BPU: A Blockchain Processing Unit for Accelerated Smart Contract Execution

Tao Lu and Lu Peng
Division of Electrical & Computer Engineering, Louisiana State University

Abstract—**Modern blockchains use smart contracts to implement automatic and decentralized programs, which are the foundations of Decentralized Applications (DApp). The poor performance on general purpose computers has become the bottleneck that limits the blockchain and smart contracts from being widely used. In this paper, we present BPU, a high-performance modularized blockchain processing unit. BPU aims at bringing performance and flexibility to the blockchain and DApp processing. Our design achieves significant speedup compared against the software implementation on an Intel CPU.**

Index Terms—**Blockchain, Transaction Processing, Smart Contract, Accelerator**

I. Introduction

Blockchain is a cutting-edge technology that was first introduced along with the invention of Bitcoin [1]. Ever since then, the unique features of its decentralization [2] and anonymity [3] have drawn much attention from both academia and industry. Soon after, the concept of Smart Contract (SC) was introduced by another blockchain project Ethereum [4] that enabled blockchain to process more sophisticated functions and logic in an automatic and decentralized manner. Due to this add-on feature, blockchain technology gained the potential to be applied in well-expanded fields such as supply chain [5], Internet of Things (IoT) [6] and Financial Industries [7].

Smart contracts [8] are the executable code stored on blockchain that are programmed to be executed automatically when triggered. To trigger the functions in SC, users need to broadcast a transaction to the network targeting the contract account. In blockchain, these transactions are collected by all the participating nodes and executed locally in a distributed manner. This procedure is called the transaction processing.

Transaction processing is very time consuming and inevitable for all full nodes in blockchain. When a new node joins the network, it needs to sync all the blockchain history from peers and re-exeute all the transactions locally in order to build up its own history copy. This syncing process usually takes several weeks even on a well equipped workstation with high-end CPU, SSD and gigabit ethernet. On the other hand, once the synchronization is done, the node continues to process new blocks received in the network as a maintainer. At this time, its poor performance on processing transactions also limits the throughput of the whole system, which prevents blockchain from replacing current established centralized systems.

Researchers have been exploring ways to improve the processing performance of blockchain. Until now, many works have been done at the software or the protocol level [9]–[12]. Meanwhile, hardware approaches have also been proposed to boost blockchain performance. However, most of the efforts are limited on improving the mining performance by using ASIC or FPGA [13], [14]. Hardware accelerators for time-consuming blockchain transaction processing remain as a blank. To our knowledge, this is the first work to explore the architectural design for smart contract execution and blockchain processing.

In this paper, we present BPU, a modularized processor for smart contract execution and blockchain processing. BPU out-performs the current software implementation on CPU with the help from architectural optimization. We then measure the performance of BPU's submodules and study the SC execution pattern by analyzing the operation frequencies using real-case benchmarks. Finally, we implement and evaluate our design on a Xilinx's FPGA platform.

Our contributions can be summarized as follows:

- We proposed a modularized architecture using pipeline optimization to improve the performance of blockchain processing, especially the SCs execution.
- We adopted application-oriented optimization to design co-processors, and implement an ERC20Core for Ethereum, which gains up to x191 speedup compared to Intel i7-7700k CPU.
- We applied detailed analysis on the execution trace in order to understand the pattern of blockchain execution.
- We implemented our design on a Xilinx's Zynq-7000 platform and evaluated the hardware utilization and performance.

II. Background and motivation

A. Block Processing and its Challenge

The blockchain can be viewed as a distributed ledger which records the consensus states of all accounts. The transactions can be viewed as changes to these states. Block processing is also the procedure of applying these state changes when appending a new block to the current blockchain. For example, a token transfer transaction will change the account balance and a SC transaction will modify the state of the contract storage. Compared to token transfer transactions, smart contract transactions are much slower and more computationally intensive. Meanwhile, the proportion of smart contracts in a block is increasing as the time goes. Inevitabley, more and more DApps will be built as blockchain technology becomes more popular and technically capable. However, current blockchain designs has experienced less-optimal behavior in processing these smart contracts.

To make things worse, special events can also boost the SC proportion significantly during a short period of time and drag down the performance significantly. Using Ethereum as an example, the well-known Initial Coin Offering (ICO) bubbles and the blockchain game *Crypto-kitties*, at its popularity peak, both pushed stress of computation burden on the blockchain and revealed the bottleneck of blockchain processing. As more attentions have been drawn to improve the performance, along with the boosting investment on financial technologies (FinTechs), we have reasons to believe the future proportion of smart contracts will be higher, thus it is urgently required to improve the performance of processing smart contracts.

