Auto-Tuning with Reinforcement Learning for Permissioned Blockchain Systems

Mingxuan Li[1,2,3]
People's Public Security University of China
limingxuan2@iie.ac.cn

Yazhe Wang
Zhongguancun Laboratory
wangyz@zgclab.edu.cn

Shuai Ma
SKLSDE Lab, Beihang University
mashuai@buaa.edu.cn

Chao Liu[2,3]
Chinese Academy of Sciences
liuchao1@iie.ac.cn

Dongdong Huo[2,3]
Chinese Academy of Sciences
huodongdong@iie.ac.cn

Yu Wang[2,3]
Chinese Academy of Sciences
wangyu@iie.ac.cn

Zhen Xu[2,3]
Chinese Academy of Sciences
xuzhen@iie.ac.cn

ABSTRACT

In a permissioned blockchain, performance dictates its development, which is substantially influenced by its parameters. However, research on auto-tuning for better performance has somewhat stagnated because of the difficulty posed by distributed parameters; thus, it is possible only with difficulty to propose an effective auto-tuning optimization scheme. To alleviate this issue, we lay a solid basis for our research by first exploring the relationship between parameters and performance in Hyperledger Fabric, a permissioned blockchain, and we propose Athena, a Fabric-based auto-tuning system that can automatically provide parameter configurations for optimal performance. The key of Athena is designing a new Permissioned Blockchain Multi-Agent Deep Deterministic Policy Gradient (PB-MADDPG) to realize heterogeneous parameter-tuning optimization of different types of nodes in Fabric. Moreover, we select parameters with the most significant impact on accelerating recommendation. In its application to Fabric, a typical permissioned blockchain system, with 12 peers and 7 orderers, Athena achieves a throughput improvement of 470.45% and a latency reduction of 75.66% over the default configuration. Compared with the most advanced tuning schemes (CDBTune, Qtune, and ResTune), our method is competitive in terms of throughput and latency.

PVLDB Reference Format:
Mingxuan Li, Yazhe Wang, Shuai Ma, Chao Liu, Dongdong Huo, Yu Wang, and Zhen Xu. Auto-Tuning with Reinforcement Learning for Permissioned Blockchain Systems. PVLDB, 16(5): 1000-1012, 2023.
doi:10.14778/3579075.3579076

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

Yazhe Wang is the corresponding author.
[1]School of Criminal Investigation, People's Public Security University of China
[2]Institute of Information Engineering, Chinese Academy of Sciences
[3]School of Cyber Security, University of Chinese Academy of Sciences

Proceedings of the VLDB Endowment, Vol. 16, No. 5 ISSN 2150-8097.
doi:10.14778/3579075.3579076

1 INTRODUCTION

Thanks to Turing-complete smart contracts, permissioned blockchains have evolved into distributed transactional management systems that can handle almost all transactions the database can execute [1, 2]. Due to the security and transparency of blockchains, permissioned blockchains are increasingly being used by companies and government departments to replace or supplement their database services [3]. Nevertheless, transactions [4–6] include a significant number of encrypted signature calculations, network communication, and trust verification, which curtails the performance and further development [7–9] of the permissioned blockchain. A typical example is Hyperledger Fabric (hereinafter "Fabric"), the first open-source permissioned blockchain platform for enterprise application scenarios [6]. Transactions in Fabric are completely replicated in the entire network and replay on all nodes involved with the same tasks. Existing research [13] shows that the performance of Fabric cannot be effectively improved by simply stacking hardware, as is done in traditional distributed databases. Consequently, an innovative optimization method is necessary.

Recent research [1] has shown that blockchains are comparable to distributed databases due to their similarity to distributed transactional management systems. Currently, there is a trend in the adoption of database features in blockchains (e.g., FastFabric[29], Fabric++[16]) or vice versa [19, 20]. For instance, Sharma et al. [16] tuned only one parameter (i.e., MaxMessageCount) from 16 to 32, and the throughput of Fabric increased by nearly 70%. However, Fabric has hundreds of configurable parameters. Thus, the performance boost observed by tuning Fabric parameters is merely the tip of the iceberg. Nonetheless, the large number of parameters makes manual tuning according to the rules neither efficient nor effective. This complexity of configuration issues is overwhelming even for experts. Recognized as a feasible solution in distributed databases, auto-tuning has emerged as a powerful optimization method. Existing auto-tuning schemes of distributed databases are divided into three classes: search-based methods (e.g., Bestconfig [31]), traditional machine-learning-based methods (e.g., OtterTune [26], ResTune [15]), and deep reinforcement learning (DRL) methods (e.g., CDBTune [30], Qtune [28]). Nevertheless, directly borrowing the auto-tuning method of the distributed database to optimize Fabric is complicated for three reasons.

First, Fabric's throughput calculation method differs from that of a distributed database. In a distributed database system, a transaction is executed only once for the individual state in the system [1], and existing auto-tuning schemes only need to record one response from any node to calculate throughput. In contrast, Fabric enables trust by simulating a single transaction on multiple nodes in parallel, and the execution process involves each participant node in the cluster. Thus, Fabric schemes require collecting mixed responses from all nodes that execute transactions [6]. Consequently, using a database benchmark suite to measure Fabric is difficult for users.

Second, state-of-the-art research methods termed dynamic configuration tuners in the database field adjust parameters according to workload changes [21–24]. This research highlights the importance of workload changes in database tuning. Accordingly, most of these tuners first collect the workload information in real-time and then predict the optimal configuration. Finally, they allocate some nodes in the cluster to reconfigure the optimal parameters to ensure the availability of the system in the parameter-reconfiguration process. However, such dynamic configuration tuners cannot fulfill Fabric-tuning needs because they utilize the workload to which the performance of Fabric is less sensitive. For example, although the performance of two workload applications (i.e., YCSB and Smallbank) in a distributed database (i.e., H-STORE) differs by nearly 6.6 times, their evaluations in Fabric only fluctuate within 10% [2]. This disparity is occasioned by Fabric's distinct working organisms. Specifically, unlike the execution mode, where a distributed database directly replicates the workload to its storage layer, Fabric requires additional consensus and a validation process to ensure the reliability of transactions, eventually involving these two processes in performance tuning. Moreover, the consensus and validation phases are far more time-consuming compared with the execution phase. The literature [1] states that the time spent on the consensus and validation phases is several times more than the time spent on execution in the case of unsaturated workload and even considerably more in the case of saturated workload. This feature of Fabric weakens the impact of workload changes on the system so that they are gradually ignored. Therefore, the tuning of Fabric should focus not only on the endorser peer nodes that execute transactions but also on all nodes (i.e., peers, orderers) that participate in the consensus and validation phases. Meanwhile, the parameters in the nodes control the execution process of consensus and validation, which has a nontrivial impact on the performance of Fabric.

Third, the auto-tuning method for a distributed database requires an expensive learning cycle, which in turn requires significant time to find the best pairing of parameters and performance in the search space for performance optimization. Due to the continuity of parameters and their dependence, the size of the parameter space grows exponentially following the increase of parameters. An optimal scheme of distributed databases can overcome this problem by selecting important parameters [23, 26, 27], as substantiated by the experimental results. To improve the performance of Fabric, all parameters of the peer, orderer and other types of components need to be considered as much as possible. However, since there is no previous knowledge to guide the selection of important parameters in tuning Fabric, a method for selecting important parameters of Fabric is urgently required to reduce the training time of tuning.

In this paper, we propose a novel scheme to auto-tune parameters for Fabric, namely Athena (a framework that automatically tunes the Hyperledger network of Fabric). The scheme uses centralized training with decentralized execution to build a DRL model to predict the best configurations. It can recommend a set of configurations that dramatically improve the performance of Fabric to attain higher throughput and lower latency. The performance maintains a competitive advantage even when there is a change in the smart contract (chaincode), workload, or network structure. Specifically, Athena observes the status information and performance indicators of all nodes, ensuring the effectiveness of the measurement of the performance of Fabric, which solves the problem that the existing database methods cannot accurately measure Fabric's real performance. At the same time, Athena performs heterogeneous configurations for different types of nodes in Fabric, making the best of each node for the performance of Fabric and solving the collaborative optimization problem in the distributed Fabric network. Moreover, we quantify the importance of the parameters to reduce the number of parameters that the auto-tuner must tune, alleviating the burden of an enormous search space during model training and improving system efficiency. Finally, we apply Athena to the most popular permissioned blockchain system, i.e., Fabric. Our results show that with the configuration set recommended by Athena, Fabric achieves a 470.45% higher throughput and a 75.66% lower latency over the default settings under 12 peers and 7 orderers. Our main contributions are as follows:

- To the best of our knowledge, this is the first time that an auto-tuning system has been proposed for the optimal performance of Fabric.
- We propose a centralized training with a decentralized execution method to realize heterogeneous tuning of Fabric.
- We design an efficient reward function, which significantly shortens training time.
- We analyze the collected data to filter out the most important configuration parameters, which significantly accelerates the auto-tuning efficiency.
- Finally, using two official workloads (Smallbank and Simple), we conducted an extensive experimental study. (a) The tuning results of Athena are consistently better than those of other competitors. (b) The important parameters related to performance are identified, and when the top 20 parameters are tuned, the performance is equivalent to that of tuning all parameters under C1 (i.e., three orderers, and four peers), and the training time is reduced by 48.79%.

The remainder of this paper as follows. Section 2 describes the related work, Section 3 shows the system architecture, Section 4 deals with auto-tuning as a DRL problem, Section 5 identifies important parameters, Section 6 describes the experiments, Section 7 describes the discussion. Finally, Section 8 provides the conclusion.

