throughput increases by 17%, and latency decreases by 2 times. Figure 11b illustrates that performance degradation of sysbench containers is negligible in marbles chaincode as well (decreasing by 1%) and CPU utilization of Fabric and sysbench containers increases by 13% and 14%, respectively.



(**a**) Fabric throughput and latency. (**b**) CPU utilization and sysbench throughput.

Figure 11. Impact of QiOi on Fabric using marbles chaincode.

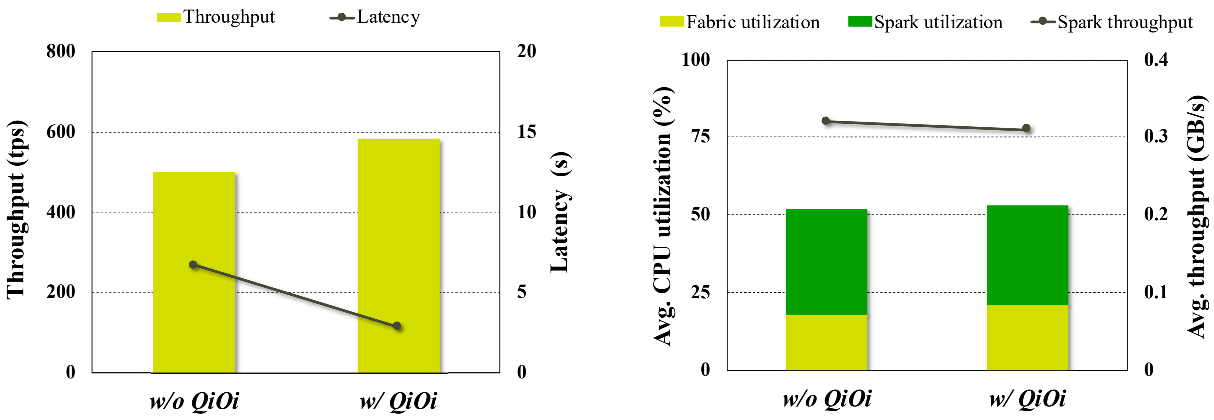
The results show that there are some differences in the performance improvement rate between Fabric chaincode (i.e., smallbank, simple, and marbles). In terms of Fabric throughput, the improvement rate is high in the order of smallbank (see Section 5.1), simple, and marbles (increasing by 22%, 18%, and 17%, respectively). In terms of Fabric latency, the improvement rate is high in the order of smallbank, marbles, and simple (decreasing by 2.5 times, 2 times, and 1.8 times, respectively). Each chaincode has different program code and is simulated on the peers. Fabric performance depends on the complexity of the chaincode, the time required for simulation, and the micro-architecture metrics required [40]. Regardless of chaincodes, QiOi focuses on minimizing the scheduling delay of components. For this reason, there is a difference in the performance improvement rate for each chaincode. Smallbank has the highest improvement rate in both throughput and latency. Simple and marbles have a similar improvement rate.

In summary, this experiment shows that the QiOi can mitigate performance degradation and improve latency in various chaincodes of Fabric.

5.3. Support for Practical Co-Located Tasks

This subsection evaluates whether QiOi can provide performance isolation when Fabric components run with a practical task other than sysbench. We chose Spark [26] as it is a widely-used cloud service, which processes data analytic operations in a distributed manner to enable fast big data processing. There are map and reduce phases in Spark: the map phase involves processing the input data and creating several small chunks of data, while the reduce phase involves processing the data that come from the map phase and producing a new set of outputs. Spark uses a master–worker architecture. A master is the worker cluster manager that accepts jobs to run and schedules resources among workers. Each worker executes the jobs. For experiments, we utilize eight workers running as containers with two threads (i.e., Spark containers) and *Terasort* job with a 47 GB input file for Spark. *Terasort* job sorts some number of data as quickly as possible.

Figure 12 illustrates the impact of QiOi when Fabric runs with eight Spark containers on a host server. Figure 12a shows Fabric throughput (left y-axis as bars) and latency (right y-axis as a solid line) compared between *w/o QiOi* and *w/ QiOi*. When QiOi runs with Fabric, Fabric throughput increases by 16%, and latency decreases by 2.4 times. On the other hand, the throughput change of Spark containers (right y-axis as a solid line) is negligible, which is only 3% of a decrease as in Figure 12b. Because Spark containers perform the compute-intensive map phase [35,41,42], the Fabric components can experience scheduling delay similar to the case with sysbench containers. Note that CPU utilization of Fabric increases by 17%, and that of Spark containers decreases by 5%.



