An Algorithm and Architecture Co-design for Accelerating Smart Contracts in Blockchain

Rui Pan
Hunan University
Changsha, Hunan, China
prui@hnu.edu.cn

Chubo Liu*
Hunan University
Changsha, Hunan, China
liuchubo@hnu.edu.cn

Guoqing Xiao
Hunan University
Changsha, Hunan, China
xiaoguoqing@hnu.edu.cn

Mingxing Duan
Hunan University
Changsha, Hunan, China
duanmingxing@hnu.edu.cn

Keqin Li
State University of New York, New Paltz
New York, USA
Hunan University
Changsha, Hunan, China
lik@newpaltz.edu

Kenli Li
Hunan University
Changsha, Hunan, China
lkl@hnu.edu.cn

ABSTRACT

Modern blockchains supporting smart contracts implement a new form of state machine replication with a trusted and decentralized paradigm. However, inefficient smart contract transaction execution severely limits system throughput and hinders the further application of blockchain.

In this work, we achieve efficient multi-layer parallelism of transactions through an algorithm and architecture co-design. Based on the analysis of smart contract invocations, we find that transactions are widely redundant in spatio-temporal distribution and reveal a restriction of scheduling benefits resulting from redundant operations. We propose a spatio-temporal scheduling algorithm to solve this restriction and design a multi-transaction processing unit (MTPU) to accomplish the optimization of transaction parallelism and redundancy. Transactions are asynchronously parallel in the spatial dimension through decoupling of execution and scheduling. Fine-grained data and instruction reuse enables transaction de-redundancy in the time dimension. In addition, we collect execution information for frequently invoked hotspot contracts and perform deep optimization in the idle time slice. Finally, our evaluation shows that the algorithm and architecture co-design is able to achieve a further acceleration of 3.53× − 16.19× compared to existing schemes.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; • **Hardware → Analysis and design of emerging devices and systems**.

*Corresponding author.

ISCA '23, June 17–21, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589067

KEYWORDS

Blockchain; Smart contract; Parallelism

ACM Reference Format:
Rui Pan, Chubo Liu, Guoqing Xiao, Mingxing Duan, Keqin Li, and Kenli Li. 2023. An Algorithm and Architecture Co-design for Accelerating Smart Contracts in Blockchain. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579371.3589067

1 INTRODUCTION

A blockchain is essentially a distributed ledger, consisting of a data structure called a block [35]. Each block contains a batch of transactions and a hash of the previous block, thus forming a tamper-proof chain. These blocks are jointly maintained by all nodes in a P2P network, as opposed to the centralized management found in traditional distributed databases. It is because of these excellent qualities of decentralization, tamper-proof and traceability that blockchain is being used in many critical areas, including cryptocurrencies [26], supply chains [28], security audits [50], healthcare systems [10], and the metaverse [47].

The hallmark of modern blockchain systems is the support of smart contracts [45]. Written in Turing-complete language [40], smart contracts are used to describe the complex business logic of blockchain applications. A transaction in the blockchain is either a token transfer or an invocation of a smart contract (Fig. 1 lower part). After being negotiated by the P2P nodes, these transactions are added to the chain in the form of blocks. All nodes execute these transactions to complete a consistent update to the system state. Thus, the throughput rate of a blockchain can be expressed as the ratio of the number of transactions within a block to the block generation interval, as shown in Fig. 2. The throughput rate is an important metric for evaluating the performance of a blockchain and is widely used.

Compared to existing centralized database systems, low throughput blockchains are completely unable to meet the real-time requirements of scenarios such as finance and industry [3]. This paper focuses on optimizing transaction execution to improve throughput. On the one hand, in a mature blockchain system, the block generation interval remains largely constant (Fig. 2(a)), as it is related to

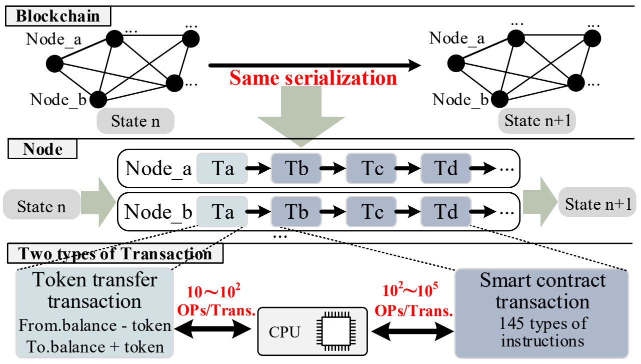
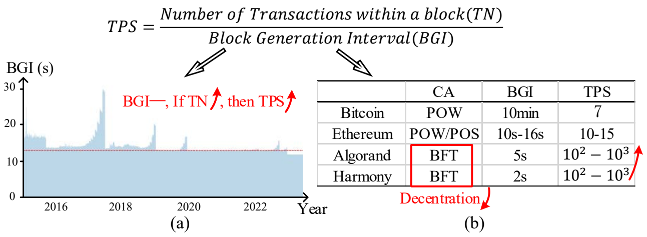


Figure 1: Sequentially execution of transactions in the blockchain.



	CA	BGI	TPS
Bitcoin	POW	10min	7
Ethereum	POW/POS	10s-16s	10-15
Algorand	BFT	5s	$10^2 - 10^3$
Harmony	BFT	2s	$10^2 - 10^3$

Figure 2: (a) Ethereum block generation interval. (b) Performance of blockchain systems with different consensus algorithms (CAs) [18, 20].

	2017	2018	2019	2020	2021
Daily Transaction	282K	688K	665K	932K	1265K
Proportion of SCTs	37.23%	50.57%	63.52%	67.94%	68.40%
Execution overhead of SCTs	72.44%	81.83%	87.97%	90.43%	90.81%

Table 1: Ethereum statistics from 2017 to 2021 [15].

the security and stability of the overall system. By improving the efficiency of transaction execution, the system can pack more transactions into blocks and process them within a fixed block interval, increasing throughput. On the other hand, some blockchain systems have compromised in order to achieve higher throughput rates. Efficient but not fully decentralized consensus algorithms are deployed (Fig. 2(b)), which limits node size and makes additional security assumptions [6, 13, 22, 34]. In contrast, accelerating transaction execution requires almost no changes to the blockchain system and is a foundational technology with full backward compatibility [12].

The execution of a smart contract largely determines the efficiency of the overall transaction execution. Compared to token transfer transactions, transactions that invoke smart contracts (i.e., smart contract transactions, SCTs) increase the operational complexity by up to three orders of magnitude. Furthermore, as smart contracts are widely invoked, the execution of SCTs dominates the overhead ($> 90\%$). Unfortunately, as shown in Fig. 1, these complex SCTs are always executed sequentially to ensure consistency of execution results between nodes.

The existing approaches to accelerate the execution of SCTs mainly focus on exploiting rough parallelism among different transactions from software perspective [4, 5, 8, 9, 14, 25, 38, 49]. Few hardware works design a dedicated architecture for smart contract [31], but the high performance is only applicable to specific scenarios. To overcome the limitation of acceleration, we design the multilayer parallel architecture based on the characteristics of transaction dependency and redundancy in spatio-temporal distribution.

Transactions that call the same smart contract are considered redundant transactions. When executing these transactions, the processor will construct almost the same context and execute almost the same logic. On the one hand, with redundancy among non-dependent transactions that are freely executed in the space dimension, most of the parallel resources are wasted in redundant operations. On the other hand, dependent transactions executed in strict order in the time dimension are the critical path of parallelism, and the redundancy among these transactions limits the benefits of scheduling.

First, redundancy is common because some smart contracts that define common or popular functions tend to be invoked much more frequently (we refer to hotspot contracts). For example, up to 37% of SCTs invoking TOP5 smart contracts [1]. Second, de-redundancy enables adequate performance. We find that reusing only smart contract bytecode will reduce the loading of the vast majority of contextual data. The sharing of the latest state data will reduce instruction access latency. For transactions with the same smart contract and entry function, their execution paths overlap almost exactly, and this predictable execution presents more optimization opportunities.

In view of this, we design a spatial-temporal scheduling algorithm. Scheduling does not violate blockchain consistency due to the ability to obtain transaction dependencies before execution. By decoupling execution and scheduling, it enables asynchronous execution of transactions to maximize spatial parallelism benefits. Redundant transactions are executed sequentially by the same processing unit, providing opportunities for redundancy reduction in the time dimension. To better support the spatial-temporal scheduling algorithm and multi-layer parallelism, we propose the Multi-Transaction Processing Unit (MTPU). In MTPU, we construct a three-layer memory architecture for efficient access and reuse of transaction data. For smart contracts, we build an efficient instruction parallel pipeline and optimize the instruction parallelism relationship. The benefits of these base designs are further amplified, as efficient parallel instructions are reused between subsequent redundant transactions. Last but not least, we persistently store the execution information of hotspot contracts and perform deep optimization of instruction execution and data access during the block generation interval.

The main contributions of this work are summarized as:

- We design an accelerator framework for blockchain transaction execution. At the system level, the parallelism among transactions is fully utilized through an algorithm and architecture co-design. At the microarchitecture level, pipelining, instruction and data optimization are used to accelerate execution.
- We propose a spatial-temporal scheduling algorithm that enables asynchronous execution of transactions in the spatial dimension and provides redundancy reduction in the temporal dimension.