2 RELATED WORK

2.1 Hyperledger Fabric

Hyperledger Fabric is an enterprise-grade permissioned blockchain framework established under the Linux Foundation [6]. In Fabric, the transaction flow follows three phases, namely, execution, ordering, and validation. We briefly introduce these phases using the

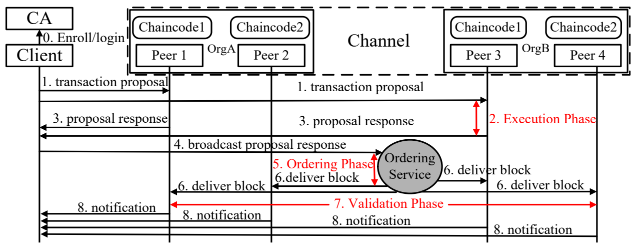

Figure 1: Fabric high-level transaction flow

example in Figure 1. Before executing the transaction, the client first registers in the certificate authority (CA) to obtain the legal identity to join the Fabric network. **Execution Phase**: The client sends to all the endorser peers according to the endorsement policy after the transaction is replicated. The peers simulate the execution of the transaction by invoking a chaincode (smart contract) and generating a read/write set. The endorsers then send a response (read/write set with all signatures) back to the client. The client collects the responses and sends them to the ordering service. **Ordering Phase**: The leader node of the ordering service orders the received transactions and creates a block using a consensus protocol (e.g., Raft). The leader node of the ordering service then delivers these blocks to peers. **Validation Phase**: The peers perform two validations (endorsement policy and serializability) on the transactions in the received blocks, commit the transactions to the ledger, and finally, send a notification to the client.

Thus, Fabric involves the client, peers, chaincodes, and orderers to complete a transaction. The time taken to execute a transaction is determined by the performance of various nodes and the communication between them. The types of multiple nodes cooperatively executing transactions render this process more difficult. In response to these challenges, this study focuses on Fabric's automatic tuning parameter to improve its performance.

2.2 Fabric Optimization

Little attention has been paid to the optimization of the parameters. In this regard, one important study is that by Thakkar et al. [7], who discuss the impact of five configurations on performance through experimental analysis and provide some guidelines. Further, they conducted nearly 1,000 experiments. However, the inefficiency of manual tuning eliminates its advantages. Some studies [16, 29] mainly use the block size as an experimental comparison item, but fail to analyze the parameters' influence on performance comprehensively. Zhu et al. [43] proposed to characterize the parameters of Fabric at the micro-architecture level, which is in a different layer from that of the impact on parameters about which we are concerned. Chacko et al. [17] proposed a new tool, i.e., HyperLedgerLab, to analyze the impact of different parameters of Fabric on transaction failure rates. This method focuses on small-scale parameters (e.g., block size) to reduce transaction failures. It cannot tap the tremendous potential of permissioned blockchain parameters to improve performance. Other optimization methods for Fabric include that of Gorenflo et al. [29], who proposed FastFabric, which implements parallel and encrypted information caching methods in the validation phase. However, this scheme is confined to laboratories and cannot be applied in industrial production. For example, the hash table is stored in the memory; data loss easily occurs once there is downtime or a node is offline. [16, 18, 33] refer to numerous principles and methods of the database in the performance optimization and functional transformation of the blockchain. However, all these optimization methods require the modification of the components of Fabric, and there are many hidden uncertainties regarding the security of Fabric. Our approach does not change the source code of Fabric, and is therefore orthogonal to the aforementioned optimizations (e.g., FastFabric).

2.3 Database Tuning

Existing methods for database tuning can be divided into four main categories [28]. **(1) Rule-based methods.** These provide users with expert method experience to tune parameters, which is suitable for quick guidance and is a typical solution as in [32]. However, this kind of scheme requires a deep understanding of the internal mechanism of the system and is time-consuming. Therefore, this scheme is not investigated in the present study. **(2) Search-based methods.** Tuning is performed by searching and adjusting the parameter space. Bestconfig [31] is a typical solution that divides the high-dimensional parameter space into subspaces. Furthermore, it uses search-based methods to retrieve the optimal parameters continuously, and in reality, it is likely to overlook the optimal configuration. Therefore, this method is also not considered in the present study. **(3) Traditional machine-learning-based methods.** These use traditional machine-learning technology [10, 11, 14, 15, 26] to tune the database by learning the experience from historical data. They mainly use a learning-based algorithm (e.g., Bayesian optimization) for parameter tuning. However, this algorithm requires a large amount of high-quality historical data, which are difficult to obtain. Therefore, this approach is not selected. ResTune [15] is the most popular method in this field, and is mainly resource-oriented for optimizing various objectives (e.g., CPU utilization). This is different from our approach, which focuses on throughput and latency. OtterTune [26] is the other typical solution. Notably, parameters that are unrelated to performance are eliminated, thereby improving the tuning efficiency. Therefore, this approach is followed when ranking the importance of the Fabric parameters. **(4) DRL-based methods.** DRL-based solutions are currently the most popular for database tuning. CDBTune [30] and Qtune [28] are typical solutions that can effectively adapt to changes in workloads and hardware environments. However, existing methods are based on a single agent, which loses some information about the relationship between parameters and performance.

Although achievements have been made in solving the complex database auto-tuning problem, it is still not possible to handle the new architecture of the permissioned blockchain. Because the permissioned blockchain needs to replicate the entire transaction to different nodes for repeated execution, performance is jointly affected by multiple nodes. Therefore, the auto-tuning of Fabric has become a new problem for multi-node cooperation, which has not yet been considered by the existing methods [26, 30, 31].

3 SYSTEM ARCHITECTURE

Figure 2 shows the overview workflow of Athena. Athena handles a tuning request as follows: The user provides a request for the controller, and the controller automatically deploys the environment

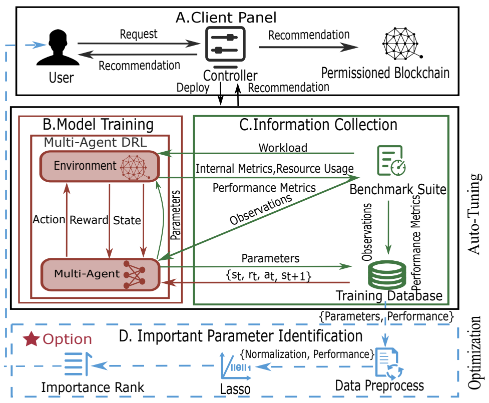

Figure 2: Athena architecture

based on the default configuration and initializes the multi-agent. The controller then activates the benchmark suite, collecting resource usage and internal metrics and calculating performance metrics (throughput and latency). The collected data and parameters are then stored in the training database. Resource usage and internal metrics, collectively called observations, are processed and provided for the multi-agent to reconfigure the environment. This process is continuously repeated with the environment configured each time using the processed data from the previous process until the set threshold is reached. By sampling the data stored in the training database, Athena trains a multi-agent DRL model and adjusts the output configuration parameter strategy. The two-step process of information collection and model training is repeated until the performance metrics conform and stabilize. Consequently, the best parameters are generated and recommended to the controller. Afterwards, the user decides whether to use the recommended configurations in real scenarios. Additionally, as an option, Athena uses the least absolute shrinkage and selection operator (LASSO) [37] to weight parameters and establish the important subset of parameters, which can reduce the overall training time while maintaining promisingly high performance for the next tuning.

Client Panel: Client panel is enabled by the controller, which can receive tuning requests from the user and then coordinate and control the operation of the entire Athena system. The panel includes (1) network architecture, such as Fabric architecture configuration that users are tuning to target (e.g., four peers and five orderers); (2) the chaincode, similar to smart contracts in concept, which defines the types of data and calculation logic that need to be recorded in the blockchain; (3) the workload, i.e., the data sent to the blockchain when Caliper (a blockchain performance benchmark framework) [36] performs testing since the chaincode and workload always appear in pairs (we will use the chaincode name to refer to both e.g., Smallbank stands for chaincode Smallbank and its workload); (4) the selected parameters, i.e., the configuration parameters on each node that the user chooses to tune; and (5) the reward coefficient, which is required by the reward function we designed. After receiving the recommended configuration parameters returned by the training model, the user can send them to the controller for deployment in an instance of Fabric network.

Information Collection: Information collection is executed along with the training model, and the collected data are provided for training the DRL model and identifying important parameters. Caliper and Prometheus (a status monitoring system) [38] are used as the benchmark suites to collect information. Caliper evaluates the performance, producing a set of performance metrics such as throughput, latency, and resource usage (i.e., traffic, average CPU, and memory utilization of the node) of Fabric. Prometheus pulls the internal metrics of each node after Caliper completes its operation. For training the DRL model, an index includes the agent type, number of executions, parameters, throughput, latency, and all the nodes' internal metrics and resource usage. To identify important parameters, we store the recommended parameter values, and performance metrics (throughput and latency) obtained each time after a benchmark suite test and form a data pair.

Model Training: The auto-tuning of Fabric aims to determine the best pairing of parameters and performance in continuous space, which is an NP-hard problem. Reinforcement learning has a strong decision-making ability and can learn from limited samples [30] in the initial stage. It is a key technology for solving the problem of tuning in continuous space. Due to the different types of nodes that participate in transactions, the contribution of each type of node to performance is differentiated. Accordingly, we designed a new multi-agent DRL algorithm that can achieve heterogeneous tuning of parameters for different types of nodes. Section 4.2 introduces the definition of the components of DRL. Section 4.4 explains the proposed algorithm in detail.