(**a**) Fabric throughput and latency. (**b**) CPU utilization and Spark throughput.

Figure 12. Impact of QiOi on Fabric running with Spark containers together.

To sum up, QiOi dynamically controls the *CPU share* of the orderers and peers, it can improve Fabric performance running with Spark containers simultaneously.

5.4. Overhead Measurement

Finally, we evaluate the overhead of QiOi. Note that the algorithm of QiOi has O(1) time complexity. We measure CPU utilization and Fabric performance when the Fabric components run with QiOi (*w/ QiOi*) and without QiOi (*Alone*) when there are no co-located tasks. Note that we run the smallbank chaincode.

Table 5 shows that the average CPU utilization of Fabric components decreases by 1.4% (peers) and 1.9% (orderers), respectively, with QiOi. This shows that QiOi does not incur additional overhead in terms of CPU utilization. In addition, Fabric throughput and latency decrease by 0.2% and 27%, respectively. This indicates that QiOi does not decrease Fabric performance.

Table 5. Comparison of Fabric performance and average CPU utilization.

	Throughput (tps)	Latency (s)	Peer CPU Utilization (%)	Orderer CPU Utilization (%)
Alone	787	1.27	22.3	5.4
w/ QiOi	785	0.93	22.0	5.3

6. Discussion

Providing performance isolation in distributed environments: We evaluated QiOi in a host server because this paper focuses on providing performance isolation for Fabric components running in a host server. However, we believe that QiOi is also extensible to distributed environments where each Fabric component runs in different host servers. For example, QiOi can be implemented as a plugin for Kubernetes [43], the most popular container orchestration platform. When Kubernetes creates the Fabric components as containers, QiOi can send requests to assign *CPU share* values to corresponding Fabric components. Since Kubernetes supports cgroups features that can reserve or limit computing resources for containers, QiOi can exploit the functionality of Kubernetes to assign the *CPU share* values to the Fabric components.

Providing performance isolation for multiple tenants: When QiOi runs with Fabric in a cloud data center, it should offer multi-tenancy support. Fabric utilizes channels to offer multi-tenancy in a Fabric network. Channel [8] is a private subnet in a Fabric network, which conducts confidential transactions between two or more specific network members. Hence, each tenant is allocated in different channels while the ordering service is shared by all channels in the Fabric network. Peers can belong to multiple channels and have ledgers for each channel. Furthermore, only components in a channel can participate in processing transactions of the corresponding channel and can access the ledger of the channel. We believe that QiOi can support multiple channels by extending the architecture. First, for performance monitoring, QiOi can monitor the actual performance of each channel by analyzing the transaction logs of peers in different channels. In addition, QiOi can control the *CPU shares* of the peers belong to different channels by managing per-tenant performance policy. The per-tenant performance policy means that QiOi can only control the *CPU shares* of the tenants that achieve lower performance than the isolation threshold.

7. Related Work

7.1. Performance Evaluation and Characterization

Since Hyperledger Fabric was introduced in 2015 [7,8], many studies have characterized the performance of Fabric. Most studies aim to evaluate throughput and latency with different parameters. For example, Nasir et al. [44] analyzed the performance of two versions of Hyperledger Fabric (v0.6 and v1.0). They showed that Fabric v1.0 outperforms Fabric v0.6. Furthermore, Fabric v1.0 provides stable performance independent of the number of peers. In [1,45,46], the authors demonstrated the impact of different parameters such as batch sizes, transaction types, different endorsement policies, and the number of peers on Fabric performance. They showed that small batch sizes reduce performance, while reducing the number of endorsements in the endorsement policy improves performance. Furthermore, performance decreases as the number of peers increases. In addition, the experimental results in [1] indicate that the complex endorsement policy leads to performance degradation and the increase of the CPU and network utilization.

Javaid et al. [16] found that different types of databases affect Fabric performance. For example, when the block size increases in GoLevelDB and CouchDB, latency of the validate phase in CouchDB is approximately four times higher than that in GoLevelDB. Kuzlu et al. [47] analyzed the performance of blockchain networks depending on the parameters such as transaction type, number of transactions, transaction sending rate, and concurrent execution of several chaincodes. When the number of concurrently executing transactions increases, throughput also increases. However, when throughput reaches its maximum value, latency increases without further throughput increase.