[1]The data from sampling statistics of Ethereum Blocks from 2020.9 to 2022.9.

- We perform deep optimization for hotspot contracts in the block generation interval. At the data level, we prefetch access data on critical paths. At the code level, we identify and optimize constant instructions and implement partial pre-execution.
- We implement a prototype of MTPU in RTL and synthesized it into a 45nm ASIC. Compared to single core, 3.53× − 16.19× speedup ratio is achieved with parallel execution of multiple transactions.

2 BACKGROUND AND MOTIVATION

2.1 Background

Ethereum, as a representative of modern blockchains, is currently the largest permissionless blockchain system that supports smart contracts [45]. Like many other software programs, smart contracts are developed using high-level programming languages. They are compiled into bytecode and executed in the Ethernet Virtual Machine (EVM) on each P2P node.

Transactions are network transported and persisted by recursive length prefix (RLP). The data format is shown in Fig. 3(a). Smart contracts are assigned unique addresses when they are deployed on the blockchain. There is a mapping relationship between the address and the smart contract bytecode. EVM loads the smart contract bytecode according to the *To* field. The specific invocation information is represented by the *Input* field, including the entry function (represented by the function identifier) and the incoming parameters.

Unlike normal programs, each instruction in a smart contract has a corresponding gas overhead [2]. Each transaction has a gas limit (*Gas* field). The EVM must check that there is sufficient gas before executing each instruction, and if there is not, the transaction is aborted. Furthermore, the gas overhead of a successfully executed transaction is eventually deducted from the sender's account in the form of a fee. Therefore, a transaction has only one uniquely determined gas overhead.

The transaction actually goes through a three-stage model from initiation to execution [12], as shown in Fig. 4. In the dissemination stage, nodes broadcast transactions to be confirmed across the network. Nodes collect these transactions and store them locally. In the consensus stage, the elected node packages the transactions and their execution results into a new block and broadcasts it; In the execution stage, other nodes sequentially execute transactions in the new block and verify the results. This model restricts nodes to complete the execution of all transactions in the current block before processing the next block, since the construction and verification of new blocks require the latest state. Consequently, transaction execution is always on the system throughput critical path.

Note that the consensus stage and the execution stage are decoupled. Triggering the execution stage requires only new blocks that have gone through consensus. Thus, no matter how the consensus algorithm is optimized, transaction execution is a completely independent process. Accelerating transaction execution is a technique that is completely orthogonal to consensus optimization.

[2]EVM is Turing-complete, allowing the execution of any code. To prevent network downtime caused by malicious programs running all the time, Ethernet introduces a different gas cost for each operation.

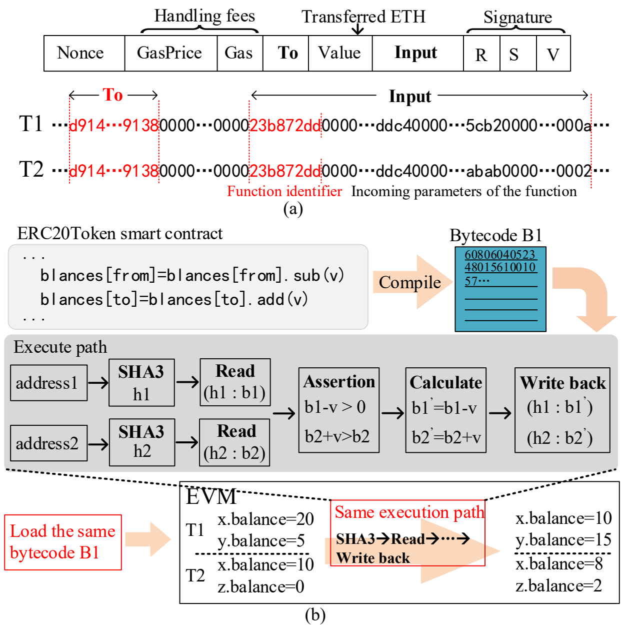

Handling fees
Transferred ETH
Signature
Nonce
GasPrice
Gas
To
Value
Input
R
S
V
To
Input
T1 ···d914···9138 0000···000 23b872dd 0000···ddc40000···5cb20000···000a···
T2 ···d914···9138 0000···000 23b872dd 0000···ddc40000···abab0000···0002···
Function identifier
Incoming parameters of the function
(a)
ERC20Token smart contract
blances[from]=blances[from].sub(v)
blances[to]=blances[to].add(v)
Compile
Bytecode B1
60806040523
48015610010
57···
Execute path
address1
SHA3 h1
Read (h1 : b1)
address2
SHA3 h2
Read (h2 : b2)
Assertion b1-v > 0 b2+v>b2
Calculate b1'=b1-v b2'=b2+v
Write back (h1 : b1') (h2 : b2')
EVM
Load the same bytecode B1
T1 x.balance=20 y.balance=5
T2 x.balance=10 z.balance=0
Same execution path
SHA3→Read→···→ Write back
x.balance=10 y.balance=15
x.balance=8 z.balance=2
(b)

Figure 3: (a)Transaction data structure. (b)Redundancy between transactions T1 and T2: the same smart contract bytecode is loaded and the same execution path is executed.

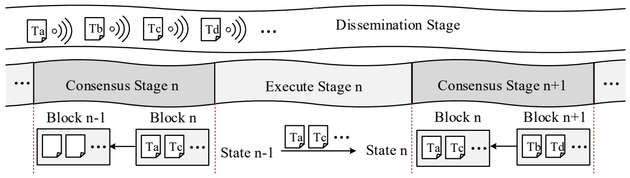


Figure 4: A three-stage model of Dissemination-Consensus-Execution.

2.2 Motivation

There has been a lot of research on using concurrency control techniques to execute transactions in parallel under consistency constraints [4, 5, 14, 38, 49]. Unfortunately, the parallelism benefit is constrained by the execution speed of smart contracts at the software level. Designing dedicated accelerators for smart contracts enables more thorough acceleration [31], but only for a certain class of smart contracts. In this work, we analyze the acceleration opportunities from multiple perspectives, inspiring us to explore a better algorithm and hardware architecture.

2.2.1 Redundancy between transactions. As shown in Fig. 3(b), there is redundancy between $T1$ and $T2$ because they call the same smart contract (same *To* field) and have the same execution path (same function identifier). Execution contexts are shared between redundant transactions to avoid loading duplicate data, especially bytecode and partial read/write sets. As shown in Table 2, bytecode makes up for a relatively large portion of the data loaded during the transaction execution. During execution, the read/write sets of these transactions are likely to contain public variables. It is also important to reuse this data to optimize the latency of instruction accesses. Furthermore, the execution paths of such transactions overlap. The same bytecode and instruction control logic provide

Smart Contract	Function	Size(Byte)			
		Bytecode		Other Data	
Tether	Transfer	5759	92.72%	452	7.28%
WETH9	Withdraw	1607	90.74%	164	9.26%
CryptoCat	createSale Auction	12500	95.33%	612	4.67%
Ballot	Vote	1203	85.99%	196	14.01%

Table 2: The proportion of bytecode in the loaded data.

opportunities for optimizing the design of instruction pipelines and caches. Finally, we analyze a sample of about 14.8 million transactions, and the percentage of transactions calling the TOP 5 smart contracts is as high as 37%. Redundancy between transactions is widespread, and we see a strong need for optimization.

2.2.2 *Adequate scheduling opportunities.*

First, Dependencies between transactions is represented by a directed acyclic graph (DAG), which is discovered by nodes in the consensus stage through concurrency control or software transaction memory. It has been proven that nodes are well motivated to put the DAG into a block [3] so that the whole network nodes benefit from it [2, 4, 5, 14, 25]. Second, the information about the transaction invoking the smart contract is fully known before execution (*To* and *Input* fields). Pre-static analysis of this information provides ample scheduling opportunities. Conflict-free transactions will be executed in parallel. Redundant transactions will be assigned to the same processing unit to achieve execution optimization.

2.2.3 *Dynamic optimization of hotspot contracts.*

In different scenarios, different application needs lead to hotspot contracts with different logic. For example, token-based contracts in the financial scenario are no longer hot in the industrial or medical scenarios. Moreover, hotspots change over time. For example, the once extremely hot CryptoCat on Ethereum, which at its peak was invoked by 14% of transactions, is hardly active anymore. This means that some studies optimized for specific types of smart contracts may not be widely applicable [31]. There is a tradeoff between flexibility and acceleration performance when performing hotspot contract optimization.

2.2.4 *Intermittent transaction execution.*

As described in the three-stage model, the transaction execution that affects throughput occurs only during the execution stage. Other idle time slices provide ample opportunity for offline optimization.

3 ALGORITHM AND ARCHITECTURE CO-DESIGN

3.1 Insight

The key insights of our design are multi-layer parallelism and redundancy optimization, as shown in Fig. 5.

(1) Transaction-level parallelism (TLP). At the software level, a transaction is a single invocation of a smart contract. The processor traverses and executes the transactions in the entire block to update the status. State consistency imposes serial semantics on transaction execution. The most straightforward optimization is serializable concurrent scheduling of transactions.

[3]There are only data dependencies between transactions, and DAGs are serialised and persistently stored in blocks.

