IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 15 March 2023, accepted 4 April 2023, date of publication 14 April 2023, date of current version 18 May 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3267277

RESEARCH ARTICLE

EVMTracer: Dynamic Analysis of the Parallelization and Redundancy Potential in the Ethereum Virtual Machine

XIAOWEN HU[1], BERND BURGSTALLER[2], AND BERNHARD SCHOLZ[1,3]
[1]School of Computer Science, The University of Sydney, Sydney, NSW 2006, Australia
[2]Department of Computer Science and Engineering, Yonsei University, Seoul 03722, South Korea
[3]Fantom Research, Grand Cayman KY1-1104, Cayman Islands

Corresponding author: Bernd Burgstaller (bburg@yonsei.ac.kr)

This work was supported in part by the Fantom Foundation, in part by the Australian Government through the Australian Research Council (ARC) Discovery Project funding scheme under Grant DP210101984, in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) funded by the Korean Government (MSIT) under Grant 2021-0-00853, and in part by the National Research Foundation of Korea (NRF) funded by the Korean Government (MSIT) under Grant 2019R1F1A1062576.

ABSTRACT Ethereum is one of the first blockchains executing smart contracts, i.e., financial applications directly executed on the ledger using a virtual machine. High transaction volumes caused by financial applications, including decentralized finance and non-fungible tokens, slow down the Ethereum Virtual Machine. Hence, there is a need to detail the execution characteristics of the Ethereum Virtual Machine before its performance can be improved. This work introduces an off-line Ethereum virtual machine tracer called `EVMTracer` that produces runtime data dependence graphs from executed transactions as an alternative program representation. From the runtime dependence graphs, we can deduce valuable metrics about program execution characteristics, including the degree of parallelization and computational redundancies in smart contracts. Our experiments encompass all blocks up to 12 M on the Ethereum mainnet. We found a geometric mean of 1.90× theoretical maximum speedup when executing the smart contracts in parallel and identified 34.97 % of `SLOAD` instructions as redundant.

INDEX TERMS Blockchain, smart contracts, Ethereum, tracing, metrics, parallelism, redundant computations.

I. INTRODUCTION

The Ethereum blockchain supports trustless and permissionless execution of smart contracts. Smart contracts received a lot of attention from applications including trade [1], [2], IoT [3], [4], trust management [5], [6], [7], banking [8], [9], [10], [11], decentralized finance (DeFi) [12], [13], [14], governance [15], [16], [17], non-fungible tokens (NFTs) [18], [19], [20] and metaverse [21], [22]. By the end of 2021, Ethereum contained 14 M blocks with over four million smart contracts and has processed over one billion transactions. More than 177.6 billion USD were locked in the area of DeFi alone by May 2022 [23].

The associate editor coordinating the review of this manuscript and approving it for publication was Mueen Uddin.

The Ethereum Virtual Machine (EVM) is the underlying, Turing-complete execution platform that processes the bytecode of smart contracts through transactions [24]. With the increasing complexity of blockchain applications, the efficiency of the virtual machine becomes paramount for smart contract execution. According to Ethereum inventor and co-founder Vitalik Buterin, scaling the transaction throughput from currently 15 tx s^{-1} to 100 000 tx s^{-1} is one of the most pressing issues of the Ethereum blockchain [25]. Visa, in comparison, is a direct competitor of Ethereum in terms of total transaction volume [26], claims a throughput of up to 65 000 tx s^{-1} [27].

Recent experiments with Ethereum have identified the EVM as a key performance bottleneck. The average time for the EVM to interpret a smart contract has been reported as 0.2 ms in [28], implying a throughput limit of 5000 tx s^{-1}.

Likewise, in an experiment that executed the initial 9 M blocks of the Ethereum mainnet, the EVM's throughput was below 4208 tx s^{-1} [29]. Both experiments measured the net performance of the EVM itself, excluding the consensus protocol and world-state database updates. The reported figures are thus indicative of the bytecode interpretation overhead and are not mitigated by Ethereum's recent consensus protocol shift from proof-of-work (POW) to proof-of-stake (POS) [30].

It is necessary for the community to understand the execution characteristics of smart contracts to reduce the EVM's execution overhead and to design and implement more efficient virtual machines and environments for the evergrowing blockchain applications.

However, in the blockchain domain, no established benchmark suite exists that could be adopted for the performance evaluation of the EVM. This is in stark contrast to the ubiquity of standardized benchmark suites in general-purpose computing areas, including embedded, compute, JVM-based, edge, cloud, and HPC [31], [32], [33], [34], [35], [36], [37], [38], [39]. Although many attempts [25], [40], [41], [42], [43], [44] have already been taken to understand the performance characteristics of smart contracts, they relied on the availability of the source code, hence focused more on high-level language elements. Note that only 5.1 % of all smart contracts deployed on the Ethereum blockchain have been open-sourced [45], up from 2.2 % by 2018 [25]. Given the lack of available source code, we argue that performance analysis based on open-sourced smart contracts will not be representative of real-world blockchain workloads.

We thus turn our attention to *bytecode*, which is the program representation for the deployment of smart contracts on the blockchain. Programmers typically write smart contracts in a high-level, specialized language such as Solidity [46], [47] or Vyper [48], which is then compiled into an immutable bytecode representation for the EVM. Bytecode is deployed persistently on the blockchain and invoked for execution on the EVM through transactions. Because all deployed bytecode and the entire execution history of all transactions—including input data—is available on the chain, we base our workload analysis on smart contract bytecode.

To provide a scalable and representative benchmarking tool for blockchains, we introduce an offline dynamic tracing system called `EVMTracer`, which replays the transactions locally and collects the *runtime dependence graph* of the transactions. The runtime dependence graph is an alternative representation of the original program's execution trace containing data flow and control information. This information can be used to determine the execution characteristics of smart contracts.

