PaVM: A Parallel Virtual Machine for Smart Contract Execution and Validation

Yaozheng Fang, Zhiyuan Zhou, Surong Dai, Jinni Yang, Hui Zhang, *Senior Member, IEEE* Ye Lu*

***Abstract*—The performance bottleneck of blockchain has shifted from consensus to serial smart contract execution in transaction validation. Previous works predominantly focus on inter-contract parallel execution, but they fail to address the inherent limitations of each smart contract execution performance. In this paper, we propose PaVM, the first smart contract virtual machine that supports both inter-contract and intra-contract parallel execution to accelerate the validation process. PaVM consists of (1) key instructions for precisely recording entire runtime information at the instruction level, (2) a runtime system with a re-designed machine state and thread management to facilitate parallel execution, and (3) a read/write-operation-based receipt generation method to ensure both the correctness of operations and the consistency of blockchain data. We evaluate PaVM on the Ethereum testnet, demonstrating that it can outperform the mainstream blockchain client Geth. Our evaluation results reveal that PaVM speeds up overall validation performance by 33.4×, and enhances validation throughput by up to 46×.**

***Index Terms*—Smart contract, Virtual machine, Architectural design, Blockchain.**

I. Introduction

Mainstream blockchain techniques usually necessitate frequent transaction validation to guarantee data consistency among massive blockchain nodes [1], [2]. A transaction includes a function invocation of a smart contract. The validation of transactions refers to the execution of a batch of smart contracts [3]–[5]. Previous studies have illustrated tha the performance bottleneck of blockchain has shifted from consensus to serial contract execution during transaction validation [6], [7]. For example, modern Practical Byzantine Fault Tolerance (PBFT) consensus achieves more than 3,000 transactions per second (TPS), but the serial contract execution only nearly 100 TPS [6] and 20 TPS [8] on average in private and public Ethereum, respectively. The reason is that the serial execution pattern has limited the transaction validation performance. Thus the transaction validation indeed suffers from high latency.

Several approaches have been proposed to speed up transaction validation by enabling parallel execution among contracts [6], [7], [9]–[11], as illustrated in Fig. 1. Smart contracts in transactions can be executed in parallel with each other. However, these approaches exhibit limitations in their ability to achieve thorough acceleration, because they only operate in parallel at the transaction level (we call it as inter-contract) rather than the function level (we call it as intra-contract). The function execution within a smart contract in the execution environment, such as Ethereum Virtual Machine, is still serial.

In fact, the validation performance can be further improved by enabling intra-contract execution in parallel within a single contract. Furthermore, previous attempts to optimize inter-contract execution [7] have frequently resulted in inconsistent blockchain state, and the receipt generation method based on uncertain transaction validation orders can also lead to state synchronization failures. Therefore, based on the above we can draw three observations as follows:

Firstly, although modern general CPU architecture provides powerful parallel processing capability [12], [13], which presents an opportunity for accelerating contract execution, the design of a typical smart contract virtual machine is still based on a single thread [14], [15]. Additionally, incorporating multiple threads mechanism directly into the VM may disrupt the correct read/write order. For instance, the random data read/write operations on storage spaces lead to inconsistent smart contract outputs and blockchain states across different blockchain nodes [16].

Secondly, traditional data conflict detection mechanisms are imprecise because they rely on incomplete runtime information. The data outside the contract storage, such as data R/W logs, cannot be detected; only dirty reads and dirty writes inside the contract storage can be monitored [17]. Since the data outside the contract storage is temporarily stored during contract runtime [18], [19], the entire runtime information cannot be transferred into the validation. This renders the detection mechanism unable to take advantage of the runtime context to address the conflicts precisely. In addition, detection methods based on static semantic analysis assume that all data operations in branches such as `if` statements will occur [20]. Such a pessimistic assumption expands the scope of conflicts and degrades overall performance.

Thirdly, parallel transaction validation is incompatible with the transaction receipt generation of public blockchains, which

This work is partially supported by the National Natural Science Foundation (No. 62372253, No. 62002175), the Natural Science Foundation of Tianjin Fund (No. 23JCYBJC00010), the CCF-Baidu Open Fund (No. CCF-Baidu202310), the CCF-Huawei Populus Grove Fund (No. CCF-HuaweiTC2022005), and the Open Project Fund of State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences (No. CARCHB202016), and the Open Project Foundation of Information Security Evaluation Center of Civil Aviation, Civil Aviation University of China (No. ISECCA-202102).

Y. Fang, S. Dai, J. Yang, and Y.Lu are with the College of Computer Science, Nankai University, China, and with the Tianjin Key Laboratory of Network and Data Science Technology.

Y. Lu is also with the College of Cyber Science, Nankai University, China, and the State Key Lab of Processors, ICT, CAS, and the Key Laboratory of Data and Intelligent System Security, Ministry of Education, China (DISSec), and the Information Security Evaluation Center of Civil Aviation, Civil Aviation University of China, Tianjin, China.

Z. Zhou and H. Zhang are with the Blockchain Platform Division, Ant Group.

Corresponding author: Ye Lu, Email: luye@nankai.edu.cn

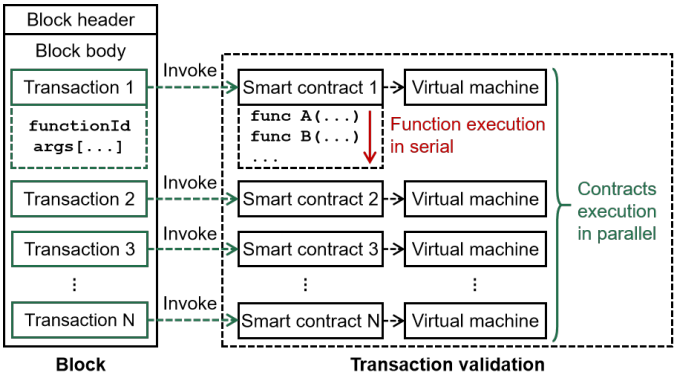


Fig. 1: The parallel transaction validation paradigm.

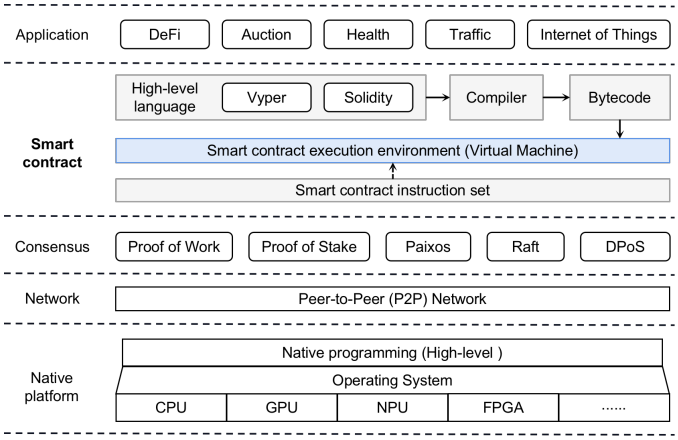


Fig. 2: The blockchain hierarchical architecture.

can result in data synchronization failures between blockchain nodes. Specifically, transaction validation produces transaction receipt in terms of intermediate blockchain state [21], [22]. In parallel validation, because the transaction validation order is random, and the intermediate state remains uncertain, thus resulting in transaction receipts being different between blockchain nodes. These different receipts can cause inconsistencies in block headers among nodes, and lead to both validation and data synchronization failures [23].

To address the aforementioned issues, we propose PaVM, the first smart contract virtual machine that supports both inter-contract and intra-contract parallel execution to accelerate transaction validation. The key idea of PaVM which supports parallel execution is to describe runtime information at a fine-grained level and enhance the runtime system by introducing thread state and management to improve contract execution efficiency. Experimental results highlight that PaVM can improve the overall transaction validation performance by about **33.4**×, compared with Geth, the mainstream client of Ethereum. The implementation of PaVM is built upon the optimized SmartVM [4]. The contributions of this work can be summarized as follows:

- We propose three kinds of instructions that record the entire runtime read/write information at the instruction level precisely. The key interfaces and instructions are also proposed for developers. The throughput of transaction validation can be improved by **25**× on average and **46**× on maximum. The additional recording time burden is only 0.3%, which can be ignored.
- We enhance and extend a runtime system for PaVM, which can realize machine state and thread management to perform parallel execution. The speedup of execution reaches **20**× on average.
- We present a read/write-operation-based receipt generation method in PaVM that can generate determined receipts in terms of intermediate operations to handle random validation. The method can also keep state consistency and is adaptive for parallel validation.
- We implement PaVM as a building block embedded into the Ethereum testnet Geth client, with programming interfaces, instructions, and the runtime system. The comparison experiments are performed by three benchmarks with eight kinds of contracts. Compared with the original Geth, Geth integrated with PaVM improves the transaction validation performance by **33.4**× and achieves 99.8% hardware resources CPU utilization which is improved by nearly **38**×.

II. Background and Motivation

A. Smart contract

The blockchain is a distributed ledger that continuously grows with a series of interconnected blocks. The blocks are securely linked together via cryptographic hashes [24]. As shown in Fig. 2, the blockchain runs on the operating system and hardware. The blockchain is implemented by common high-level programming languages (e.g., Golang). The nodes in blockchain are interconnected via a peer-to-peer network with various communication protocols. The consensus mechanism defines the rule of block generation, realizes block synchronization, and maintains data consistency across multiple blockchain nodes. As a crucial component of the blockchain, the smart contract enables trusted on-chain computations. To make the computations trusted, the computation results of smart contracts are validated by all blockchain nodes. The smart contracts can support many distributed applications such as smart traffics and auction systems. Our PaVM primarily focuses on optimizing the smart contract system to improve the validation and execution performance.

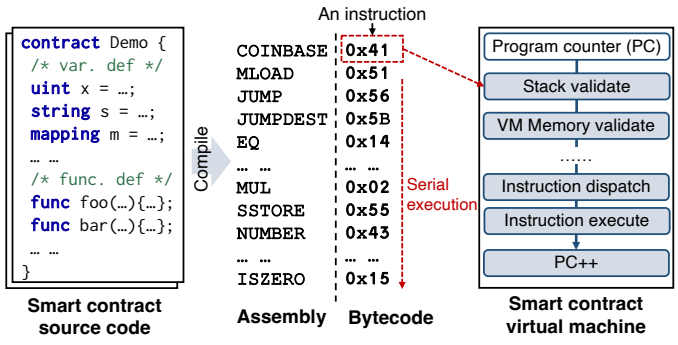


Fig. 3: The contract execution.

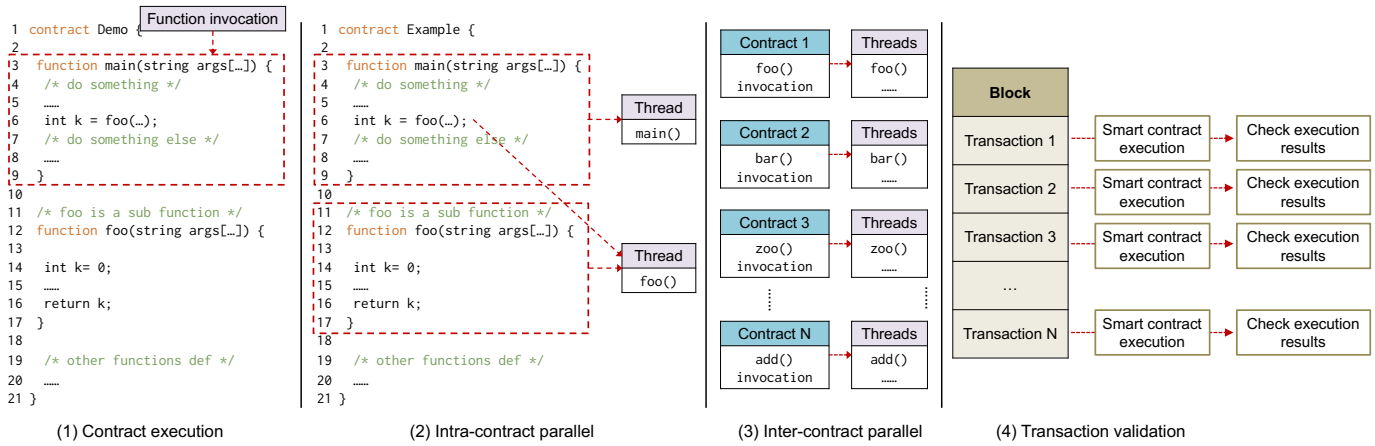


Fig. 4: The preliminary concepts of PaVM.