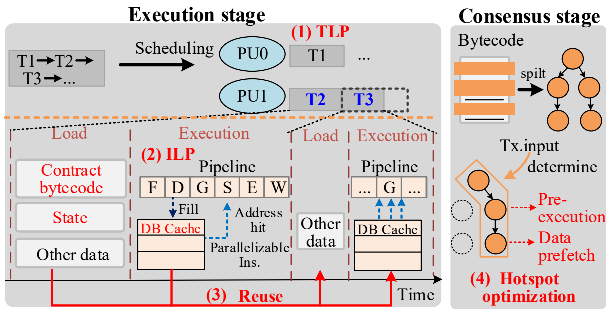


Figure 5: (1)Transaction-level parallelism; (2)Instruction-level parallelism; (3) Reuse between transactions; (4) Hotspot optimization.

(2) Instruction-level parallelism (ILP). In EVM, bytecodes are sequentially decoded and then executed. Implementing ILP by designing an efficient instruction pipeline can fundamentally improve the efficiency of smart contract execution. However, ILP needs to be applied carefully in blockchain scenarios. The biggest difference from other programs is that smart contract instructions need to perform gas calculations before execution. Dynamic/aggressive ILP is no longer suitable, as instruction conflicts or branch prediction failures will lead to rollbacks. Deducting and recovering gas would introduce additional computation and concurrent reads/writes, significantly increasing the cost of instruction parallelism. Therefore, we use a conservative ILP.

(3) Reuse between transactions. At the data level, redundant transactions have partially overlapping execution contexts. In EVM logic, the context is reconstructed before each transaction is executed. This redundant memory access is completely unnecessary. At the instruction level, redundant transactions have exactly the same bytecode and partially the same control logic. Instruction optimization for one transaction can also be partially applied to all subsequent transactions with which it has a redundant relationship.

(4) Hotspot optimization. Some smart contracts are called repeatedly and become execution hotspots. Unlike other database systems, transaction execution in a blockchain is intermittent and occurs only during the execution stage. We place hotspot optimization on a free time slice outside of the execution stage to avoid contention for computational resources. Since hotspots are executed frequently, the execution path is relatively clear, including dependencies between instructions and data streams. Caching this information for hotspot contracts provides more opportunities to optimize instructions and data.

Therefore, we first design a spatial-temporal scheduling to achieve serializable parallelism between transactions (1). Transactions that have redundant relationships are scheduled on the same processing unit as much as possible to gain more reuse opportunities (3).

In the MTPU, we add a decoded bytecode cache (DB cache) to the instruction pipeline bypass (2). It collects decoded bytecodes (instructions) and fills the same cache line with instructions that have no dependencies. When an address hits a cache line, all instructions in that line are executed in parallel. Instruction rollback does not occur, because dependencies are handled at the point of instruction filling and no aggressive branch prediction is performed. In addition, we optimize instruction dependencies so that each cache line is filled with as many instructions as possible, allowing for more

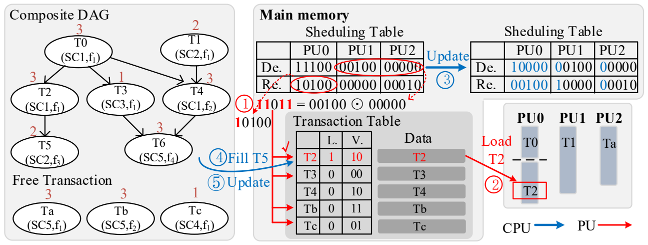


Figure 6: Spatial-temporal scheduling.

parallelism. More importantly, the DB cache can be reused between redundant transactions (3), and the above design will yield many times the benefit.

We design a three hierarchical memory architecture (3). The separation of shared and private memory enables both isolation between parallel transactions and reuse of data and instructions between redundant transactions.

Perform hotspot optimization offline (4). Base on the execution logic collected, the bytecode of a hotspot contract can be easily split into multiple chunks. Furthermore, the vast majority of transactions are known to the node before to the execution stage [12]. Thus, for hotspot contract transactions, part of the execution path is determined prior to execution. We perform pre-execution and data prefetching for the bytecode chunks on the execution path.

3.2 Spatial-temporal Scheduling Algorithm

As shown in Fig. 6, the core data structure of the scheduling algorithm consists of a composite DAG (inter-transaction dependencies and contract invocation information), a Scheduling Table, and a Transaction Table. The CPU selects a batch of candidate transactions to write to the main memory. The information about these transactions is updated to the Scheduling Table and the Transaction Table. The processing unit (PU) reads the most suitable transaction into its own private memory based on these two tables. After the read is completed, the CPU writes the new transaction to the main memory again.

3.2.1 Write transactions. In the composite DAG, directed edges represent the priority order of transactions. For example, $T0 \rightarrow T2$ means that the execution of transaction $T2$ depends on the execution result of transaction $T0$. The value of a node indicates the degree of redundancy. For example, the value of the $T0$ node indicates that the $SC1$ invoked by $T0$ will be executed three more times.

In the initial phase or after the PU finishes reading the transaction, the CPU writes new transactions and their associated data to the main memory. To ensure conflict-free parallelism in the spatial dimension, the indegree of these transactions is 0. Meanwhile, to provide the opportunity for de-redundancy in the time dimension, transactions that call the same smart contract as the transactions being executed are prioritized, and if not, transactions with the larger value.

3.2.2 Read transactions. The main memory holds m candidate transactions, and the PU selects the best transaction based on the Scheduling Table and the Transaction Table.

Scheduling Table. The dependency (*De*) and redundancy (*Re*) items are represented by m bits. If i-th bit is 1, it indicates that the transaction being executed in the PU has a dependency/redundancy relationship with the i-th candidate transaction. For example, the candidate transactions *T2, T3, T4* depend on the running transaction *T0* in *PU0*, so the *De* of *PU0* is represented as 11100 (Fig. 6). The redundancy relationship (*Re*) is similar.

Transaction Table. To ensure that transactions are read by only one PU, we use locks (*L*) to mark the availability of transactions. The *V* corresponds to the node value in the composite DAG and indicates a priority of the candidate transaction.

As shown in Fig. 6, the flow of transaction selection is as follows. When *PU0* finishes *T0*, the transaction it selects must not have a dependency with the transaction being executed (*T1, Ta*). Therefore, PU0 calculates 11011 based on the *De* of *PU1* and *PU2* (00100 and 00000), indicating that the transactions that *PU0* can select include *T2, T3, Tb* and *Tc* (①). Then, based on the *Re* (10100) of *PU0*, *T2* is read in priority (②). If there is no redundancy, the transaction with the largest V is read. This transaction will be locked until the processing unit has completed its read operation. The CPU subsequently updates the *De* and *Re* (③) and writes *T5* to the main memory (④). Finally, the CPU updates the Transaction Table (⑤).

Due to the asynchrony between the CPU and the PUs, there is a delay in the update operation of the Scheduling Table. To avoid dirty reads (i.e., another PU reads the Scheduling Table before the CPU completes the update operation), we add a bit to the dependency item to indicate whether the data is valid or not. Invalid dependencies are treated as all zeros because the completed transaction no longer affects the execution of other transactions.

3.2.3 Efficiency. (1) Scheduling and execution are decoupled and performed by CPU and PUs, respectively, hiding the time overhead of scheduling. (2) Transactions are executed dynamically and asynchronously. A small amount of memory overhead is added to mitigate data sharing and asynchronous read/write conflicts. (3) Transaction selection (①, ②) is located on the critical path of asynchronous execution, but its overhead is limited to $O(n)$ bits of logical operations.

3.3 Hardware Design

Fig. 7 shows the components and data flow of the MTPU architecture. Each PU of the MTPU reads data from the main memory asynchronously and executes transactions concurrently. All updated state is written to the State Buffer and written back to the main memory at the appropriate time.

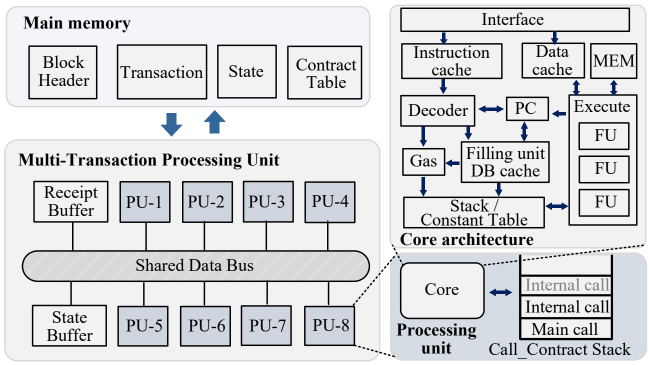


Figure 7: MTPU architecture.

Name	Opcode	Bytecode
Arithmetic	ADD, MUL, SUB, DIV, SDIV, MOD, SMOD, ADDMOD, MULMOD, EXP,SIGNEXTEND	0x01-0x0b
Logic	LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, NOT	0x10-0x1d
SHA	SHA3	0x20
Fixed access	ADDRESS, ORIGIN, CALLER, CALLVALUE, GASPRICE, CALLDATALOAD,CALLDATASIZE, CALLDATACOPY, CODESIZE, GASPRICE, BLOCKHASH,...,GASLIMIT, PC, GAS	0x30, 0x32-0x3a, 0x3d, 0x3e, 0x40-0x45, 0x58, 0x5a
State query	BALANCE, EXCODESIZE, EXCODECOPY, EXCODEHASH	0x31,0x3b,0x3c, 0x3f
Memory	MLOAD,MSTORE, MSTORE8, MSIZE, LOG0,...LOG4	0x51-0x53, 0x59, 0xA0-0xA4
Storage	SLOAD, SSTORE	0x54,0x55
Branch	JUMP,JUMPI,JUMPDEST	0x56, 0x57,0x5b
Stack	POP, PUSH1,...PUSH32, DUP1,...DUP16, SWAP1,...SWAP16	0x50,0x60-0x9f
Control	STOP, RETURN, REVERT	0x00,0xf3, 0xfd
Context switching	CREATE, CALL, CALLCODE,DELEGATECALL, CREATE2, STATILCCALL	0xf0-0xf2, 0xf4, 0xf5, 0xfa

TABLE 3: Functional units.