In [19,48], the authors compared different consensus protocols such as Solo, Kafka, and Raft. Yusuf and Surjandari [19] demonstrated that Raft is superior to Kafka. Wang and Chu [48] investigated the performance characterization of each phase, including execute, order, and validate phase. The authors found that the validate phase can become a bottleneck due to the low validation speed of chaincodes.

In short, previous studies have carried out experiments to achieve a maximum performance in environments where sufficient computing resources were provided, using individual VMs with a large number of virtual CPUs (e.g., 16 or 32). However, they did not analyze the characteristics of each component in terms of CPU scheduling and did not consider performance interference when Fabric components run with other services concurrently. Thus, QiOi is complementary to these studies. By considering the different characteristics of Fabric components, QiOi prevents performance interference with other services.

7.2. Performance Improvement

Several studies have proposed to improve the performance of Fabric by optimizing Fabric architecture or enhancing the processing routines (as shown in Table 6). Previous research [1,16] pointed out the following performance bottlenecks: (1) repeated de-serialization of chaincode information including identity and certificate, (2) sequential validation of transactions during VSCC, and (3) delay of multi-version concurrency control (MVCC), which ensures no read–write conflicts between the transactions. In order to resolve these bottlenecks, they propose to cache chaincode information, to execute VSCC in parallel, and to use bulk read–write during MVCC validation. Their evaluation results [16] showed that the proposed techniques improve the Fabric performance by 2 times and 1.3 times in CouchDB and GoLevelDB, respectively.

Table 6. Comparison of related work in terms of Fabric analysis.

	Typical Characterization of Fabric	Optimization for Performance Improvement	Performance Interference in Fabric
Nasir et al. [44]	✓	✗	✗
Balig et al. [45]	✓	✗	✗
Thakkar et al. [1]	✓	✓	✗
Kuzlu et al. [47]	✓	✗	✗
Javaid et al. [16]	✓	✓	✗
Kwon and Yu [36]	✓	✓	✗
Gorenflo et al. [37]	✓	✓	✗
Shalaby et al. [46]	✓	✗	✗
Yusuf and Surjandari [19]	✓	✗	✗
Nakaike et al. [49]	✓	✓	✗
Wang and Chu [48]	✓	✗	✗

In addition, some studies have suggested enhancing the strategies for endorsement or database access. For example, Kwon and Yu [36] improved read transaction processing by separating read and write transactions in the endorsement of peers. They also created a new consensus protocol without using an external project. As a result, they improved the overall latency and throughput by 20%. Gorenflo et al. [37] proposed replacing the database that stores the latest key-value pair in the ledger with an in-memory hash table to enhance the speed of database access. They improved Fabric throughput up to 20,000 tps in Fabric v1.4. Nakaike et al. [49] also pointed out database access overhead in chaincodes and suggested disabling the compression of GoLevelDB and reducing the size of StateDB and the number of reads and writes in a transaction.

To sum up, previous studies focused on analyzing performance bottlenecks and optimized Fabric architecture to improve Fabric performance. QiOi differs from these studies in that QiOi addresses the performance interference in Hyperledger Fabric when the Fabric components run on cloud data centers.

7.3. Performance Isolation and Resource Management

Performance isolation is critical for service quality in cloud environments. Thus, there have been many research efforts to develop techniques for resource allocation and scheduling that assign server resources, such as CPU time, disk bandwidth, and network bandwidth.

Li et al. [25] proposed PINE, a technique for performance isolation in container environments. PINE can adaptively allocate the storage resources for each service according to their performance behaviors (e.g., latency-sensitive or throughput-first services) through dynamic resource management and I/O concurrency configuration. Kim et al. [23] suggested CPU cap management to ensure different QoS enforcement levels in a platform with shared resources. They adjusted the number of working threads per QoS class and dispatched the outstanding task along with the associated events to the most appropriate working thread. Xu et al. [24] designed a network bandwidth management system working between Kubernetes and network plugins called NBWGuard. NBWGuard supports three QoS classes consistent with the Kubernetes approach. Khalid et al. [22] discussed the reason for performance degradation of containers when co-located containers running network-related applications consume CPU resources aggressively. In order to prevent performance degradation, the authors suggested fairly allocating CPU by accounting CPU consumption for network packet processing and applying the CPU consumption in CPU scheduling.

