2023 IEEE International Conference on Blockchain (Blockchain) | 979-8-3503-1929-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/Blockchain60715.2023.00061

SCU: A Hardware Accelerator for Smart Contract Execution

Tao Lu
Electrical and Computer Engineering
Louisiana State University
Baton Rouge, LA, USA
tlu4@lsu.edu

Lu Peng
Department of Computer Science
Tulane University
New Orleans, LA, USA
lpeng3@tulane.edu

Abstract—**Smart contracts and the blockchain have recently been widely used in many application fields. Current smart contracts are executed on general-purpose CPUs and still have a large room to improve performance. In this paper, we first analyze the most popular public blockchain platform Ethereum and characterize smart contracts running on its ecosystem. After identifying its performance limitations, we propose a heterogeneous processor Smart Contract Unit (SCU), which is a hardware-based accelerator in place of the current EVM design. With our proposed novel RISC-style SCU ISA and heterogeneous architecture, SCU can leverage instruction-level parallelism and transaction-level parallelism during smart contract processing and boost its execution performance. Furthermore, SCU can be configured and adapted to different workloads in order to remove bottlenecks. We implement and evaluate the proposed SCU design on a Xilinx FPGA platform. Our design achieves a significant speedup compared to the software implementation on an Intel CPU and runs a few times faster than state-of-the-art design.**

Index Terms—**Blockchains, Smart contracts, Hardware acceleration, Multicore processing**

I. Introduction

Blockchain technology originated from the invention of Bitcoin [13] in 2008 was quickly adopted to build a series of cryptocurrencies in the years after. In 2015, it went beyond just digital currency by adopting the concept of Smart Contract (SC) [8] introduced by the Ethereum [18] project. Smart contracts enable software developers to program sophisticated code and business logic that can be executed and validated in the distributed network. Thanks to the robustness stemming from its decentralized behavior, many businesses have started to utilize blockchain and smart contracts to build their infrastructure, including financial technology (Fintech) [15], Internet of Things (IoT) [14], and supply chain [11]. However, these applications are far from maturity due to blockchain technology's poor performance and scalability.

The blockchain is a chain of blocks recording transactions that can atomically alter the global state. Each block must be first proposed by one peer and then endorsed by all other validator peers to ensure correctness in the distributed network. During both the proposal and validation stages, each transaction in a block must be executed locally by the peer. We call this block processing, and its underperformance has been the bottleneck of the overall system. Several works have been done to address this issue using both software and hardware approaches. Dickerson et al. [9] proposed a method to run transactions concurrently through a heuristic algorithm. Javaid et al. proposed an FPGA-based co-processor to offload the CPU at the network level [10]. Chen et al. [6] proposed a speculative transaction execution approach to improve Ethereum performance. BPU [12] was proposed as an FPGA accelerator with High-Level Synthesis (HLS) to address the EVM execution. In this paper, we are the first to propose a hardware-based heterogeneous architecture and a RISC-V-like Instruction Set Architecture (ISA) to improve the performance at a micro-architecture level and achieve significant improvement.

In this paper, we analyzed the most popular blockchain Ethereum and spotted several interesting observations which led to the motivation for this work. First, we investigated the current execution model, the Ethereum Virtual Machine (EVM), and found opportunities to achieve instruction-level parallelism by using pipelining and dynamic execution. However, in the current design, instructions are based on a stack-based architecture and forced to be executed in order, which limits the performance at the micro-architecture level. We thus proposed RISC-V-like register-based architecture to replace the current design and break the limitation. Second, we look into each block and found that the data dependencies between transactions are not heavy, which leaves room for transaction-level parallelism. Independent transactions can be executed in parallel by utilizing multiple cores and assigned smartly through a scheduling algorithm. The results are validated in order to guarantee the final state is correct. Third, after checking the block history, we found several smart contracts are always called more frequently than others in a short period, which we call hotspots. Hotspots exist because a popular project in the real world can always draw a lot of attention during a short period, typically around its launch time or commercial advertisements and promotions. This pattern suggests that an application-based accelerator is beneficial. Last, in the second half of 2022, the Ethereum network had a major update: the adoption of the new Proof-of-Stake (PoS) consensus and the Beacon chain [1]. After this update, the execution layer was separated from the consensus layer, reducing the consensus time significantly. Therefore, the execution layer has been the

979-8-3503-1929-3/23/$31.00 ©2023 IEEE
DOI 10.1109/Blockchain60715.2023.00061

bottleneck for the overall performance, and it is decoupled from the consensus, which strengthens our motivation above to explore the better performance of the block processing.

Our contributions can be summarized as follows:

- We perform a detailed analysis of the Ethereum history record and discover the trend of smart contracts and the architectural level limitations of the current execution model (EVM). Such observation is used to guide our architectural design below.
- We propose a RISC-V-like ISA [2] that enables a dynamically scheduled pipeline that exploits instruction-level parallelism (ILP) and a heterogeneous multicore that supports thread-level parallelism (TLP) for SC execution.
- We implemented SCU on an FPGA platform and evaluated the performance against state-of-the-art works. The result shows a significant performance improvement compared to state-of-the-art implementations and an Intel CPU.