Smart contract is a piece of executable code stored in the blockchain [25], [26]. The smart contract realizes complex computations by defining state variables and functions [27]. As shown in Fig. 3, the contracts programmed in high-level contract-oriented languages (e.g., Solidity) are compiled into bytecode [28] before being executed in a smart contract virtual machine (SCVM) such as Ethereum Virtual Machine (EVM). The bytecode is composed of several SCVM instructions such as `PUSH` and `ADD`. Traditional SCVMs perform the contracts by executing the bytecode instructions in serial based on a program counter [4]. The instructions are defined in a smart contract instruction set and implemented by native programming languages. The function in a smart contract can be invoked through a blockchain transaction, which typically includes the data of the function identifier and corresponding arguments. Hundreds of transactions can be encapsulated as a block. Once the nodes receive a block, they are obligated to validate the transactions contained within the block. The validation involves executing the functions which are related to call data in smart contracts.

Before introducing the motivation of PaVM, there are several key concepts should be explained as follows:

- *Contract execution.* A smart contract consists of several variables and functions, which can be invoked by users. The logic in the invoked function is executed in smart contract execution environments such as EVM. For example, in Fig. 4(1), the user invokes the `main` function, which is executed in EVM. The function execution process can also be called as contract execution.
- *Intra-contract parallelism.* As shown in Fig. 4(2), the `main` function invokes the `foo` function. These two functions can ideally be executed in parallel when they have no data or control dependencies. The two functions can be run in separate threads. We name this case as intra-contract parallelism.
- *Inter-contract parallelism.* As shown in Fig. 4(3), when multiple smart contracts are invoked and there are no dependencies between each of two invoked contracts, these contracts can be executed in separate threads. We name this case as inter-contract parallelism.
- *Transaction validation.* In the blockchain system, a smart contract invocation is encapsulated into a transaction, and transactions from dozens to hundreds can form a block, as shown in Fig. 4(4). The smart contract invocations in the transactions are re-executed by all blockchain nodes. This process is called as transaction validation.

B. Virtual machine for smart contract

Smart contract virtual machine (SCVM) serves as a virtual execution environment with a rudimentary runtime system designed for the execution of smart contract bytecode. SCVMs typically include two temporary storage spaces designed for storing runtime information: *Stack* and *VM Memory* [29]. During contract execution (see Fig. 3), the SCVM fetches instructions from bytecode by referencing the program counter. The SCVM subsequently validates the status of storage spaces before dispatching and executing the specified instruction.

Compared to traditional multiple-thread virtual machines, such as Java Virtual Machine (JVM) [30], SCVMs are usually more lightweight, compact, and simpler. SCVMs execute the instructions in serial without any support for multiple threads management [4]. Utilizing the multiple thread mechanism directly to improve the execution efficiency of SCVMs can result in random and disordered R/W orders, ultimately leading to inconsistent execution output. The reason is that all the threads can read/write the storage spaces such as *Stack* and *VM Memory* in any order. This motivates us to enhance the runtime system to support the parallel execution of smart contracts.

Moreover, the multiple thread mechanisms in a traditional VM for parallel execution are too heavy for a smart contract system [31]. Those mechanisms impose overhead on all blockchain nodes during validation and execution. All blockchain nodes should launch the SCVM to re-execute all the contracts [32]. It is worth to note that the new research SmartVM [4] on parallel contract execution at the instruction level aims at facilitating high-performance on-chain deep neural network (DNN) computations. This approach implies the *potential of intra-contract parallel mechanism*. Although SmartVM incorporates parallel execution for DNN-oriented instructions, its parallelism is limited to only allowing matrix

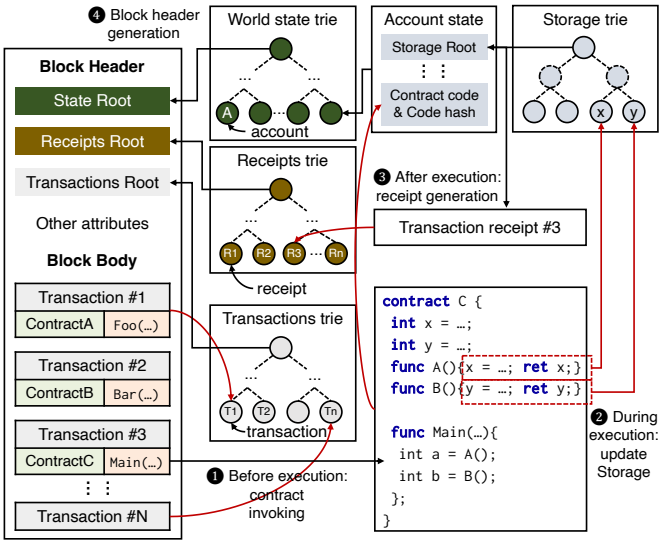


Fig. 5: The overview of smart contract execution.

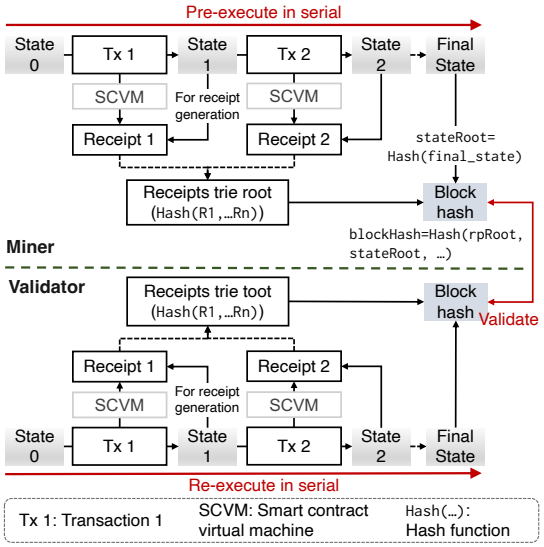


Fig. 6: The miner-validator style for transaction validation.

operations, while other operations still need to be performed in serial. In addition, SmartVM does not manage data read/write operations performed by numerous threads, possibly leading to inconsistent computational results. These observations motivate us to propose the thread management and operation order control methods to achieve data consistency at the function level in PaVM design.

Contract execution changes the blockchain state, which refers to the data stored in the blockchain such as account balance, smart contract code, etc [33]. In the traditional SCVM, validating a transaction involves invoking and executing corresponding contracts in serial.

The execution will update the storage trie, which stores the contracts' variables. Each variable resides in a separate leaf node of the storage trie. As shown in Fig. 5, we take Ethereum as an example. The blockchain state is maintained by three kinds of tries [4]. Transaction details, blockchain accounts, and transaction receipts are stored in the leaf nodes of the transaction trie, world state trie, and receipt trie, respectively. During transaction #3 validation, ❶ the `Main` function of `Contract C` is invoked in the transaction fields. ❷ The subfunctions within `Main`, `A` and `B`, are executed in serial. In fact, they can be executed in parallel, because there are no R/W operation conflicts between variable `x` and variable `y`. Function `A` and function `B` write data to distinct leaf nodes. However, the original serial execution pattern of the two functions within the contract constrains the overall performance. ❸ After the contract execution, the outcomes and operation logs are utilized to generate the transaction receipt. ❹ The headers of three tries are recorded in the block header. Any inconsistency in the data of a trie can cause inconsistent block headers, leading to failures to synchronize blocks and transactions. The serial pattern limitations motivate us to explore the intra-contract parallelism method which conforms to the consistency.

C. Transaction Validation

In fact, based on our analysis, there are minimal instances of data races and dependencies existent in real transactions and smart contracts. We analyzed 10,000 blocks in the Ethereum, each block includes 200 to 300 transactions. The results reveal that more than 90% transactions and smart contracts can be validated and executed in parallel. The transaction validation obeys a miner-validator style [6] as shown in Fig. 6. The miner pre-executes the smart contracts serially to generate a new block with the block hash, and the validator re-executes the contract in serial to generate a block hash, which will be compared with the one in the miner. Traditional validation commonly regards a SCVM as a black box, thus the valuable runtime information cannot be fully utilized for validation and receipt generation.

As shown in Fig. 6, the smart contract in the transaction is executed by the SCVM, with outputs consisting of both the receipt and the updated state. During the execution, the runtime information is temporarily stored in the SCVM and discarded afterward. The runtime data occupies only 0.1% working memory [4]. However, the entire runtime information is transferred to the validation process. The data conflicts can only be detected by observing the dirty read and write operations in the state. It should be noted that the conflicts do not manifest in the state will remain undetectable. The imprecise conflict detection can cause different blockchain states. This motivates us to collect the entire runtime information which records atomic operations with only a little working memory burden.

We should also consider the receipt generation depicted in Fig. 6. With each contract execution, an intermediate state is generated, which subsequently serves as the basis for receipt creation [34]. For instance, the receipt 1 for transaction 1 is generated according to State 1. During the parallel validation, the receipt in the miner and the receipt in the validator are inconsistent, because the random orders of transaction execution cause their intermediate state uncertain. The two

inconsistent receipts lead to two different block hashes, thus failing to synchronize the transaction validation and state among the blockchain nodes. This encourages us to re-design a receipt generation method in PaVM to keep the consistency of receipts.

D. Summary of Limitations & Challenges

We summarize three limitations and challenges faced in contract execution, runtime information, and receipt generation as follows:

The prevailing smart contract virtual machines are primarily designed following a single-thread pattern, which fails to support parallel execution and sufficient utilization of hardware resources. The bytecode instruction executing in serial in conventional SCVM leads to poor performance. As depicted in Fig. 7, the smart contract fails to make full use of multi-core hardware process capabilities. Only one out of the 40 available cores is actively engaged in execution, while the remaining 39 cores have a utilization rate of only 0.5%. The intra-contract parallel method is hard to design due to the requirement for a determinate data read/write order among multiple threads. However, data read/write operations are inherently random when using multi-threading technology in smart contracts. The non-deterministic order of data R/W operations in intra-contract parallelism eventually leads to an inconsistent blockchain state among blockchain nodes.

Coarse-grained and imprecise transaction grouping leads to frequent rollbacks and block synchronization failures in parallel transaction validation. Imprecise transaction grouping results from imprecise detection of data conflicts. Precise conflict detection requires the recording of the complete runtime information such as data read/write operations. The complete information should be recorded at the both database level and the instruction level. Existing runtime information recording can only record the operations from the coarse-grained database level by the corresponding interfaces. SCVM lacks the instruction support to record the instruction operations at the instruction level. For example, in EVM[1], the database-level runtime information such as `storageChange` can be recorded by the data dirty read/write interfaces, while the instruction-level information such as `addLogChange` cannot be recorded during execution. The coarse-grained and imprecise transaction grouping limits parallel transaction validation.

Uncertain intermediate state during receipt generation frequently leads to block inconsistencies and transaction synchronization failures. The intermediate state refers to the blockchain state after smart contract execution. In serial validation, the generation of receipts from these intermediate states aims to ensure the correctness of the operations in SCVM during contract execution [35]. However, in parallel validation, random validation order leads to non-deterministic intermediate state. The receipt generation based on the non-deterministic intermediate state generates inconsistent receipts, thereby causing inconsistency in the blockchain state among blockchain nodes [23]. Consequently, it is difficult to design an SCVM that can simultaneously guarantee receipt consistency and the accuracy of operations in parallel transaction validation.