`EVMTracer` is based on an existing efficient transaction replay system [29] that enables us to collect runtime information for millions of transactions without the substantial overhead experienced with client software [49]. We collect the runtime dependence graphs for all transactions in the initial 12 M blocks of Ethereum and use them to determine two performance characteristics of smart contracts. Two metrics are presented to showcase that `EVMTracer` can discover useful metrics from millions of blocks and help the community to get a better understanding of overall runtime characteristics of transactions. The first metric is the degree of parallelism at the bytecode instruction level. This metric can help the community to understand the potential parallelism in smart contracts and determine whether developing a contract-level parallel execution model is beneficial for the EVM. The second metric is the number of redundant memory and storage computations in the smart contract runtime, in which we count the number of redundant instructions[1] that the EVM executes. This metric reveals the potential for overlooked optimizations and the execution characteristics of a stack-based instruction set.

This paper makes the following contributions:

- An offline dynamic tracing system that produces *runtime dependence graphs* of smart contract executions for large volumes of transactions.
- At-scale investigation of contract-level parallelism on the Ethereum blockchain.
- At-scale investigation of redundant computations for memory and storage IO operations on the Ethereum blockchain.

We have released the `EVMTracer` framework as open source, as described in the paper's availability statement.

The remainder of this paper is organized as follows: In Section II, we provide the background for Ethereum and the EVM. In Section III, we discuss technical details about the system and the runtime dependence graph. In sections IV and V, we explain how to obtain the contract-level parallelism and redundant computation metrics from the runtime dependence graph. Section VI contains the experimental results. We discuss the related work in Section VII and draw our conclusions in Section VIII.

II. BACKGROUND

Ethereum can be viewed as a transaction-based state machine that maintains a *world state* as shown in Figure 1. The world state comprises information about the accounts on the blockchain. Each account is referred to by its account address and contains the following information: (1) A *nonce*, which is a counter used to prevent replay attacks (double-spending), (2) an account *balance* representing the endowment of the account, (3) a *code* section with the smart contract bytecode (although an account is allowed to have an empty code section), and (4) the *storage* space of the account. The storage space provides smart contracts with a persistent state across transactions. Storage is represented as a key-value map where the key is the storage address, and the value is the actual data stored at the address.

[1] The EVM specification [24] describes *operations* (I/O, arithmetic, computations, etc.) categorically and *instructions* (e.g., "the `SSTORE` instruction performs an I/O operation on the storage of an account."). We adopt this convention in this paper.

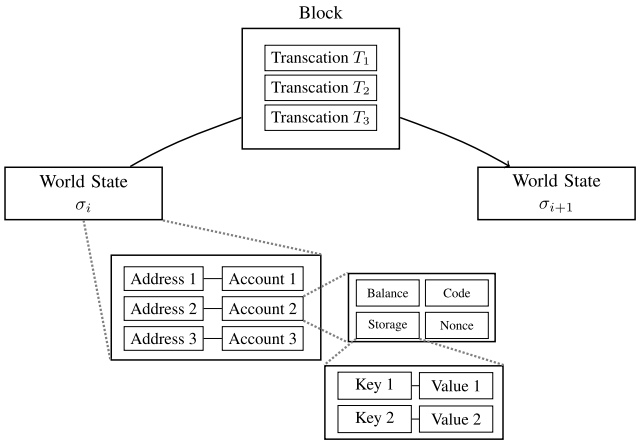


FIGURE 1. Overview of Ethereum as a transaction-based state machine.

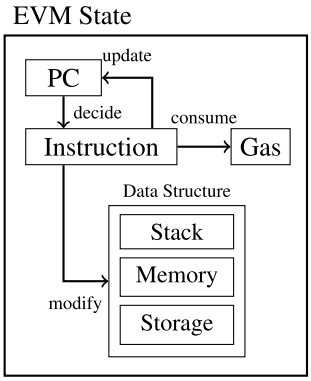


FIGURE 2. Interworking of the components that comprise the internal EVM state.

Each transaction results in a side effect that advances the state of Ethereum. Transactions are further collated into blocks, and each block's transactions are applied sequentially on the Ethereum blockchain. We classify transactions into three types. First, an *asset transfer* transaction takes an asset from the sender and transfers it to the recipient. Next, a *contract creation* transaction creates a new account on the chain and initializes and stores the smart contract bytecode in the storage space of the account. Finally, a *message call* transaction invokes a target contract on the blockchain.

Dynamic tracing [50] refers to the action of collecting and investigating runtime behavior of the target program by recording the executed instructions at program runtime. Hence, we enable dynamic tracing only on transactions that result in the execution of smart contracts, i.e., tracing is performed on message call transactions. Note that tracing depends on the program's input, unlike static program analysis [51] such as abstract interpretation [52], which is not concerned with a concrete input.

For our purpose, we need to understand four of the necessary fields in a message call transaction. A *from* and a *to* address, which specifies the sender (caller) and recipient (callee) addresses. A *gas limit* variable, which specifies the amount of gas that the sender is willing to pay to execute the contract. Finally, a *call data* section that is part of Ethereum's application binary interface (ABI), specifies the entry function to call and its input arguments. Once the miners confirm a block, the EVM executes the block's transactions sequentially.

The EVM is a stack-based virtual machine that is specified in the Yellow Paper [24] and interprets (executes) the bytecode of smart contracts. The following components define the internal state of the EVM:

1) *Stack:* A stack data structure that stores 256 bit values and has a maximum depth of 1024 stack slots.
2) *Memory:* An unlimited, linear byte array that supports random access at runtime. Memory is accessed through the `MLOAD` and `MSTORE` instructions of the EVM. Memory grows dynamically. The `MSIZE` instruction reports the current memory size (in bytes).
3) *Storage:* The global storage contains all storage variables in Ethereum. Unlike memory and stack, storage variables are persistent and thus part of the global state of Ethereum. Storage is accessed through the `SLOAD` and `SSTORE` instructions of the EVM.
4) *Program counter (PC):* The PC points to the EVM's next instruction.
5) *Gas:* The remaining gas available for the current execution. The EVM will terminate a transaction that runs out of gas.