II. Background and Motivation

In this section, we introduce the background and our observations that lead to our motivations.

A. Ethereum and Block Processing

Ethereum is a blockchain-based platform that supports the creation of decentralized applications (DApps) and smart contracts. Initially, Ethereum relied on a proof-of-work (PoW) consensus algorithm to validate transactions and add new blocks to the blockchain. However, this approach had several drawbacks, including high energy consumption and a potential centralization of mining power. To address these issues, Ethereum is in the process of upgrading to a proof-of-stake (PoS) consensus algorithm known as Ethereum 2.0 [1]. With PoS, rather than relying on miners to validate transactions, the network relies on validators who hold a certain amount of Ethereum as collateral. These validators are incentivized to act honestly and secure the network, as they risk losing their collateral if they act maliciously. The PoS upgrade is expected to significantly reduce energy consumption and increase transaction throughput, making Ethereum more scalable and efficient.

Ethereum's execution layer is part of the platform that executes smart contracts and processes transactions. The execution layer operates on top of the Ethereum Virtual Machine (EVM), which is a software environment designed to execute code written in the Solidity programming language. When a user initiates a transaction on the Ethereum network, the transaction is broadcast and processed by nodes (computers) participating in the network. The transaction is then validated and included in a block, along with other validated transactions. Once the block is added to the blockchain, the execution layer processes the transactions within the block, executing the smart contracts and updating the global state of the network. This process involves running the code contained within each contract, updating the state of the network accordingly, and recording the results on the blockchain. The whole procedure is known as block processing, and we find its performance issues that need to be addressed.

B. EVM ISA and Execution Model

In the Ethereum design, smart contracts are coded using high-level programming language and compiled into a low-level bytecode according to the Instruction Set Architecture (ISA) defined in the Ethereum Yellow Paper [18]. This ISA is used by the execution model, the EVM, which executes the Ethereum SC and updates the database. The current EVM uses a simple stack-based architecture with additional memory and storage buffer. The input to the EVM has two main parts. One is the transaction going to be processed, and the other is the state of the current database, including the code and the storage of the target account. As we introduced above, EVM will create a local copy of the target account with the code and storage field. Then it will fetch the bytecodes from the code buffer and execute them one by one. The result of each operation will be pushed into the 256-bits integer stack and accessed by the following operations. A large chunk of data could be saved into the Memory buffer, which is a byte array. The data in Memory are temporary and will be discarded when the execution is finished. On the other hand, any changes intending to be updated to the global state database will be first applied to the local storage copy and only update to the global when EVM finishes execution successfully. If any exceptions happen during the execution, the current local storage will be discarded, and nothing will be changed.

We analyze the execution model of the current EVM architecture to identify performance challenges. Figure 1 shows the data flow and the logic of the EVM design. The processing of a valid transaction starts from the up-left step 1. EVM will first get the target address from the transaction and search the state database for the target account. Then in step 2, EVM needs to check whether the target account has occupied the code field. If the code field of the target account is empty, EVM will discover that the transaction is a token transfer transaction and thus will change the balance directly (go to step 3, after checking the legality) and then update State DB (step 4). Alternatively, if the target account is a smart contract, the EVM will set up a local copy using fetched code and storage in step 5. Then it will enter the execution loop of 6, 7, 9, 10. During the execution, any exceptional halt (such as unknown bytecode or invalid jump destination) will result in a termination immediately and the changes will be reverted (steps 8, 11). The execution will continue until a STOP or RETURN operation is encountered and EVM will exit normally (step 12). Then EVM will start to calculate the gas (step 13), which is a small amount of fee charged for each computation on the Ethereum blockchain. If the provided gas in the transaction is enough (step 14), it will be consumed, and a receipt will be generated (step 16). At last, EVM will exit to finalize the state changes (step 4). Alternatively, an insufficient fund will result in an Out-of-Gas error, and reversion of any changes has been made (steps 15, 11).

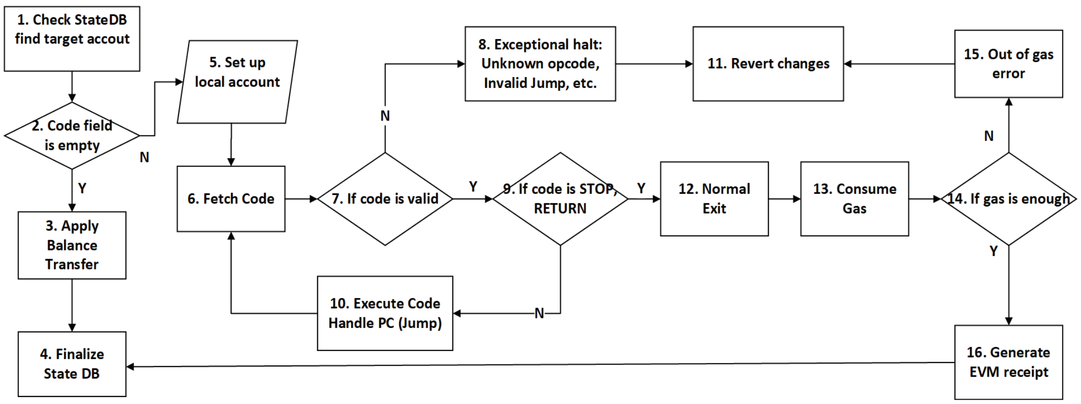


Fig. 1. EVM Data Flow

C. Limitations of the EVM

Based on the above analysis, the current EVM design has several limitations that prevent it from achieving good performance.