B. Standardized Smart Contracts and Templates

Different from software, SCs on blockchain are often templated and standarized. There are several standard SCs existing in the blockchain ecosystem. They act as public standard libraries and design references. Their existence reduces the burden of developing complicated DApps and the cost of re-inventing the wheel. More importantly, they are widely accepted and used in reality. For example, in Ethereum, one of the most well-known standardized smart contracts is the ERC20 contract [15], which defines a series of essential functions to launch user-customized cryptocurrency. This template has been widely adopted ever since its proposal in 2015. In Sept. 2018, the entire set of 64,393 ERC20 token networks capture 19.45 million unique addresses, which corresponds to nearly 45.9% of all addresses on the Ethereum blockchain [16]. At the current time of Nov. 2019, there have already been 227,048 ERC20 contracts existing on Ethereum, according to Token Tracker from etherscan.io [17].

On the other hand, these standard templates are developed and well tested with the concern of security. In the past, the DAO attack [18], a less-securely designed non-standard smart contract, resulted in $60M loss of money and the hard fork of Ethereum Classic (ETC). Thus, we believe the standardized smart contract and libraries will be hugely needed in the future. Therefore, it is well worth applying optimization for these predominant smart contracts.

C. *Generalized Smart Contract*

Diverse blockchains usually define their own instruction set architecture (ISA) and runtime environment to support their SC execution. Even though the coding languages are different, the philosophy and dataflow behind it are quite similar. SCs can be viewed as the hard-coded rules to modify blockchain states if specific requirements meet. Having this vision, all SCs can be treated as a condition checking logic and a state updating method, even though they could be implemented differently. Therefore, without loss of generality, we use the most well-known blockchain Ethereum as an example to explore the potential optimization. Furthermore, based on this observation, we propose the generalized dataflow optimization, which will be introduced in next section.

III. Design and Architecture

In this section, we introduce the design and architecture of BPU, which has the configurable and modularized components to be generalized on different blockchain platforms. In this paper, we target BPU at the most well-known SC blockchain Ethereum, and implement App Engine as ERC20 engine and GSC engine as Ethereum Virtual Machine (EVM) engine. Note that the BPU design can be also applied to other blockchains. In the following sections, we will introduce BPU's components first and discuss the multi-core design and transaction level parallelism.

A. *BPU Design*

Figure 1 describes the components and dataflow of BPU architecture. At a high level, BPU will read data from Input Buffer via a Scheduler, process the transactions in the Transaction Processor (TP), and finally write the result to the Receipt Buffer. The essential parts of TP are an Application Engine and an General Smart Contract (GSC) Engine. The former is deeply optimized to accelerate specific applications or standardized SCs, and the latter handles the general-purpose SCs. In addition, BPU can be configured into either single-core mode or multi-core mode.

1) Input Buffer and Data Structure: The Input Buffer holds all the raw data fed into the BPU. It is composed of two parts: The Transactions Buffer (TransBuf) and the State Database (StateDB). TransBuf stores the transactions to be executed, which contains a series of fixed-length parameters, such as the 160-bits *address*, the 80-bits *value* etc., and an arbitrary length byte array *Inputdata*. We stored the fixed-length part and variable-length part separately in Transaction information buffer (TransInfo) and transaction input

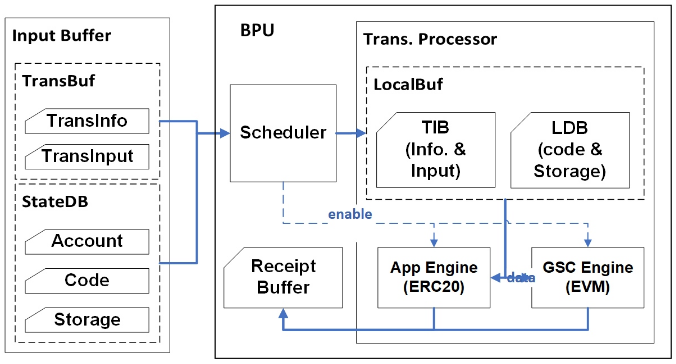


Fig. 1: BPU architecture

buffer(TransInput) respectively. This enables us to concatenate the fixed-length data together and read them as a single piece at one cycle. On the other hand, we can save space when allocating dynamically for the variable-length Inputdata. Meanwhile, it also makes it possible for App Engine to read both sections concurrently.