PU design. We implement the smart contract instruction set through a modular design. For the unique gas feature of smart contracts, we propose a dedicated instruction pipeline and implement ILP using the DB cache. In addition, we eliminate some of the instruction dependencies to improve parallelism.

Memory design. To better support data reuse, we design a three hierarchical memory architecture. Moreover, we perform data flow optimization to efficiently support multiple PUs to execute transactions in parallel.

3.3.1 *Modular design.*

As shown in Table 3, we implement the instruction set motivated by [45]. According to the operation logic of instructions, we divide them into several categories, each of which corresponds to a functional unit. Based on the modular design, we can optimize the execution of different instructions separately, and also facilitate the subsequent expansion of the instruction set.

3.3.2 *Instruction Pipeline.*

The pipeline is designed to follow the logic of the EVM executing the bytecode, as shown in Fig. 8 (a). First, the instruction is fetched according to the Program Counter (PC). Then, the decoding unit pre-decodes the instruction and generates a control signal that is sent to the execution unit. Next, the Gas unit subtracts the gas overhead of this instruction. If it is insufficient, an exception is returned and the transaction is aborted. Subsequently, the instruction is executed by the corresponding functional unit. If necessary, the operands are fetched from the stack before execution. Finally, the execution result is written back.

3.3.3 *ILP under Consistency Constraints.*

The consistency of the blockchain requires that the amount of gas consumed by each transaction is uniquely determined. Each instruction must be checked for gas as it is executed, so the gas margin must be correct at all times. However, dynamic and aggressive ILP allows instructions to be executed that should not be. In this case, a smaller margin of gas will cause subsequent checks to fail. Therefore, we use a low-cost DB cache and redesign the cache line to achieve deterministic ILP.

As shown in Fig. 8 (b), we add a fill unit and a DB cache outside the pipeline. The fill unit collects the decoded bytecode and fills the cache lines according to data dependencies and control logic. All instructions in the same line are combined together and can be issued in the same cycle and executed in functional units simultaneously.

In the DB cache, the length of each line is related to the number of functional units. Each functional unit corresponds to a fixed-length field. When the fill unit fills a decoded bytecode into the

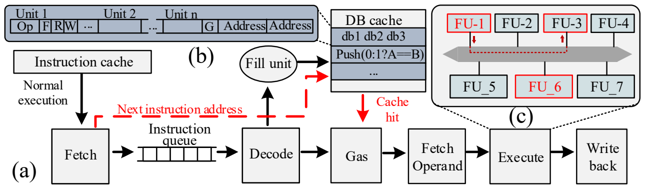


Figure 8: (a) Six-stage pipeline; (b) DB cache; (c) Data forward.

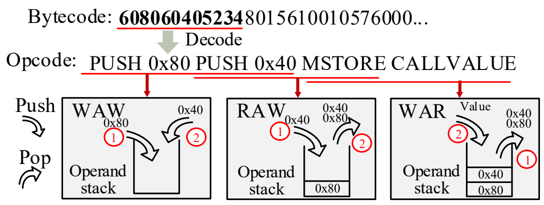

Bytecode: 608060405234801561001057600...
Decode
Opcode: PUSH 0x80 PUSH 0x40 MSTORE CALLVALUE
Push
Pop
WAW
RAW
WAR
Value
Operand stack
0x80
0x40

Figure 9: There are data dependencies between every two adjacent instructions.

cache line, if there is a dependency between instructions or the functional unit field requested by the instruction is not empty, the filling of this line ends. The next instruction is filled from a new line. Since the number of instructions contained in each line is uncertain, the address of the next instruction is recorded at the end of the line, which is provided to the branch unit.

Each line is identified by the address of the first filled instruction. If the address of the next instruction hits a line in the DB cache, all instructions of this line will take precedence over the normal execution path and skip the decoding stage. Before multiple instructions in the same line are executed, the sum of gas consumed will be deducted at one time. This information is recorded at the end of the line (G).

3.3.4 *Parallelism Improvement.*

The benefit of instruction parallelism is closely related to the space utilization of the DB cache. We expect each line in the DB cache to be populated with as many instructions as possible. However, there is a widespread data dependency between instructions in the smart contract instruction set, resulting in cache lines being filled too quickly. For example, Fig. 9 shows that stack-out and stack-in operations during instruction execution constitute write-after-read (RAW), read-after-write (WAR), and write-after-write (WAW).

To eliminate WAR and WAW, we add read and write sequence numbers (R, W) to the cache line to indicate the amount of data read from and written to the stack at one time. The functional unit takes the corresponding operand from the top of the stack according to R and writes the result to the corresponding location in the buffer according to W.

Stack instructions other than POP (56.18% of all instructions) perform write operations to the stack, imposing a large number of RAW dependencies. Therefore, we add data forwarding between functional units to amortize their impact. According to the frequency and logic of instruction execution, we divide functional units into reconfigurable units and non-reorganizable units, and further subdivide the cycle into the upper half cycle and the second half cycle. The execution logic of the reconfigurable unit is relatively simple, and its execution process only takes half a cycle. Only the results of their execution are forwarded to each other. To indicate the source of the forwarded data, we add a new field (F)

to the line. Note that the forwarding technique can only eliminate one RAW dependency. When the second RAW dependency occurs between instructions, the line filling is completed.

Although the above optimizations enable more instructions to be parallelized, there are still a large number of redundant stack operations between these instructions. We perform pattern recognition and instruction folding on the decoded instructions to eliminate some redundant operations. When a foldable pattern occurs, the fill unit fills the synthesized instruction directly into the cache line. Since there are countless patterns that can be folded, only the most common patterns are checked. Note that instruction folding does not occur on the critical path of instructions, and its impact on performance is very limited.

The following bytecode demonstrates the benefits of these optimizations. Almost all smart contract bytecodes contain function jump logic:

PUSH4 0xCC80F6F3 EQ PUSH2 0xB6 JUMPI

(1) PUSH4 0xCC80F6F3: Push the function identifier;

(2) EQ: Take the top two elements of the stack, judge whether they are equal, and push the result into the stack;

(3) PUSH2 0xB6: Push the branch address of the jump;

(4) JUMPI: Take the top two elements of the stack (condition and address), and execute the conditional branch.

We perform folding on the first and second instructions, and obtain a synthetic instruction, PUSH (0:1? top == 0xCC80F6F3). At the fetch operand stage, the stack top element and function identifier are sent directly to the logical function unit to complete the judgment. More importantly, this folding eliminates a RAW dependency without using forwarding technology. Therefore, this folding instruction and the remaining two instructions can be placed in the same line (eliminating the RAW dependency between them through forwarding technology). The function jump logic, which originally takes four cycles, can be reduced to one cycle.

3.3.5 *Reuse of the DB cache.*

With the help of upper-level scheduling, the DB cache achieves more instruction parallelism among transactions. Redundant transactions are scheduled on the same PU as much as possible. The instruction parallelism information in the DB cache is reused during execution. This is because the bytecodes executed by redundant transactions are exactly the same, and the execution difference is only in the input data and state. Therefore, the instruction addresses of the overlapping parts of the execution path will hit the DB cache line directly. Ultimately, more parallelism is achieved without increasing the size of the DB cache.

3.3.6 *Memory Hierarchy.*

To better support execution optimization and transaction parallelism, we design a three-level memory architecture. First, each processing core has proprietary caches for storing data or instructions. Second, the execution environment buffer outside the core holds the context of transaction execution, including the input (initial state, block information, and contract invocation information) and the output (updated state and generated receipt information) of the transaction. Third, the main memory serves as the last level of the memory hierarchy between processing units and off-chip memory, holding the transactions that will be processed in the next execution stage.

Data		Explanation	Length
Block Header	*Height*	Block number	Fixed
	Timestamp	Approximate time of block generation	
	Coinbase	Miners address	
	Difficulty	Difficulty target of mining	
	GasLimit	Gas limit of block	
	Hash[256]	Hash of the first 256 blocks	
Transaction	*Nonce*	The number of transactions sent by the sender	Fixed
	gaslimit	Gas limit of transaction	
	gasPrice	The gas price paid by the sender	
	From	Initiator address	
	To	Receiver address	
	CallValue	The number of tokens transferred	
	DataLen	Length of input data	
	Data[]	Additional input data	Variable
State	*Address*	Account address	Fixed
	nonce	Transaction or contract serial number	
	Balance	Account balance	
	CodeLen	Length of contract code	
	CodeHash	Hash of contract code	
	Code	Contract code of contract account	Variable
	Storage	Storage content of contract account	

TABLE 4: Data in the main memory.