1. Lack of parallelism and slow speed: The current EVM implementation is based on a software virtual machine that bytecodes are processed one by one in a fetch, execute, and finalize loop, which means little parallelism is employed in this procedure. Moreover, the nature of software virtual machines makes them run very slowly. According to our measurement, the CPU utilization rate is usually around 10%.

2. Stack-based model: The current EVM design is using an antiqued stack-based model, which means all the intermediate results must be pushed into the stack before they can be reused. One instruction will read its operand from the stack only when it is being executed, which stops the parallelism. In addition, when an instruction reads its operands from the stack, it must use the value at the top of the stack. This brings up two main issues. Firstly, if we want to read an operand and also save it for future use, the compiler must insert DUP instructions to create a copy. This brings unnecessary delays just for preparing the operands. Secondly, when we want to reuse a deep value, the compiler must add swap or DUP operations to move the deep value to the top or create a copy at the top of the stack; this also brings additional control operations and hurts the performance.

3. Variable length instruction: The Ethereum bytecode mixes the instruction and the value of the PUSH operation together. This results in the fact that instructions can have variable lengths. Therefore, Ethereum bytecodes cannot be parsed after being fetched until the length of each instruction is determined. This generates difficulties in directly applying the conventional pipeline technique, which needs to fetch instructions for each cycle.

These limitations lead to our proposed work, which will be discussed in the following sections.

III. Hardware Architecture Design

To overcome the challenges spotted above, we propose an innovative architecture for smart contract processing, *the Smart Contract Unit (SCU)*, where we make three major improvements. First, we propose a register-based RISC ISA, which can be translated from the stack-based bytecode seamlessly at runtime. Second, based on our novel ISA, we implement instruction-level parallelism using dynamic pipelining, register renaming, and in-order commitment. Last, we propose a heterogeneous architecture to combine the configurability and flexibility of our design to suit different applications and further improve performance. In the following sections, we present our design details for these improvements.

A. Instruction-level Parallelism

EVM is implemented using a bytecode format and stack-based machine because it results in a smaller size of the code. Since SC code must be stored on the blockchain forever, the cost of storage is considered huge. On the other hand, the stack-based architecture is not efficient in exploiting parallelism. Therefore, not Ethereum specifically, for all the blockchain systems, there is a need to translate a storage-friendly ISA on-chain to a performance-friendly ISA off-chain. In this paper, we propose a RISC-style ISA to bridge the gap.

1) Register-based RISC: As mentioned, the original Ethereum stack-based ISA has limitations that stop instruction-level parallelism, such as dynamic execution. Therefore, we propose SCU ISA, which is a RISC-style register-based instruction set. It is compatible with the current Ethereum's bytecode ISA when an interpreter is introduced. Compared to CISC ISA (like x86), RISC, with its streamlined, efficient instruction set and simpler, more power-effective hardware design, aligns well with the growing demands for energy efficiency and high performance in mobile and embedded systems. Its adaptability and straightforward design, exemplified by architectures like ARM and open-standard RISC-V, are setting the trend, offering a potentially superior approach in modern computing scenarios where simplicity, scalability, and power efficiency are paramount.

We use 32 256-bit registers to replace the current operator stack. The width of 256 is determined because it is the length of the hash value used in Ethereum and the defined longest value in Ethereum's overall design. The number of registers is also chosen carefully. Although Ethereum's original stack is 1000 items deep, it can rarely be used more than 32. This is because the design rationale of Ethereum and the existence of gas discourage developers from deploying complex calculations on Ethereum because they are not financially efficient (fees are high). Based on our observation of the history of smart contract execution, using a total of 32 registers is sufficient and redundant. If in any rare case, these architectural registers are not enough, the system will just wait until occupied registers are released. This will only not hurt the correctness nor the security of our system.

2) SCU Execution Engine: The design above is implemented by a hardware architecture, which we call SCU Execution Engine. Figure 2 shows the architecture details and its companion Input Buffer. The input buffer holds every information needed to execute a transaction, including the transaction, the code, and the input set. On the right is the SCU execution engine, which is in charge of executing the code and producing the result state. The execution engine is equipped with a series of submodules to implement the Out-of-Order (OoO) pipeline. Each instruction is handled by a variety of execution units (EU) such as Adder, Logic, etc. The number of each type is flexible, and we divide them into two groups: the basic EU and the Configurable EU. The basic EU has one unit of each type so that it is fully functional and capable of handling any smart contracts. The configurable EU has the extra resources that enable more instructions to run in parallel, and this part is explained in the next subsection.