Figure 2 shows the interworking of the components that comprise the internal state of the EVM. The PC determines the next instruction to be executed. The executed instruction may have a side effect that modifies the stack, memory, or storage. The executed instruction consumes gas and updates the PC to point to the next instruction to be executed.

We present a smart contract in Solidity syntax in Figure 3 to familiarize readers with basic EVM operations. For the sake of demonstration, the depicted bytecode is a simplified version of the bytecode generated by the Solidity compiler. In line 2, the 32 B wide unsigned storage variable `MyVar` is declared. The contract contains a single public function `setStorage`, which sets the value of `MyVar` to input argument `val`. A message call transaction has a call data section (a byte array specifying the target function to be invoked and its input arguments). In the call data, the first four bytes represent the ID of the target function, which we assume to be the function `setStorage` in this example. After receiving the call data, the EVM identifies the target function, dispatches control to the specified function, and starts execution.

The right-hand side of Figure 3 shows the bytecode representation of function `setStorage`. The purpose of the first line is to push the value 0×4 onto the stack, which is the offset of the first argument in the call data. In the second line, the `CALLDATALOAD` instruction pops the top of the stack, uses it as the offset and pushes a 32 B value from call data onto the stack. (This effectively pushes the argument value `Val`

```
1 contract SimpleStorage_1 {
2     uint32 MyVar;
3
4     function setStorage(uint32 val)
          public {
5         MyVar = val;
6     }
7 }
```

```
1 PUSH 0x4       ;position of 1st arg
2 CALLDATALOAD   ;load arg onto the stack
3 PUSH 0x0       ;address of MyVar
4 SSTORE
```

FIGURE 3. Smart contract (left) and the corresponding EVM bytecode of function `setStorage` (right).

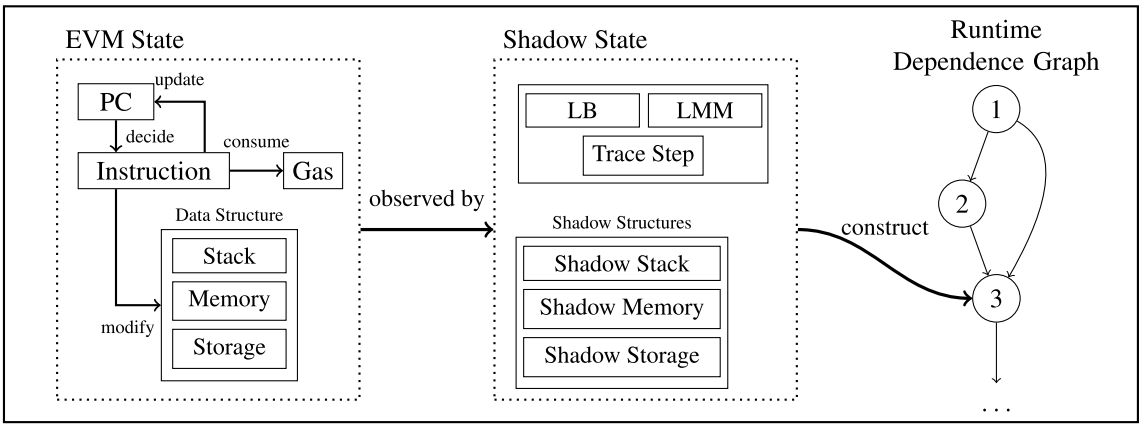


FIGURE 4. Tracer: Dependence graphs are constructed from the shadow state, which is the tracer's extension of the EVM's internal state with runtime dependence information.

onto the stack.) Finally, the code in lines 3–4 stores the result at the address of variable `MyVar` in storage.

In Figure 1, the storage variables that belong to a particular account are located in the account's `<Key, Value>` map that is stored in the `<Address, Account>` map in the Ethereum world state. The `SSTORE` instruction implicitly takes the current contract as the execution environment. The `Key` of `MyVar` in our example is 0×0. Hence, the code in line 3 pushes 0×0 onto the stack. In line 4, `SSTORE` will pop the `Key` and `Value` from the stack and perform the write operation on the storage.