To summarize, previous studies are not easily applicable to Fabric that consists of heterogeneous components. Therefore, this paper proposes QiOi to provide component-level performance isolation for the heterogeneous characteristics of components.

8. Conclusions

In cloud data centers, the Fabric services may run concurrently with co-located services. In such an environment, Fabric services may experience performance interference. We demonstrate that performance interference of Fabric indeed occurs and it is due to the scheduling delay caused by co-located services. To mitigate performance interference, we present QiOi, component-level performance isolation technique for Hyperledger Fabric. The technique is based on dynamically controlling the *CPU share* of Fabric components. Our evaluation results show that QiOi mitigates performance degradation of Fabric by 22% and improves Fabric latency by 2.5 times without sacrificing the performance of co-located services.

Author Contributions: Conceptualization, J.K. and K.L. (Kyungwoon Lee); methodology, J.K. and K.L. (Kyungwoon Lee); software, J.K. and J.I.; validation, K.L. (Kyungwoon Lee) and G.Y.; investigation, J.K., K.L. (Kyungwoon Lee), G.Y., K.L. (Kwanhoon Lee) and J.I.; writing—original draft preparation, J.K. and K.L (Kyungwoon Lee); writing—review and editing, J.K., K.L. (Kyungwoon Lee), G.Y., K.L. (Kwanhoon Lee), J.I., and C.Y.; visualization, J.K.; supervision, C.Y.; funding acquisition, C.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partly supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2015-0-00280 (SW Starlab) Next generation cloud infra-software toward the guarantee of performance and security SLA). This research was also supported by Next Generation Engineering Researcher Program of National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (No. NRF-2019H1D8A2105513).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Acknowledgments: We appreciate the anonymous reviewers.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Thakkar, P.; Nathan, S.; Viswanathan, B. Performance benchmarking and optimizing Hyperledger Fabric blockchain platform. In Proceedings of the 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Milwaukee, WI, USA, 25–28 September 2018; IEEE: New York, NY, USA, 2018; pp. 264–276.
2. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. Available online: https://bitcoin.org/bitcoin.pdf (accessed on 16 March 2021).
3. Ripple. Available online: https://ripple.com/ (accessed on 16 March 2021).
4. Enzyme Finance—On-Chain Asset Management. Available online: https://enzyme.finance/ (accessed on 16 March 2021).
5. Morgan, J.; Wyman, O. *Unlocking Economic Advantage with Blockchain. A Guide for Asset Managers*; JP Morgan Reports: New York, NY, USA, 2016.
6. Lim, J.M.; Kim, Y.; Yoo, C. Chain Veri: Blockchain-Based Firmware Verification System for IoT Environment. In Proceedings of the 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), Halifax, NS, Canada, 30 July–3 August 2018; IEEE: New York, NY, USA, 2018; pp. 1050–1056.
7. Hyperledger Fabric. Available online: https://github.com/hyperledger/fabric (accessed on 16 March 2021).