In Figure 2 we illustrate the submodules and the seven pipeline stages marked by colored arrows. Step 1 is to fetch the code from the instruction buffer into an instruction queue (IQ) entry with a window size of 32. In step 2, the decoder will decode the bytecode and translate the instruction into our SCU ISA instructions. Then in step 3, when the instructions are issued to the reservation station, a register renaming approach will be applied to further remove unnecessary register dependencies. When the required operands and execution units are ready, the reservation station dispatches the awaiting instruction to its proper target execution unit, as step 4 indicates. Then in step 5, each execution unit will operate the corresponding executions, and the result will be written on the common data bus (CDB) to update the reservation station and the reorder buffer (ROB). Since the execution sequence in steps 2-5 can be out-of-order, we use an in-order commit to finalize the changes to the register file, memory, and the local storage buffer in step 6. Finally, the last step is to generate the EVM receipt and write it back to the local storage, which is used to generate the output set at the end of execution.

3) Configurable Units: As we mentioned in the smart contract characterization, there exist some frequently used hotspot SCs. Therefore, it would be beneficial if we could have customized optimization on these specific smart contracts. We propose configurable units, where we can equip the SCU execution engine with more extra units to avoid potential structure hazards and further boost the performance, as the bold box shown in Figure 2.

To make our design functional, using at least one of each unit introduced above is necessary. With the help of additional resources, more instructions can be executed simultaneously and improve performance. However, there are situations where extra resources are not worth the cost. For example, the additional multiplier may not be beneficial because, in the current smart contract ecosystem, the appearance of consecutive multiplication is very rare. It means when the previous multiplication is done, and typically, the later one has not been issued yet. Therefore, the second multiplier will rarely be used. Furthermore, data dependency will also result in wasting additional resources. For example, if a division follows an exponential operation, it cannot be executed in two multipliers concurrently due to data dependency. Essentially, adding more execution units may be helpful but not always good. Therefore, we designed our SCU as a configurable design on FPGA that can adjust a number of execution units to suit different tasks. Another reason that we are targeting the FPGA platform is that it allows protocol changes and is cheaper compared to ASIC. ASIC could be faster but will fix the circuit, and it is much more expensive when adopted at the early stage. Given the fact that the blockchain industry is rapidly changing, we believe the FPGA solution is better in terms of balancing performance and cost.

We understand the golden rule is to maximize overall performance with minimum overhead. However, determining the best configuration for different benchmarks is non-trivial. To explore the design space, we follow a heuristic approach to get the numbers of extra functional units. We first profile the target smart contract and get its instruction breakdown, which is the proportion of each type of our ISA. Next, we allocate extra functional units according to the popularity of each type. The often-used instruction type will be paired with more execution units. Then we trace how the new configuration works by monitoring whether the additional resource is used. If the utilization rate is less than the ratio of hardware cost increase, we stop adding units. This is a simpler method compared to a comprehensive design space exploration where we can find the Pareto Optimality [17]. In this case, a software simulator needs to be developed to exhaustively search all possible configurations. Our method needs relatively less work but can still achieve a good performance improvement, which will be shown in section 6.

B. Transaction-level Parallelism

In Ethereum design, transactions must be executed in the order of their appearance in the block. This is because the potential data dependency between transactions will be broken if they are executed in the wrong order. However, running multiple transactions in parallel is still achievable. We propose two techniques: two-step validation and heterogeneous multicore design to address this issue.