[1] github.com/ethereum/go-ethereum/blob/master/core/state/journal.go

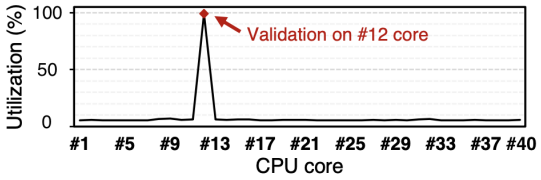


Fig. 7: Utilization of each CPU core during validation.

III. PaVM Design

PaVM aims to provide a parallel virtual machine with programming interfaces, a bytecode instruction set, and a runtime system to support both inter-contract and intra-contract parallel execution. The entire runtime information generated by atomic operations can be recorded by the proposed instructions with minimal memory burden. PaVM re-designs the machine state to handle massive amounts of thread data and implement data order control among multiple threads. Moreover, PaVM provides a transaction receipt generation method based on R/W operations, specifically tailored for parallel transaction validation.

A. Overview

Fig. 8 shows the overview of PaVM. The core design of PaVM can be decomposed into three parts: programming interfaces, bytecode instruction set, and runtime system. (1) The programming interfaces are divided into thread control, data order control, and runtime recording control. These interfaces are supplied to enable developers to achieve data operation order consistency and accurate runtime information recording, thereby facilitating parallel validation. (2) The bytecode instruction set is designed to facilitate the compilation of programming interfaces. The instruction set contains thread control, data control, and runtime recording instructions. PaVM facilitates parallel transaction validation and receipt generation by recording the entire runtime information with the help of runtime recording instructions. All the information is generated by contract instructions such as `SLOAD`, the proposed recording instructions can record the instruction action and its context. (3) The runtime system is the execution environment of smart contracts. It includes machine state, thread state, and thread management. The machine state is re-designed to support the storage of subthread data. The Stack stores instruction operands. The persistent data is stored in Persistent Storage. The complex temporary data such as the array is stored in Temporary Data. The call data of subfunctions are stored in Thread Data. During contract execution, PaVM assigns the subfunctions to different threads and executes them in parallel. Each thread reads/writes its own Thread Data area and updates its state in Thread State Table. The return values of threads are stored in Temporary Data according to the metadata in Thread Data. The runtime information of both the

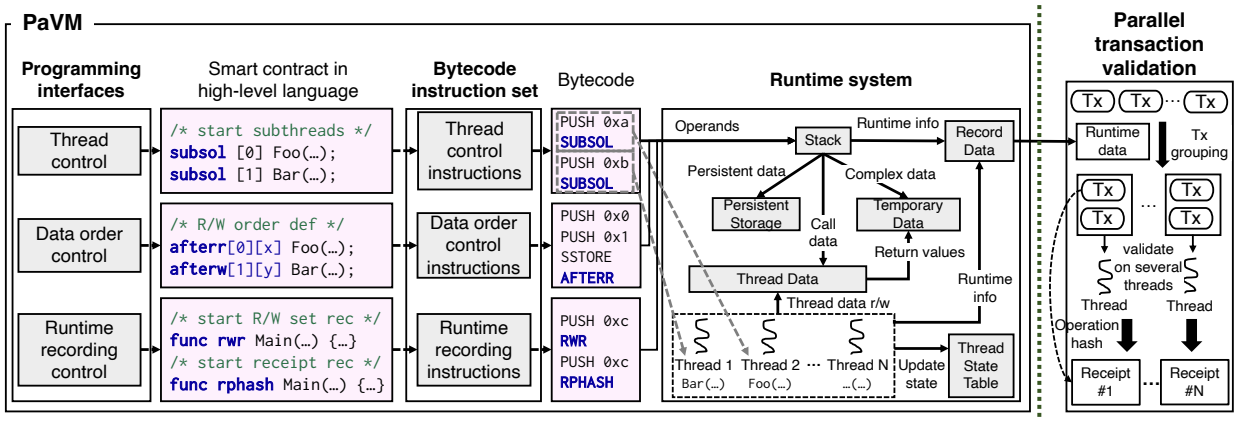


Fig. 8: The overview of PaVM.

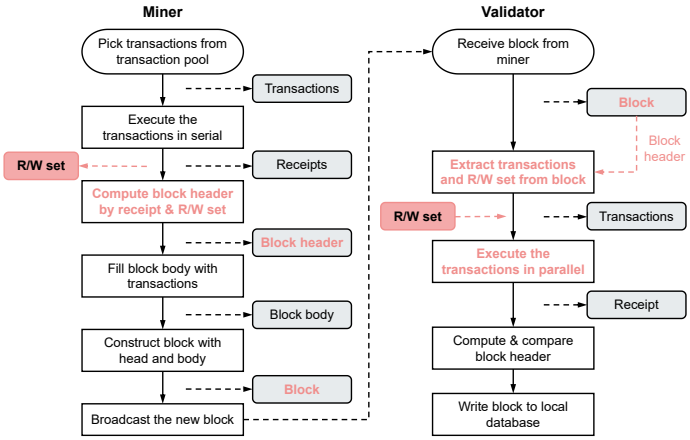


Fig. 9: The block cycle in the blockchain with PaVM.

main thread and sub-threads is stored in Record Data. The data in Record Data is read in the transaction validation process.

PaVM supports parallel validation by recording runtime information precisely and entirely. The transactions are organized into groups based on the runtime information, thus avoiding data conflicts and maintaining blockchain state consistency. While the transactions are validated on different threads, the hash values of data R/W operations are computed to generate the receipt. The operation-based receipt generation ensures the correctness of operations and the consistency of the final state.

We summarize the block cycle in the blockchain with PaVM in Fig. 9. The miner generates and broadcasts a new block, and the validator(s) validate the newly generated block. First, the miner picks transactions from the transaction pool according to specific rules such as transaction fee priority. Then the miner executes the picked transactions in serial and generates the receipts. In PaVM, to accelerate the block cycle and validation, an R/W set will be generated during the miner executing the transactions in serial. The R/W set records the data read and write sequence. Therefore, the miner can divide the transactions into different parallel groups according to the R/W set. The R/W set and group information can be stored in the block header with negligible overhead.

The block header hash is calculated by receipts and the generated R/W set, and the block body is filled with the picked transactions. Subsequently, the miner constructs a new block by block header and body, then broadcasts the new block to other nodes (validators). The validator receives the new block from the miner and extracts the transactions from the received block. With the help of the R/W set and group information, the validators can execute the transactions and smart contracts in parallel. The parallel contract execution can improve the validation performance. Lastly, the block is stored in the local database of the validator.

B. Key interfaces

PaVM provides programming interfaces for developers to realize intra-contract parallelism. The key interfaces in PaVM are listed in Table. I. There are three kinds of interfaces: thread control, data control, and runtime recording control. The thread control interfaces support thread management. For instance, a thread can be started to run a subfunction by `subsol`, and be suspended by `sleep`. The data control interfaces are used for defining data operation order to ensure blockchain state consistency among blockchain nodes. The runtime recording control interfaces declare runtime information recording at the instruction level. The recorded data can support parallel transaction validation. The aforementioned interfaces are encapsulated in keywords in practice.

The thread control interfaces are provided for developers to start and terminate threads. The two functions, `Foo` and `Bar`, can run in parallel by the statements `subsol Foo(...)` and

Key programming interfaces		
Type	**Keyword**	**Syntax**
Thread control	subsol	subsol [*tid*] [*func*]
Thread control	sleep	sleep [*tid*] [*time*]
Thread control	revertt	revertt [*tid*]
Data control	afterw	afterw [*tid*] [*var*] [*func*]
Data control	afterr	afterr [*tid*] [*var*] [*func*]
Data control	aftera	aftera [*tid*] [*func*]
Runtime record	rr	func(...) [*rwr*]
Runtime record	rphash	func(...) [*rphash*]

TABLE I: Key programming interfaces in PaVM.

`subsol Bar(...)`, respectively. PaVM allows handling the threads in a fine-grained way: developers can define the thread identifier to control threads. For instance, in the statement `subsol [9] Foo(...)`, the thread with identifier nine runs the function `Foo`.

In order to run subfunctions in parallel without thread R/W conflicts, we provide programming interfaces to explicitly specify the R/W order. This approach ensures the consistency of computed results. The data R/W order is ruled by `afterw`, `afterr`, and `aftera` interfaces in terms of the thread identifier, operation type, and variable name. For instance, the statement `afterw[6][x] Foo(...)` means that all operations on variable `x` in `Foo` should occur after thread #6 has written the variable `x`. Similarly, the statement `aftera[9] foo(...)`, means that all the operations on any variable should occur after thread #9 has read/written all variables. PaVM will not lead to incorrect execution results when the developer does not write any data control, because PaVM has a default data R/W sequence. PaVM advises against introducing data dependencies between functions running on different threads, as it may lead to data R/W blocking.

The runtime recording interfaces enable runtime data recording at the instruction level. For example, `Foo(...) rr` allows runtime records within the `Foo` function. The recorded data includes data R/W order, logging, and others. The keyword `rphash` is used for computing the data operation hashes, which support R/W operation-based receipt generation.

PaVM provides concise interfaces for developers to minimize development burden. Users can enable PaVM features with minimal modifications, as shown in Fig. 10. Furthermore, PaVM seamlessly integrates with EVM, as traditional smart contracts can also be executed in PaVM.

As shown in Fig. 10, we use the *WordCount* contract as an example to demonstrate the utilization of key interfaces. The *WordCount* contract returns the number of each word (e.g., $< word, frequent >$) in the input text. The contract code is given in Fig. 10(a). There are two functions in the contract: *main* and *count*. The *main* function (line 13) splits the input text into two segments and dispatches them to the *count* function (line 3) to calculate the number of each word.

In Fig. 10(b), we use *rr* and *rphash* keywords to enable runtime information recording and hash computation (line 13), respectively. With the *rr* keyword, after the contract is executed, PaVM generates a data read/write set to support parallel transaction validation. With the *rphash* keyword, the hash value of each data r/w operation can be computed when it occurs. Since the input text can be split into different segments for parallel processing, the *subsol* keyword can be used for starting two threads (lines 16 and 17) to perform the *count* function simultaneously thereby improving the process performance. The numbers in brackets, such as `[1]` and `[2]`, are the thread IDs within PaVM. Other keywords, such as *sleep* and *revertt*, can easily control the corresponding thread through specific thread IDs (lines 18 and 19).

C. Key instructions

PaVM provides a bytecode instruction set to support contract compilation and realize precise runtime information recording. The instruction set contains thread control instructions, data control instructions, runtime recording instructions, and normal instructions. Table II shows the key instructions. The first three kinds of instructions are designed to support the programming interfaces during compilation. The normal instructions include some basic instructions for computations such as `ADD`. The Opcode is the specific byte of the instruction in bytecode. The context refers to the arguments required by the instruction.

Key instructions			
Type	**Mnemonic**	**Opcode**	**Context**
Thread control	SUBSOL	0xA1	PUSH, PUSH
Thread control	SLEEP	0xA2	PUSH, PUSH
Thread control	REVERTT	0xA3	PUSH
Data control	AFTERW	0xA4	PUSH, MLOAD
Data control	AFTERR	0xA5	PUSH, MLOAD
Data control	AFTERA	0xA6	PUSH, MLOAD
Runtime record	RR	0xA7	PUSH, MLOAD
Runtime record	RPHASH	0xA8	PUSH, MLOAD
Runtime record	RDR	0xA9	PUSH
Runtime record	WTR	0xAA	PUSH

TABLE II: Key instructions in PaVM.