Main memory. We analyze the smart contract code and classify the data accessed during execution into the following three categories: *Block Header*, *Transaction*, and *State* (Table 4). In order to improve access efficiency, we store the fixed-length parameter information and the variable-length parameter information separately. We connect fixed-length parameters that can be read entirely in a single cycle. By dynamically allocating space for variable-length parameters, memory utilization can be maximized. The two parts of data stored separately can be read in parallel by processing units. In addition, the Contract Table stores the execution information of hotspot contracts, which is the key data for hotspot optimization.

Execution environment buffer. The buffer is divided into a public part (the State Buffer and the Receipt Buffer) and a private part (the Call_Contract Stack).

The State Buffer holds state data and supports parallel read/write, because there is no dependency between transactions executed by all PUs at the same time. The modified state is written back only after the transaction is successfully executed. If an exception occurs, the modified state is discarded without affecting the original state. In addition, the state of dependent transactions is kept for a period of time so that subsequent transactions are able to access it directly. Reuse of the latest state in the State Buffer effectively reduces redundant accesses to off-chip memory. The Receipt Buffer holds the receipt generated at the end of the transaction execution.

The Call_Contract Stack is private to each processing unit and holds information about each smart contract call [4], which is the key data for executing transactions, including the caller, contract bytecode, input data, and so on. The contract bytecode dominates the loading overhead and is reused when executing redundant transactions.

In-core cache. The core interacts with the external memory through the interface, and the obtained instructions and data are stored in the cache. The MEM holds intermediate results and can be read and written directly by the corresponding instructions. The maximum depth of the operand stack is 1024, and each element is 256 bits. The Constants Table holds the operands of the constants

[4]It is specified in EVM that its maximum depth cannot exceed 1024. And there are four types of bytecodes for calling other smart contracts: CALL, CALLCODE, DELEGATECALL, STATILCCALL.

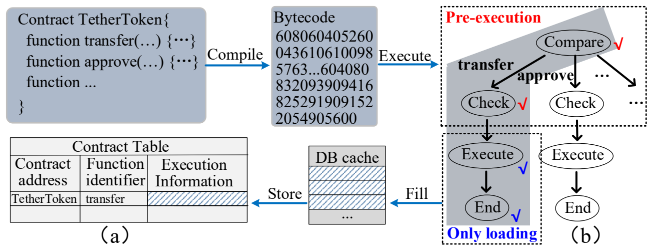


Figure 10: TetherToken is one of the hottest cryptocurrencies in Ethereum. (a) Contract Table. (b) Pre-execution of chunks (✓) and bytecode loading optimization (✓).

instruction for subsequent code optimization of the hotspot contract (see section 3.4.3).

3.4 Optimization of hotspot contracts

There is value in optimizing for frequently executed hotspot contracts. Typically, smart contracts deployed on the blockchain are not allowed to be modified again, so the optimization results are always valid for the lifetime of the contract. Since hotspot contracts are dynamically changing, we design a common approach for all smart contracts: bytecode and data level optimization based on tracking the execution path of hotspot contracts. To avoid competing with transaction execution for limited compute resources, we perform optimization within the block interval.

3.4.1 Collecting Executive Information. When filling the DB cache, there will not be a cache line that holds only one instruction (which is discarded), because fetching a single instruction from the DB cache is considered to be inefficient [19]. To keep a complete track of the execution path of a hotspot contract, we divide a small extra space in the DB cache to record single instructions. These instructions will not be hit during execution, so only the addresses need to be stored.

As shown in Fig. 10 (a), the execution path of hotspot contracts is persisted to the Contract Table. Only transactions that call the same smart contract and have the same entry function have almost completely overlapping execution paths, so we use the contract address and function identifier as labels.

3.4.2 Bytecode Chunking. Based on the analysis of the execution path, the bytecode is divided into chunks according to the execution logic (Fig. 10 (b)): (1) Compare: compare the function identifier to determine the entry function; (2) Check: check *CallValue*; (3) Execute: parse the incoming parameters and execute the function body; (4) End: return and end.

Pre-execution of partial chunks. As described in the three-stage model of transactions (Fig. 4), the dissemination stage is always ongoing. Before receiving a new block, the node has already collected the vast majority of transactions (91.45% - 98.15%) [12]. That is, there is a sufficient interval between when the transactions are heard and when they are executed. The execution of the Compare chunk and the Check chunk depends only on transaction attributes (*To*, *Input Data*, *CallValue*), which are fully known before execution. Therefore, these chunks are pre-executed in the interval.

Optimization of bytecode loading. The *To* and *Input Data* fields mark which bytecode chunks will be executed. Other chunks outside the execution path will not be loaded. Taking TetherToken as

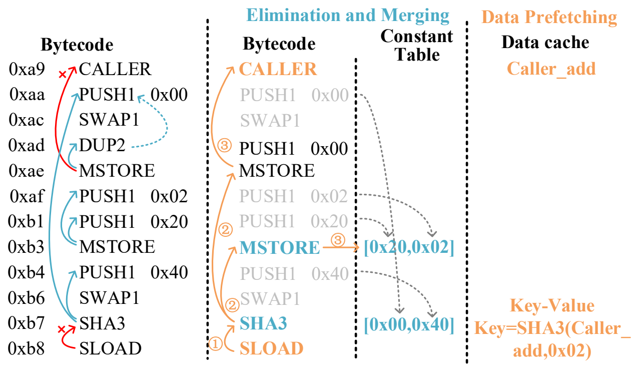


Figure 11: This is a snippet of TetherToken bytecode. Instruction elimination and merging (⟶), data prefetching (⟶)

an example, after chunking and pre-execution optimization, the bytecode loaded when executing the transfer function is only 8.2% of the original.

3.4.3 Instruction Elimination and Merging. Bytecode execution on a stack-based accelerator is constrained by access to operands in the stack. A large number of stack instructions are used to adjust the elements in the stack to provide the correct operands for computation, access, control, and other instructions. For example, based on a statistical analysis of 145 types of smart contract instructions, stack instructions account for 62.54%. We attempt to alleviate this limitation by separating a portion of the operands from the stack.

The most common stack instruction is the PUSH instruction (26% of all instructions), which pushes a constant onto the stack, such as an instruction address, a memory address, an offset, and so on. The DUP (20.32%) and SWAP (11.39%) instructions are used to copy and swap elements on the stack. This large number of stack instructions creates a high probability that the operands in the stack are constants. Therefore, the execution of most instructions is completely known, since their operands are fixed constants. As shown in Fig. 11, we determine whether the operand of the instruction is a constant by backtracking. If it is, we call them constant instructions (e.g., 0xb3 MSTORE and 0xb7 SHA3). Their constant operands are separated and stored in the Constant Table. In the instruction pipeline, the operands for these constant instructions are taken directly from the Constant Table instead of from the stack. This eliminates some of the redundant stack instructions and shortens the bytecode length. More importantly, the dependencies between instructions are greatly simplified, providing more opportunities for instruction parallelism. For example, the 0xb3 MSTORE and the 0xb7 SHA3 no longer form a dependency and can be executed in parallel.

3.4.4 Data Prefetching. We try to speed up the external access instructions located in the execution path by prefetching data. As shown in Table 3, these instructions are divided into fixed access instructions and dynamic access instructions (state query instructions and SLOAD).

Fixed access instructions obtain specific attributes of a transaction without operands, such as CALLER, CALLVALUE, etc. The access logic of these instructions is completely fixed and enables 100% data prefetching before execution.

Architectural Components		Size or Number	Area (mm^2)
Core	Instruction cache	16KB	0.227
	Data cache	64KB	0.547
	MEM	128KB	2.238
	Stack	32KB	0.337
	Gas	32B	0.013
	DB cache	234KB	3.006
	Exectution unit	N/A	0.916
	Else	N/A	0.097
Processing Unit	Core	1	7.381
	Call_Contract Stack	417KB	4.785
Transaction Processor	Processing Unit	4	48.644
	Receipt Buffer	512KB	5.483
	State Buffer	2MB	25.473
	Total	N/A	79.623

Table 5: Key design parameters and area breakdown.

Dynamic access instructions get value in State by key (operand). If they are treated as constant instructions by instruction elimination and merging, the data they access is deterministic and thus prefetched. For non-constant dynamic access instructions, we determine the origin of the operands by backtracking. This class of instructions also enables data prefetching if the operands can be obtained by performing arithmetic logic operations on fixed values (including constants and transaction attributes, which are invariant during execution). By taking three steps back, the operands of the SLOAD instruction in Fig. 11 are obtained by hashing the constant and the address of the caller.

The prefetched data is stored in the data cache in the core before execution, thus reducing the access overhead during execution.

4 EVALUATION

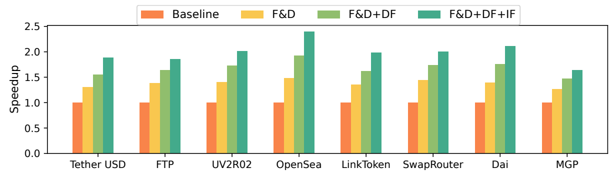
4.1 Methodology

We have implemented MTPU using Chisel [7] and converted it to Verilog RTL. Table 5 lists the parameters of the main architectures. The Synopsys Design Compiler with SMIC 45nm technology is used to synthesize the design. The performance of the off-chip main memory is simulated using Ramulator. The average power consumption is evaluated using Synopsys PrimeTime PX. The average on-chip power consumption of an MTPU configured with four PUs was measured to be 8.648W at a 300MHZ clock frequency.