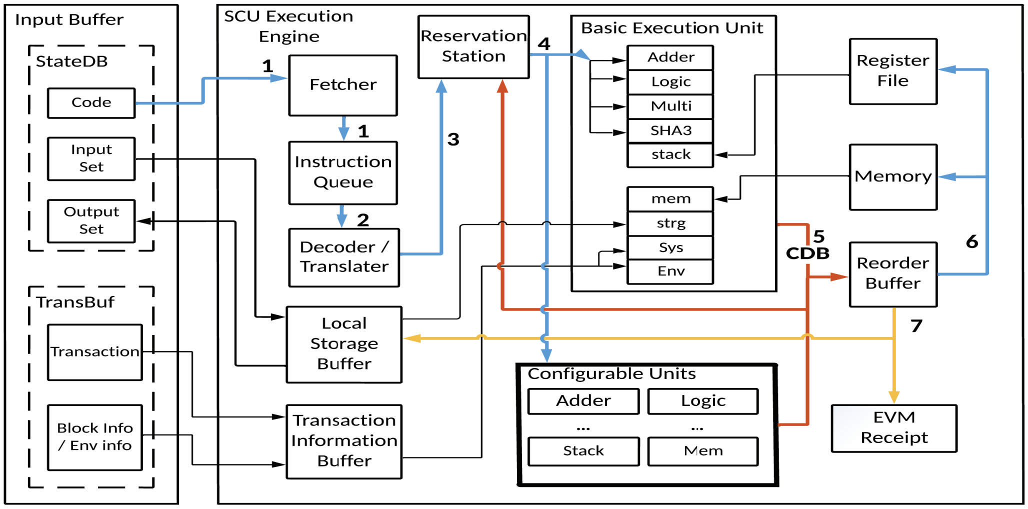


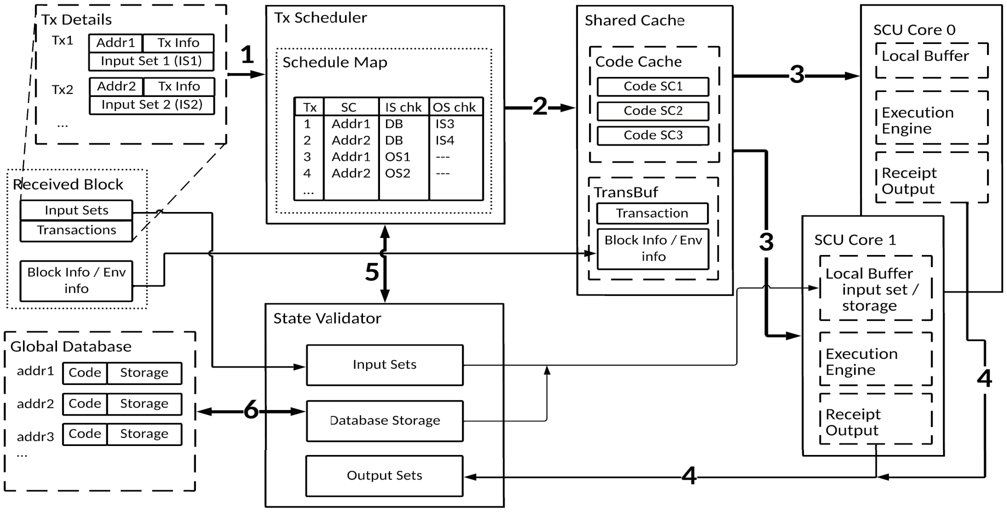
Fig. 2. Micro-architecture of SCU Core

1) Two-step Validation: Normally, a transaction will only affect the storage field of their target smart contract and have no influence on the other ones. Therefore, data dependency only exists between transactions sent to the same target smart contract. In such cases, the sequence of their execution is deterministic since the resulting state of the leading transaction will be used as the input state of the followers. Thus, we define the two-step validation that separates the execution from the validation of the state and these two steps can run in parallel.

We divide the whole validation process into *execution validation* and *state validation*. The *execution validation* is to check the correctness of the EVM execution result, given the transaction and its input state. The input state is the state of the storage field of the target smart contract, which is a key-value map that stores the permanent storage of the application. Normally, each transaction will query a series of data from the local database. We define the *input set* (IS) as the key-value pairs that are queried by each transaction. Such IS can be monitored by tracking the storage reading operation SLOAD. Similarly, the *output set* (OS) is defined as the key-value pairs that get modified during the execution. Such information is collected by tracking the storage access operations (SSTORE). The IS can be collected by the miner and broadcast to validation peers along with the block so that we know which key values are used for a specific transaction. On the receiver side, we load the key-value pairs to the local buffer using the received IS so that transactions can run before the state of its predecessor is locally reproduced, which allows transaction-level parallelism.

On the other hand, the *state validation* is to check the correctness of the input set and the output set. Both sets need to reflect the real state of the global database, which is checked by the state validator module. Only if the input set and output set are both valid, the state is considered valid. Meanwhile, for a specific transaction, only if both its state validation and execution validation pass, the execution of this transaction is considered to be successful, and the result is finalized. Since we divide the whole process into two isolated phases and run them in parallel, we call this approach the two-step validation. This approach is an efficient correctness check at the execution layer, which is not an enhanced security feature. Neither does this approach bring more security or privacy concerns, nor does this strengthen the existing security at the consensus layer.

2) Design Overview and Transaction-level Parallelism: Figure 3 shows the SCU design overview and the TLP dataflow. Upon receiving a new block, the transaction scheduler will scan the transactions and schedule the sequence of their execution, as arrow 1 indicated. Then the scheduler will load the shared cache with SC's code(flow 2). The shared cache is designed with a content-aware approach to increase the hit ratio. Since the SCU engine works as a validator, the miner can broadcast hotspot SCs to all validators. We prioritize the hotspot SCs' code to stay in the cache instead of using the conventional least recently used (LRU) algorithm. Next, the scheduler dispatches the validation task to the available SCU core (flow 3). These cores are equipped with a local buffer and the SCU Engine, which is introduced in III-A2. Detailed architecture of the input buffer and Execution Engine are illustrated in Figure 2. In this example, we present a setup of two cores, which could be either big cores or small cores. Therefore, the scheduler will load the shared cache with the target SC's code section and forward the transaction to the two engines separately. Upon receiving the transaction, SCU engines load its local buffer with the input and storage section, fetch the code from the shared memory, and start its execution. When the task is completed, the output set will be written to the state validator for correctness checking (flow 4). At the



Tx	SC	IS chk	OS chk
1	Addr1	DB	IS3
2	Addr2	DB	IS4
3	Addr1	OS1	---
4	Addr2	OS2	---
...			

Fig. 3. SCU Design Overview and Heterogeneous Multicore Design

same time as the above flow, the state validator will use the schedule map from the transaction scheduler (via flow 5) and access the global database (flow 6 right) to perform the state validation. After the validator finishes its job, an update to the global database will finalize the processing of the current block, as the left arrow of flow 6 shows.