The thread control instructions can start, suspend, and terminate a specific thread. Once a thread control instruction is executed, the corresponding thread will receive a message to perform the corresponding action.

The data control instructions enforce the user-defined data R/W order. Once the current operation on a variable does not conform to the semantics of data control interfaces, the operation will be blocked until other threads complete the operation on this variable.

In the existing SCVM, all runtime information is generated from atomic operations, including database R/W and log appending. These atomic operations are performed by instruction execution, so we record instruction actions in the runtime system of PaVM to record the entire information to support transaction grouping. For example, the data read and write can be recorded by `RDR` and `WTR` instructions, respectively. The `RDR` instruction records the data location when reading data, and the `WTR` records the data location and the new value when writing data.

Runtime data recording captures the entire sequence when the miner pre-executes transactions to ensure result consistency. The recorded data includes the count of variable R/W operations and identifies which transaction wrote/read the variable first. When the validators execute the transactions in parallel, the sequence will be checked when the data is written/read. When a read/write operation happens, the thread will check whether the previous operations occur or not. As a result, the transactions can be executed in parallel even though the transactions invoke the same contract function. With the read/write sequence checking, PaVM will not restrict the parallelism between transactions when two transactions invoke the same smart contract.

The runtime recording instructions generate a data R/W set after contract execution. The data R/W set contains contract address, database identifier, and other relevant information. After contract execution, the R/W set is stored in the transaction

```
contract WordCount {

 function count(string textSegment) {
  mapping string[int] ret;
  for(int i = 0; i < len(textSegment); i++) {
    string word = textSegment[i];
    ret[word] == NULL ? ret[word] = 0 : ret[word]++;
  }
  return ret;
 }

 /* Main function for process the input text */
 function main(string text) {
  int length = len(text);

  mapping tempRet1 = count(text[0 : length/2]);
  mapping tempRet2 = count(text[length/2 : length]);

  /* Join the two temporary values by embedded function */
  return join(tempRet1, tempRet2);
 }
}
```

(a) Traditional smart contract programming.
The two `count(…)` functions are executed in serial.

Enable data r/w recording

Enable parallelization

```
contract WordCount {

 function count(string textSegment) {
  mapping string[int] ret;
  for(int i = 0; i < len(textSegment); i++) {
    string word = textSegment[i];
    ret[word] == NULL ? ret[word] = 0 : ret[word]++;
  }
  return ret;
 }

 /* Enable runtime record and hash computing in main function */
 function rr rphash main(string text) {
  int length = len(text);

  mapping tempRet1 = subsol [1] count(text[0 : length/2]);
  mapping tempRet2 = subsol [2] count(text[length/2 : length]);
  sleep [1] 1000; // No.1 thread sleeps for 1000ms
  revertt [2]; // Revert the operations made by No.2 thread

  /* Join the two temporary values by embedded function */
  return join(tempRet1, tempRet2);
 }
}
```

(b) Smart contract programming with PaVM.
The two `count(…)` functions can be executed in parallel.

Fig. 10: Example for key interfaces. The developers can enable PaVM features with some tiny modifications.

metadata by the RWS instruction. The R/W set is the input of the transaction grouping.

D. Runtime System

1) Machine state: Machine state stores the temporary data such as instruction operands and contract variables during contract execution [36]. In traditional EVM, the machine state only stores the data generated by a single thread. When utilizing multiple threads to execute functions, these threads may concurrently access the same machine state location. To address this issue, we propose Thread Data and Thread State Table to store thread data independently to avoid the R/W conflicts in the machine state. Besides, managing massive return values in the machine state requires considering how to store the generated data by multiple threads in appropriate locations. We store the indexes of the return values in Thread Data and the specific values in Temporary Data. We redesign the traditional machine state to solve the above problems in PaVM.

Thread Data stores the call data and return value metadata for subthreads. Call data includes the function selector and arguments. The function selector indicates the function to be invoked, while the arguments represent the inputs for the invoked function. The return value metadata is written to Thread Data upon completion of a subthread and includes offset and size in Temporary Data. The return value metadata can be empty, as a function may return without any values.

Record Data stores the runtime information generated by the runtime recording instructions such as RDR and WTR. There are several kinds of runtime data supported by PaVM: data R/W set, Temporary Storage R/W set, etc. The data in Record Data is delivered to support transaction grouping and receipt generation. All the runtime information is stored in Record Data until it is consumed in the validation process.

Thread State Table is provided to record the state of threads. For instance, a thread updates its state to *Running* to the Thread State Table when it begins to run. At the end of thread execution, it updates its state to *Return* and writes its return value metadata to the corresponding Thread Data.

Besides, the following structures in traditional SCVM are required to store the main thread data:

Stack stores instruction operands and temporary variables. The maximum height of Stack is defined as 1024 in PaVM for compatibility with traditional SCVM like EVM. The Stack is operated by stack-related instructions such as PUSH. The Stack also stores call data of threads through thread control instructions. The call data in Stack is transferred to Thread Data.

Persistent Storage stores persistent data such as global variables within a contract. The data in Persistent Storage is stored in blockchain persistently after contract execution, and it is accessible to all threads.

Temporary Data stores complex variables (e.g., mapping, array) and return values of the subthreads. The return values of each specific thread are written to Temporary Data based on their respective offset and size.

2) Thread State and Management: To monitor whether a thread is running or terminated, it is necessary to design the thread state in runtime system. The thread state design in PaVM is inspired by the Goroutine's state in Golang[2]. PaVM excludes the state involving data competition, which can be avoided through data control instructions.

The thread state in PaVM includes: *Preparing*, *Running*, *Waiting*, *Return*, and *Revert*. *Preparing* refers to the execution environment validation (e.g., Stack overflow). *Running* denotes that the function is executing on a thread with associated call data. *Waiting* indicates that the thread is waiting for other

[2]github.com/golang/go/src/runtime/runtime2.go

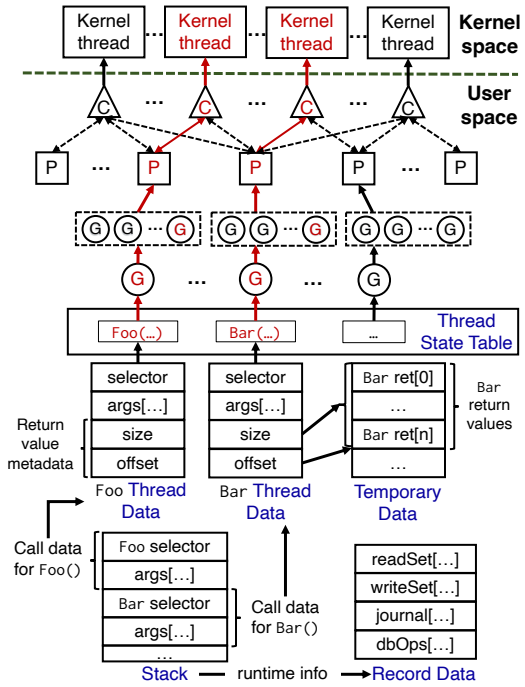


Fig. 11: The runtime system of PaVM.

threads' R/W operations. *Return* represents that the thread has written the return values to the corresponding location (otherwise the state is *Running*). *Revert* is important as a transaction may be reverted. All the operations by the reverted thread should be withdrawn. During contract execution, the main thread maintains a thread state table to record the thread state. The main thread appends the new thread state to the table when it uses system calls (e.g., `clone()`) to create a subthread. The state in the table is then updated by the corresponding subthread. The main thread constantly monitors the state of each thread. Once the thread state is *Return* or *Revert*, the main thread retrieves the return value from the corresponding thread's temporary data area. Then the subthread's temporary data area is released.

The thread management is implemented in the runtime system. PaVM executes multiple functions in multiple kernel threads during contract execution. As shown in Fig. 11, the subthread call data is stored in Stack by `subsol` instruction. The call data is then transferred to the corresponding Thread Data. The thread state is maintained in Thread State Table. In PaVM, we exploit the GCP model [37] to realize parallel execution. The subthreads that run different functions are started by several different Goroutines [38]. For example, `Foo` and `Bar` functions are loaded in different Goroutines. The different Goroutines are appended to different G queues mounted on distinct Processors. Lastly, the different Processors are bound to different Cores. Each Core corresponds to a kernel thread. In this case, `Foo` and `Bar` functions run on two kernel threads, then run on different CPU cores after being scheduled by the operating system. During main thread and subthreads execution, the runtime information is recorded in Record Data to support parallel transaction validation.

We give a snippet of a smart contract with new interfaces (i.e., `subsol`) to illustrate the schedule of parallel execution. As shown in Fig. 12, the contract uses the keyword `subsol` to define two threads to execute `foo` and `bar` functions in parallel, respectively. PaVM runtime first stores the invoke data such as function identifier and arguments in Stack, and stores the thread states in Thread State Table. During contract execution, PaVM runtime maintains three kinds of struct to realize thread management: goroutine (G), processor (P), and Core (C). G is a lightweight coroutine with contract function information, P is a coroutine queue manager, and C is a mapping between user thread and kernel thread. PaVM runtime starts two Gs in user space, the two Gs record the function information such as entry address. Then runtime puts the Gs to queues under different Ps according to the load balancing principle. P decides which G can be executed according to scheduling rules such as short-task first. After that, P picks a free C to execute the contract function in the kernel thread. The schedule of kernel threads used in PaVM is the default rule of Linux, i.e., Completely Fair Scheduler (CFS).

Overall, the GPC model with both user space and kernel space scheduling can hold vast user threads, and different contract functions can be assigned to different kernel threads to be executed in parallel.

E. Parallel Transaction Validation

The transaction validation benefits from runtime recording instructions in PaVM. PaVM is able to transfer the complete 14 kinds of runtime information such as `storageChange` and `nonceChange` from the runtime system to validation. The traditional methods can only discover two kinds of information (`storageChange` and `balanceChange`) related to data R/W operations. For the miner-validator style, the miner executes the contracts to record the entire runtime information, then the transactions are grouped by the disjoint set algorithm according to the recorded data. The grouping information is recorded in the block header, which will be broadcast to validators. The validator performs parallel validation between transaction groups based on the grouping information. In PaVM, transaction receipts are generated based on intermediate operations rather than intermediate states. The R/W operation-based receipt generation can ensure both the correctness of operations in SCVM and the consistency of the blockchain state.

Transaction grouping. PaVM adopts a disjoint set to group transactions, because the disjoint-set algorithm just brings small-scale computations [39]. We use `Txs` to represent the transactions in a block, `RS` and `WS` denote the read set and write set generated by runtime recording instructions during execution. The `group` is defined to store grouping information. Each transaction initializes a single group. Next, iterating through the transactions from the first transaction in `Txs`. The transactions without read-write, write-write, or write-read conflicts can be divided into the same group. Lastly, the algorithm returns the grouping information. In parallel validation, the transactions are processed in parallel between groups

and serially within groups. During grouping, the algorithm analyses both the data R/W conflicts in traditional methods and other runtime information such as log recording and nonce changing. The grouping with all runtime information brought by PaVM is more precise, ensuring the consistency of the blockchain state.

R/W operation-based receipt generation. In PaVM, transaction receipts are generated based on R/W operations conducted in each smart contract. The receipts are computed by using intermediate data R/W operations rather than the intermediate state. The R/W operations are recorded via runtime recording instructions during execution. The R/W operation-based receipt generation is shown in Fig. 13. The data R/W set is produced by runtime recording instructions, and the set is composed of data R/W operations and orders. Each data R/W operation has several fields such as storage location and value. Each operation is summarized as a hash value by `wHash` or `rHash` instruction. All operation hashes are summarized as transaction receipts. Benefits from the runtime recording instructions, all the operations can be entirely recorded for generating receipts which help ensure the correctness of the intermediate operations and consistency of the blockchain state.