8. Androulaki, E.; Barger, A.; Bortnikov, V.; Cachin, C.; Christidis, K.; De Caro, A.; Enyeart, D.; Ferris, C.; Laventman, G.; Manevich, Y.; et al. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In Proceedings of the Thirteenth EuroSys Conference, Porto, Portugal, 23–26 April 2018; pp. 1–15.
9. Digitizing Global Trade with Maersk and IBM. 2018. Available online: https://www.ibm.com/blogs/blockchain/2018/01/digitizing-global-trade-maersk-ibm/ (accessed on 16 March 2021).
10. SecureKey: Building Trusted Identity Networks. Available online: https://securekey.com/ (accessed on 16 March 2021).
11. Everledger—Tech for Good Blockchain Solutions. Available online: https://www.everledger.io/ (accessed on 16 March 2021).
12. Hyperledger Blockchain Performance Metrics. Whitepaper. Available online: https://www.hyperledger.org/wpcontent/uploads/2018/10/HL_Whitepaper_Metrics_PDF_V1 (accessed on 16 March 2021).
13. Soltesz, S.; Pötzl, H.; Fiuczynski, M.E.; Bavier, A.; Peterson, L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, Lisbon, Portugal, 20–23 March 2007; Association for Computing Machinery: New York, NY, USA, 2007; pp. 275–287.
14. Wang, P.; Meng, J.; Chen, J.; Liu, T.; Zhan, Y.; Tsai, W.T.; Jin, Z. Smart contract-based negotiation for adaptive QoS-aware service composition. *IEEE Trans. Parallel Distrib. Syst.* **2018**, *30*, 1403–1420. [CrossRef]
15. Silva, M.; Ryu, K.D.; Da Silva, D. VM performance isolation to support QoS in cloud. In Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, Shanghai China, 21–25 May 2012; IEEE: New York, NY, USA, 2012; pp. 1144–1151.
16. Javaid, H.; Hu, C.; Brebner, G. Optimizing Validation Phase of Hyperledger Fabric. In Proceedings of the 2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Rennes, France, 22–25 October 2019; IEEE: New York, NY, USA, 2019; pp. 269–275.
17. LevelDB Key-Value Database in Go. Available online: https://github.com/syndtr/goleveldb (accessed on 16 March 2021).
18. Apache CouchDB. Available online: https://couchdb.apache.org/ (accessed on 16 March 2021).
19. Yusuf, H.; Surjandari, I. Comparison of Performance Between Kafka and Raft as Ordering Service Nodes Implementation in Hyperledger Fabric. *Int. J. Adv. Sci. Technol.* **2020**, *29*, 3549–3554.
20. Ongaro, D.; Ousterhout, J. In search of an understandable consensus algorithm. In Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC), Philadelphia, PA, USA, 19–20 June 2014; pp. 305–319.
21. Apache Kafka. Available online: http://kafka.apache.org/ (accessed on 16 March 2021).
22. Khalid, J.; Rozner, E.; Felter, W.; Xu, C.; Rajamani, K.; Ferreira, A.; Akella, A. Iron: Isolating Network-based CPU in Container Environments. In Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Renton, WA, USA, 9–11 April 2018; pp. 313–328.
23. Kim, Y.K.; HoseinyFarahabady, M.R.; Lee, Y.C.; Zomaya, A.Y.; Jurdak, R. Dynamic control of cpu usage in a lambda platform. In Proceedings of the 2018 IEEE International Conference on Cluster Computing (CLUSTER), Belfast, UK, 10–13 September 2018; IEEE: New York, NY, USA, 2018; pp. 234–244.
24. Xu, C.; Rajamani, K.; Felter, W. NBWGuard: Realizing Network QoS for Kubernetes. In Proceedings of the 19th International Middleware Conference Industry, Rennes, France, 10–14 December 2018; pp. 32–38.
25. Li, Y.; Zhang, J.; Jiang, C.; Wan, J.; Ren, Z. PINE: Optimizing performance isolation in container environments. *IEEE Access* **2019**, *7*, 30410–30422. [CrossRef]
26. Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauly, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), San Jose, CA, USA, 25–27 April 2012; pp. 15–28.
27. Bouron, J.; Chevalley, S.; Lepers, B.; Zwaenepoel, W.; Gouicem, R.; Lawall, J.; Muller, G.; Sopena, J. The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS. In Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC), Boston, MA, USA, 11–13 July 2018; pp. 85–96.
28. Wong, C.; Tan, I.; Kumari, R.; Lam, J.; Fun, W. Fairness and interactive performance of O (1) and CFS Linux kernel schedulers. In Proceedings of the 2008 International Symposium on Information Technology, Kuala Lumpur, Malaysia, 26–29 August 2008; IEEE: New York, NY, USA, 2008; Volume 4, pp. 1–8.
29. Kulkarni, S.G.; Zhang, W.; Hwang, J.; Rajagopalan, S.; Ramakrishnan, K.; Wood, T.; Arumaithurai, M.; Fu, X. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, Los Angeles, CA, USA, 21–25 August 2017; pp. 71–84.
30. Turner, P.; Rao, B.B.; Rao, N. CPU bandwidth control for CFS. In Proceedings of the Linux Symposium, Ottawa, ON, Canada, 13–16 July 2010; pp. 245–254.
31. Hyperledger Caliper—Hyperledger. Available online: https://www.hyperledger.org/use/caliper (accessed on 16 March 2021).
32. Hyperledger Caliper Benchmarks. Available online: https://github.com/hyperledger/caliper-benchmarks (accessed on 16 March 2021).
33. What's New in Hyperledger Fabric v2.x. Available online: https://hyperledger-fabric.readthedocs.io/en/release-2.3/whatsnew.html (accessed on 16 March 2021).
34. Sysbench—A Modular, Cross-Platform and Multi-Threaded Benchmark Tool. Available online: http://manpages.ubuntu.com/manpages/trusty/man1/sysbench.1.html (accessed on 16 March 2021).