TLP is exploited using the Tx scheduler, the state validator, and multiple SCU engines. As mentioned before, the dependencies between transactions are detected, and a schedule map is generated following the format in Figure 3. The Tx column shows the index of transactions. The SC column lists the address to fetch the SC code. Then the IS_chk and OS_chk columns indicate which OS or IS should be checked in the state validation respectively. As we can see in this example, Tx1 has no predecessor so IS1 must be checked against the DB. Tx3 is dependent on the result of Tx1 so IS3 must be checked against OS1. In other words, IS3 and OS3 should match, and such correctness checking is done by the state validator. Meanwhile, there are no dependencies between Tx1 and Tx2 since Addr1 and Addr2 are different. Therefore, Tx1 and Tx2 can be assigned to Engine0 and Engine1 in parallel. The Engines will use the received IS1 and IS2 to start the execution validation without waiting for the state validation between IS1 and DB.

3) Heterogeneous Multicore Design: To achieve better performance, we propose two types of cores, which we call big and small. A big core is equipped with an dynamic execution engine and focuses on the best performance with a cost of more hardware resources. A small core is equipped with an in-order pipeline engine, which consumes fewer hardware resources but is more efficient. With these two cores in hand, our accelerator can be configured in a homogeneous design, which uses several cores of the same type, or a heterogeneous design, which uses a mixed type of cores. In this paper, we coded the design with the number and the type of the cores. For example, our SCU_2B design is equipped with two big cores, and our SCU_1B2S design is equipped with one big core and two small cores.

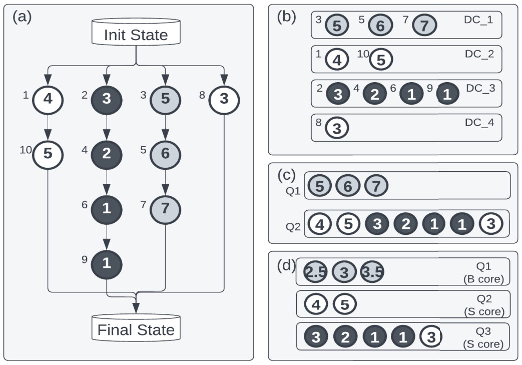


Fig. 4. Task Scheduling with Dependency

For multi-core design, the challenge is how to schedule transactions into different cores. There are several techniques that can be used in multi-processor scheduling, such as [16], [20], and [7]. Our approach is based on a dependency graph as shown on the left of Figure 4 (a). This graph is generated by gathering the transaction dependency and the execution time of each transaction. The dependency can be extracted by checking the target address and the index of each transaction. The execution time can be broadcasted by the miner to all validators, and this information can be used to schedule the task more efficiently. This is the unique advantage of blockchain validation with a known transaction execution time from the miner. In Figure 4, each node stands for a transaction, and its index is labeled on the left shoulder.

The arrows between nodes illustrate the dependency between transactions and the transactions calling the same SC forms a dependency chain (DC). The number in the node denotes the execution time of the transaction. We use this information to schedule transactions to the cores. Algorithm 1 elaborates the scheduling strategy. We start by initializing an empty queue for each core and generating the dependency graph described above. Given the graph, we then calculate the total weight of each DC by adding up the execution time along the chain and sorting all chains in descending weight order. Then starting with the heaviest unassigned DC, we assign it to the lightest queue. The weight of the queue is the total weight of all assigned tasks in it. Later, when the execution starts, each core will pop a transaction from its own queue. Note that we are trying to balance the load between cores to minimize the idle time. However, they do not need to be perfectly balanced. When a core's queue is empty, it is allowed to steal transactions from other cores' queues, red which means queued DC can be moved from the busy core to the idle core. In this way, the execution time of each queue will likely be very close. Furthermore, this algorithm works well naturally for symmetric designs since all the cores are the same, and the execution time is consistent among cores. However, for asymmetric or heterogeneous designs, the execution times are different between big and small cores. Therefore a correction parameter can be used to adjust the time to ensure that the total weight is relatively close between cores. For example, if the big core has a speedup of x over the small core, the weight of a DC on the small cores will be x times the one on the big cores because the small cores are slower.

Algorithm 1: Transaction Scheduling for Multicores

```
Initializing Queues for each core: Q1, Q2, ...;
Generating dependency graph for the block;
Calculating the weight of each chain using its
  execution time from the miner;
Sorting the chains in the order of descending weight ;
while there is unassigned chain Ci do
    calculating the weight of each queue (Q1, Q2...);
    adjusting the weight if a heterogeneous design;
    assigning Ci to the Qk, which has the minimal
      weight;
end
```

Figure 4 shows an example of our scheduling algorithm. Group (a) illustrates the dependency tree of an example block and the execution time is normalized. Given this graph, we can calculate the weight of each dependency chain (DC) and rank them in descending order as shown in group (b). Then starting from DC_1, we put each chain as a whole into the lightest queue. This algorithm works for both homogeneous and heterogeneous designs with any number of cores. Group (c) and group (d) show the scheduling result of a homogeneous duo-core design and a heterogeneous 1B2S design respectively. Note that for the duo-core design (Group c), we fill Q1 and Q2 with DC_1 and DC_2 first. Then DC_3 will be put in Q2 because the weight of Q2 is the smaller one at this time. At last, DC_4 will go to Q2 by comparing the total weight of each queue. Group (d) shows the scenario of a heterogeneous design where we have one Big core and two Small cores. In this example, Q1 belongs to the big core and we assume a big core is twice faster than the small core, thus we have an adjustment parameter of 0.5. When the DC_1 is scheduled to Q1 on B core, we saw the weight is adjusted to half to reflect the speedup of the big cores. In this case, the first three DCs will go to each idle core, which results in the total weight of each queue being 9, 9, and 7 respectively. Therefore the last task DC_4 will go to the lightest Q3, even though the Core_1 (big core) doubles the performance.