IV. Evaluation

To validate the design point of PaVM and demonstrate its performance benefits, we deploy experiments on the Ethereum testnet and the multiple-core equipped CPU platform. The objectives of the evaluation are fourfold: (1) testing the performance improvement of PaVM compared with Geth in transaction validation; (2) testing the performance improvement of PaVM compared with EVM in contract execution; (3) providing insights of PaVM's outperforming its peers; and (4) studying the impact of PaVM on the original system.

A. Experimental setup

PaVM is built based on the following hardware and software:

Hardware. According to the previous work [6], we employ four servers equipped with Xeon E5-2630 CPUs (2.3 GHz, 40 Cores) and 96GB of memory to establish PaVM's testbed. We run 25 blockchain nodes to evaluate the overall experiments and eight nodes for other experiments. We leverage more nodes in the overall experiments to demonstrate the scalability of our solution.

Software. The smart contract programming language is Solidity, the corresponding compiler is Solc (v0.4.20). The Ethereum test network is built by Geth (v1.10). The numerical results are recorded by the Web3PY framework [40] from the `Process` function[3] (transaction validation) and the `Run` function[4] (contract execution).

	Workload	Used for	Explanation
1	MatrixMul	Intra-contract parallelism	Different threads compute different parts of the matrix.
2	MatrixAdd		
3	Substring		Different threads process different text segments.
4	Histogram		Different threads process different input array segments.
5	ERC20	Inter-contract parallelism	The R/W operations on balance/account are atomic.
6	Fibnacci		The Fibnacci computations are serial.
7	CPUHeavy		The sort algorithm is serial and cannot be spilt.
8	KVStore		The R/W operations on blockchain state data are atomic.

TABLE III: The workload characteristic description.

Benchmarks. Considering that different workloads have different computational characteristics, the workloads we used have two categories: one part is for evaluating the performance of inter-contract parallelism, and the other part is for evaluating the performance of intra-contract parallelism. For example, the *Substring* contract is suitable for intra-contract parallelism because each thread in the contract can handle different string segments. The *KVstore* contract simulates the reading and writing operations of blockchain data. This contract is more suitable for inter-contract parallelism evaluation because the read and write operations are atomic operations. Overall, as shown in Table III, the workloads suitable for multi-threading

[3] github.com/ethereum/go-ethereum/blob/master/core/state_processor.go
[4] github.com/ethereum/go-ethereum/blob/master/core/vm/interpreter.go

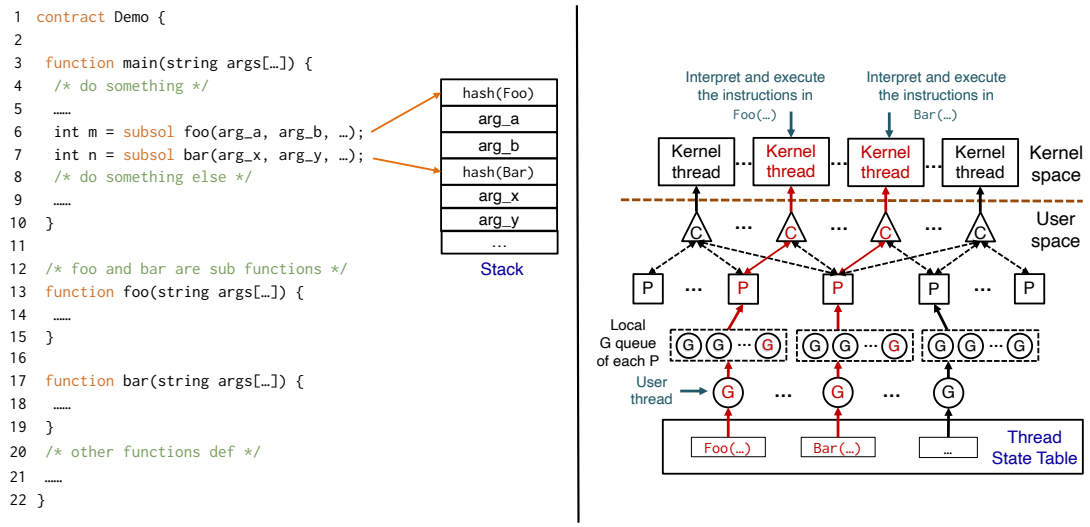


Fig. 12: Thread schedule in PaVM with the GCP model.

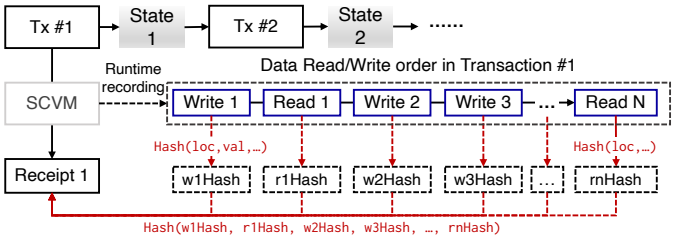


Fig. 13: The receipt generation based on R/W operations.

are utilized to evaluate intra-contract parallelism, and the single-threaded applications are employed to evaluate inter-contract parallelism.

The smart contracts for transaction validation are ERC20 [41], Fibonacci, CPUheavy [42], and the KVstore contract [42]. ERC20 contract implements the functions related to blockchain tokens. Fibonacci contract computes the Fibonacci sequence by a recursion function. CPU heavy contract implements a quick sort for a given array. KVstore contract performs massive reading and writing operations for contract variables. The smart contracts for evaluating the performance of contract execution are MatrixMul [4], MatrixAdd [4], Substring [43], and Histogram contract [43]. The first two contracts implement matrix multiplication and addition. The Substring contract implements matching substrings within a long string. The Histogram contract is used for finding the numbers in an array that satisfies the input range.

Measure Metrics. We compare PaVM with Geth and EVM in terms of the following metrics: (1) Latency. The main goal of PaVM is accelerating contract execution and transaction validation. (2) Throughput. We evaluate the throughput of transaction validation with both inter-contract and intra-contract parallelism. (3) Hardware utilization. PaVM pursues higher hardware utilization to achieve fast transaction validation and contract execution.

B. Latency

We evaluate latency from three aspects: (1) Overall latency to demonstrate the effectiveness of PaVM with both inter-contract and intra-contract parallelism, (2) contract execution latency comparison between PaVM and EVM, (3) transaction validation latency comparison between our proposal and Geth.

1) Overall latency: We compare the overall latency of PaVM with Geth on seven contracts including Fibonacci, Histogram, etc., with different computational scales (contract inputs), as shown in Fig. 14. PaVM enables both inter-contract and intra-contract parallel execution.

The transaction validation has three main steps: extracting transactions from the block, contract execution, and receipt generation. The overall latency refers to the time from transaction extraction to receipt generation. The results show that compared with Geth, PaVM achieves 17.5×, 17.7×, 18.5×, 33×, 26.7×, 33.6×, and 21× speedups on average for seven different contracts, respectively. The latency can be reduced by more than 95.4% on average for these contracts. Results also show that the overall transaction validation can achieve the speedup by 3×, 1.3×, and 2.7× on average among the eight contracts, compared with the methods in [6], [7], and [44], respectively. PaVM can significantly improve the performance of transaction validation with the help of inter-contract and intra-contract parallelism.

PaVM is particularly well-suited for executing tasks such as Substring, since a Substring task can be broken down into smaller computations that can be processed in parallel. Note that as the task size increases, the time to process the task increases, but the speedup ratio is similar because the number of CPU cores limits the parallelism of PaVM, which ultimately affects performance.

2) Contract execution latency: The execution latency is evaluated on MatrixMul, MatrixAdd, Substring, and Histogram contracts. We perform addition and multiplication on 40 matrices, matching the quantity of CPU cores. We split the string and array on Substring and Histogram contracts into 40 slices to facilitate parallel contract execution, as shown in Fig. 15.

In MatrixAdd, as shown in Fig. 15a, the latency can be reduced by 69.1%, 74.8%, 73.7%, 80%, and 83.6% with different matrix sizes (from 90×90 to 210×210). The matrix size is changed from 40×40 to 80×80 in MatrixMul. Compared with EVM, the latency is reduced by 86.3%, 89.1%, 92.6%, 91.9%, and 93.9% under different matrix sizes in MatrixMul (see Fig. 15b). For the matrix operations, the latency reduction reaches 83.5% on average. With the increment of matrix size, the intra-contract acceleration gets more effective.

For Substring, the string size changes from 12,000 to 28,000, as shown in Fig. 15c. In PaVM, the latency can be reduced by 95.3%, 95.3%, 95.3%, 95.4%, and 95.5% under different string lengths, compared with EVM, achieving 22× speedups on average. For Histogram, the integer array size changes from 30,000 to 70,000. The results in Fig. 15d show that PaVM reaches 20.9× speedups on average under different array sizes. The latency is significantly reduced by 95.2% in PaVM for the Substring and Histogram contracts on average, compared with EVM.

3) Transaction validation latency: We conduct transaction validation latency on ERC20, Fibonacci, CPUheavy, and KVstore contracts with different computational scales. We examine the function type, length of Fibonacci sequence, array size for quick sort, and the numbers of data read/write of ERC20, Fibonacci, CPUheavy, and KVstore contracts, respectively.

The number of transactions for validation is 300, which roughly equals the average transaction number in a block in Ethereum. Fig. 16a shows the results of ERC20. Compared with Geth, the parallel validation in PaVM reduces the latency by 75%, 68.3%, and 68% for `approve`, `transfer`, and `transferFrom` function, respectively. For Fibonacci, the latency of PaVM can be reduced by 94.3% on average with the length of the Fibonacci sequence changes from 50,000 to 350,000 (see Fig. 16b). For CPUHeavy, in Fig. 16c, with the array size increased (from 90 to 210), the latency is reduced from 94.1% to 94.5% gradually in PaVM. The validation latency is significantly reduced from 12.7s in Geth to 0.68s in PaVM with the same array size of 210. For the KVstore