Important Parameter Identification: Figure 2 (D) demonstrates how to apply LASSO [37] to find important parameters that affect performance. We first acquire the parameter-performance pair from the training database. Before applying the LASSO model, we need to normalize the sample data. As far as we know, LASSO can provide high-quality results when the features are continuous and of approximately the same order of magnitude. After data normalization, the processed parameter-performance pairing is sent to the LASSO model. We then use incremental methods to rank the parameters that affect the performance, thereby reducing the dimensionality of the performance features of Fabric as well as the training time. Section 5 describes the exact optimization method.

4 AUTO-TUNING AS A DRL PROBLEM

This section first defines the relationship between parameters and performance in Fabric and formulates auto-tuning as a DRL model. Moreover, this section describes the shortcomings of traditional reinforcement-learning solutions for solving multi-agent collaboration problems. Finally, this section presents our multi-agent DRL model for auto-tuning Fabric.

4.1 Parameter and Performance

It is challenging to delineate the impact of parameters on the performance of Fabric. The first and critical step is to define the model of the relationship between the parameters and the performance of Fabric. We use latency and throughput as the performance metrics in this study. In [39], the transaction latency is defined as "the transaction confirmation time minus the transaction submission time", which is essentially the sum of endorsement latency, broadcast latency, ordering latency, and commit latency [7]. Mathematically, the transaction latency can be presented as follows: $Latency = T_c - T_s$, where T_c is the confirmation time and T_s is the

Table 1: List of parameters, their names, descriptions, range, and types.

Parameter Name	Description	Range	Type
AbsoluteMaxBytes	The absolute maximum number of bytes allowed for the serialized messages in a batch.	512 kb-10240 kb	orderer
CORE_PEER_KEEPALIVE_MININTERVAL	The minimum permitted time between client pings.	3-60 s	peer
CORE_PEER_GOSSIP_MEMBERSHIPTRACKERINTERVAL	Interval for membershipTracker polling.	0.25-5 s	network
...	...	...	...

Forty-eight parameters are omitted due to space limitations. The completed parameter list is at https://github.com/Matthewbalala/Athena.

submission time. A transaction latency can also be presented as: $Latency = T_{el} + T_{bl} + T_{ol} + T_{cl}$, where T_{el} is the endorsement latency time, T_{bl} is the broadcast latency time, T_{ol} is the ordering latency time, and T_{cl} is the commit latency time. However, the EOV architecture of Fabric is greatly inspired by the traditional database system's optimistic concurrency control mechanisms. Therefore, under the multi-version concurrency control mechanism of Fabric, which can execute multiple transactions concurrently, we use the average of latency as the our performance metric, which is defined as: $Latency_{aver} = \frac{1}{k}\sum_{i=0}^{k} Latency_i$, where k denotes the total number of totals successful transactions, and $Latency_i$ is the latency of the ith transaction. The other performance metric is throughput, which is calculated as: $Throughput = \frac{k}{T_{lc} - T_{fs}}$, where k represents total successful transactions, T_{lc} is the last committing time in global time, and T_{fs} is the first submitting time. Through the defined latency and throughput, we found that the critical factor affecting the performance of Fabric is the time spent by transactions in different stages. Among them, endorsement latency and commit latency are consumed by peers, ordering latency is consumed by orderers, and broadcast latency is consumed by communication with peers and orderers. Specifically, the performance of these nodes and networks is controlled by parameters on their nodes as previously mentioned. Due to the uncertain progressive relationship between performance metrics, various components, and component parameters, we define p_i as the ith parameter of a node of a component type. Each parameter has its limit value. Therefore, we define the limit of each parameter as follows:

$$\begin{bmatrix} l_1 \\ l_2 \\ \cdots \\ l_n \end{bmatrix} \le \begin{bmatrix} p_1 \\ p_2 \\ \cdots \\ p_n \end{bmatrix} \le \begin{bmatrix} u_1 \\ u_2 \\ \cdots \\ u_n \end{bmatrix} \tag{1}$$

where l_i is the lower-limit value of the parameter and u_i is the upper-limit value of the parameter. Table 1 presents the specific values. We then define the mapping relationship between parameters and performance metrics (i.e., throughput and latency) as f, and the performance metrics can be represented as $Performance = f(p_1^P, \cdots, p_n^P; p_1^O, \cdots, p_n^O; p_1^N, \cdots, p_n^N)$, where p_i^P is the ith parameter of peer node, p_i^O is the ith parameter of orderer node, and p_i^N is the ith parameter of network communication. Finally, the purpose of the performance optimization problem is under (1) to compute $f(p_1^P, \cdots, p_n^P; p_1^O, \cdots, p_n^O; p_1^N, \cdots, p_n^N)$. The types of nodes, the parameter types, and the impact of nodes on throughput and latency are all different. Accordingly, we transform the auto-tuning parameter problem of Fabric into a multi-agent cooperation problem.

4.2 Transformation of the Problem

Reinforcement learning (RL) is a learning framework that maximizes rewards by guiding an agent on how to take actions in the environment. Unlike supervised learning, RL does not require a large amount of labeled data in the initial stage of modeling [25, 30]. Through trial-and-error, the agent can repeatedly optimize the behavior selection strategy. At the same time, through an exploration and exploitation mechanism, it can balance the unexplored space and existing knowledge to avoid falling into the local optimum. In recent years, excellent results have been achieved with auto-tuning [28, 30]. Therefore, we attempted to use the DRL method to solve the tuning problem of Fabric.

4.2.1 Environment. The environment is our target for tuning (an instance of Fabric). As shown in Figure 1, five types of nodes including peer, orderer, CA, client, and chaincode together with the network (communications protocol, i.e., Gossip and gRPC) constitute the Fabric network architecture. The actual combination and number of nodes depend on the specific business requirement (e.g., four peers, five orderers, two chaincodes, and two CAs). Note that *Client* in Figure 1 denotes the benchmark suite, which sends transaction tests, after which Fabric executes the transaction.

4.2.2 Agent. An agent is an algorithm used to adjust the output strategy according to the state and reward, which is our DRL model. The peer and orderer occupy most of the time for executing transactions, and all parameters on Fabric are configured on the peer and orderer, so we design agents for them. As for the network, although its parameters are configured in peers and orderers, cross communications must be carried out between nodes of the same or different types in Fabric, resulting in a highly complex relationship between communication and performance. To capture and analyze this relationship, we separate the network-related parameters from the peer and orderer and design a separate agent specifically for the network. Accordingly, we design three agents in total.

4.2.3 State. The state space of Fabric is composed of all observations (internal metrics and resource usage) of participating nodes in the transaction observed by different types of agents. We adopted the official documents of Fabric [34] and Caliper [36] to define the state of Fabric. Two folds define the state space of the peer and orderer, internal metrics for maintenance index data (e.g., *deliver_blocks_sent*, indicating the number of blocks sent by the delivery service), and the physical resource occupied (e.g., average CPU utilization). There are 30 internal metrics and 2 physical-resource metrics (average CPU and memory utilization of the node) in the peer state. The orderer has 30 internal metrics and 2 physical-resource status information (average CPU and memory utilization of the node) in the state. We selected 37 internal metrics (Gossip and gRPC, i.e., 10 from the orderer and 27 from the peer) and 4 physical-resource status information elements (i.e., resource consumption state of the network: traffic in, and traffic out) from the peer and orderer as the state of the network.

4.2.4 Actions. We take the parameters of each node in the Fabric network as an action, as set out before. We aimed to cover as many parameters as necessary to conduct a thorough and systematic analysis of the relationship between parameters and performance, undoubtedly excluding some parameters that are not obviously related to performance, such as listenAddress, the address at local network interfaces on which this peer will listen. Finally, as presented in Table 1, we selected 51 parameters as our action: 12 for the peer, 10 for the orderer, and 29 for the network.

4.2.5 Rewards. Defining the reward function is essential in RL and determines the actual effect of the RL model. Quantification and high sparsity are the two main problems encountered when designing rewards in most practical scenarios. Intuitively, it is effective to use optimization goals as a reward directly; however, we cannot analyze the measured performance results quantitatively. Accordingly, we considered the performance change of the previous time and the performance change of the initial time. Based on the above, we use r, T, and L to represent reward, throughput, and latency, respectively. We define T_0 and L_0 as the throughput and latency before tuning, respectively. T_t and L_t, T_{t-1} and L_{t-1} are the performance metrics at time t and $t-1$, respectively. We use Eqs. (2) and (3) to calculate ΔT and ΔL, respectively. The growth rates of *throughput* and *latency* are represented by ΔT and ΔL, not only considering changes between initial and current values but also between the current measurement and the immediately previous measurement. Since latency and throughput are two opposite indicators, the direction of these two optimization indicators should be consistent. Therefore, when we calculate the ΔL of latency, we add a negative sign in front of its equation, which is in the same format as that of ΔT. Thus, when our RL model calculates reward, increasing throughput or decreasing latency will give positive feedback.

$$\Delta T = \begin{cases} \Delta T_{t\to 0} = \frac{T_t - T_0}{T_0} \\ \Delta T_{t\to t-1} = \frac{T_t - T_{t-1}}{T_{t-1}} \end{cases} \tag{2}$$

$$\Delta L = \begin{cases} \Delta L_{t\to 0} = \frac{-L_t + L_0}{L_0} \\ \Delta L_{t\to t-1} = \frac{-L_t + L_{t-1}}{L_{t-1}} \end{cases} \tag{3}$$