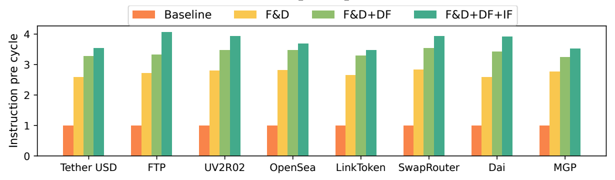
We use real blocks as our dataset, a randomly selected fraction of blocks generated in the most recent year (2020.9–2022.9) of the Ethernet mainnet. We filter out the 8 most frequently invoked smart contracts from them (removing of some smart contracts with undisclosed source code). Table 6 shows the analysis results of the bytecodes of these smart contracts. We intend to evaluate the following three aspects:

(1) We test the benefits of instruction-level parallelism. We use a single PU without any parallelism as the baseline. For each of these smart contracts, we build transactions that call different entry functions and run through all the execution paths of that smart contract as much as possible. The final speedups obtained are averaged.

(2) Combined with the underlying hardware, we evaluate our spatial-temporal scheduling algorithm. The superiority of this algorithm over common scheduling algorithms lies mainly in asynchronous execution and redundancy optimization. We test the performance improvements for different proportions of dependent transactions and different numbers of PUs.



(a) Speedup



(b) Instruction pre cycle

Figure 12: The performance from F&D, DF and IF.

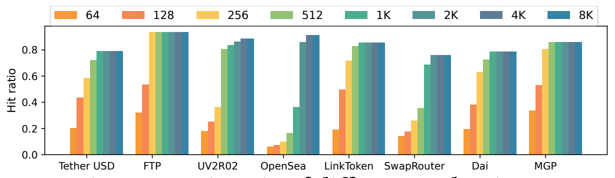


Figure 13: Hit ratio of different cache sizes.

(3) We evaluate the acceleration benefit from the optimization of hotspot contracts. We mark TOP8 smart contracts as hotspot contracts. The performance is compared with no hotspot contract optimization.

4.2 Instruction Level Optimization

We first evaluated the performance of individual PU. The optimizations for it include: (1) adding a fill unit and a DB cache outside the critical path (F&D); (2) using data forwarding to eliminate RAW dependencies between reconfigurable units (DF); (3) performing pattern detection and instruction folding (IF). We assume that the hit rate of the DB cache is 100%. Fig. 12 shows the upper bound of the performance improvement from these optimizations. With a stack-based instruction set, there are many redundant operations on the stack. The experimental results show that the DB cache is able to provide better performance improvement. Efficient ILP can be further achieved by data forwarding and instruction folding.

The hit rate of cache is very low (3%-10%) when actually processing a single transaction, because the main application of smart contracts in Ethereum is currently cryptocurrency, with less circular logic. Nevertheless, when processing new blocks, there is a lot of redundancy in a batch of transactions. The benefits of ILP will be further amplified by reusing the DB cache. Therefore, we test the performance of a batch of transactions invoking the same smart contract with different sizes of cache, and the hit rate is shown in the Fig. 13. When the cache is small, the frequent replacement of cached content limits the hit rate. As the size of the cache gradually increases, the hit rate gradually rises, but the magnitude is affected by the code size of the smart contract and the proportion of transactions calling different entry functions. In the end, the hit rate is basically stable at the cache size of 2K entries (around 85%). For a sufficiently large cache, it is mainly due to the cold misses. Further, we test the execution performance of each smart contract

Smart Contract	Arithmetic	Logic	SHA	Fixed access	State query	Memory	Storage	Branch	Stack	Control	Context switching
Tether USD	8.52%	11.29%	0.90%	1.97%	0.13%	5.62%	2.62%	5.11%	61.42%	2.30%	0.13%
UniswapV2Router02	9.19%	7.55%	0.08%	4.77%	0.31%	7.09%	0.52%	4.82%	63.30%	2.05%	0.32%
FiatTokenProxy	6.62%	8.61%	0.26%	4.77%	0.13%	5.43%	0.66%	9.01%	61.85%	2.38%	0.26%
OpenSea	13.73%	6.30%	0.08%	2.35%	0.04%	8.53%	0.35%	3.73%	64.15%	0.64%	0.09%
LinkToken	9.25%	8.11%	1.53%	2.67%	0.10%	8.16%	0.94%	4.85%	62.59%	1.68%	0.10%
SwapRouter	8.40%	7.38%	0.03%	3.24%	0.24%	7.47%	0.05%	7.10%	63.82%	1.98%	0.29%
Dai	9.39%	7.55%	1.31%	2.53%	0.00%	8.50%	1.13%	3.95%	64.05%	1.57%	0.03%
MainchainGatewayProxy	5.94%	14.09%	0.29%	3.93%	0.00%	3.74%	3.36%	7.96%	56.76%	3.84%	0.10%
Avg	8.88%	8.86%	0.56%	3.28%	0.12%	6.82%	1.20%	5.81%	62.24%	2.06%	0.16%

Table 6: Instruction breakdown of TOP8 smart contracts.

Smart	Upper Limit		2K		Compare	
Contract	IPC	Speedup	IPC	Speedup	IPC	Speedup
Tether USD	3.53	1.88	2.73	1.67	-22.71%	-11.18%
FTP	4.06	1.85	3.50	1.69	-13.91%	-8.67%
UV2R02	3.94	2.02	3.57	1.96	-9.27%	-3.03%
OpenSea	3.70	2.40	3.23	2.23	-12.69%	-7.14%
LinkToken	3.47	1.98	2.91	1.80	-16.22%	-8.88%
SwapRouter	3.94	2.00	2.68	1.69	-31.90%	-15.25%
Dai	3.91	2.11	2.90	1.82	-25.78%	-13.89%
MGP	3.53	1.64	2.87	1.53	-18.62%	-6.67%
Avg	3.76	1.99	3.05	1.80	-18.99%	-9.36%

Table 7: Performance of a single transaction processor when the cache entry is 2K.

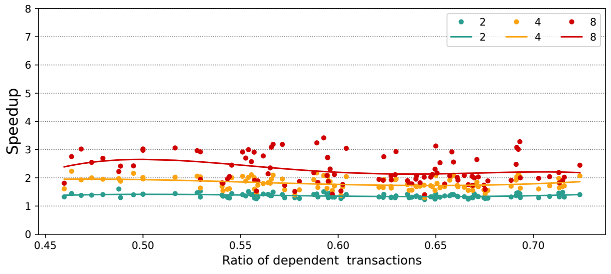
transaction with a cache size of 2K, as shown in Table 7. Compared with the upper limit shown in Fig. 12, the performance loss due to cache miss is 9.34%. Finally, the implementation of ILP can achieve a speedup of 1.80× for the execution of smart contract transactions.

4.3 Transaction Parallelism

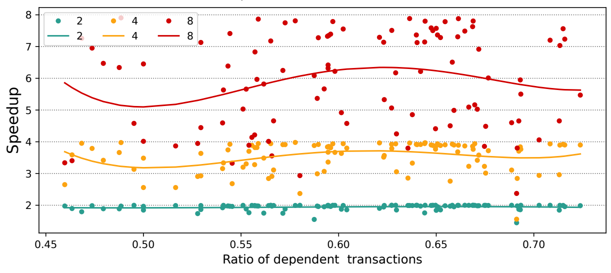
In the case of transaction processors with multiple PUs, we evaluate the performance gains from transaction parallelization. We select blocks with different dependency ratios and execute these transactions in parallel. Using the sequential execution of a PU as a baseline, we compare four approaches: (1) Synchronous execution of transactions; (2) Spatial-temporal scheduling; (3) Spatial-temporal scheduling + Redundancy optimization; (4) Spatial-temporal scheduling + Redundancy optimization + Hotspot optimization, and the experimental results are shown in Fig. 14 , Fig. 15, Fig. 16.

The experimental data has fluctuations because the parallelism benefit is also affected by other factors, such as the longest path in the dependency graph, the proportion of smart contract transactions, etc. The overall trend (as shown in the fitted curves) still illustrates that the spatial-temporal scheduling we designed is able to utilize computational resources more fully and achieve higher speedup benefits than the synchronous execution of transactions.

Further, we compare the benefits from redundancy optimization, as shown in Fig. 16(a). Compared with Fig. 14(a), reuse of context data and DB cache can bring performance improvement even on a single PU. In multiple PUs, more parallel opportunities are found. We mark the TOP8 smart contracts as hotspot contracts and implement continuous acceleration for all hotspot contract transactions to further improve performance, as shown in Fig. 16(b).

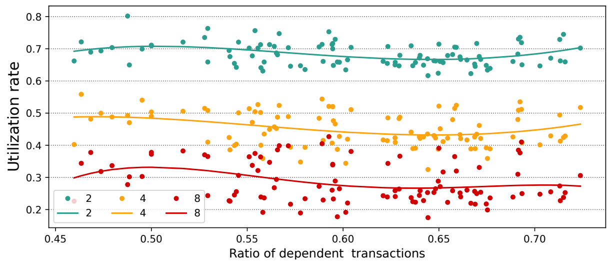


(a) Synchronous execution

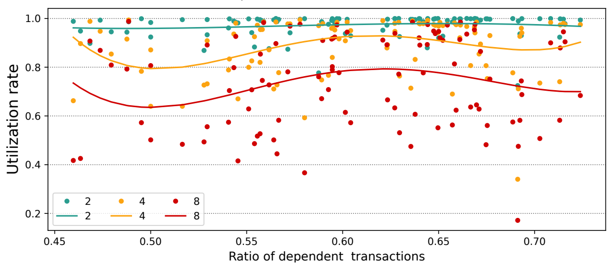


(b) Spatial-temporal scheduling

Figure 14: Speedup with different ratios of dependent transactions.