StateDB contains the state before the transactions get processed, including all the related accounts and target contracts. Each account has four fields: balance (fixed size), nonce (fixed size), code field (any size), and storage field (any size). For the same reason of seperating fixed-length and variable-length parts, we have the Code and Storage buffers and AccountInfo buffer seperately.

2) Scheduler: The Scheduler is a module that reads data from the input buffer and fills the Local Buffer (LocalBuf) with the appropriate format. More importantly, it will identify the function being called and distribute the transaction to either EVM Engine or Application engine. As we introduced in the background, the first eight bytes of the input data is the identifier of the function. The scheduler will firstly retrieve a transaction and get the input data. Then it will decode the identifier from the input data and compare with a series of pre-defined identifiers to distinguish the general SC and optimized SC. For example, in our experiment, we use constant value "0xa9059cbb" as the identifier for optimized ERC20 transaction, which is the same as standard transfer() function in ERC20 contracts.

Meanwhile the scheduler will search for the target account in StateDB and retrieve the target code field and storage field. Finally, the scheduler will save these data to the LocalBuf in order to set up the local copy for Transaction Processor (TP). In multi-core mode, there will be multiple TPs running in parallel. The scheduler will also handle the way transactions are dispatched to different TPs, which will be introduced in next section.

3) Local Buffer and Transaction Processor: Local Buffer is a piece of cache that saves a copy of the target contract. It is composed of two parts: Transaction Information Buffer (TIB) and Local Database (LDB). TIB holds the current transaction being executed, which provides the target information such as contract address, sender address and input argument. LDB serves as a copy of the pre-processing state. During the execution, any temporary changes of the state are written to LDB. After the execution, only if the transaction is executed successfully, LDB will overwrite the StateDB. Or if any error happens during the execution, LDB will be discarded so that StateDB is unchanged, which is also called "revert".

TP is where transactions get processed. At the transaction level, TP is fed by the scheduler one transaction at a time. Only after the previous one retires, the next transaction can be read in TP. In single-core mode, better performance comes from the micro-architecture optimization in EVM Engine and App Engine. In next section, we

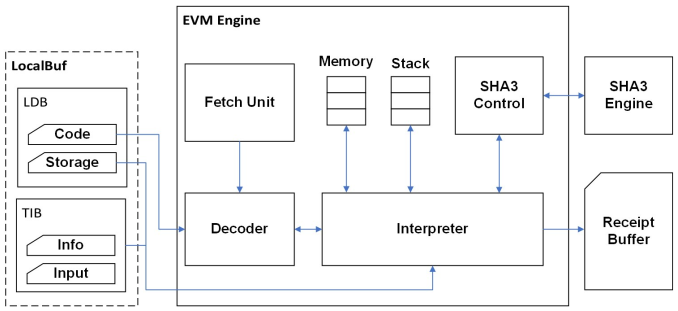


Fig. 2: Structure of EVM engine

Name	Opcode	Bytecode
Stack	PUSH, DUP, SWAP, POP, BYTE	0x60 ... 0x9f, 0x50, 0x1a
Arith	ADD, MUL, SUB, DIV, SDIV, MOD, SMOD, ADDMOD, MULMOD, EXP, SIGNEXTEND	0x01 ... 0x0b
Logic	LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, NOT	0x10 ... 0x19
Env	Address, Balance, Origin, Caller, Callvalue, Calldataload, Calldatasize, Calldatacopy, Codesize, Codecopy, Gasprice, Extcodesize, Extcodecopy, Returndatasize, Returndatacopy	0x30 ... 0x3e
Block	Blockhash, Coinbase, Timestamp, Number, Difficulty, Gaslimit	0x40 ... 0x45
Memory	Mstore, Mstore8, Mload	0x51 ... 0x53
Storage	Sload, Sstore	0x54, 0x55
System	Return, Revert, Invalid, Selfdestruct	0xfc ... 0xff

TABLE I: Submodules of EVM engine

discuss the multi-core design targeting at parallelism at the transaction level.