Due to the similarity between the permissioned blockchain and the distributed database, we attempted to use the scheme in [30] and achieved robust results in the database field to form our reward function; however, the scheme is not suitable. The main reason for this is that its reward function is linear. Hence, notwithstanding every tuning parameter improves performance, the reward obtained in the multi-agent field is insignificant. Empirically, we find that the model is difficult to converge. Therefore, we use an exponential reward function to pay more attention to changes in performance. Once the performance change is significantly improved, the reward is satisfactory, and the model converges quickly. We use Eqs. (4) and (5) to calculate the reward via throughput and latency. An exponential reward function may not only capture performance changes better but is also more sensitive to such changes. Moreover, as in all systems, the performance varies for multiple runs of the same task for various direct and indirect reasons. Thus, we use η as an impact factor for scaling performance change. When users use our scheme, the improvement of performance metrics is minimal when the value of reward changes significantly, and η, therefore, needs to be reduced. When the values of reward and performance-metrics change are minimal, η needs to be increased.

$$r_T = \begin{cases} e^{\eta \Delta T_{t\to 0} * \Delta T_{t\to t-1}}, \Delta T_{t\to t-1} > 0 \\ -e^{-\eta \Delta T_{t\to 0} * \Delta T_{t\to t-1}}, \Delta T_{t\to t-1} \leqslant 0 \end{cases} \tag{4}$$

$$r_L = \begin{cases} -e^{\eta \Delta L_{t\to 0} * \Delta L_{t\to t-1}}, \Delta L_{t\to t-1} > 0 \\ e^{-\eta \Delta L_{t\to 0} * \Delta L_{t\to t-1}}, \Delta L_{t\to t-1} \leqslant 0 \end{cases} \tag{5}$$

$$r = C_T * r_T + C_L * r_L \tag{6}$$

As shown in Eq. (6), we consider both throughput and latency when calculating the final reward. Therefore, we multiply them by a coefficient and summate them to calculate the reward of each agent. The coefficients of C_T and C_L require user adjustment, and the sum will be 1.

4.3 RL for Auto-Tuning

Traditional single-agent RL can be divided into two categories, namely, value-based and policy-based RL. Q-learning and deep Q-network (DQN) are the most classic algorithms in value-based RL. If these are applied to a multi-agent system, each agent must treat other agents as the environment, making the model difficult to converge [41]. In a multi-agent system, if other agents are regarded as the environment, the implication is that the environment becomes dynamic and constantly changes owing to the continuous optimization of the other agents. Therefore, this is a dynamic Markov decision process, contrary to the static invariance of the Markov decision process (where the probability and reward are unchanged). The other approach involves policy-gradient algorithms. If the algorithm is directly applied to a multi-agent system, the value function depends on the policies of other agents. An increase in the number of agents results in a large discrepancy between the directions of the calculated policy gradient and the actual gradient, making it difficult for the model to converge even with reverse optimization [41]. The deep deterministic policy gradient (DDPG) has achieved good performance in auto-tuning database problems [28, 30]. It uses a replay buffer to remove correlations in the input experiences and exploits target network approaches to stabilize the training process. Nonetheless, it is still limited by the aforementioned problems faced by a single agent. In other words, single-agent RL methods cannot directly contribute to the parameter tuning and optimization of multi-agent cooperative systems. Therefore, in the next paragraphs, we describe the use of multi-agent DRL to optimize Fabric.

4.4 PB-MADDPG for Auto-Tuning

Multi-agent deep deterministic policy gradient (MADDPG) [41] is a DRL method for multi-agents that combines the DDPG algorithm with a cooperative multi-agent learning architecture. Figure 3 (A) illustrates the MADDPG framework. Each agent has two networks: an actor network μ and a critic network Q. The actor network calculates the action to be executed based on the state acquired by the agent, whereas the critic network evaluates the action calculated by the actor network to improve the performance of the actor network. In the training phase, the actor network only obtains observation information from itself, while the critic network acquires information such as the actions and observations of other agents. In the execution phase, the critic network is not involved, and each agent only requires an actor network. MADDPG uses the collective-behavior value function to train the agent to consider the impact of the parameters of all nodes that participated in the transaction on the performance of Fabric, which can alleviate the turbulence caused by

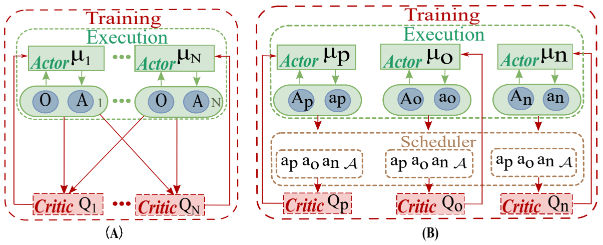

Figure 3: MADDPG and PB-MADDPG

the unstable environment of the multi-agent system. At the same time, each agent only needs to make independent decisions based on local observation, which can tune the different parameters of different types of nodes in Fabric and realize distributed control of the multi-agent. In addition, MADDPG has all the advantages of DDPG, and it can directly output the specific action to deal with the high-dimensional and continuity problems of the parameters.

However, the MADDPG algorithm requires setting an agent for each node. Each agent corresponds to an actor and a critic. With many agents in the process, there are many models that are not friendly to training. Notably, the MADDPG algorithm requires all the agents' state and action information as input when training the critic. Since Fabric is scalable, the input dimensions will rapidly and significantly increase with an increase in users, which is disastrous for model training. Therefore, we further extended MADDPG and implemented an algorithm, i.e., permissioned blockchain MADDPG (PB-MADDPG), which fits the actual scenario of Fabric. Figure 3 (B) presents the framework of PB-MADDPG. We design a two-part optimization. First, as nodes of the same type play similar roles in Fabric, we set one agent for each type. We divide the participating nodes into three groups: peers, orderers, and networks. The same type of node has the same configuration, thereby reducing the number of trained models and accelerating overall training. Second, motivated by [40], we add a scheduler to the original MADDPG architecture to further abstract the observations of all agents and calculate the average value of the observed node states separately according to their type. For example, when we calculate the abstraction, there are four peers and three orderers. We therefore the collect states of four peers and calculate the average value to build an array with 32 dimensions. Moreover, we simultaneously collect the states of three orderers and calculate the average value to construct an array with 32 dimensions. After collecting the network states of all peers and orderers to compute the average, we build an array with 41 dimensions. We then combine the three arrays into a new array as an abstraction. Therefore, the array length of each abstraction is constant (105 dimensions).

Formally, we define the state of the ith node in terms of internal metrics as s_i^m, and the state of the ith node in terms of the physical resource occupied is denoted s_i^p. The observation of the ith node is $o^i = (s_1^m, s_1^p)$. We assume that there are α peers and β orderers in the Fabric network. Further, we define the state space of peers as $o_p = \left\{\left(s_1^m, s_1^p\right), \ldots, \left(s_\alpha^m, s_\alpha^p\right)\right\}$, and the state space of orderers as $o_o = \left\{\left(s_1^m, s_1^p\right), \ldots, \left(s_\beta^m, s_\beta^p\right)\right\}$. Note that the state of the network is obtained from peers and orderers. Thus, the state space of the network is $o_n = \left\{\left(s_1^m, s_1^p\right), \ldots, \left(s_{\alpha+\beta}^m, s_{\alpha+\beta}^p\right)\right\}$. We then define the state space of Fabric as $s = \{o_p, o_o, o_n\}$. Next, we define the abstract of the nodes observed by

Algorithm 1: PB-MADDPG for Auto-Tuning

```
for episode = 1 to M do
    Initialize a random process $\mathcal{N}$ for exploration of action $a$
    Collect initial state $s$ and calculate abstraction $\mathcal{A}$
    for t= 1 to max-episode-length do
        Each Node agent i, select their parameter as action
          $a_i = \boldsymbol{\mu}_{\theta_n}(A_i) + \mathcal{N}_t$ from the action space, w.r.t the
          current policy and exploration
        Execute actions $a = (a_p, a_o, a_n)$, collect state $s'$,
          calculate abstraction $\mathcal{A}'$ and rewards $r$.
        Store $(\mathcal{A}, a, r, \mathcal{A}')$ in replay buffer $D$
        $\mathcal{A} \leftarrow \mathcal{A}'$
        for agent i= 1 to 3 do
            Sample a random minibatch of S samples
              $(\mathcal{A}^j, a^j, r^j, \mathcal{A}^{j\prime})$ from $D$
            Set $y^j = r_i^j + \gamma Q_i^{\mu'}\left(\mathcal{A}'^j, a_1', \ldots, a_N'\right)\Big|_{a_k'=\mu_k'\left(A_k^j\right)}$
            Update critic by minimizing the loss
            $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^\mu\left(\mathcal{A}^j, a_1^j, \ldots, a_N^j\right)\right)^2$
            Update actor using the sampled policy gradient:
            $\nabla_{\theta_i}\mathcal{J} \approx \frac{1}{S}\sum_i \nabla_{\theta_i}\mu_i\left(A_i^j\right)\nabla_{a_i}$
                $Q_i^\mu\left(\mathcal{A}^j, a_1^j, \ldots, a_i, \ldots, a_N^j\right)\Big|_{a_i=\mu_i\left(A_i^j\right)}$
        end
        Update target network parameters for each agent i:
        $\theta_i' \leftarrow \tau\theta_i + (1-\tau)\theta_i'$
    end
end
```

the peer agent as $A_p = \frac{\sum_{i=1}^{\alpha} o_p^i}{\alpha}$, where o_p^i represents the observation sets of the ith peer node. We then define the abstract of the nodes observed by the orderer and network agent as $A_o = \frac{\sum_{i=1}^{\beta} o_o^i}{\beta}$ and $A_n = \frac{\sum_{i=1}^{\gamma} o_n^i}{\gamma}$. Finally, we define the abstract of all the agents' observations as $\mathcal{A} = \{A_p, A_o, A_n\}$.