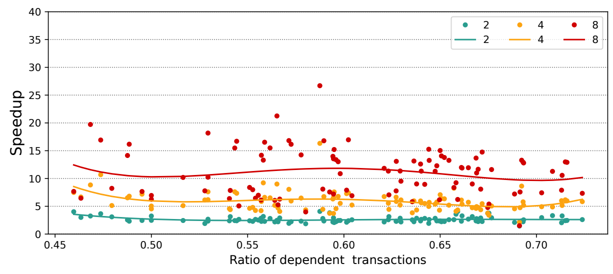


(a) Synchronous execution

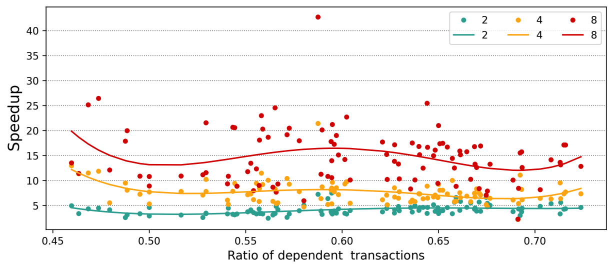


(b) Spatial-temporal scheduling

Figure 15: Resource utilization with different ratios of dependent transactions.



(a) Spatial-temporal scheduling + Redundancy optimization



(b) Spatial-temporal scheduling + Redundancy optimization + Hotspot optimization

Figure 16: Speedup with different ratios of dependent transactions.

	100%	80%	60%	40%	20%	0%
BPU	12.82x	3.40x	2.23x	1.63x	1.33x	1x
MTPU	2.79x	2.14x	2.16x	2.05x	2.00x	1.71x

Table 8: Acceleration of BPU and MTPU for different proportions of ERC20 transactions (single core)

	100%	80%	60%	40%	20%	0%
BPU	3.51x	3.80x	4.69x	4.95x	5.76x	7.4x
MTPU	8.68x	9.36x	9.87x	12.01x	12.08x	15.25x

Table 9: Acceleration of BPU and MTPU for different proportions of dependent transactions (quad-core)

4.4 Comparison with Other Works

BPU is the first dedicated accelerator for accelerating smart contracts [31]. There are two engines in BPU: the GSC engine for executing general smart contracts and the App engine designed specifically for executing the most common ERC20 smart contracts. We use a single GSC engine as a baseline. We first compare the acceleration performance of the BPU and MTPU in the single-core case when executing blocks with different proportions of ERC20 (Table 8). For the ERC20 smart contract, the acceleration achieved by our design is relatively low. On the one hand, our architecture is more general and aims to accelerate all smart contracts. As the ERC20 ratio decreases, the performance of BPU decreases very significantly, while the performance of MTPU is more stable. On the other hand, the strength of our architecture lies in exploiting extensive parallelism. We randomly select blocks with different dependency transaction ratios and compare the performance of both in the quad-core case. As shown in the Table 9, MTPU is able to achieve more parallel benefits through fine-grained scheduling compared to BPU, and transaction dependencies have less impact on it. Our design increases the area and energy overhead by about 17% and 10%, respectively, compared to BPU due to the additional hardware added to implement multiple layers of parallelism.

5 RELATED WORK

There are many studies dedicated to improving the throughput of blockchain systems, which can be broadly classified into two categories: increasing the frequency of block generation and accelerating transaction execution.

5.1 Increase the Frequency of Block Generation

The following approaches are adopted: (1) Splitting a blockchain into multiple partitions or running multiple parallel chains to generate multiple blocks simultaneously. [27, 32, 36, 43, 48]. (2) Making improvements to the consensus protocol to reduce the consensus time of blocks [11, 13, 22, 24, 34, 42, 46]. (3) Using DAG-based chain structures to organize blocks and increase the block generation rate [1, 29, 41, 44]. (4) Developing off-chain solutions to offload certain workloads [21, 23, 30, 33, 37, 39].

5.2 Parallel Execution of Transactions

Parallelism is the most direct way to accelerate transaction execution. The two ways to achieve parallelism are intra-node and inter-node. For the former, methods for analyzing dependencies between transactions fall into two categories: static analysis is used to obtain the read/write sets of transactions to observe conflicts between transactions [8, 9]; dynamic conflict detection speculatively executes transactions in parallel and uses software transaction memory techniques to capture the dependencies between transactions [4, 5, 14, 25, 38, 49]. For the latter, a trust community is built. Nodes within the community share the results of transaction execution to avoid duplicate execution. However, this often requires additional security assumptions and specialized hardware [8, 16, 17].

5.3 Hardware Accelerator

There is currently very little work on this part. BPU is the first hardware accelerator for accelerating blockchain transaction processing [31]. By analyzing the execution path of transactions, BPU uses pipelining techniques to accelerate the execution of smart contracts and application-oriented optimization to design a dedicated data flow for ERC20 contracts, ultimately achieving high performance.

6 CONCLUSIONS

Inefficient transaction execution restricts blockchain throughput. Based on the analysis of transaction execution characteristics, we propose an algorithm and architecture co-design for accelerating smart contract execution, including a spatial-temporal scheduling algorithm, multi-layer parallelism, de-redundancy, and hotspot contract optimization. Experiments show that our design is able to achieve a further acceleration of 3.53× − 16.19× compared to existing schemes.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their constructive comments. This work was supported in part by the National Key Research and Development Program of China (No. 2021YFA1000600 and 2021YFB0300300), the Programs of National Natural Science Foundation of China (No. 62072165), and the Fundamental Research Funds for the Central Universities.

REFERENCES

[1] Igor D. Alvarenga, Gustavo Franco Camilo, Lucas Airam C. de Souza, and Otto Carlos M. B. Duarte. 2021. DAGSec: A hybrid distributed ledger architecture for the secure management of the Internet of Things. In *2021 IEEE International Conference on Blockchain, Blockchain 2021, Melbourne, Australia, December 6-8, 2021*. IEEE, 266–271. https://doi.org/10.1109/Blockchain53845.2021.00043

[2] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*. IEEE, 1337–1347. https://doi.org/10.1109/ICDCS.2019.00134

[3] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. Permissioned Blockchains: Properties, Techniques and Applications. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2813–2820. https://doi.org/10.1145/3448016.3457539

[4] Parwat Singh Anjana, Hagit Attiya, Sweta Kumari, Sathya Peri, and Archit Somani. 2020. Efficient Concurrent Execution of Smart Contracts in Blockchains Using Object-Based Transactional Memory. In *Networked Systems - 8th International Conference, NETYS 2020, Marrakech, Morocco, June 3-5, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12129)*. Springer, 77–93. https://doi.org/10.1007/978-3-030-67087-0_6

[5] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. 2019. An Efficient Framework for Optimistic Concurrent Execution of Smart Contracts. In *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2019, Pavia, Italy, February 13-15, 2019*. IEEE, 83–92. https://doi.org/10.1109/EMPDP.2019.8671637

[6] L. M. Bach, Branko Mihaljevic, and Mario Zagar. 2018. Comparative analysis of blockchain consensus algorithms. In *41st International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2018, Opatija, Croatia, May 21-25, 2018*. IEEE, 1545–1550. https://doi.org/10.23919/MIPRO.2018.8400278

[7] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. 2012. Chisel: constructing hardware in a Scala embedded language. In *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*. ACM, 1216–1225. https://doi.org/10.1145/2228360.2228584

[8] Shrey Baheti, Parwat Singh Anjana, Sathya Peri, and Yogesh Simmhan. 2022. DiPETrans: A framework for distributed parallel execution of transactions of blocks in blockchains. *Concurr. Comput. Pract. Exp.* 34, 10 (2022). https://doi.org/10.1002/cpe.6804

[9] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. 2021. A theory of transaction parallelism in blockchains. *Log. Methods Comput. Sci.* 17, 4 (2021). https://doi.org/10.46298/lmcs-17(4:10)2021

[10] Pronaya Bhattacharya, Sudeep Tanwar, Umesh Bodkhe, Sudhanshu Tyagi, and Neeraj Kumar. 2021. BinDaaS: Blockchain-Based Deep-Learning as-a-Service in Healthcare 4.0 Applications. *IEEE Trans. Netw. Sci. Eng.* 8, 2 (2021), 1242–1255. https://doi.org/10.1109/TNSE.2019.2961932

[11] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X. Zhang. 2020. Combining GHOST and Casper. *CoRR* abs/2003.03052 (2020). arXiv:2003.03052 https://arxiv.org/abs/2003.03052

[12] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. 2021. Forerunner: Constraint-based Speculative Transaction Execution for Ethereum. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 570–587. https://doi.org/10.1145/3477132.3483564