B. General SC Engine - EVM

EVM is the kernel execution model of Ethereum, which is a stack-based state machine as defined in [4]. In this paper, we implement our GSC as an EVM engine. As Figure 2 shows, it is composed of a local Stack, a local Memory, a Fetch Unit, a Decoder, a SHA3 engine and an Interpreter. Among them, the stack and memory save the local volatile data; the Fetch Unit fetches the code and handles the branches; the Decoder decides which one of the submodules in the Interpreter should be sent the current instruction; SHA3 engine is a stand-alone module specifically optimized for hash function, which will be introduced in the next section. The Interpreter is a collection of multiple submodules which handle different operations;

1) Submodules and Pipeline: Table I lists out the names of EVM submodules, also called units, and their supported operations. These units implement the ISA defined in [4] and thus are fully compatible with the current Ethereum architecture. Our highly modularized design of EVM Engine enables us to apply the pipeline optimization to achieve better performance. We design an in-order pipeline on the EVM Engine as shown in Figure 3. The execution is divided into four stages. Namely, operation fetching, decoding, execution and writeback to stack or memory. There are registers that carry the data between the stages.

Before the execution, the scheduler will move the target data from Input Buffer into the local buffer. When the execution starts, the controller will fetch the target code using Program Counter (PC) and apply a pre-decoding check. The STOP code (0x00) will terminate the execution immediately. If the execution continues and the Decoder is ready, the controller pushes the instruction into the decoding stage. The Decoder will then decode the operation and choose the appropriate submodule of interpreter to send. Then in the Interpreter, the operation will be executed, and the temporary result is latched and stored to the Stack and Memory in the writeback stage. Finally, when the EVM Engine decides to terminate, it will generate an EVM receipt based on the status of the local memory and stack, and return to the Receipt Buffer.

In addtion, we apply special optimization on these opcode executions in order to achieve better performance. In early versions of Ethereum ISA, there is no bit shifting operations. Such operations are always implemented by 256-bit division (DIV) operation. For example, a typical procedure at the beginning of execution is to fetch the function identifier, which is a 32-bits integer locating as the first eight bytes of the input section. Due to the lack of shifting, compilers often implement these by using the 256-bit integer divided by constant value 2^{224}, in order to get first 8 bytes. Similarly, when the first 16 bytes are needed, another constant value 2^{192} is needed. Such big constants are not pre-calculated but usually generated by exponent (EXP) operation at runtime. In practice, such big number division and exponent operation is very time consuming if implemented by dividers and multipliers, so we optimize such special cases using bit shifting registers, which significantly saves the time and reduces the additional cost. Nevertheless, the general-purpose division and exponent is still necessary because EVM is intended to support Turing complete computations.

2) SHA3 Submodule: We also design a special SHA3 unit to do the time consuming SHA3 operation (bytecode 0x20). We need to point out that the SHA3 here is NOT the standard SHA3 algorithm (defined by FIPS 202 [19]). Instead, it refers to the Keccak-256 algorithm [20], which uses different parameters in calculation. In this paper, we use the conventional name “SHA3” but all of them represent the Keccak-256 algorithm.

In Ethereum, SHA3 is often used to calculate the 256-bit key of the storage index in order to get the current state from the storage. It will firstly pad the input into fixed length data (1088-bits long) and feed it into the sponge function. Our SHA3 engine has a 64-bit input paired with a valid signal. When the valid signal is set, SHA3 engine will start reading the input continually and save them in the buffer. It takes up to nine cycles to fill up the 1088-bits buffer, and then this big value will be consumed all at once to do the sponge function. Another “last byte” control signal is used to stop feeding ahead of schedule if needed. The SHA3 algorithm does 24 rounds of calculation on the sponge function and result will be saved in the output buffer and ready to be read. Inspired by the design [21], we implemented a two rounds per cycle design at the hardware level, which reduces the execution time of the sponge function to 12 cycles.

To summarize, our EVM Engine has full support for Ethereum’s defined Turing complete operations. We optimized several time consuming operations, such as SHA3, DIV and EXP, and implement pipeline optimization for better performance.

C. Application Engine - ERC20

As we introduced in the background, a majority of SCs are templated and limited to some certain functions (libraries). This important feature enables us to do deep optimization as long as we define the function’s dataflow clearly. In this paper, we propose a general dataflow optimization targeted at maximizing parallelism and explore the potential benefits for such optimization using the Ethereum ERC20 template as an example.

1) Dataflow Template: As we introduced in the background, many SCs can be treated as an assertion logic and a state changing method.