As shown in Figure 3 (B), we use decentralized execution and centralized training methods under actor and critic architectures. In the execute phase, each actor can take appropriate actions a according to the state (A_p, A_o, or A_n) itself. As a result, the state or action of other agents will not be required. When each critic calculates the Q value in the training stage, the scheduler will send all agents' actions and the abstraction. The critic trains according to the estimated Q value and the actual Q value, and the actor updates the strategy according to the estimated Q value feedback from the critic. The input of each critic ensures that the particular dimension will stay uninfluenced despite the increase in agents, avoiding the explosion of state spaces and improving the convergence efficiency.

Next, we present our PB-MADDPG algorithm, which solves Fabric's parameter tuning problem. Algorithm 1 presents the pseudocode of the PB-MADDPG algorithm, and its detailed description is as follows. (1) First, we initialize the online and target network of actors and critics for the agent of each node type. We then initiate a random process $\mathcal{N}$ for exploration action, deploy the Fabric network, receive the collected initial state s of Fabric and calculate the abstraction $\mathcal{A}$ (lines 1-3). (2) Next, we execute $max{-}episode{-}length$ iteration, and each agent selects actions (a_i, parameters) according to its policy ($\boldsymbol{\mu}_{\theta_n}$ (A_i), where $\boldsymbol{\mu}_{\theta_n}$ is the actor policy and A_i is the abstract of the agents' observations) and exploration noise $\mathcal{N}_t$. The entire Fabric network will be redeployed after each node assigns new actions(lines 4-5). We then use the benchmark suite (Caliper) to test the performance of Fabric with regard to the reward r, which is calculated using Eqs. (2-6). Note that our goal is the highest overall performance; so, the reward of each agent is the same. At the same time, the benchmark suite is deployed to collect the state s' and calculate the abstraction $\mathcal{A}'$ of each node in Fabric. The current states $\mathcal{A}$, the performed action a, the rewards r, and the new states $\mathcal{A}'$, are then stored as a tuple $(\mathcal{A}, a, r, \mathcal{A}')$ in the experience replay buffer D. Therefore, in the execution phase, the agent can output the recommended configuration a according to the current state $\mathcal{A}'$. All nodes are reconfigured with output configurations, and the benchmark suite performs testing, collecting the state, and calculating $\mathcal{A}'$ of all nodes. Afterwards, we update the state $\mathcal{A}'$ to $\mathcal{A}$ and use it as the state $\mathcal{A}$ at the beginning of the next iteration(lines 6-8). (3) After executing an entire episode, each agent will perform the following steps to update its actor and critic network. First, the agent of each node randomly selects a small batch of S samples from experience replay buffer D. We then set the target value of Q-function into y^j, where r_i^j is the reward received by the ith agent from the jth sample, and γ is a discount factor. $Q_i^{\mu'}$ $(\mathcal{A}'^j, a'_1, \ldots, a'_N)$ is a centralized action-value function, which takes the actions of all agents, $a'_1, \ldots, a'_N$, in addition to abstraction $\mathcal{A}'^j$ as inputs, and the Q-value for agent i as outputs. a'_k is obtained by feeding the agent abstraction A_k to the policy μ'(lines 9-11).

The θ of the critic network is updated by minimizing the difference between y^j and $Q_i^{\mu}\left(\mathcal{A}^j, a_1^j, \ldots, a_N^j\right)$ in the sample (lines 12-13). Similarly, we update the θ of the actor network based on the gradient ascent (lines 14-16). (4) Finally, we update the online network of actors and critics, after which their target network parameters are updated under a soft update (lines 17-18).

5 IMPORTANT PARAMETER IDENTIFICATION

In our architecture, we use LASSO as a filter to rank the importance of parameters with performance. It has been widely applied to the selection of feature importance and has achieved outstanding results [26, 37]. LASSO obtains a more refined model by combining the least-squares method and an L1 regularized function constructed on its basis, compressing some coefficients and setting some to zero. According to the coefficient, the variables are screened while fitting the generalized linear model [26]. The training data are the recommended parameter values and throughput obtained each time the MADDPG is trained. Research [26, 42] has shown that LASSO can provide high-quality results when the features are continuous, in an approximate order of magnitude and with similar variances. Therefore, we first normalize the data (value minus mean divided by the standard deviation) to feature them with the same order of magnitude before we use LASSO to rank the importance of the parameters. We then use LASSO regression to rank the importance of the parameters. Mathematically, Athena solves the following optimization problem: $\arg\min_{\theta} \frac{1}{n}\|Y - X\theta\|_2^2 + \lambda\|\theta\|_1$, where n is the number of training samples, Y is the throughput of Fabric, X is the vector of Fabric parameters, θ represents the coefficients of each feature, and λ is a hyperparameter used to adjust the sparse strength of the coefficients of each parameter. We start by giving LASSO a very high penalty item and then decrease the penalty and record each new parameter. The order of parameter appearance implies their impact on performance.

6 EXPERIMENTS

This section evaluates Athena's performance. We chose a variety of the most representative static configuration tuners and performance-optimization methods for Fabric as the baselines:

Default: The default configurations are provided by Fabric.

Manual: Manual tuning by a blockchain expert engaged in tuning and optimizing Fabric for four years.

FastFabric: FastFabric [29] is a performance-optimization approach that implements techniques such as parallel and encrypted message caching based on v1.4. We ported the primary optimization method, cached unmarshaled blocks, parallelized validation, and hashed table to Fabric v2.4.3; we replicated it in our hardware environment for comparison with the tuning methods.

Bestconfig: Bestconfig [31] is a tuner method based on search. We stripped out the divide-and-diverge Sampling (DDS) and recursive bound and search (RBS) algorithms from Bestconfig and applied them to Fabric tuning parameters.

OtterTune: OtterTune [26] is a tuner method based on Bayesian optimization. We used the parameters and performance metrics collected from several other tuning schemes as training data to form the Gaussian process (GP) model of OtterTune. We then randomly selected the configuration to find the local optimum using stochastic gradient descent in the GP model.

CDBTune: CDBTune [30] is a tuner method based on single-agent DRL. We used an agent to adjust the parameters of all nodes.

Qtune: Qtune [28] is a query-aware database tuning system with a DRL model. We adopted its workload-level query optimization method. It differs from CDBTune in that it utilizes a pre-trained model to predict internal metrics.

ResTune: ResTune [15] is a tuner method based on Bayesian optimization. The main difference with OtterTune is the extension of a single GP to a Gaussian regression process with multiple resource types (e.g., CPU) and the addition of service-level-agreement (SLA). In our implementation, we optimize the two indicators, i.e., throughput and latency.

Athena: Athena is our approach that uses a multi-agent DRL to tune Fabric.

Athena-FastFabric: Athena-FastFabric uses Athena to tune parameters on FastFabric.

6.1 Setup

In all our experiments, we used Fabric v2.4.3 with Raft. The endorsement policy was set to "OR." All components ran inside Docker containers. Moreover, the experiments were run on cloud servers. Every two nodes were deployed on an instance using an Intel Xeon(Ice Lake) Platinum 8369B (8 vCPU) with 32 GB memory. The machines were connected via 10 Gigabit Ethernet. Ubuntu 18.04 was the operating system. We used Python 3 and TensorFlow to implement all of Athena's algorithms and components, and we used Python 3 to implement all the other algorithms.

We used Simple and Smallbank from Caliper benchmarks, which have been used as representative workloads in previous studies [16, 29]. Simple and Smallbank perform the benchmark run using 16 worker processes, and submit 200000 TXs at a fixed 800 txn/sec send rate in each round. We considered advanced evaluation schemes such as [31], [26], and [30] in our experiment. The metrics to which we need to pay attention include training time, throughput, and latency. Note that we set three main Fabric network configurations for evaluation: ***C1, three orderers and four peers; C2, five orderers and eight peers; C3, seven orderers and twelve peers.*** The experiments were performed three times, and the average values were reported.

6.2 Execution Time Breakdown

This section divides one training-tuning step into five stages:

1. Benchmark suite test (BST): the benchmark suite sends the workload to the Fabric network.

2. Metrics collection (MC): collecting observations in the Fabric cluster, throughput, and latency calculated by Caliper, and storing them in the training database.

3. Model update (MU): updating the model parameters with collected training data.

4. Recommendation (R): inputting observations to output recommended parameters.

5. Deployment (D): deploying the Fabric network according to the recommended parameters.

As presented in Table 2, we used three network configurations (C1, C2, and C3) based on Simple and Smallbank in the evaluations: First, Athena reduces the deployment time to under 1.5 min by automatically restarting Fabric. However, on average, the deployment time still accounts for 68% of the entire scheme. It is unavoidable in any optimization scheme being the premise for most parameters of Fabric to take effect. Second, MU and MC only consume a small amount of time (overhead, the percentage of MU and MC in the total time for a tuning step is 1.8% - 2.1% in our model), which is almost negligible compared with the duration for the deployment of the Fabric network and BST (execute transactions).