35. Shi, J.; Qiu, Y.; Minhas, U.F.; Jiao, L.; Wang, C.; Reinwald, B.; Özcan, F. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proc. Vldb Endow.* **2015**, *8*, 2110–2121. [CrossRef]
36. Kwon, M.; Yu, H. Performance Improvement of Ordering and Endorsement Phase in Hyperledger Fabric. In Proceedings of the 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), Granada, Spain, 22–25 October 2019; IEEE: New York, NY, USA, 2019; pp. 428–432.
37. Gorenflo, C.; Lee, S.; Golab, L.; Keshav, S. FastFabric: Scaling Hyperledger Fabric to 20000 transactions per second. *Int. J. Netw. Manag.* **2020**, *30*, e2099. [CrossRef]
38. Kim, J.; Lee, K.; Yang, G.; Lee, K.; Im, J.; Yoo, C. Exploring the Characteristics of Hyperledger Fabric in Resource Consumption. In Proceedings of the 2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS), Paris, France, 28–30 February 2020; IEEE: New York, NY, USA, 2020; pp. 208–209.
39. The Ordering Service. Available online: https://hyperledger-fabric.readthedocs.io/en/latest/orderer/ordering_service.html (accessed on 16 March 2021).
40. Zhu, L.; Chen, C.; Su, Z.; Chen, W.; Li, T.; Yu, Z. BBS: Micro-Architecture Benchmarking Blockchain Systems through Machine Learning and Fuzzy Set. In Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), San Diego, CA, USA, 22–26 February 2020; IEEE: New York, NY, USA, 2020; pp. 411–423.
41. Ousterhout, K.; Rasti, R.; Ratnasamy, S.; Shenker, S.; Chun, B.G. Making sense of performance in data analytics frameworks. In Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), Oakland, CA, USA, 4–6 May 2015; pp. 293–307.
42. Samadi, Y.; Zbakh, M.; Tadonki, C. Performance comparison between Hadoop and Spark frameworks using HiBench benchmarks. *Concurr. Comput. Pract. Exp.* **2018**, *30*, e4367. [CrossRef]
43. Kubernetes. Available online: https://kubernetes.io/ (accessed on 16 March 2021).
44. Nasir, Q.; Qasse, I.A.; Abu Talib, M.; Nassif, A.B. Performance analysis of Hyperledger Fabric platforms. *Secur. Commun. Networks* **2018**, *2018*, 3976093. [CrossRef]
45. Baliga, A.; Solanki, N.; Verekar, S.; Pednekar, A.; Kamat, P.; Chatterjee, S. Performance characterization of Hyperledger Fabric. In Proceedings of the 2018 Crypto Valley Conference on Blockchain Technology (CVCBT), Zug, Switzerland, 20–22 June 2018; IEEE: New York, NY, USA, 2018; pp. 65–74.
46. Shalaby, S.; Abdellatif, A.A.; Al-Ali, A.; Mohamed, A.; Erbad, A.; Guizani, M. Performance Evaluation of Hyperledger Fabric. In Proceedings of the 2020 IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT), Doha, Qatar, 2–5 February 2020; IEEE: New York, NY, USA, 2020; pp. 608–613.
47. Kuzlu, M.; Pipattanasomporn, M.; Gurses, L.; Rahman, S. Performance analysis of a Hyperledger Fabric blockchain framework: throughput, latency and scalability. In Proceedings of the 2019 IEEE International Conference on Blockchain (Blockchain), Atlanta, GA, USA, 14–17 July 2019; IEEE: New York, NY, USA, 2019; pp. 536–540.
48. Wang, C.; Chu, X. Performance Characterization and Bottleneck Analysis of Hyperledger Fabric. *arXiv* **2020**, arXiv:2008.05946.
49. Nakaike, T.; Zhang, Q.; Ueda, Y.; Inagaki, T.; Ohara, M. Hyperledger Fabric Performance Characterization and Optimization Using GoLevelDB Benchmark. In Proceedings of the 2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), Rhodes Island, Greece, 2–6 November 2020; IEEE: New York, NY, USA, 2020; pp. 1–9.