III. ETHEREUM DYNAMIC TRACING SYSTEM

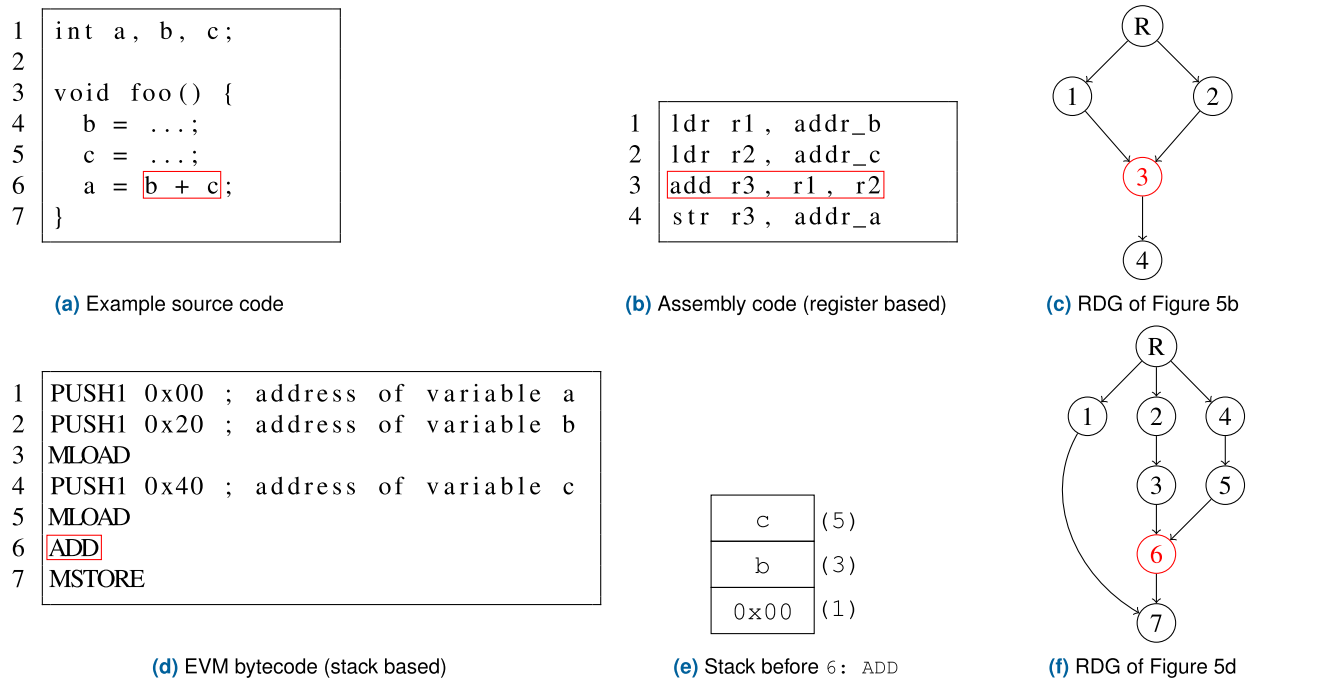
The runtime dependence graph (RDG)—or dynamic program dependence graph [53], [54]—is a dynamic variation of the *program dependence graph* [55], [56], [57] (PDG) encoding data and control dependencies among statements in a program. The PDG has been used in compiler optimizations [58] such as program parallelization [59], and program analyses such as slicing [60].

As a variation of the PDG, our RDG encodes the *dynamic* data and control dependencies for executing the EVM *bytecode* for a *concrete* state on the ledger. A node in the RDG represents an executed bytecode instruction of a smart contract execution, and edges represent data- and control-flow dependencies between executed EVM instructions. Our RDGs are acyclic, whereas the original graphs [53], [54] treated each statement as an individual node and produced cyclic multigraphs.

Our `EVMTracer` uses the RDGs to reveal the execution characteristics of smart contracts. Computations may involve EVM's stack, memory or storage operations. The data flow of the EVM is produced from the inputs and outputs of the computations and is captured in the RDG in the form of edges. The task of `EVMTracer` is to capture the data flow of the EVM faithfully. In addition to data flow, control flow must be captured as well. Control-flow dependencies are induced by jumps and calls while executing a smart contract.

`EVMTracer` extends the EVM interpreter to build RDGs as a side effect of the smart contract execution; we call this component the *tracer*. For each message-call transaction, `EVMTracer` invokes the tracer to replay the transaction, instruction by instruction. During replay, the tracer observes how run-time data is combined via the stack, memory and storage for each instruction. From these observations, RDGs are constructed. The sequence of executed instructions constitutes an instruction trace, or trace, for short.

Figure 4 illustrates the internals of the tracer. The internal state of the EVM (cf. Section II) consists of five elements (i.e., PC, memory, storage, stack, and message buffers). During execution, these elements affect each other and produce data dependencies. To record the effect of an executed instruction, the state of the EVM is observed by the tracer and recorded into the *shadow state*, which contains:



(a) Example source code **(b)** Assembly code (register based) **(c)** RDG of Figure 5b

(d) EVM bytecode (stack based) **(e)** Stack before `6: ADD` **(f)** RDG of Figure 5d

FIGURE 5. **Register-based and stack-based program representations and their corresponding RDGs.**

1) *Trace step*: An integer variable i that starts from 1 and increments by one for each instruction executed until the contract terminates. We can then uniquely identify the state of the EVM by referring to a specific trace step.
2) *Last Branch (LB)*: This variable records the trace step of the last branch instruction, such as `JUMPI` and `JUMP`. It is used to construct control-dependence edges in the graph.
3) *Shadow Structures*: The shadow structures record the data flow in the corresponding parts of the EVM state and are used to construct the data-dependence edges in the graph.
4) *Last Memory Modifier (LMM)*: An ad-hoc variable that is used to record the last instruction that extended the memory size of the EVM, which is used for recording the dependency for the `MSIZE` instruction.

Our RDG consists of trace steps and edges representing data and control dependencies. Data dependencies describe the data flow of the instructions, such that the instruction at trace step i must use the result of another instruction at trace step j to compute the correct state of the EVM. The control dependencies describe the program's path through the control-flow graph. An instruction at trace step i is executed only if a certain path was taken from a branch statement at trace step j. Formally, the graph $G(V, E)$ is defined by a set of nodes $G.V$ representing the trace steps i, and a set of edges $G.E \subseteq V \times V \times L$ where L indicates the type of dependency. We employ the following four dependence types from the literature [56], [57]: read-after-write (RaW), write-after-read (WaR), write-after-write (WaW), and control dependency. For each RDG G, we use a unique entry node *Root* and a unique exit node *End*. The *Root* node represents trace step 0 and is used for graph initialization. The *End* node represents the last trace step in the graph. Both nodes are used to cover corner cases that we will discuss later.

A. SHADOW STRUCTURES

Figure 5(a) depicts a source code example where we want to determine the dependency relations for the add operation in line 6. In the process of compiling this source code to assembly code of a register-based CPU, variables `b` and `c` have been assigned to registers `r1` and `r2`, which are the input operands of the `add` instruction in line 3 of Figure 5(b). To reveal those data dependencies at runtime, a tracer would have to record for each CPU register the trace step of the most recent update (line 1 for register `r1`, and line 2 for register `r2`). When execution reaches line 3, the register operands induce the dependencies of the `add` instruction for trace steps 1 and 2. Thus, the RDG in Figure 5(c) contains the dependence edges (1, 3) and (2, 3).