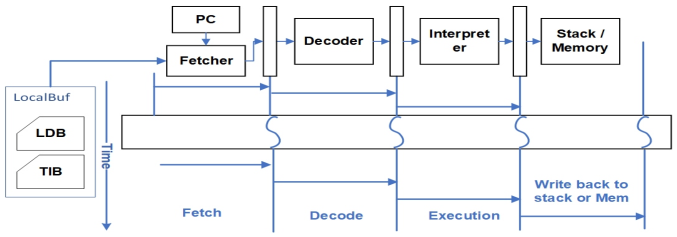
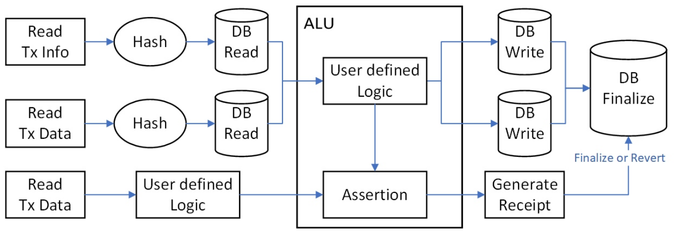
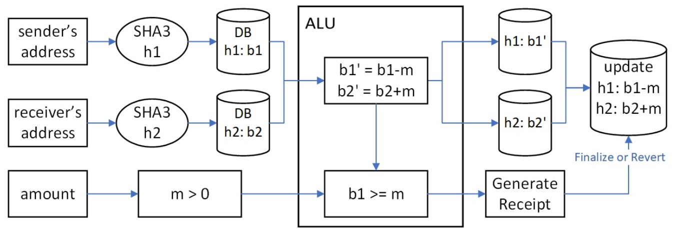


Fig. 3: Pipeline of EVM engine



(a) General data flow template



(b) ERC20 transfer function data flow

Fig. 4: Application Engine data flow template

Based on this observation, we propose the optimized dataflow template as shown in Figure 4a. The whole design is composed of a series of basic units: Read, Hash, DB Read, DB Write, Arithmetic Logic Unit (ALU) and Receipt. These input units will read not only from the input data of the transaction, but also from the local buffer to achieve transaction information such as the sender's address. Figure 4b shows an example of mapping ERC20 transfer function to our template. Two input units will read the 160-bits receiver's address and sender's address respectively, and then calculate the corresponding keccak256 hash value. The other input unit will read the transaction data field and get the amount of tokens to be transferred. The ALU will calculate the output value. The Assertion unit is used to revert the execution by discarding the current state changes, if specific conditions do not meet. In this case, it needs to guarantee the sender has enough tokens to be transferred (no less than the amount). Finally results will be passed to the DBWrite unit and Receipt generation unit to finalize the transaction. All the above modifications are done to the local database and if everything goes well, the world state will be overwritten at the last step.

Our design divides the dataflow based on their dependency and takes advantage of the parallelism. The two input module reads from different location. One is from the variable length *data* segment, and another is from fixed length transaction information segment, which contains sender's address, to_address, etc. These segments are stored in different buffers in order to enable concurrent reading and other benefits, which will be introduced in next single-core section.

2) Optimization and App Engine: The conventional EVM implement for smart contract is usually less-optimal and lack of support for parallelism, which does not take fully advantage of the modern hardware. For example, when compiled with solidity v0.5.13, it takes more than 200 operations to implement a simple transfer function. A lot of them are redundant data pushing, duplication and swapping on the stack. Therefore, we implement App Engine using our proposed dataflow to achieve better performance.

In our design, we fully parallelize the three dataflow and merge them only at the ALU because they do not have dependencies on each other. For simplicity and the trade-off between the performance and hardware area, our ALU in App Engine only supports the ADD and multiplication (MULT) operations. However, more complicated computation can be done using the companion EVM Engine and pass to the assertion unit to approve the transaction.

D. Multi-core Design and Transaction-Level Parallelism

In our design, the BPU can be configured to have multiple TPs running in parallel in order to enhance the performance. In this scenario, the transactions may be processed out of order. If two transactions try to call the same contract, there will be possible Read-after-Write (RAW) and Write-after-write (WAW) dependency issues. Furthermore, a transaction will possibly trigger any other transaction. There is no way to learn the target contract of the triggered transaction except by processing the first one. Therefore, it is impossible to determine the dependency before the actual execution. The paper [9] proposed a speculative execution to solve this issue by saving the miner's execution sequence in the block as supporting information.

In this paper, we profile the target block and determine the transaction level dependencies in advance and save such information in the InputBuf. In multi-core mode, we use the Scheduler to dispatch dependent transactions to the same TP in the right timing sequence. Independent transactions, on the other hand, can be distributed to any available TP and processed out of order. We can benefit from this design in two aspects. Firstly, it will guarantee the data dependency to be resolved. Secondly, processing the dependent transactions in the same TP will remove the redundant loading from the database. If these transactions are executed separately, TPs must writeback to the world state first, and load their LocalDB again. Using our distribution strategy, the TP can reuse its LocalDB as the up-to-date state.