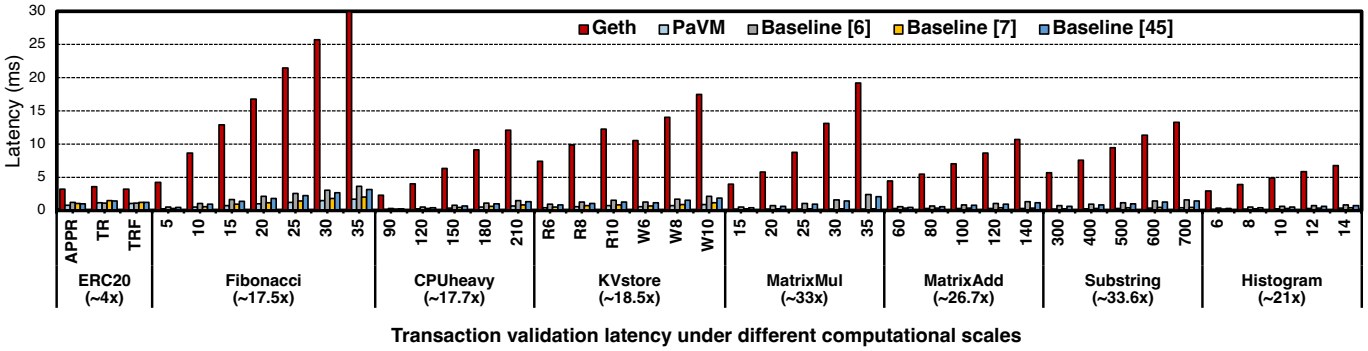
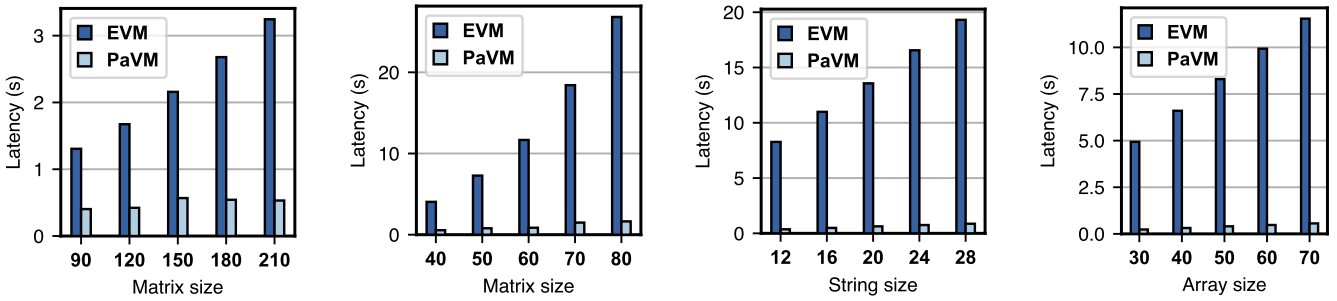
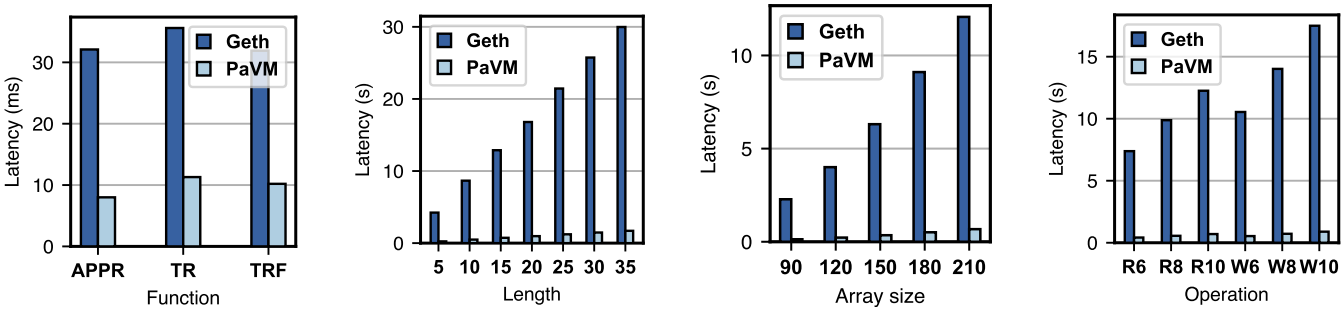


Fig. 14: The overall latency comparison between baselines and PaVM on eight contracts with different computational scales.



(a) MatrixAdd (b) MatrixMul (c) Substring (d) Histogram

Fig. 15: Contract execution latency results comparison between EVM and PaVM with different computational scales.



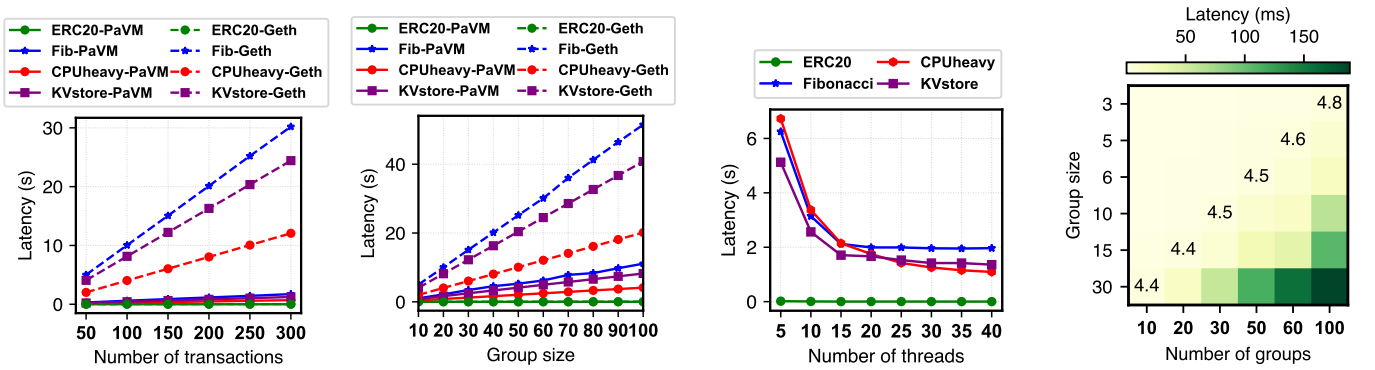
(a) ERC20 (b) Fibonacci (c) CPUheavy (d) KVstore

Fig. 16: Transaction validation latency results with different contract inputs. The transaction number in the block is 300.

contract, we explore the impact of data read and write on the latency during transaction validation. In Fig. 16d, R6 means reading a variable 6,000 times, and W10 means writing a variable 10,000 times. The results show that PaVM reduces the latency by 94.3% and 94.9% with massive data read and write, respectively.

Next, we change the number of transactions in a block (Fig. 17a) and the group size to observe validation latency (Fig. 17b). With different numbers of transactions in a block, PaVM demonstrates that it can reach 13×, 17.2×, 16.2×, and 19.2× speedups on average for ERC20, Fibonacci, CPUheavy, and KVstore, respectively, compared with Geth. The maximum speedup reaches 27× in the ERC20 contract with the number of transactions of 300. In PaVM, the transactions are assigned into different groups to be validated in parallel. The effect of group size (total number of transactions is 300) on validation latency is given in Fig. 17b. With the increment of group size, the validation latency is increased from 2.78s to 81.9s in Geth, and from 0.57s to 5.9s in PaVM validation, respectively, because the number of groups that can be validated in parallel is reduced. For various group sizes, the latency of PaVM is reduced by 64.8%, 78.6%, 79.4%, and 79.8% on average for the four contracts, respectively, compared with Geth.

Then, we explore the impact of parallelism on transaction validation by adjusting the number of threads in PaVM. Fig. 17c shows the results. More threads bring lower validation latency, as more transactions can be validated in parallel. For `transferFrom` function in the ERC20 contract, the latency is reduced from 17ms to 1.8ms (9.4× speedups) when the number of threads varies from 5 to 40. For the Fibonacci contract with the length of 350,000, PaVM achieves 3.2× speedups. For CPUheavy, the validation latency is reduced from 6.7s to 1.1s with the increment of threads. For KVstore contract, the latency is reduced by 74.5% (from 5.1s to 1.3s) in PaVM. The average reduction of validation latency can reach 79.1% for the four contracts.



(a) Different transaction numbers (b) Different group sizes (c) Different thread numbers (d) Recording and grouping time

Fig. 17: Transaction validation latency and recording time results with different transaction scales.

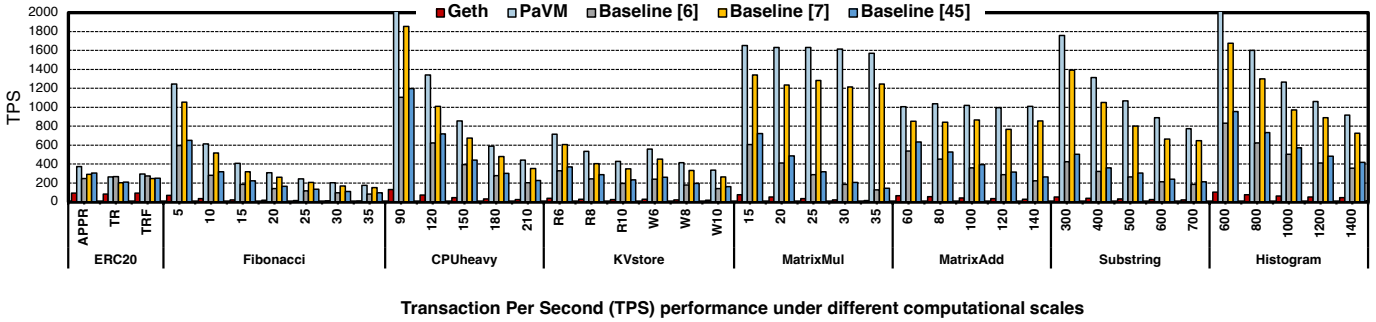


Fig. 18: The throughput comparison between baselines and PaVM on eight contracts with different computational scales.

Lastly, we consider the impact of PaVM on the original Geth through recording runtime information and transaction grouping. Fig. 17d shows the results. The x-axis represents the number of transaction groups that can be validated in parallel, the y-axis represents the group size, and each cell represents the time. The product of the x-axis and y-axis of each cell represents the total transaction number in a block. The upper left part of the heatmap represents a small number of transactions, while the bottom right represents a large number. The lighter the color of the heat map implies the shorter the time, and vice versa. It is obvious that the recording and grouping latency is proportional to the number of transactions. The average latency for recording 300 transactions is only 4.5ms in PaVM. In practice, the recording and grouping time can be ignored during transaction validation. With the transaction number of 300, the recording time only accounts for 0.3% of the parallel validation total time.

C. Transaction Throughput

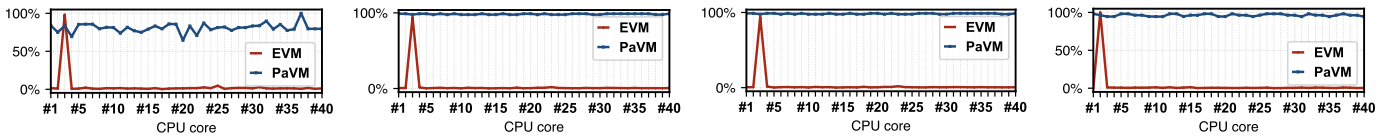
Latency can demonstrate the efficiency of PaVM in processing different kinds of tasks and is closely related to user experience. In this section, we examine the throughput of PaVM because it reflects the number of tasks or transactions that PaVM can handle in a fixed period. Throughput exhibits the PaVM's capability to process a large number of transactions and tasks.

We evaluate the throughput of transaction validation on seven contracts from Fibonacci to Histogram with different computational scales as shown in Fig. 18. The unit of throughput is TPS (transaction per second). The evaluation is conducted with the default Proof-of-Work consensus. Overall, the throughput of the entire system can be improved by 24.4× compared with Geth. Compared with the methods in [6], [7], and [44], the throughput improvement achieves 3× on average among the eight contracts. Results show that for the Fibonacci contract, the throughput can be improved from 26 TPS to 457 TPS on average. Especially, with a scale of 10, the throughput is improved by 2.71× in PaVM, compared with Geth. For CPUheavy contract, the throughput reaches 2,240 TPS with a scale of 90, and more than 440 TPS even with the largest computational scale in PaVM. For KVstore contract, the average throughput (27 TPS in Geth v.s. 498 TPS in PaVM) is smaller than that of other contracts.

The throughput of MatrixAdd and MatrixMul reaches more than 1,650 TPS. For MatrixMul, with the increment of computational scale (from 10 to 35), PaVM can keep the TPS above 1,550, while the TPS in Geth is 17. The TPS in Substring contract reaches nearly 1,800 with a size of 300 in PaVM, while the TPS is only 60 in Geth. The TPS is maintained at nearly 800 in PaVM even with the largest string size. For Histogram contract, the highest throughput is 2,142 with a size of 600 in PaVM, while the throughput is 916 with a size of 14,000. The maximum throughput in Geth is only 103.