However, the EVM uses a stack-based execution model, where instructions encode the operation only, while operands are implicitly consumed from the stack and results produced onto the stack. The `ADD` instruction in the EVM bytecode in Figure 5(d) thereby relies on the `MLOAD` instructions in lines 3 and 5 to push its operands on the stack. Additional bookkeeping is thus needed in the tracer to capture these relations. Figure 5(e) shows the EVM stack for the bytecode in Figure 5(d) before the execution of the `ADD` instruction. For each stack slot, the numbers in parentheses depict the

trace step from which the stack value originated. The tracer maintains this bookkeeping information at runtime. Thereby it becomes evident that the top-most two stack slots that constitute the operands `b` and `c` of the `ADD` instruction originated from lines 3 and 5 in the executed bytecode, which is reflected in the dependence edges (3, 6) and (5, 6) in the RDG in Figure 5(f). Each `MLOAD` instruction depends on a prior `PUSH` instruction for the addresses of variables `b` and `c`, reflected in the dependence edges (2, 3) and (4, 5).

Similar bookkeeping information is required for the memory of the EVM, and for the storage of an account. For this purpose, we introduce shadow structures. Instead of storing actual values from the EVM state, the shadow structures store a set of trace steps by observing the state of their corresponding parts in the EVM state. In this section, we explain the details of each shadow structure. We apply function $\text{ts}_{\text{op}}(i)$ to map from trace step i to the opcode of the instruction executed at trace step i.

1) SHADOW STACK

Most of the intermediate results within the EVM are transferred through the stack. Therefore, dependencies occur when data from instruction a is transferred through the stack to instruction b. In this case, instruction a must strictly execute before instruction b, which constitutes a RaW dependency.

We implement a *shadow stack* named `SSTACK` to trace stack dependencies. The shadow stack stores trace steps and observes the effect that instructions executed by the EVM exhibit on the stack (we call the reader's attention to the subtle but vital distinction between the terms shadow stack and stack for the remainder of the exposition). We utilize the shadow stack and the following stack-related rules (s1) and (s2) to construct the dependence graph G.

(s1) When a value is pushed onto the stack at trace step i, the trace-step value i is pushed onto the shadow stack.

(s2) When a value is popped from the stack at trace step i, we pop the shadow stack's top-of-the-stack j and add the dependence edge (i, j, RaW) to the dependence graph G.

Intuitively, this means that when an instruction x pushes a value onto the stack, the subsequent instruction that uses that value has a RaW dependency on instruction x.

Two EVM instructions do not create values on the stack and need special treatment, i.e., `SWAP` and `POP`.

The EVM specification [24] employs a zero-indexing scheme for the stack such that `stack`[0] denotes the top-of-the-stack (the first stack item), `stack`[1] denotes the stack slot below the top (the second stack item), and `stack`[n] generally denotes the n^{th} stack slot below the top (stack item $n+1$). We adopt this indexing scheme for the shadow stack.

Let $\text{ts}_{\text{op}}(i)$ be a `SWAP`$\langle n \rangle$ instruction which swaps the values of stack slots `stack`[0] and `stack`[n]. After execution of trace step i, we create dependencies (i, `SSTACK`[0], RaW) and (i, `SSTACK`[n], RaW). Finally, we overwrite the values at `SSTACK`[0] and `SSTACK`[n] by i.

The `POP` instruction pops the top-of-the-stack and discards it, i.e., the discarded value is not needed for the computation. Therefore, no data nor control dependencies are required for this instruction—it is ignored in the dependence graph.

2) SHADOW MEMORY

The EVM provides a memory area representing a linear, zero-indexed sequence of bytes. Memory instructions like `MLOAD` and `MSTORE` can read from and write to this memory. Instructions like `CALL` and `CALLCODE` rely on memory to retrieve function arguments and provide a return value. The EVM automatically extends the memory size whenever necessary, and the `MSIZE` instruction returns the current memory size in bytes.

We apply an idea similar to the shadow stack by implementing a data structure called *shadow memory* (`SMEM`), which stores a linear sequence of trace steps. We use index operation `SMEM`[i] to represent the i^{th} value stored in the shadow memory, corresponding to the i^{th} byte of memory, i.e., `mem[i]`. The EVM initializes memory to zero. To comply with this convention, we initialize the shadow memory with our artificial *Root* node. Unlike the stack, memory is not restricted to last-in-first-out semantics, which allows WaR and WaW dependencies in addition to the RaW dependencies that occur on the stack.

If the instruction at trace step i interacts with memory at index n, we employ the following two memory-related rules (m1) and (m2) for constructing the dependence graph G.

(m1) For a value read from `mem[n]`, we create a RaW dependency from trace step i to `SMEM[n]`.

(m2) For a value written to `mem[n]`, we create a WaW dependency from trace step i to `SMEM[n]`. We then search the dependence graph G for nodes $\{v \mid (v, \texttt{SMEM[n]}, \mathit{RaW}) \in G.E\}$. For each node v, we create a WaR dependency from step i to step v. Finally, we update `SMEM[n]` to i to record the trace step that wrote memory location n.

Rule (m1) models the standard RaW dependency we have discussed so far where an instruction depends on the result of a previous instruction. The reason for rule (m2) is that if an instruction at trace step i overwrites the result of trace step j, we need to ensure that all instructions use the result of step j before executing step i. Otherwise, the input of those instructions would be overwritten by step i. Note that WaR and WaW are both known as false dependencies because they can be eliminated by using different memory locations to store different values [57]. E.g., a WaR dependency between $\text{ts}_{\text{op}}(i)$ and $\text{ts}_{\text{op}}(j)$ can be eliminated if the program is rewritten such that $\text{ts}_{\text{op}}(i)$ and $\text{ts}_{\text{op}}(j)$ operate on different memory locations. However, such transformations are not in the scope of this paper.