IV. METHODOLOGY AND RESULTS

A. Platform

We implement the BPU on a Xilinx's Zynq-7000 ZC706 evaluation kit [22]. All modules of BPU, except the stand-alone SHA3 engine, are coded in C/C++ and synthesized into Vivado IPs using High Level Synthesis (HLS). The exception, the SHA3 engine, is coded using Verilog directly in order to apply direct control on the timing for optimization. At last, Vivado IPs are integrated and tested on the FPGA. The clock frequency of the overall design is 100MHz. In terms of input buffers, Block RAMs (BRAM) are ideal for storing large arrays. In our design, we used 692 (out of 1090 available) on-board BRAM-18K to compose the input buffer, storing the state database and the transactions. These BRAMs are configured as dual port ROMs so that data can be accessed by two cores at the same time.

We run the software implementation of Ethereum as baseline, which is coded in C/C++ and compiled using GCC version 8.1. The CPU we used is an Intel i7-7700k quad-core at 4.2GHz. A multi-run averaged execution time is measured and compared against BPU performance on the target FPGA. Meanwhile, BPU is set up with different configuration in order to evaluate individual engines as shown in Table II.

Config	# of EVM	# of App	Used in test
SimpleCore	1	0	u-bench, real-bench
ERC20Core	1	1	u-bench, real-bench
ERC20Duo	2	2	real-bench

TABLE II: BPU configuration in experiments

Bench	Block#	Txs	ERC	Gnrl SC	ETH	ERC/SC
B1	6653186	190	31%	13%	56%	70%
B2	6653197	102	29%	29%	41%	50%
B3	6653232	115	46%	13%	41%	78%
B4	6653208	78	22%	42%	36%	34%
B5	6653220	90	33%	33%	33%	50%
B6	6653209	159	40%	18%	43%	69%
B7	6653205	16	38%	6%	56%	86%
B8	6653205	80	38%	6%	56%	86%

TABLE III: Details of real-case benchmarks

B. Benchmarks

1) *Real-case benchmarks:* In our study, we use real-case blocks retrieved from Ethereum's history record as our benchmarks. We carefully select the blocks with various numbers of transactions and smart contracts in order to satisfy generality and universality as shown in Table III. We also count the number of total transactions, ERC20 transactions, General SC (non-ERC20 SC), ETH token (non-SC) and ratio of ERC20 within SCs as shown in the columns respectively. The real block data were retrieved from the Etherscan.io [17] website via API, and saved into the input buffer.

2) *Micro-benchmarks:* We also use micro-benchmarks to test the performance of different submodules which are discussed in EVM Engine section. Each of these micro-benchmarks will only target one specific type of operation. To improve the accuracy, we repeat the same kernel code 1000 times, as presented in Table IV. In addtion, as shown in the bottom half of Table IV, we also add three block level micro-benchmarks to test the performance of App engine. The EMPTY is an empty SC without computation, which stands for the overhead of loading SC to BPU. For 100% ETH, we package 20 non-SC ETH transfers. The major ETH operation is global database search and modification. For 100% ERC20 benchmark, we select one ERC contract and generate 20 ERC20 transfers to call the transfer() function. These transactions are compute-bounded and can be benefited from hardware acceleration.

3) *Operations analysis:* In order to recognize the behavior of smart contracts, we analyzed the processing trace of all benchmarks and calculate the frequency of each type of operations. As the Table V presents, we categorize the operations into different groups depending on which submodule handles the bytecode, which is the same relationship defined in Table I.

C. Results

In our experiments, we equipped the BPU with different number of EVM engines and App engines and test in different sections, as illustrated in Table II. Note that the SimpleCore and ERC20Core are both in single-core mode and the ERC20Duo is in duo-core mode with a scheduler to distribute transactions.