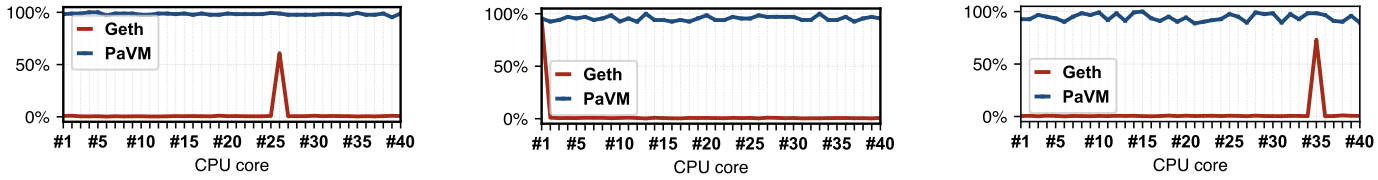
D. Hardware Utilization

We evaluate the utilization of CPU cores to represent the hardware utilization in validation and execution. Higher CPU utilization means more sufficient usage of hardware resources



(a) MatrixAdd (matrix size=80) (b) MatrixMul (matrix size=80) (c) Substring (string length=80) (d) Histogram (array size=80)

Fig. 19: CPU utilization results for contract execution.



(a) Fibonacci contract (b) CPUheavy contract (c) KVstore contract

Fig. 20: CPU utilization during transaction validation on different smart contracts.

and stronger processing capabilities. PaVM enables both inter-contract parallelism and intra-contract parallelism on different CPU cores. The subfunctions in PaVM are started by the `subsol` keyword in PaVM. The CPU utilization is recorded by the *pprof* toolkits in Golang.

1) Contract execution: We evaluate the CPU utilization during contract execution with MarixMul, MatrixAdd, Substring, and Histogram contracts. PaVM utilizes multiple CPU cores to run matrix multiplication and addition for the first two contracts, and process the slices of the input array for the last two contracts. The CPU utilization in PaVM is significantly higher than EVM, because the EVM can only run all computations in a single core, while PaVM can utilize all CPU cores and achieve load balancing.

The results of each CPU core utilization with the four contracts are shown from Fig. 19a to Fig. 19d, respectively. For the MatrixAdd contract with the matrix size of 80, PaVM maintains the CPU utilization between 64.6% and 100%, while the average utilization is only 6.4% in EVM. For MatrixMul contract, the average CPU utilization is only 3.1% on 40 cores in EVM, while the utilization is 93.7% in PaVM. In EVM, the maximum CPU utilization is 98.12% on the #3 core. In PaVM, the minimum utilization is up to 83%, and the average utilization on 40 cores is 99.8%. In MatrixAdd contract, the average CPU utilization in PaVM is slightly less than MartrixMul, and PaVM has fluctuations in CPU utilization. The reason is that MatrixAdd has smaller computations compared to MatrixMul, and the computations are evenly assigned to different CPU cores to achieve load balancing.

For Substring contract with a string length of 80, since each core is arranged to perform a string slice, the utilization of all cores is higher than 90% in PaVM. However, in EVM, only the #2 core is used for contract execution with nearly 98%. Similar to the results of the Substring, the performance of Histogram contract is significantly improved by PaVM. With the help of parallel execution in PaVM, the average CPU utilization of 40 cores is up to 99.91% with an array size of 80. In contrast, the average CPU utilization in EVM is only 3%.

2) Transaction validation: The CPU utilization of validation is evaluated on Fibonacci, CPUheavy, and KVstore contracts. Fig. 20 shows the results. For Fibonacci contract, as shown in Fig. 20a, only core #25 is used for validation with 60.9% core utilization in Geth. The other cores with only 0.6% utilization on average (for system services). In PaVM, all cores are activated for validation with 98.3% utilization on average. Specifically, 26 out of 40 cores' utilization is greater than 98%. The lowest utilization in PaVM reaches 95.3%. For CPUheavy contract, in Fig. 20b, Geth only uses the #1 core to perform validation with 94.7% utilization. The average utilization of other cores (from #2 to #40) is only 0.7%, which is used for other system services such as FTP and SSH. Compared with Geth, PaVM with parallel validation significantly improves the average utilization of all 40 cores to 95.4%. The maximum and minimum utilization in PaVM are 92.4% (#11) and 100% (#12), respectively.

The results for KVstore contract are as shown in Fig. 20c. In Geth, only core #35 is actively engaged in validation with 73.2% utilization. The other 39 cores' utilization is 0.52% on average. Such situation is optimized by PaVM: all 40 cores are enabled for validation, and the average utilization reaches 94.4%. Compared with Geth, PaVM improves the CPU utilization by 40.3× (from 2.34% to 94.4%) across all 40 cores. According to the above results, PaVM fully utilizes hardware resources in comparison to Geth. The reason is that Geth assigns the transactions to only a single core for validation, while PaVM validates transactions on multiple cores with thread management.

E. Discussion

According to the above results, it is clear that enabling multiple cores for validation can obviously reduce latency, enhance throughput, and improve CPU utilization. We discuss the impact of PaVM in comparison to the original Ethereum blockchain as follows.

1) System aspect: **The I/O burden.** PaVM brings no I/O burden to the original Geth or EVM, as it has no read/write operations on the hard disk during transaction validation and contract execution. Besides, since PaVM does not change the process of the miner-validator model, it has no impact on the network communication latency.

Network overhead. In both PoW-based and PBFT-based blockchain systems, the newly created block with a R/W set is broadcast to all validator nodes. Specifically, PBFT-based systems utilize a three-phase protocol to reach consensus on the new block [45], [46]. Once consensus is reached, the validators proceed to validate the transactions contained in the new block. PaVM optimizes the transaction validation at the smart contract execution environment level. Therefore, PaVM does not introduce extra network overhead even when the consensus mechanism is modified.

Memory footprint. PaVM stores the thread data in Thread Data Storage, which resides in memory (RAM). Each Thread Data Storage occupies only nearly 1KB memory (RAM), as it is implemented using a byte32 array with four elements. Besides, PaVM imposes no disk storage burden, as PaVM information and data for parallel execution and validation are not stored on the hard disk. The information and data are deleted after validation and execution.

Extra overhead. The extra overhead in parallel validation in PaVM mainly stems from data R/W recording and disjoint-set algorithm based transaction grouping. On one hand, instead of complex static or dynamic analysis, the recording process is simply making marks during pre-execution in the miner-validator model. The recording time is proportional to the number of reading and writing storage, so smart contracts should minimize the reading and writing of storage. Furthermore, according to the experimental results, the R/W set size amounts to 2.22KB across various real block sizes. The R/W set size typically makes up just about 1% of the total block size. On the other hand, the grouping time is proportional to the number of transactions too. When the dependency between transactions within a block is weak, the time for parallel validation can cover the grouping time. In practice, the extra overhead is very low, because the recording and algorithm just need a few milliseconds, while the validation latency is several seconds.

Design principle/balance. In the traditional miner-validator model, the contract execution can be accelerated on both the miner side and the validator side. In PaVM, we focus on the validator side, as remaining the pre-execution step can allow for more precise transaction grouping. Besides, in the blockchain with Proof-of-Work (PoW) consensus, the bottleneck is the re-execution step rather than the pre-execution step. Once the bottleneck shifts from consensus, the acceleration on the miner side will bring higher performance.

2) Other aspects: **Gas cost.** In mainstream blockchains, gas serves as a unit for representing the computational complexity of a smart contract. In traditional virtual machine instructions, most instructions are used for performing computations, such as `ADD`, `SUB`, and `SSTORE`. However, since the new instructions in PaVM do not involve any additional computations, PaVM does not define the specific gas cost of new instructions. For example, `SUBSOL` instruction is only used for starting new threads to run the subfunctions, `RWRITE` is only used for recording the variable writing situations to the memory.

The gas mechanism is also designed to limit the abuse of computation resources. PaVM is proposed to support the applications in private scenarios. In private scenarios, because the members are trusted in the system, the computation resources will not be abused by default. To facilitate system management, PaVM provides a gas cost design interface in the form of JSON files for system administrators to specify the gas costs of new instructions.

Consensus mechanism. In modern blockchain system, the consensus mechanism generally consists of three steps: block generator/owner election, block broadcast, and block validation [46]. PaVM focuses on the last step (block validation), as it aims to improve the performance of transaction validation and contract execution through virtual machine design. Hence, the different consensus algorithms have no impact on PaVM performance.

V. Related Works

The related works accelerate transaction validation by parallel processing, which is enabled by transaction grouping and other approaches.

Transaction grouping is based on a data read/write set. The transactions are grouped according to whether the data read-/write conflicts or not in the set. Transactions within a group are processed in sequence, while all groups are processed in parallel. The data read/write set can be generated in two ways as follows: (1) **Dynamic speculative execution**. The speculative execution of smart contracts can detect data conflict information by pre-execution. For instance, Dickerson et al. [9] adopt a speculatively execution and rollback mechanism to record parallel execution paths, then they are sent to validators for parallel execution and validation. Jin et al. [6] also adopt the optimistic concurrency control and rollback mechanism to realize parallel execution in the validator. Nevertheless, in speculative execution methods, for transactions with a high number of data read/write conflicts, frequent rollbacks result in high latency for speculative execution. (2) **Static analysis**. For the static analysis methods, Yu et al. [44] propose a parallel smart contract model, which utilizes static analysis methods to find the common variables to group the transaction. Aelf [47] blockchain groups the transactions based on the mutex of transactions. The mutex of transactions is determined by shared variables in the transactions. Besides, FISCO BCOS [48] blockchain is designed to group the transactions according to a transaction dependency graph, which is also generated by the shared variables. The independent nodes in the graph can be executed in parallel. However, the static methods generally assume that all common variables are bound to generate read/write conflicts, this pessimistic concurrency control leads to inefficiencies. PaVM can narrow the scope of the data conflicts by dynamically recording the data read/write set during the pre-execution of smart contract in the miner.

In addition, other methods are also explored to improve the transaction validation performance. There are some approaches [49]–[51] that focus on blockchain sharding to improve blockchain performance. These approaches shard the blockchain into different areas, which can reduce the complexity of consensus. Moreover, the transactions can be validated in parallel in different areas. However, the performance improvement brought by blockchain sharding is not

thorough enough, because it can only enable inter-contract parallel execution. Besides, sharding the blockchain may raise security concerns. Therefore, PaVM provides the runtime environment and key instructions to realize the function-level parallelism, and then enable intra-contract parallelism. Liu et al. [7] design a paradigm for smart contract execution that uses some nodes to execute contracts in parallel, but this paradigm seriously affects the blockchain security because not every node validates transactions and the blockchain is vulnerable to distributed denial-of-service (DDoS) attacks. Garamvolgyi et al. [52] ealize parallel execution by breaking up the application conflict chains using the technologies such as partitioned counters and commutative instructions. However, the solution cannot make full use of the contract runtime information to resolve the application conflicts precisely. PaVM can record the entire runtime information by instruction design, and provide the information to resolve application conflicts.

In summary, the aforementioned research works primarily focus on the transaction level parallelism, they can not realize thorough acceleration for validation. PaVM presents a runtime system that supports both inter-contract and intra-contract parallelism, thus achieving a significant improvement in transaction validation.

VI. Conclusion

We present PaVM, the first smart contract virtual machine that supports both inter-contract and intra-contract parallel execution to accelerate transaction validation. PaVM facilitates parallel execution by collecting runtime information in fine-grained and enhances the runtime system with thread state and management to improve contract execution efficiency. Experimental results highlight that compared with Geth, PaVM significantly improves the overall transaction validation efficiency by nearly 33.4× on average, and improves the throughput by 46× on maximum.

References

[1] S. Ponnapalli, A. Shah, S. Banerjee, D. Malkhi, A. Tai, V. Chidambaram, and M. Wei, "{RainBlock}: Faster transaction processing in public blockchains," in 2021 USENIX Annual Technical Conference (USENIX ATC 21), 2021, pp. 333–347.

[2] A. A. Zarir, G. A. Oliva, Z. M. Jiang, and A. E. Hassan, "Developing cost-effective blockchain-powered applications: A case study of the gas usage of smart contract transactions in the ethereum blockchain platform," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 30, no. 3, pp. 1–38, 2021.