IV. Evaluation

To evaluate the effectiveness of SCU architecture, we implemented our design on an FPGA platform. In this section, we introduce our experimental setups and then display the result of performance speedups.

A. Experimental Setup

1) Benchmarks: To test the improvement from ILP and OoO execution, we use some real blocks to evaluate our design. For the block and transaction information, the data was retrieved from the Etherscan.io [3] website via API, which is an Ethereum history browser. For the database data, we reproduce the state before execution based on the result from tracing transactions. In order to be accessible by FPGA, we developed a Python tool to interpret the raw data from JSON format into the FPGA readable format. Then the data is loaded from the host machine to the FPGA via PCIe using Xilinx XRT tools.

2) Platform: We implement the SCU on Xilinx's latest data center level FPGA Alveo 250 [19], which has an A-U250-P64G-PQ-G FPGA on board. We design the pipeline components using Verilog and System Verilog at the RTL level. The design is packaged into a Vitis kernel to be compatible with the Au250 board. At last, the kernel is loaded to the FPGA board connected via PCIe, which is clocked by an optimized frequency at 300MHz. On the other hand, we run the software baseline for performance test on an Intel i7-7700k [4] quad-core CPU at 4.2GHz.

3) Architecture Setup: To test the speedup contribution from TLP and heterogeneous architecture, we test and compare the results between five different setups. As Table I shows, SCU_1B and SCU_1S are respectively equipped with one big core (dynamic core) and one small core (in-order core). The big core is equipped with configurable execution units and a 128-slot ROB to support the seven-stage pipeline as introduced before. The small core is equipped with only the basic execution units and follows a four-stage in-order pipeline: fetch, decode, execute, and writeback. Therefore, by comparing these two cores, we can show the single-core performance and the contribution of dynamic execution of the big cores. Similarly, SCU_2B and SCU_2S are equipped with

TABLE I
EXECUTION TIME OF SCU BENCHMARKS

Bench	Block	# of Txs	CPU (us)	BPU (us)	SCU_1S (us)	SCU_1B (us)	SCU_2S (us)	SCU_2B (us)	SCU_1B2S (us)
B1	6653186	190	66040	1708.4	913.24	475.81	546.1	364.59	318.51
B2	6653197	102	56669	1410.9	791.32	356.77	466.7	335.48	301.68
B3	6653232	115	57129	1129.9	813.59	372.94	656.2	290.73	235.76
B4	6653208	78	37037	1410.9	647.9	358.89	407.4	261.99	243.73
B5	6653220	9	4184	165.7	99.74	83.54	70.1	73.91	52.21
B6	6653205	159	46092	1595.8	1112.88	545.92	735.5	431.62	372.01
B7	6653209	16	3696	218.4	115.36	64.63	83.9	64.61	64.65

two identical big and small cores, respectively, which shows the contribution of the TLP by distributing tasks among two identical cores. Lastly, we add a heterogeneous architecture SCU_1B2S, which is equipped with one big core and two small cores. The number of small cores is two because the hardware resource used by two small cores are close to one big core. Therefore, the total cost of 2B is close to 1B2S. Comparing these two groups shows whether the trade-off between big and small cores makes a difference.

TABLE II
HARDWARE RESULTS OF THE DESIGNS

Configs	FPGA utility				Freq.
	DSP	FF	LUT	BRAM18	MHz
BPU	402	67646	82246	25	100
SCU_1S	225	9799	142816	206	300
SCU_1B	405	78163	204448	206	300
SCU_2S	450	20687	301501	378	300
SCU_2B	810	165011	431612	378	300
SCU_1B2S	855	98850	505947	422	300

B. Results

Table I records the execution time of all the benchmarks run by native CPU implementation Geth, the previous BPU implementation, and five SCU versions. The CPU result is measured by running the original Geth (v1.9.26) [5] implementation multiple times and recording the average execution time. The BPU design implemented an in-order pipeline without a heterogeneous design and the result is reproduced using the same setups as introduced in [12]. The rest of the table reflects our latest implementation of SCU and its heterogeneous configurations. SCU_1S has one small core, which is the in-order pipelined execution engine, and SCU_1B has one big core, which is the dynamic pipelined execution engine. As the name suggests, a small core consumes fewer hardware resources and area, and a big core provides better performance, benefiting from the OoO techniques. All measured time has a unit of microseconds. In addition, SCU_2S and SCU_2B are recorded to show the duo-core speedup against its single-core version. Lastly, the SCU_1B2S version is compared to show the advantage of the heterogeneous design.