Finally, the `MSIZE` instruction reports the current size of the memory at runtime. The EVM grows the memory on demand. E.g., if the current size of the memory is 32 B and a write operation occurs that writes to the 42^{nd} byte in memory, the EVM will resize the memory to satisfy the request of the operation. Therefore, we record the last instruction's trace

step that extended the memory size in variable *Last Memory Modifier* (*LMM*) of the shadow state initialized to the *Root* node. Execution of instruction `MSIZE` entails the creation of a RaW dependency on the trace step stored in variable *LMM*.

3) SHADOW STORAGE

Unlike linearly-indexed memory, access to a variable in EVM storage requires two key-value maps: an `<Address, Account>` map to retrieve the target account and a `<Key, Value>` map to retrieve the value of the targeted storage variable of the selected account. For the purpose of this paper, it is sufficient to regard storage as a single `<Key, Value>` map where the `Key` comprises the account address and the key of the variable. Storage accesses exhibit the same ordering constraints as memory, i.e., RaW, WaR, and WaW. Similar to our approach with memory, we create a *shadow storage* (named `SSTORAGE`) to express dependencies between executed storage instructions.

Like the EVM storage, our shadow storage is a key-value map. Similarly, as the already introduced shadow data structures, it stores trace steps instead of actual values for tracing dependencies. Conceptually, each value in the shadow storage is set to the *Root* note before the start of the tracer. Practically—as the index space of the key-value map is infinite, and the actual keys are unknown ahead of the trace—the tracer starts with an empty key-value map. Whenever a non-existing entry is visited, it is initialized to the *Root* node to represent the initial state of the execution. The EVM instruction set contains only one storage write instruction, `SSTORE`, and one storage read instruction, `SLOAD`.

We collect dependencies similar to our approach for memory. In the following, let k be the storage key accessed at trace step i.

(p1) For a value read from storage at index k, i.e., `storage`[k], we create a RaW dependency from step i to `SSTORAGE`[k].

(p2) For a value written to `storage`[k], we create a WaW dependency from step i to `SSTORAGE`[k]. We then search the dependence graph for nodes $\{v \mid (v, \texttt{SSTORAGE[k]}, \text{RaW}) \in G.E\}$. For each node v, we create a WaR dependency from step i to step v. Finally, we update the shadow storage `SSTORAGE[k]` to i to record the trace step that wrote storage location k.

4) CONTROL DEPENDENCIES

A control dependency is a constraint due to the control flow of a program [56]. E.g., the instruction following a branch instruction can only be executed after the branch target has been decided. For this purpose, we employ variable *LB* to record the tracing step of the last branch instruction (variable *LB* is initialized to the *Root* node). For each instruction at trace step i, add edge (i, *LB*, *Ctrl*) to graph G. This effectively creates a strict control dependency among all basic blocks, so each basic block must be executed in order. Finally, after the trace is completed, we create an artificial node *End* such that *End* is dependent on all leaf nodes in the dependence graph.

5) EFFICIENT DEPENDENCE GRAPH CONSTRUCTION

In this section, we provide the technical details of how dependence graphs are represented inside `EVMTracer`. In our implementation, shadow structures store references to graph nodes instead of raw trace steps. (We chose trace steps in the exposition of the paper to facilitate reading and highlight the correspondence between graph nodes and trace steps.) A node in the dependence graph contains references to its dependent nodes (i.e., outgoing edges), dependencies (i.e., incoming edges), dependency types, and the corresponding trace step. During tracing, whenever a new node is created, its dependencies are fetched from the shadow structures in the form of references to graph nodes. Therefore, construction can be done immediately by creating a new graph node and connecting it with its dependencies which become the new node's incoming edges. A WaR dependency represents a special case that requires the tracer to search one level deeper: instead of fetching the immediate dependency directly from a shadow structure, the outgoing edges of the dependency need to be searched. This pertains to the implementation of Rule (m2) of the shadow memory and Rule (p2) of the shadow storage as introduced above.

B. RUNTIME DEPENDENCE GRAPH EXAMPLE

The left-hand side of Figure 6 depicts an example of a smart contract written in Solidity. It contains two storage variables `storage1` and `storage2` and a function `reset` that sets both storage variables to the value 42. The right-hand side of Figure 6 shows the EVM bytecode snippet corresponding to the `reset` function.

We show the process of the RDG construction in Figure 7. The example contains only RaW dependencies. Because the bytecode contains no branch instruction, it generally holds that line i of the bytecode from Figure 6 is executed at trace step i. Figure 7 provides one subfigure (i.e., 7(a)–7(g)) per trace step (i.e., 1–7). For each subfigure, the caption specifies the trace step and the executed instruction.

In trace steps 1 and 2, the EVM interpreter pushes two values onto the stack. The shadow stack observes those effects and records each stack value in the trace step. I.e., trace step values 1 and 2 are pushed on the shadow stack.

In trace step 3, `DUP2` is executed, duplicating the stack's second item. Therefore, a RaW dependency is created in the RDG from trace step 3 to trace step 1, which is the trace step that produces the duplicated value. (Note that the RDG node labeled "R" represents the *Root* node. Trace steps 1 and 2 do not depend on any actual trace step. To keep the RDG connected for practical matters to be discussed later, trace steps without dependencies depend on the *Root* node.)

In trace step 4, instruction `SWAP1` swaps the first two values on the stack. As shown in the previous shadow stack in Figure 7(c), those stack values were created in trace steps 2 and 3. Thus trace step 4 has RaW dependencies on

```
1 contract SimpleStorage_2 {
2     uint storage1;
3     uint storage2;
4
5     function reset() public {
6         storage1 = 42;
7         storage2 = 42;
8     }
9 }
```

```
1 PUSH1 0x2A   ;value 42 in base 10
2 PUSH1 0x0    ;address of storage1
3 DUP2
4 SWAP1
5 SSTORE
6 PUSH1 0x1    ;address of storage2
7 SSTORE
```

FIGURE 6. Smart contract (left) and the corresponding EVM bytecode of function `reset` (right).