[3] M. Fang, Z. Zhang, C. Jin, and A. Zhou, "High-performance smart contracts concurrent execution for permissioned blockchain using sgx," in 2021 IEEE 37th International Conference on Data Engineering (ICDE). IEEE, 2021, pp. 1907–1912.

[4] T. Li, Y. Fang, Y. Lu, J. Yang, Z. Jian, Z. Wan, and Y. Li, "Smartvm: A smart contract virtual machine for fast on-chain dnn computations," IEEE Transactions on Parallel and Distributed Systems, 2022.

[5] P. Ruan, T. T. A. Dinh, D. Loghin, M. Zhang, G. Chen, Q. Lin, and B. C. Ooi, "Blockchains vs. distributed databases: dichotomy and fusion," in Proceedings of the 2021 International Conference on Management of Data, 2021, pp. 1504–1517.

[6] C. Jin, S. Pang, X. Qi, Z. Zhang, and A. Zhou, "A high performance concurrency protocol for smart contracts of permissioned blockchain," IEEE Transactions on Knowledge and Data Engineering, vol. 34, no. 11, pp. 5070–5083, 2021.

[7] J. Liu, P. Li, R. Cheng, N. Asokan, and D. Song, "Parallel and asynchronous smart contract execution," IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 5, pp. 1097–1108, 2021.

[8] "The ethereum blockchain explorer." [Online]. Available: https://etherscan.io/

[9] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," in Proceedings of the ACM Symposium on Principles of Distributed Computing, 2017, pp. 303–312.

[10] Z. Chen, H. Zhuo, Q. Xu, X. Qi, C. Zhu, Z. Zhang, C. Jin, A. Zhou, Y. Yan, and H. Zhang, "Schain: a scalable consortium blockchain exploiting intra-and inter-block concurrency," Proceedings of the VLDB Endowment, vol. 14, no. 12, pp. 2799–2802, 2021.

[11] J. M. Faleiro, D. J. Abadi, and J. M. Hellerstein, "High performance transactions via early write visibility," Proceedings of the VLDB Endowment, vol. 10, no. 5, 2017.

[12] Y. Xiang and H. Kim, "Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference," in 2019 IEEE Real-Time Systems Symposium (RTSS). IEEE, 2019, pp. 392–405.

[13] T.-W. Huang, D.-L. Lin, Y. Lin, and C.-X. Lin, "Taskflow: a general-purpose parallel and heterogeneous task programming system," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 41, no. 5, pp. 1448–1452, 2021.

[14] Y. Kim, S. Jeong, K. Jezek, B. Burgstaller, and B. Scholz, "An {Off-The-Chain} execution environment for scalable testing and profiling of smart contracts," in 2021 USENIX Annual Technical Conference (USENIX ATC 21), 2021, pp. 565–579.

[15] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart contract development: Challenges and opportunities," IEEE Transactions on Software Engineering, vol. 47, no. 10, pp. 2084–2106, 2019.

[16] X. T. Lee, A. Khan, S. Sen Gupta, Y. H. Ong, and X. Liu, "Measurements, analyses, and insights on the entire ethereum blockchain network," in Proceedings of The Web Conference 2020, 2020, pp. 155–166.

[17] K. Wüst, S. Matetic, S. Egli, K. Kostiainen, and S. Capkun, "Ace: Asynchronous and concurrent execution of complex smart contracts," in Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 587–600.

[18] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He et al., "Soda: A generic online detection framework for smart contracts." in NDSS, 2020.

[19] T. Lu and L. Peng, "Bpu: A blockchain processing unit for accelerated smart contract execution," in 2020 57th ACM/IEEE Design Automation Conference (DAC). IEEE, 2020, pp. 1–6.

[20] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the laws of order in smart contracts," in Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, 2019, pp. 363–373.

[21] S. Bistarelli, G. Mazzante, M. Micheletti, L. Mostarda, D. Sestili, and F. Tiezzi, "Ethereum smart contracts: Analysis and statistics of their source code and opcodes," Internet of Things, vol. 11, p. 100198, 2020.

[22] B. Pillai, K. Biswas, Z. Hóu, and V. Muthukkumarasamy, "Burn-to-claim: An asset transfer protocol for blockchain interoperability," Computer Networks, vol. 200, p. 108495, 2021.

[23] Q. Wang and R. Li, "A weak consensus algorithm and its application to high-performance blockchain," in IEEE INFOCOM 2021-IEEE Conference on Computer Communications. IEEE, 2021, pp. 1–10.

[24] Y. Zhou, A. N. Manea, W. Hua, J. Wu, W. Zhou, J. Yu, and S. Rahman, "Application of distributed ledger technology in distribution networks," Proceedings of the IEEE, 2022.

[25] M. Ali, J. Nelson, R. Shea, and M. J. Freedman, "Blockstack: A global naming and storage system secured by blockchains," in 2016 USENIX annual technical conference (USENIX ATC 16), 2016, pp. 181–194.

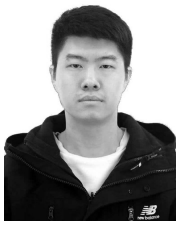
[26] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds," in 2022 IEEE Symposium on Security and Privacy (SP). IEEE, 2022, pp. 161–178.

[27] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," IEEE Transactions on Software Engineering, 2020.

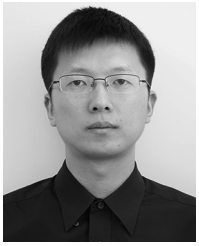
[28] P. Zheng, Z. Zheng, and X. Luo, "Park: accelerating smart contract vulnerability detection via parallel-fork symbolic execution," in Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, pp. 740–751.

[29] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, 2018, pp. 9–16.

[30] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defectchecker: Automated smart contract defect detection by analyzing evm bytecode," IEEE Transactions on Software Engineering, 2021.
[31] O. Farhat, K. Daudjee, and L. Querzoni, "Klink: Progress-aware scheduling for streaming data systems," in Proceedings of the 2021 International Conference on Management of Data, 2021, pp. 485–498.
[32] M. Rodler, W. Li, G. O. Karame, and L. Davi, "{EVMPatch}: Timely and automated patching of ethereum smart contracts," in 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 1289–1306.
[33] J.-Y. Kim, J. Lee, Y. Koo, S. Park, and S.-M. Moon, "Ethanos: efficient bootstrapping for full nodes on account-based blockchain," in Proceedings of the Sixteenth European Conference on Computer Systems, 2021, pp. 99–113.
[34] L. Su, X. Shen, X. Du, X. Liao, X. Wang, L. Xing, and B. Liu, "Evil under the sun: understanding and discovering attacks on ethereum decentralized applications," in 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 1307–1324.
[35] N. He, R. Zhang, H. Wang, L. Wu, X. Luo, Y. Guo, T. Yu, and X. Jiang, "{EOSAFE}: Security analysis of {EOSIO} smart contracts," in 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 1271–1288.
[36] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "ethor: Practical and provably sound static analysis of ethereum smart contracts," in Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 621–640.
[37] M. Chabbi and M. K. Ramanathan, "A study of real-world data races in golang," in Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2022, pp. 474–489.
[38] Z. Liu, S. Xia, Y. Liang, L. Song, and H. Hu, "Who goes first? detecting go concurrency bugs via message reordering," in Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022, pp. 888–902.
[39] P. Veličković, L. Buesing, M. Overlan, R. Pascanu, O. Vinyals, and C. Blundell, "Pointer graph networks," Advances in Neural Information Processing Systems, vol. 33, pp. 2232–2244, 2020.
[40] E. Kafeza, S. J. Ali, I. Kafeza, and H. AlKatheeri, "Legal smart contracts in ethereum block chain: Linking the dots," in 2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW). IEEE, 2020, pp. 18–25.
[41] A. Li, J. A. Choi, and F. Long, "Securing smart contract with runtime validation," in Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020, pp. 438–453.
[42] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in Proceedings of the 2017 ACM international conference on management of data, 2017, pp. 1085–1100.
[43] J. Cheng, S. T. Fleming, Y. T. Chen, J. Anderson, J. Wickerson, and G. A. Constantinides, "Efficient memory arbitration in high-level synthesis from multi-threaded code," IEEE Transactions on Computers, vol. 71, no. 4, pp. 933–946, 2021.
[44] W. Yu, K. Luo, Y. Ding, G. You, and K. Hu, "A parallel smart contract model," in Proceedings of the 2018 International Conference on Machine Learning and Machine Intelligence, 2018, pp. 72–77.
[45] H. Sukhwani, J. M. Martínez, X. Chang, K. S. Trivedi, and A. Rindos, "Performance modeling of pbft consensus process for permissioned blockchain network (hyperledger fabric)," in 2017 IEEE 36th symposium on reliable distributed systems (SRDS). IEEE, 2017, pp. 253–255.
[46] W. Li, C. Feng, L. Zhang, H. Xu, B. Cao, and M. A. Imran, "A scalable multi-layer pbft consensus for blockchain," IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 5, pp. 1146–1160, 2020.
[47] "aelf - a multi-chain parallel computingblockchain framework," 2022. [Online]. Available: https://aelf.com/gridcn/aelf_whitepaper_v1.7_en.pdf
[48] "Financial blockchain open source platform - fisco bcos," 2017. [Online]. Available: https://github.com/FISCO-BCOS/whitepaper/blob/master/FISCO%20BCOS%20Whitepaper(EN).pdf
[49] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A sharded smart contracts platform," arXiv preprint arXiv:1708.03778, 2017.
[50] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in 2018 IEEE symposium on security and privacy (SP). IEEE, 2018, pp. 583–598.
[51] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in Proceedings of the 2019 international conference on management of data, 2019, pp. 123–140.
[52] P. Garamvölgyi, Y. Liu, D. Zhou, F. Long, and M. Wu, "Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts," in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 2315–2326.



Yaozheng Fang is currently working toward his Ph.D. degree in the College of Computer Science, Nankai University. His main research interests include computer system, smart contract architecture, and blockchain system.



Zhiyuan Zhou received the master's degree in computer science from the Chinese Academy of Sciences, Beijing, China, in 2007. He is currently a Staff Engineer with Blockchain Platform Division, Ant Group, Hangzhou, China. He is currently leading a team to deliver the core engine of Antgroup's consortium blockchain system, specifically designed for Internet scale asset exchange. Besides engineering, he is doing research on blockchain scalability. He also had profound experience in cloud computing.

Surong Dai received her B.S. degree in computer science and technology from Nankai University, Tianjin, in 2020. She is currently working toward her Ph.D. degree in the College of Computer Science, Nankai University, Tianjin. Her main research interests include computer architecture, compiler design, and blockchain virtual machine.

Jinni Yang received her B.Eng. degree in Internet of Things from Nankai University in 2020. She is currently studying for a master's degree in computer science in Nankai University. Her main research is in blockchain security.

Hui Zhang (Senior Member, IEEE) received the Ph.D. degree in computer science from the University of Southern California, Los Angeles, CA, USA, in 2005. He is currently the Senior Director of the Blockchain Platform Division, Ant Group, Hangzhou, China, and also the Head of the Blockchain Laboratory, Alibaba Damo Academy, Hangzhou. He is responsible for the R&D and commercialization of Ant Group's blockchain technology. Prior to joining Ant Group, he was the Head of the Department of Systems Research, NEC Laboratories America, Princeton, NJ, USA, focusing on R&D for high-performance distributed systems and networks, especially peer-to-peer network algorithms and big data analytics.

Ye Lu received the B.S. and Ph.D. degree from Nankai University, Tianjin, China in 2010 and 2015, respectively. He is an associate professor and doctoral supervisor at the College of Computer Science, and College of Cyber Science, Nankai University now. His main research interests include computer architecture and FPGA accelerator, virtual machine, embedded system software-hardware co-design.