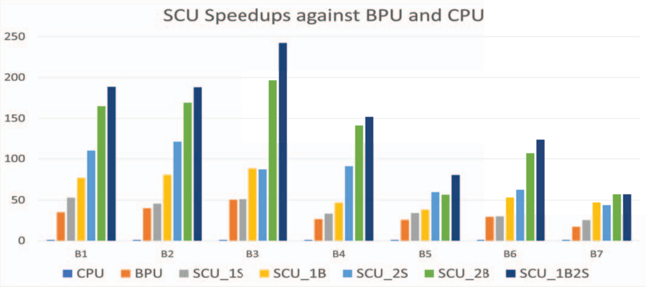


Fig. 5. SCU Performance Speedups

Figure 5 shows the speedup of our latest version with various setups against the original CPU and BPU implementation. The CPU performance is standardized as one, and the other bars stand for the speedup against the CPU version. First, we can examine the speed-up of our single core version SCU_1S and SCU_1B. As we can see, the SCU_1S performs better than BPU, although they are both using the in-order pipeline. This is because we optimized SCU_1S with a higher clock frequency and a more precise pipeline stage control using Verilog against the BPU's implementation by High-level synthesis (HLS), which limited its frequency optimization. The SCU_1B has notably better performance against the BPU due to the benefit of dynamic execution. The improvement is insignificant for some benchmarks (such as B5). This is because not all transactions benefit from the out-of-order setting. When the transaction has many short operations, it gains a relatively lower performance boost. The OoO setting is beneficial when time-consuming instructions can be bypassed without having to wait for the result.

Then we check the speedup of TLP by comparing SCU_2S against SCU_1S. We can see the duo-core over single-core is between 1.4 and 1.7. It is clear that the extra core is fully used and contributes to the speedup. The speed cannot be doubled because a critical path exists that dependent transactions must be executed in order. In some cases, a long dependency chain on one core will result in a second core being idle, which limits the speedup to 1.4. Then we can compare SCU_2B against SCU_1B, and we can see the speed up is about 1.3x. This is because the big core is much faster than the small one, so when two big cores work in parallel, the cache loading becomes the bottleneck. By checking the use of a global queue and shared cache, we see a high cache miss rate, and cores often wait for the cache to be loaded. Finally, we compare the speedup between SCU_1B2S and SCU_2B. We can observe an average 15% speedup even though the hardware resource used is similar. This is due to the help of the extra number of small cores and the critical path scheduling. We force the critical path to be executed by the big core and offload other easy transactions to the less powerful small cores. The reason for B7 and B4, where improvement is not significant, is that the total number of transactions is relatively small, and one of the smart contracts is large, so the cache loading dominates the processing time. The resource usage and frequency of the FPGA designs are listed in Table II.

V. Conclusion

In this paper, we explore the current smart contract behavior on the Ethereum platform by performing a detailed analysis of the public history and identifying its performance limitations. Based on the observations, we propose high-performance architecture designs SCU for smart contract processing. This design relies on our proposed novel RISC ISA to implement instruction-level parallelism. We have applied dynamic scheduling, Register Renaming, and heterogeneous multicores to improve performance. Meanwhile, we propose the two-step validation algorithm to decouple data dependency and achieve transaction-level parallelism. Finally, we evaluate the SCU design on a Xilinx FPGA platform. As we have shown, the SCU design has a significant speedup compared to state-of-the-art implementations and an even more substantial improvement compared to the current processing of the Ethereum client on the CPU. Finally, SCU's modularized and configurable design retains the flexibility to be tuned for different ecosystems or blockchains.

References

[1] https://ethereum.org/.

[2] https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[3] https://etherscan.io/.

[4] https://ark.intel.com/content/www/us/en/ark/products/97129/intel-core-i7-7700k-processor-8m-cache-up-to-4-50-ghz.html.

[5] https://geth.ethereum.org/docs/.

[6] Y. Chen, Z. Guo, R. Li, S. Chen, L. Zhou, Y. Zhou, and X. Zhang, "Forerunner: Constraint-based speculative transaction execution for ethereum," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 570–587.

[7] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero, "Criticality-aware dynamic task scheduling for heterogeneous architectures," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 329–338.

[8] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: foundations, design landscape and research directions," *arXiv preprint arXiv:1608.00771*, 2016.

[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*. ACM, 2017, pp. 303–312.

[10] H. Javaid, J. Yang, N. Santoso, M. Upadhyay, S. Mohan, C. Hu, and G. Brebner, "Blockchain machine: A network-attached hardware accelerator for hyperledger fabric," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2022, pp. 258–268.

[11] K. Korpela, J. Hallikas, and T. Dahlberg, "Digital supply chain transformation toward blockchain integration," in *proceedings of the 50th Hawaii international conference on system sciences*, 2017.

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

[13] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[14] M. Samaniego and R. Deters, "Blockchain as a service for iot," in *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 2016, pp. 433–436.

[15] S. Singh and N. Singh, "Blockchain: Future of financial and cyber security," in *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*. IEEE, 2016, pp. 463–467.

[16] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 213–224, 2012.

[17] T. Wilkerson, *Advanced Economic Theory*. Scientific e-Resources, 2018.

[18] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[19] Xilinx, https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html.

[20] Y. Zhang, L. Duan, B. Li, L. Peng, and S. Sadagopan, "Energy efficient job scheduling in single-isa heterogeneous chip-multiprocessors," in *Fifteenth International Symposium on Quality Electronic Design*. IEEE, 2014, pp. 660–666.