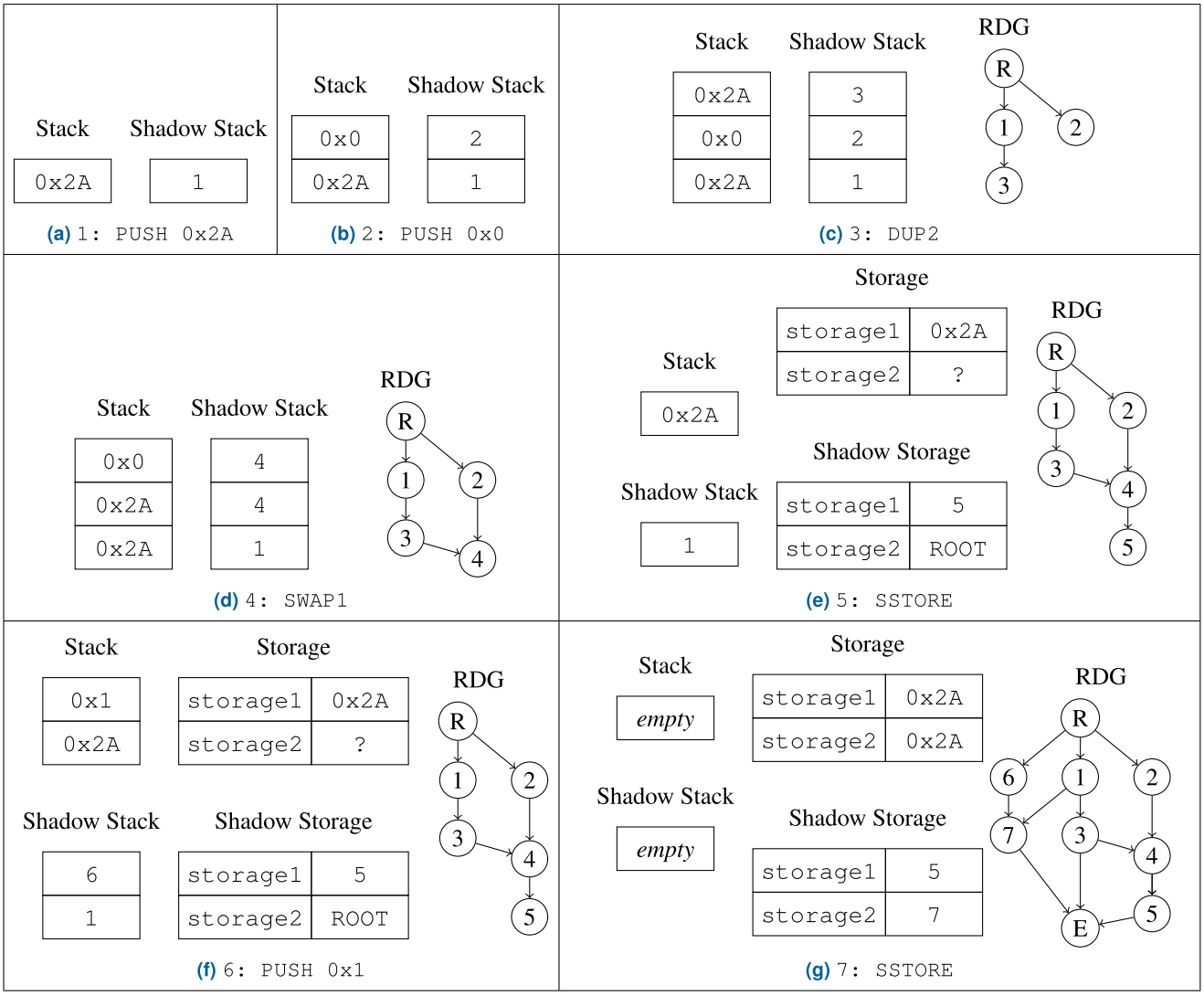


FIGURE 7. RDG construction example, all dependencies are RaW dependencies.

trace steps 2 and 3. The first two values of the shadow stack are updated to trace step 4, which produces the corresponding swapped stack values.

In trace step 5, the `SSTORE` instruction is executed, which consumes the first and second stack items. The first stack item constitutes an address in storage (for the sake of the example, we assume that the Solidity compiler has allocated storage variable `storage1` to storage location 0). The second stack item constitutes the value stored at the given address ($0 \times 2A$ in this example). The dependence edge (4, 5) is added to the

RDG to record the dependency of trace step 5. Meanwhile, the value of `storage1` in the shadow storage is rewritten to trace step 5, reflecting the last trace step that modified the storage value.

Trace steps 6 and 7 follow along the same lines. The final state of the EVM's stack and storage and the generated RDG are shown in Figure 7(g). The RDG node labeled "E" represents the *End* node.

C. EFFICIENT TRACING

Collecting the trace information of the whole chain requires the execution of all transactions. Therefore, `EVMTracer` is built on a *transaction replay mechanism* to replay transactions locally and collect metrics efficiently. With millions of blocks on the blockchain, it is crucial to have an efficient tracing system for rapid collection. However, blockchain programs are inherently sequential to execute block n, and the world state after block $n-1$ is required to be the input state. The Ethereum world state contains the data of all the accounts on the blockchain. Data is encoded with a recursive length prefix (RLP, [24, Appendix B]) and is associated with each account by key-value (KV) pairs. The KV pairs are then stored as Merkle Patricia tries (MPTs). The storage information can be accessed or updated at runtime by executing the `SLOAD` and `SSTORE` instructions. However, because of the size of the MPTs, the `SLOAD` and `SSTORE` instructions are prohibitively expensive in terms of performance [49].