6.3 Tuning Effectiveness and Efficiency Comparison

This section evaluates the effectiveness and efficiency of Athena by comparing with Default, Manual tuning, FastFabric, Bestconfig, OtterTune, CDBTune, Qtune, ResTune, and Athena-FastFabric under different workloads (Smallbank and Simple) and different Fabric network architectures. Note that when the output throughput fluctuates by less than 3 txn/sec five consecutive times, we assume that the tuning process has reached convergence. To test the scalability of our approach, four, eight, and twelve peers correspond to C1, C2, and C3, respectively. We fix the number of orderers to seven while scaling the number of peers from 16 to 32, as shown in Figure 4. Our findings were as follows.

Table 2: Execution time distribution of Athena

Workload	Network	BST (s)	MC (s)	MU (ms)	R (ms)	D (s)	Overhead (%)
Simple	C1	24.7	1.4	0.26	0.42	51.3	1.84
Simple	C2	25.6	1.6	0.31	0.45	56.4	1.96
Simple	C3	26.5	1.9	0.35	0.47	68.2	1.97
Smallbank	C1	25.0	1.4	0.27	0.43	52.5	1.84
Smallbank	C2	26.4	1.6	0.32	0.44	56.5	1.98
Smallbank	C3	27.3	2.0	0.34	0.46	70.2	2.02

1) Athena's tuning solution scales well with the number of nodes. Figure 4 shows the results with varying peer numbers from 4 to 32 by step 4. Athena achieves approximately (379.66%, 422.73%, 470.45%, 478.51%, 536.63%, 490.32%, 511.63%, and 541.89%), and (365.24%, 401.24%, 434.75%, 541.28%, 632.95%, 569.51%, 619.18%, and 703.39%) better throughput, and (71.66%, 75.23%, 75.66%, 77.24%, 77.39%, 73.98%, 73.96%, and 78.15%) and (74.31%, 75.74%, 78.06%, 78.25%, 79.63%, 75.62%, 76.25%, and 78.81%) better latency compared with Default based on Smallbank and Simple, respectively.

2) Athena is orthogonal to FastFabric. When varying peers from 4 to 32 by step 4 with Smallbank, Athena can improve FastFabric throughput and latency by (58.42%, 65.94%, 68.80%, 71.33%, 74.23%, 76.40%, 89.38%, and 97.59%) and (54.14%, 62.00%, 64.41%, 57.68%, 52.98%, 53.14%, and 54.91%), respectively. Compared with FastFabric, Athena achieves approximately (26.32%, 37.05%, 41.14%, 44.75%, 48.90%, 52.00%, 70.54%, 82.27%) better throughput and (22.93%, 33.00%, 41.53%, 45.69%, 42.81%, 38.29%, 44.63%, 43.60%) better latency based on Smallbank.

3) As experts only selected four parameters that they thought were important for tuning, the best result of manual tuning was only 134.85% higher than the default settings with C3. Moreover, unlike database experts, the optimization of a blockchain cannot grasp the whole picture of the relationships among all parameters. When we take manual tuning as the baseline, the average throughput and latency of (Athena, Bestconfig, OtterTune, CDBTune, Qtune, and ResTune) exceed (100.32%, 26.13%, 31.36%, 68.06%, 75.16%, and 83.23%) and (59.65%, 21.93%, 35.38%, 42.34%, 44.09%, and 42.63%) on Smallbank with C3, as well as (89.78%, 24.77%, 42.11%, 65.63%, 72.76%, and 78.28%) and (67.09%, 26.79%, 32.14%, 39.49%, 44.85%, and 47.40%) on Simple with C3, respectively. Athena achieved the best results. Even compared with the most advanced schemes (CDBTune, Qtune, and ResTune), our method has (31.62%, 32.63%, and 32.26%; 28.77%, 26.95%, and 25.16%; 29.91%, 29.94%, and 27.10%) and (18.97%, 13.99%, and 18.64%; 17.06%, 14.23%, and 15.21%; 17.31%, 15.56%, and 17.02%) the advantage in throughput and latency with C1, C2, and C3, respectively, based on Smallbank. From the effect of different workloads, all the tuning results of Simple are slightly better than those of Smallbank, but the gap is slight, as Fabric is not quite sensitive to workloads with similar bytes.

4) After scaling multiple nodes, we found that the time for tuning increased with the number of nodes, mainly because of the increase in startup time caused by the increase in nodes. As for efficiency, Bestconfig was the best (230 min, 343 min, and 398 min) under (C1, C2, and C3). However, as a search solution, it is easy for Bestconfig to fall into the local optimum, which on the other hand, means it has a short period. The performance improvement of Athena is (4.59, 4.73, and 4.84; 4.59, 4.64, and 4.72) times higher than that of Bestconfig on Smallbank and Simple under (C1, C2, and C3), respectively, taking manual tuning as the baseline. Therefore, even

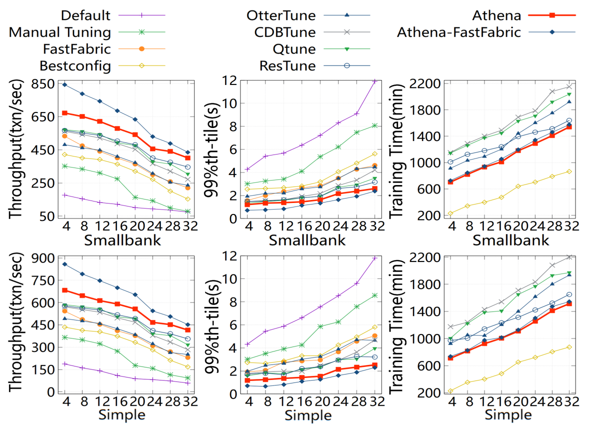

Figure 4: Comparison with Default settings, Manual tuning, FastFabric, Bestconfig, OtterTune, CDBTune, Qtune, ResTune, and Athena-FastFabric on Smallbank and Simple on Fabric

if the training time of Bestconfig is the shortest, its tuning effect is not as satisfactory. Compared with CDBTune and Qtune, though both use RL, the efficiency of Athena is (65.59%, 52.15%, and 54.93%; 41.61%, 49.82%, and 49.95%) better owing to simpler model selection.

In addition, the exponential reward function of Athena is more effective than the linear one and dramatically increases attention to throughput and latency changes. Furthermore, it could also shorten training time. To enable a fair comparison, we add the time for collecting data when computing the training time of OtterTune and ResTune, which take (30.32%, 28.96%, and 23.44%; 36.81%, 23.07%, and 23.84%) more training time than Athena under (C1, C2, and C3) with Smallbank. The efficiency of Athena is always better than that of the other algorithms. Experts can only conduct manual tuning, thus the time spent can not be precisely recorded and calculated as with other schemes, which will not be shown in the figure.

Athena can produce a good optimization effect. On the one hand, the multi-agent DRL can better capture the relationship between the performance and parameters of different types of nodes. Further, the proposed exponential reward function is more sensitive to performance changes and accelerates the training speed of the model. The results of these two aspects make Athena more effective and efficient than the baselines.

6.4 Impact of the Number of Parameters

To better demonstrate the impact of a limited number of parameters on performance in subsequent experiments, we first used LASSO [37] to rank the importance of the parameters by utilizing data collected during tuning instance C1 with Smallbank. For clarity, we only show the ten most influential parameters of these results. Figure 5 shows the weight vectors of various parameters in the regression model. At the same time, the first three are all network-related, and the fourth is related to the orderer. Three of the top ten parameters that impact performance are related to the network, two to the orderer, and five to the peer.

In addition, from the ranking results of all parameters, we conclude that previous research on the performance tuning of Fabric mainly focused on block size parameters; we found four parameters related to block rules, i.e., PreferredMaxBytes, AbsoluteMaxBytes

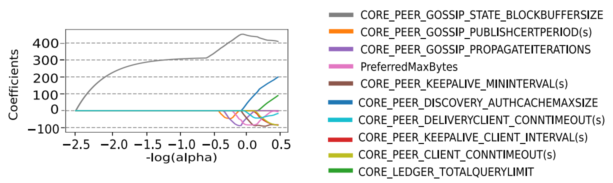

Figure 5: Parameter selection by LASSO path

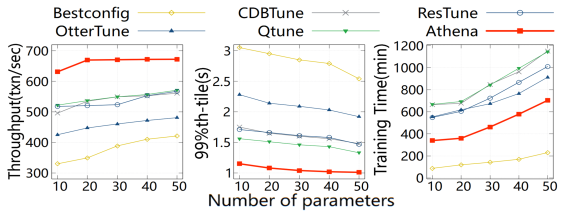

Figure 6: Performance by increasing the number of parameters

(rank 34), BatchTimeout (rank 36), and MaxMessageCount (rank 37). According to our experiment, these ranks are not as high as those in previous research, and even PreferredMaxBytes is ranked fourth, which is only mediocre. Essentially, many performance-related parameters have not yet been identified in existing research.

To evaluate the performance tuning better, we analyzed the reasons for the impact of the top five parameters on performance. CORE_PEER_GOSSIP_STATE_BLOCBUFFERSIZE is the most critical parameter, reflecting the size of the reordering buffer. We believe this parameter ranks high because increasing the cache space of block data helps peers improve the efficiency of pulling blocks, thereby assisting them in verifying quickly, which leads to higher throughput. The second is PUBLISHCERTPERIOD; the time startup certificates are included in live messages. This ensures the availability of certificates in peers and dramatically influences the continuous transmission of block information between peers. Next is CORE_PEER_GOSSIP_PROPAGATEITERATIONS, which represents the number of times a message is pushed to remote peers. A message is pushed to more peers in parallel, which can better leverage the advantages of Fabric's parallel execution and improve throughput. The fourth is PreferredMaxBytes, the preferred maximum number of bytes allowed for serialized messages in a batch. This parameter is the first rule to be triggered in Fabric slicing. Increasing the bytes of the block can include more transactions in the block, reducing the frequency of peers pulling blocks from the orderers, and thereby improving the performance. CORE_PEER_KEEPALIVE_MININTERVAL is the fifth important parameter and is the minimum permitted time between client pings. This parameter controls the ping frequency between the client and the peer. If this time is too short, the client frequently sends pings, which causes the peer and client to disconnect, resulting in the peer losing contact.