1) *Micro-benchmarks Results:* In this section, we tested the EVM Engine performance using SimpleCore. Figure 5 provides the speedup for different operation types. We can see that the speedup of MEMORY operations is very high (x70). This is because the CPU Intel i7-7700K has only 8MB last level cache [23] and our FPGA platform has 18M on-chip BRAM. Our Memory micro-benchmark generates more on-chip cache misses in CPU than BPU. In addition, the STORAGE benchmark has a bottleneck in off-chip database search. This limits

Name	Description
PUSH	1000 PUSH
STACK	1 PUSH, 1000 DUP, 1000 POP
ARITH	1000 ADD, 1001 PUSH
LOGIC	1000 PUSH, 1000 AND
STORAGE	1000 PUSH, 1000 SLOAD
MEMORY	1000 PUSH, 1000 MLOAD
EMPTY	Empty transaction with only "return 0";
100% ERC20	20 Transaction calling ERC20 Transfer() function
100% ETH	20 Ethereum token ETH transfer (not a SC)

TABLE IV: Details of micro-benchmarks

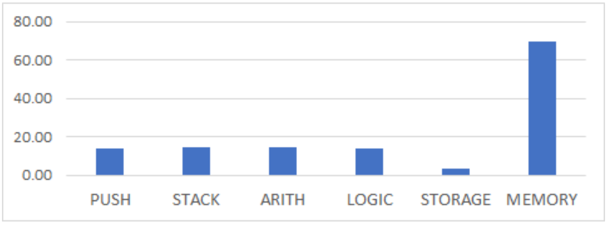


Fig. 5: Speedup of operation level micro-benchmarks

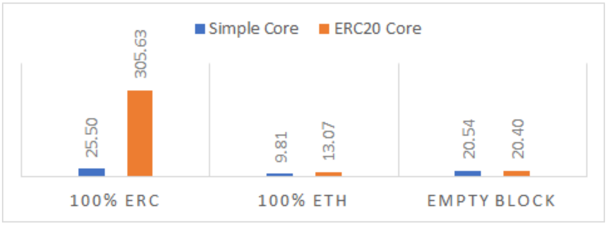


Fig. 6: Speedup of block level micro-benchmarks

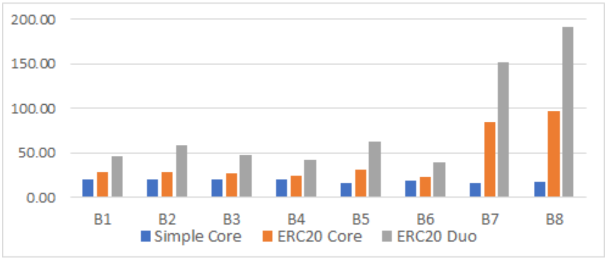


Fig. 7: Speedup of BPU against CPU

the performance advantages of the fast BPU and results in a low speedup (x3.6).

In Figure 6, we presents the result of block level micro-benchmarks. The speedup for SimpleCore on ETH benchmark is relatively low (x9.8) due to the heavy off-chip database searches. However, it has a x25.5 speedup on 100% ERC benchmark due to the benefit from pipeline optimization. When equipped with additional ERC20 engine, the ERC20Core boosts up the speedup to over x300 on 100% ERC benchmark. This proves the significant advantage of applying such application level optimization on App engine.

2) *Real-benchmarks Results:* Figure 7 illustrates the overall performance of BPU in all three configurations. We use the CPU version as our baseline and show the FPGA speedup. For SimpleCore configuration, it does not have high speedup on B5, B7, and B8. This is due to the fact of high rate of Storage operation (shown in Table V) and its poor acceleration (shown in Figure 5).

However, ERC20Core has a good performance on B7, B8 with x84, x96 speedup respectively. This result is because these blocks contain a higher proportion of ERC20 transactions (shown in Table

	SHA3	Push	Stack	Arith	Logic	Jump	Mem	Strg	Env	Block	Others	total count
B1	0.46%	36.73%	18.77%	2.91%	15.81%	16.20%	2.70%	0.63%	2.98%	0.02%	2.79%	14281
B2	0.26%	28.32%	25.04%	5.39%	16.27%	16.94%	1.65%	0.54%	1.66%	0.00%	3.92%	21063
B3	0.44%	36.50%	19.84%	2.57%	15.67%	16.53%	2.48%	0.67%	2.72%	0.03%	2.54%	9801
B4	0.14%	37.37%	19.86%	2.12%	17.61%	17.89%	1.57%	0.36%	1.51%	0.14%	1.44%	11558
B5	0.72%	34.33%	21.59%	5.09%	14.06%	13.16%	3.77%	1.32%	2.75%	0.00%	3.23%	16720
B6	0.40%	37.10%	19.73%	2.74%	16.64%	16.23%	2.33%	0.52%	1.87%	0.01%	2.42%	13499
B7	1.00%	32.18%	23.10%	4.82%	14.25%	13.35%	3.63%	1.77%	2.68%	0.09%	3.13%	2203
B8	1.00%	32.18%	23.10%	4.82%	14.25%	13.35%	3.63%	1.77%	2.68%	0.09%	3.13%	11015

TABLE V: Instruction breakdown of benchmarks

Design	LUT	FF	DSP	BRAM	Power (Watt)
SimpleCore	61202	67344	402	25	5.956
ERC20Core	65517	72464	402	25	6.104
ERC20Duo	137717	153841	806	50	6.892

TABLE VI: Hardware Utilization and On-Chip Power

III) and they benefit a lot from the App Engine. Even though B3 has a relatively high ERC20 proportion, we discover there are several time-consuming 256 bits division operations that slow down the overall performance.