[13] Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. 2022. DAMYSUS: streamlined BFT consensus leveraging trusted components. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 1–16. https://doi.org/10.1145/3492321.3519568

[14] Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2020. Adding concurrency to smart contracts. *Distributed Comput.* 33, 3-4 (2020), 209–225. https://doi.org/10.1007/s00446-019-00357-z

[15] Etherscan. 2022. Etherscan. https://etherscan.io/

[16] Min Fang, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2021. High-Performance Smart Contracts Concurrent Execution for Permissioned Blockchain Using SGX. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 1907–1912. https://doi.org/10.1109/ICDE51399.2021.00175

[17] Min Fang, Xinna Zhou, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2022. SEFrame: An SGX-enhanced Smart Contract Execution Framework for Permissioned Blockchain. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 3166–3169. https://doi.org/10.1109/ICDE53745.2022.00289

[18] Md. Sadek Ferdous, Mohammad Jabed Morshed Chowdhury, and Mohammad Ashraful Hoque. 2021. A survey of consensus algorithms in public blockchain systems for crypto-currencies. *J. Netw. Comput. Appl.* 182 (2021), 103035. https://doi.org/10.1016/j.jnca.2021.103035

[19] Manoj Franklin and Mark Smotherman. 1994. A fill-unit approach to multiple instruction issue. In *Proceedings of the 27th Annual International Symposium on Microarchitecture, MICRO 1994, San Jose, California, USA, November 30 - December 2, 1994*. ACM / IEEE Computer Society, 162–171. https://doi.org/10.1109/MICRO.1994.717455

[20] Xiang Fu, Huaimin Wang, and Peichang Shi. 2021. A survey of Blockchain consensus algorithms: mechanism, design and applications. *Sci. China Inf. Sci.* 64, 2 (2021). https://doi.org/10.1007/s11432-019-2790-1

[21] Zhonghui Ge, Yi Zhang, Yu Long, and Dawu Gu. 2022. Shaduf: Non-Cycle Payment Channel Rebalancing. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/auto-draft-254/

[22] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP 2017, Shanghai, China, October 28-31, 2017*. ACM, 51–68. https://doi.org/10.1145/3132747.3132757

[23] Zicong Hong, Song Guo, Rui Zhang, Peng Li, Yufeng Zhan, and Wuhui Chen. 2022. Cycle: Sustainable Off-Chain Payment Channel Network with Asynchronous Rebalancing. In *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2022, Baltimore, MD, USA, June 27-30, 2022*. IEEE, 41–53. https://doi.org/10.1109/DSN53405.2022.00017

[24] Ruomu Hou, Haifeng Yu, and Prateek Saxena. 2022. Using Throughput-Centric Byzantine Broadcast to Tolerate Malicious Majority in Blockchains. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 1263–1280. https://doi.org/10.1109/SP46214.2022.9833617

[25] Cheqing Jin, Shuaifeng Pang, Xiaodong Qi, Zhao Zhang, and Aoying Zhou. 2022. A High Performance Concurrency Protocol for Smart Contracts of Permissioned Blockchain. *IEEE Trans. Knowl. Data Eng.* 34, 11 (2022), 5070–5083. https://doi.org/10.1109/TKDE.2021.3059959

[26] Aggelos Kiayias, Markulf Kohlweiss, and Amirreza Sarencheh. 2022. PEReDi: Privacy-Enhanced, Regulated and Distributed Central Bank Digital Currencies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. ACM, 1739–1752. https://doi.org/10.1145/3548606.3560707

[27] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 583–598. https://doi.org/10.1109/SP.2018.000-5

[28] Wang Fat Lau, Dennis Y. W. Liu, and Man Ho Au. 2021. Blockchain-Based Supply Chain System for Traceability, Regulation and Anti-Counterfeiting. In *2021 IEEE International Conference on Blockchain, Blockchain 2021, Melbourne, Australia, December 6-8, 2021*. IEEE, 82–89. https://doi.org/10.1109/Blockchain53845.2021.00022

[29] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. 2020. A Decentralized Blockchain with High Throughput and Fast Confirmation. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. USENIX Association, 515–528. https://www.usenix.org/conference/atc20/presentation/li-chenxing

[30] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter R. Pietzuch. 2019. Teechain: a secure payment network with asynchronous blockchain access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. ACM, 63–79. https://doi.org/10.1145/3341301.3359627

[31] Tao Lu and Lu Peng. 2020. BPU: A Blockchain Processing Unit for Accelerated Smart Contract Execution. In *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*. IEEE, 1–6. https://doi.org/10.1109/DAC18072.2020.9218512

[32] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A Secure Sharding Protocol For Open Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 17–30. https://doi.org/10.1145/2976749.2978389

[33] Patrick McCorry, Surya Bakshi, Iddo Bentov, Sarah Meiklejohn, and Andrew Miller. 2019. Pisa: Arbitration Outsourcing for State Channels. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019*. ACM, 16–30. https://doi.org/10.1145/3318041.3355461

[34] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 31–42. https://doi.org/10.1145/2976749.2978399

[35] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review* (2008), 21260. https://bitcoin.org/bitcoin.pdf

[36] Lan N. Nguyen, Truc D. T. Nguyen, Thang N. Dinh, and My T. Thai. 2019. OptChain: Optimal Transactions Placement for Scalable Blockchain Sharding. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*. IEEE, 525–535. https://doi.org/10.1109/

ICDCS.2019.00059

[37] Nikolaos Papadis and Leandros Tassiulas. 2022. Payment Channel Networks: Single-Hop Scheduling for Throughput Maximization. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications, London, United Kingdom, May 2-5, 2022*. IEEE, 900–909. https://doi.org/10.1109/INFOCOM48880.2022.9796862

[38] Vikram Saraph and Maurice Herlihy. 2019. An Empirical Study of Speculative Concurrency in Ethereum Smart Contracts. In *International Conference on Blockchain Economics, Security and Protocols, Tokenomics 2019, May 6-7, 2019, Paris, France (OASIcs, Vol. 71)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:15. https://doi.org/10.4230/OASIcs.Tokenomics.2019.4

[39] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Kathleen Ruan, Parimarjan Negi, Lei Yang, Radhika Mittal, Giulia Fanti, and Mohammad Alizadeh. 2020. High Throughput Cryptocurrency Routing in Payment Channel Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. USENIX Association, 777–796. https://www.usenix.org/conference/nsdi20/presentation/sivaraman

[40] Solidity. 2022. Solidity. https://docs.soliditylang.org/en/v0.8.17/

[41] Yonatan Sompolinsky and Aviv Zohar. 2015. Secure High-Rate Transaction Processing in Bitcoin. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8975)*. Springer, 507–527. https://doi.org/10.1007/978-3-662-47854-7_32

[42] Xiao Sui, Sisi Duan, and Haibin Zhang. 2022. Marlin: Two-Phase BFT with Linearity. In *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2022, Baltimore, MD, USA, June 27-30, 2022*. IEEE, 54–66. https://doi.org/10.1109/DSN53405.2022.00018

[43] Jiaping Wang and Hao Wang. 2019. Monoxide: Scale out Blockchains with Asynchronous Consensus Zones. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*. USENIX Association, 95–112. https://www.usenix.org/conference/nsdi19/presentation/wang-jiaping

[44] Tianyu Wang, Qian Wang, Zhaoyan Shen, Zhiping Jia, and Zili Shao. 2022. Understanding Characteristics and System Implications of DAG-Based Blockchain in IoT Environments. *IEEE Internet Things J.* 9, 16 (2022), 14478–14489. https://doi.org/10.1109/JIOT.2021.3108527

[45] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32. https://files.gitter.im/ethereum/yellowpaper/VIyt/Paper.pdf

[46] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. 2022. DispersedLedger: High-Throughput Byzantine Consensus on Variable Bandwidth Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, Amar Phanishayee and Vyas Sekar (Eds.). USENIX Association, 493–512. https://www.usenix.org/conference/nsdi22/presentation/yang

[47] Murat Yilmaz, Tuna Hacaloglu, and Paul M. Clarke. 2022. Examining the Use of Non-fungible Tokens (NFTs) as a Trading Mechanism for the Metaverse. In *Systems, Software and Services Process Improvement - 29th European Conference, EuroSPI 2022, Salzburg, Austria, August 31 - September 2, 2022, Proceedings (Communications in Computer and Information Science, Vol. 1646)*. Springer, 18–28. https://doi.org/10.1007/978-3-031-15559-8_2

[48] Haifeng Yu, Ivica Nikolic, Ruomu Hou, and Prateek Saxena. 2020. OHIE: Blockchain Scaling Made Simple. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 90–105. https://doi.org/10.1109/SP40000.2020.00008

[49] An Zhang and Kunlong Zhang. 2018. Enabling Concurrency on Smart Contracts Using Multiversion Ordering. In *Web and Big Data - Second International Joint Conference, APWeb-WAIM 2018, Macau, China, July 23-25, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10988)*. Springer, 425–439. https://doi.org/10.1007/978-3-319-96893-3_32

[50] Cheng Zhang, Yang Xu, Yupeng Hu, Jiajing Wu, Ju Ren, and Yaoxue Zhang. 2022. A Blockchain-Based Multi-Cloud Storage Data Auditing Scheme to Locate Faults. *IEEE Trans. Cloud Comput.* 10, 4 (2022), 2252–2263. https://doi.org/10.1109/TCC.2021.3057771