According to the sorting result of LASSO, we used ten more parameters each time for tuning. Figure 6 shows the tuning results of Athena, Bestconfig, OtterTune, CDBTune, Qtune, and ResTune based on C1 under the workload of Smallbank. When we tuned 20 parameters, the throughput reached (99.64%, 83.00%, 93.00%, 95.30%, 94.00%, and 83.00%) with all parameters, respectively. The latency increased by (1.93%, 4.30%, 5.60%, 6.10%, 5.80%, and 9.20%) compared

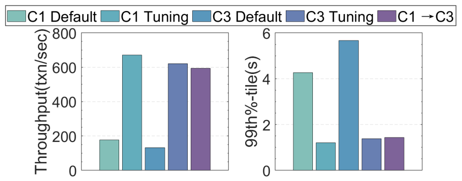

Figure 7: Performance for different Fabric network configurations

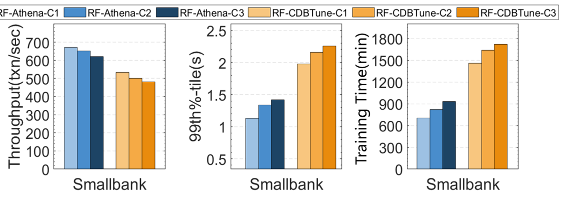

Figure 8: Comparison between RF-Athena and RF-CDBTune

to all parameters, respectively. The performance was equivalent to that of tuning all parameters. However, the benefits are apparent. The training time was reduced by (48.20%, 32.30%, 41.20%, 39.40%, 40.20%, and 48.79%) of that of tuning all parameters, respectively. Thus, we argue that the training time can be significantly reduced if important parameters are identified.

6.5 Adaptability

This section verifies the adaptability of Athena to different network configurations. Since Fabric is scalable, users can join the network according to business requirements. Therefore, this scalability naturally requires good adaptability of the tuning model. We experimented with two instances (C1 and C3, based on Smallbank workload). First, we used recommended parameters of instance C1 to C3 (C1⟶C3). We then evaluated the performance of instance C1 and C3 default parameters, C1⟶C3, C1 tuning (Athena tuning on C1), and C3 tuning (Athena tuning on C3). As shown in Figure 7, the configuration recommended by Athena is adaptable. Applying the configuration recommended by C1 to instance C3, the throughput improves by 350.4% compared with the default C3, which is 95.6% of the throughput after re-tuning.

6.6 Evaluation of Reward Functions

This section evaluates the reward functions of Athena. To verify the superiority of our exponential reward function (RF-Athena), we compared it with the linear reward function of CDBTune (RF-CDBTune). As shown in Figure 8, we use three configurations, i.e., C1, C2, and C3, and the chaincode is Smallbank, while it is fixed at C_T=C_L=0.5. We found that no matter the throughput, latency, and training time under the two configurations, RF-Athena is superior to RF-CDBTune. The throughput, latency, and training time of RF-Athena found by (C1, C2, C3) are (26.08%, 29.94%, and 28.84%), (42.93%, 37.96%, and 37.17%), and (51.85%, 50.00%, and 45.93%) better than RF-CDBTune under Smallbank, respectively. On the one hand, the reason for such a result is that the exponential reward function can scale up and down the reward value due to performance changes so that the model converges better. On the other hand, when the linear reward function is faced with a multi-agent RL model, the obtained reward value is the smallest, dramatically increasing the time for the model to converge. To summarize, compared with the

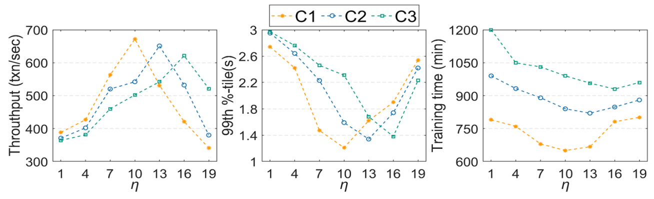

Figure 9: The η to optimize performance and training time

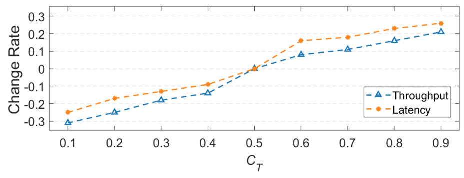

Figure 10: C_T to optimize performance

linear reward function, our proposed exponential reward function can achieve better performance and a faster convergence speed when adjusting the parameters of the permissioned blockchain.

η is a significant parameter in Eqs. (4) and (5). Its contribution is to solve the difference in performance feedback due to the Fabric network architecture or other external reasons. To evaluate the impact of the η, we varied η from 1 to 19 on C1, C2, and C3 (chaincode is Smallbank), while C_T= C_L= 0.5 is fixed. Figure 9 displays the results. We found that the best performance value for the η of (C1, C2, and C3) is (10, 13, and 16), respectively. All η values that achieve the best performance are above 10. The η value needs to be larger because the parameter adjustment space primarily results from many parameters. The η value needs to be increased to increase its sensitivity to performance changes. In this study, the η value is used as a hyperparameter to help users better optimize the auto-tuning effect.

We designed the throughput and latency weight parameters C_T and C_L to trade off the two optimization objectives in Eq. (6). To observe the impact of both on optimization performance, we set the performance of C_T=C_L=0.5 as the benchmark based on [30]. We experimented with instance C1; the chaincode is Smallbank. Figure 10 shows the results. We observe the rate of change of the two optimization objectives. With the increase of C_T, the throughput change rate increases, and the corresponding latency change rate increases. This is due to the fact that increasing C_T increases the contribution of the throughput variation to the reward. However, it will reduce the contribution of latency.

Generally, when adjusting these three parameters, the administrator first determines C_T and C_L according to the requirements. Under the requirement of high throughput, increase C_T, or reduce C_T for low latency. After determining C_T and C_L, the administrator initially defines η as ten and adjusts η according to the reward value. If the reward fluctuation is significant, reduce η; if the reward is small, increase η.

6.7 Evaluation of Fault Tolerance

This section evaluates two types of crashes: (1) Orderer crash: Fabric's consensus algorithm is Raft, which is crash fault-tolerant. We find that crashing a leading or non-leading orderer has a limited impact on performance. This is because, in the ordering service, the leading orderer performs the ordering process, and the other nodes are only backup operations. After the leader node crashes, the non-leading orderers elect a new leader in a short time. (2) Peer crash: Although other nodes can compensate for the crash of peers within the organization, this will increase the endorsement work on other nodes, thereby significantly affecting performance. To evaluate Athena's impact on clusters with peer crashes, we vary them from 1 to 6 in step 1 and fix the other parameters to their default values of C3 with Smallbank.

As shown in Figure 11, Athena achieves the best results, in which the throughput and latency tuned by Athena are (355.29%, 346.35%, 473.33%, 548.81%, 601.13%, and 721.13%) and (71.63%, 73.62%, 74.08%, 78.71%, 77.17%, and 77.28%), respectively, outperforming default crashes. The comparison demonstrates that Athena can facilitate the continuous use and excellent performance of the Fabric network in a crash state.

6.8 Summary

From these tests, we find the following.

(1) The overhead of our Athena is minimal, with each tuning process only costing 1.8% - 2.1%.

(2) Athena achieved the best tuning performance. The throughput and latency of (Athena, Bestconfig, OtterTune, CDBTune, Qtune, and ResTune) improves by (100.32%, 26.13%, 31.36%, 68.06%, 75.16%, and 83.23%) and (59.65%, 21.93%, 35.38%, 52.34%, 54.09%, and 52.63%), respectively, on Smallbank under C1, and by (89.78%, 24.77%, 42.11%, 65.63%, 72.76%, and 78.28%) and (67.09%, 26.79%, 32.14%, 49.49%, 54.85%, and 57.40%), respectively, Simple under C1, compared to manual tuning.

(3) The throughput of tuning 20 parameters equals 99.64% of the effect of tuning all the parameters under C1. The latency can only increase by 1.93%, which is nearly negligible, and the tuning time reduces by 48.79%.

(4) Despite the network change (C1→C3), the configuration recommended by Athena can realize a throughput improvement of 350.4% compared with the default configuration.

(5) The throughput, latency, and training time of Athena's exponential reward functions determined via (C1, C2, and C3) are (26.08%, 29.94%, and 28.84%), (42.93%, 37.96%, and 37.17%), and (51.85%, 50.00%, and 45.93%) better than the linear reward function under Smallbank, respectively.

(6) For fault tolerance, the throughput and latency tuned by Athena, are (355.29%, 346.35%, 473.33%, 548.81%, 601.13%, and 721.13%) and (71.63%, 73.62%, 74.08%, 78.71%, 77.17%, and 77.28%), respectively, outperforming default crashes.

7 DISCUSSION

This section discusses some details about the use of Athena in production and aims to clarify some specific points.