When configured in duo-core mode, ERC20Duo achieves a better performance up to x151 on B7 and the best x191 on B8, almost doubled the single-core performance. The benefit of duo-core over single-core depends on the content of the block's transactions. More accurately, it is how well the scheduler can distribute the task evenly. B8 is a perfect example of this situation. We duplicate the transactions in B7 into B8, so that the scheduler can distribute B8 onto two cores evenly. This results in a x1.97 speedup against single-core and x191 against CPU baseline.

3) Other Results: Table V demonstrates the operation breakdown of each benchmark, which describes the diversity between smart contracts and explains their different performance. As we can see, B7 and B8 have a higher proportion of Storage and SHA3 operations. These are the typical signs of higher rate of ERC20 transactions, which can be accelerated using App engine. On the other hand, B2 has a higher rate of Arithmetic and Stack operations, and lower rate of Storage and SHA3. This means that B2 does not contain many ERC20 transfers (also shown in Table III), thus cannot benefit a lot from ERC20Core. However, it is computation bounded and achieves a x20 speedup from SimpleCore. In addition, the utility of resources and power consumption is also shown in Table VI.

V. CONCLUSION

In this paper, we proposed BPU, a flexible and high-performance architecture design for blockchain processing. This design relies on a variety of modules to accelerate the execution of transactions, especially the smart contract transactions. At the micro-architecture level, we have applied pipelining and other optimization techniques to improve the performance. At the system level, BPU can also be configured as either single-core or multi-core mode in order to benefit from parallelism and achieve better performance.

In addition, we explored the current smart contract behavior on the Ethereum platform by doing a detailed analysis on the execution traces. Based on the result, we propose application-oriented optimization, App engine, which achieves a x305 speedup under best conditions. Finally, we implemented and evaluated the BPU design on a Xilinx's Zynq-7000 ZC706 evaluation kit. As shown above, the BPU design has an up to x96 speedup compared to the CPU implementation in processing Ethereum transactions. The dual-core mode can roughly double the performance if scheduled perfectly.

Furthermore, BPU's modularized design retains the flexibility to be tuned for different ecosystems or other blockchains other than Ethereum. The future work will be exploring the feasibility and the trade-offs between hardware cost and performance.

REFERENCES

[1] S. Nakamoto *et al.*, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] P. De Filippi, "The interplay between decentralization and privacy: the case of blockchain technologies," *Journal of Peer Production, Issue*, no. 7, 2016.

[3] M. Moser, "Anonymity of bitcoin transactions," 2013.

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

[5] 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.

[6] 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)*, pp. 433–436, IEEE, 2016.

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

[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*, pp. 303–312, ACM, 2017.

[10] M. Scherer, "Performance and scalability of blockchain networks and smart contracts," 2017.

[11] S. S. Hazari and Q. H. Mahmoud, "A parallel proof of work to improve transaction speed and scalability in blockchain systems," in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 0916–0921, IEEE, 2019.

[12] C. Riegger, T. Vinçon, and I. Petrov, "Efficient data and indexing structure for blockchains in enterprise systems," in *Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services*, pp. 173–182, ACM, 2018.

[13] Y. Sakakibara, K. Nakamura, and H. Matsutani, "An fpga nic based hardware caching for blockchain," in *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, p. 1, ACM, 2017.

[14] J. A. Dev, "Bitcoin mining acceleration and performance quantification," in *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pp. 1–6, IEEE, 2014.

[15] V. Buterin and F. Vogelsteller, "Erc20 token standard," *URL: https://theethereum. wiki/w/index. php/ERC20 Token Standard*, 2015.

[16] F. Victor and B. K. Lüders, "Measuring ethereum-based erc20 token networks," in *press*, 2019.

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

[18] M. B. Atzei, Nicola and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," 2017.

[19] S.-. S. P.-B. Hash and E.-O. Functions. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[20] https://ethereumclassic.github.io/blog/2017-02-10-keccak/.

[21] https://opencores.org/projects/sha3.

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

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