Intended user: The intended user of Athena is the administrator elected by the participating organizations in the real world, who has the privilege to access all servers and modify the Fabric configuration. The reasons for electing an administrator are as follows: (1) Ensure uniformity of parameter tuning results for each node. (2) All organizations give permissions to administrators, which can prevent data leakage between nodes. After tuning is completed, each organization configures its node configuration according to the recommendations of the administrator. It is recommended that the

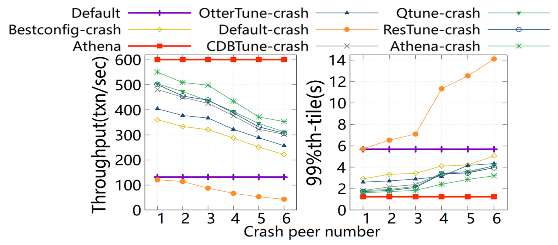

Figure 11: Robustness of tuning algorithms in nodes crash

parameters of all nodes are set to be the same because experiments show that some parameters in the nodes are different, rendering the system unusable. After the configuration is complete, each organization can disable the access authority of the administrator to ensure the security of the production environment data.

Operation procedure: In production, Athena adopts the above-mentioned centralized tuning and distributed deployment. The administrator tunes Fabric using a sample workload to generate recommendations, which are used on a real workload. Moreover, organizational users may join or quit, the hardware environment may change, or the business scenario shift. As an option, the administrator can use the data collected during the previous tuning process to rank the parameters via the LASSO. The administrator can select the important parameters of Fabric to retune all the nodes and retain the other parameters in their default configuration.

Available platforms: In general, Athena has been adapted to all versions of Fabric 1.4.x - 2.4.x. At present, we are actively promoting the adaptation of Athena to other types of permissioned blockchains (e.g., FISCO BCOS).

8 CONCLUSION

In this paper, we propose Athena, a Fabric-based auto-tuning system that can automatically provide parameters for optimal performance and only requires the controller to have administrative privileges to modify the configuration and restart Fabric in the training stage. First, we introduce the relationship between the parameters and performance at the theoretical level and transform the problem of tuning parameters in Fabric into a multi-agent coordination problem. Second, we propose a novel PD-MADDPG algorithm to solve this problem efficiently. In addition, we select the parameters that make the greatest contribution to improving the tuning efficiency. Finally, we experimentally verify that the recommendation from Athena is significantly more effective than that of the default configuration. By comparison, our method outperforms the other three most advanced solutions. Thus, we are among the first to study auto-tuning for the performance optimization of blockchains.

In future studies, we will better balance the relationship between the number of nodes and agents to solve the problem of more complex heterogeneous hardware configurations. Meanwhile, we plan to extend our approach to other blockchain systems.

ACKNOWLEDGMENTS

This work was supported by National Key R&D Program of China (No.2019YFB1706002), the Informatization Plan of Chinese Academy of Sciences (Grant No. CAS-WX2021SF-0508), NSFC (61925203), and Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing.

REFERENCES

[1] P. Ruan, T. T. A. Dinh, D. Loghin, M. Zhang, G. Chen, Q. Lin, B. C. Ooi. Blockchains vs. Distributed Databases: Dichotomy and Fusion. In *Proc. of 2021 ACM International Conference on Management of Data*, pages 1–14, 2021.

[2] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Ji Wang. Untangling blockchain: A data processing view of blockchain systems. *IEEE Trans. Knowl. Data Eng*, 30(7):1366–1385, 2018.

[3] M. Crosby, P. Pattanayak, S. Verma, and V. Kalyanaraman. Blockchain Technology: Beyond Bitcoin. In *Appl. Innov.,* vol. 2, pp. 6–10, Jun. 2016.

[4] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[5] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper,* vol. 151, pp. 1–32, Apr. 2014.

[6] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[7] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. 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)*, pages 264–276, 2018.

[8] P. Zheng, Z. Zheng, X. Luo, X. Chen, and X. Liu. A detailed and real-time performance monitoring framework for blockchain systems. In *Proc. 40th Int. Conf. Softw. Eng. Softw. Eng. Pract.-ICSE-SEIP*, pages 134–143, 2018.

[9] K. Wüst and A. Gervais. Do you need a Blockchain? In *Crypto Valley Conference on Blockchain Technology*, pages 45-54, 2018.

[10] K. Kanellis, C. Ding, B.Kroth, et al. LlamaTune: Sample-Efficient DBMS Configuration Tuning. *arXiv preprint arXiv:2203.05128*, 2022.

[11] Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. CGP-Tuner: a contextual gaussian process bandit approach for the automatic tuning of IT configurations under varying workload conditions. In *Proc. VLDB Endow.,* 14(8): 1401–1413, 2021.

[12] J. Wang, I. Trummer, and D. Basu. UDO: universal database optimization using reinforcement learning. *arXiv preprint arXiv:2104.01744*, 2021.

[13] K. Tan, Q. Cai, B. C. Ooi, W. Wong, C. Yao, and H. Zhang. In-memory databases: Challenges and opportunities from software and hardware perspectives. In *Proc. of 2015 ACM SIGMOD International Conference on Management of Data*, pages 35-40, 2015.

[14] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Billian, and Andrew Pavlo. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. In *Proc. VLDB Endow.,* 14(7):1241–1253, 2021.

[15] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. ResTune: Resource oriented tuning boosted by meta-learning for cloud databases. In *Proc. of 2021 ACM International Conference on Management of Data*, pages 2102-2114, 2021.

[16] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. Blurring the Lines Between Blockchains and Database Systems: The Case of Hyperledger Fabric. In *Proc. of 2019 ACM International Conference on Management of Data*, pages 105-122, 2019.

[17] J. A. Chacko, R. Mayer, and H.-A. Jacobsen. Why do my blockchain transactions fail? a study of hyperledger fabric. In *Proc. of 2021 ACM International Conference on Management of Data*, pages 221-234, 2021.

[18] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi. Towards scaling blockchain systems via sharding. In *Proc. of 2020 ACM International Conference on Management of Data*, pages 123-140, 2020.

[19] S. Nathan, C. Govindarajan, A. Saraf, M. Sethi, P. Jayachandran. Blockchain meets database: design and implementation of a blockchain relational database. In *Proc. VLDB Endow.,* 12(11):1539–1552, 2019.

[20] X. Yang, Y. Zhang, S. Wang, B. Yu, F. Li, Y. Li, W. Yan. LedgerDB: a centralized ledger database for universal audit and verification. In *Proc. VLDB Endow.,* 13(12):3138–3151, 2020.

[21] Ashraf Mahgoub, et al. OPTIMUSCLOUD: Heterogeneous Configuration Optimization for Distributed Databases in the Cloud. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 189-203, 2020.

[22] Ashraf Mahgoub, et al. Rafiki: A middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 28-40, 2017.

[23] Ashraf Mahgoub, et al. SOPHIA: Online reconfiguration of Clustered NoSQL Databases for Time-Varying Workloads. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 223-240, 2019.

[24] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. ibtune: Individualized buffer tuning for large-scale cloud databases. In *Proc. VLDB Endow.,* 12(10):1221-1234, 2019.

[25] V. Mnih, et al. Playing Atari with Deep Reinforcement Learning. In *Computer Science*, pages 351-362, 2013.

[26] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proc. of 2017 ACM International Conference on Management of Data*, pages 1009-1024, 2017.

[27] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. Too many knobs to tune? towards faster database tuning by pre-selecting important knobs. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[28] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. In *Proc. VLDB Endow.,* 12(12):2118-2130, 2019.

[29] C. Gorenflo, S. Lee, L. Golab, and S. Keshav. FastFabric: Scaling hyperledger Fabric to 20 000 transactions per second. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 455-463, 2019.

[30] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proc. of 2020 ACM International Conference on Management of Data*, pages 415-432, 2020.

[31] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 338-350, 2017.

[32] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 244-259, 2013.

[33] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, B. C. Ooi. A transactional perspective on execute-order-validate blockchains. In *Proc. of 2020 ACM International Conference on Management of Data*, pages 543-557, 2020.

[34] Metrics. 2023. https://hyperledger-fabric.readthedocs.io/en/release-2.4/metrics_reference.html. [Online; accessed 2-January-2023].

[35] paddlepaddle. 2023. https://github.com/PaddlePaddle/Paddle. [Online; accessed 2-January-2023].

[36] Hyperledger Caliper. 2023. https://hyperledger.github.io/caliper/. [Online; accessed 2-January-2023].

[37] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267-288, 1996.

[38] Prometheus. 2023. https://prometheus.io/. [Online; accessed 2-January-2023].

[39] Hyperledger Blockchain Performance Metrics White Paper. 2023. https://www.hyperledger.org/learn/publications/blockchain-performance-metrics. [Online; accessed 2-January-2023].

[40] Dawei Qiu, Jianhong Wang, Junkai Wang, and Goran Strbac. Multi-Agent Reinforcement Learning for Automated Peer-to-Peer Energy Trading in Double-Side Auction Market. In *IJCAI*, pages 2913-2920, 2021.

[41] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in neural information processing systems*, pages 2681–2690, 2017.

[42] W. Lyu, Y. Lu, J. Shu, and W. Zhao. Sapphire: Automatic Configuration Recommendation for Distributed Storage Systems. *arXiv preprint arXiv:2007.03220*, 2020.

[43] Liang Zhu, Chao Chen, Zihao Su, Weiguang Chen, Tao Li, and Zhibin Yu. Bbs: Micro-architecture benchmarking blockchain systems through machine learning and fuzzy set. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 411-423, 2020.