There are several options when it comes to historical transaction replay. An Ethereum client can be configured as an *Ethereum archive node* to store the entire history of the Ethereum world state locally. A server like Geth then provides a JSON RPC server, where the client sends the request to the server containing the arguments and the ID of the block it wants to replay. The server then retrieves the world state at block $n-1$, replays block n and sends the result back to the client. However, this approach suffers from scalability issues due to the overhead of the JSON RPC API and the large amount of disk space occupied by the world state database. It requires 14 TB of disk space [61] to support full chain replays up to block 14 M and can take on the order of several weeks to complete the replay [29].

An alternative approach configures the client as an *Ethereum full node*. A client can download the world state at block $n-1$ and then replay and verify block n. However, to download a particular state, the client must be able to find a peer in the P2P network that can provide the requested world state. Such a peer may not be available, and a sought world state far behind the tip of the chain will likely not be provided. If such an absence is encountered, the client must compute all the local states to produce the state at block $n-1$. The empirical data reported in [29] suggests that replay from full nodes is essentially faster than archive nodes but still suffers from severe performance and storage issues.

Finally, the above approaches all lack built-in multi-threading features. Although it is possible to run multiple Geth JSON servers to increase replay throughput with archive nodes, the performance of the archive node is inherently slow. Adding more instances does not help much in practice. For full nodes with multiple instances, it is currently impossible to allow multiple full nodes to share the same copy of the world state. Therefore, each instance must obtain its copy of the database, further aggravating the storage issue.

We built our `EVMTracer` system on top of the Substate Replayer, a scalable and efficient replay system developed in [29]. By encoding the states of each transaction to their minimal form, the Substate Replayer can reproduce the transactions efficiently and in parallel. Figure 8 illustrates the overall replay system. To record a transaction in the Substate Replayer, it first executes the transaction by the EVM. It then records the minimal information necessary to reproduce the result faithfully. This includes the storage information, which is a map of `<Address, Key>` to `Value` that are accessed or modified during the execution, the transaction description and the transaction result. That related information is stored in the substate database. Most importantly, the replayer records only the set of KV pairs accessed or modified by a transaction. In practice, a transaction only involves a few account addresses. As a result, a substate's size is much smaller than the size of the complete world state at a given block height. The recording step is required only once. After all the substates are recorded, the replay can be performed off the chain. In comparison, the substate replayer can perform 2817 tx s^{-1} when replaying blocks from 0–9 M, being on average 4.54× faster than the full node replay from Geth and consuming 59 % less disk space [29]. More importantly, the substate replayer provides multi-threading support without disk-space overheads. Those characteristics make the Substate Replayer a viable candidate for collecting execution traces for `EVMTracer`.

For our work, we first obtained the recorded substates up to 12 M blocks from the Substate Replayer. Given the substate database, we replayed those transactions locally through the `EVMTracer` system. Because the analysis of each transaction is conducted off the chain and in isolation, `EVMTracer` parallelizes the tracing of transactions through the use of multiple threads. For each transaction, a tracer is invoked to record the runtime dependence graphs of the EVM. A tracer takes as input a substate, which contains the contract bytecode and the program input, and outputs the runtime dependence graphs. The dependence graphs are collected for subsequent analyses and thus constitute the input for our parallelization and redundant computation metrics. Figure 9 demonstrates the overall workflow of the tracer.

IV. CONTRACT-LEVEL PARALLELISM

Leveraging the parallelism inherent in computations through parallel programming [62], [63], [64], [65], [66], [67], [68] is an important aspect of high-performance systems. A parallel execution model calculates a schedule based on the dependencies between tasks to take advantage of the multiple computation units available on modern hardware to execute

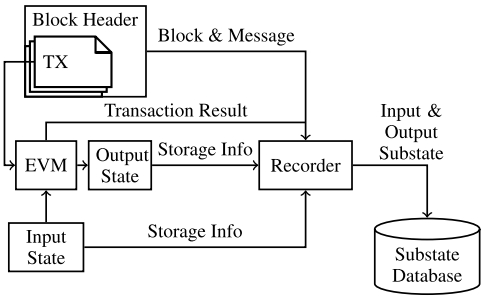


FIGURE 8. Substate Recorder Components from [29].

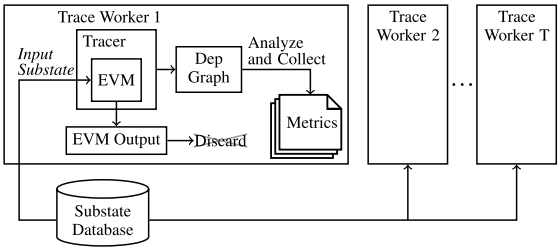


FIGURE 9. EVM tracer system (`EVMTracer`).

multiple tasks concurrently, thereby potentially increasing system performance.

In this section, we investigate the potential gain from contract-level parallelism, i.e., the performance improvement that can be obtained from scheduling the tasks of a message-call transaction so that the EVM can execute multiple instructions in parallel. The runtime dependence graph describes the dependence relations among tasks. E.g., from the final runtime dependence graph in Figure 7(g) we observe that the two `SSTORE` instructions at trace steps 5 and 7 do not have to be executed sequentially. Because `SSTORE` instructions generally incur a high execution-time overhead [49], executing them in parallel can substantially improve EVM performance.

The parallel execution model considers all dependence types, including false dependencies (WaR and WaW) and control dependencies. Although false dependencies can be avoided by replicating data in memory, extending the memory space is not free in Ethereum and consumes gas. Note that `EVMTracer` is an analysis tool and does not conduct program transformations in the manner of optimizing compilers. The model assumes zero communication cost among processors. The basic principle to calculate the theoretical speedup is first to obtain a cost function f_c which returns the estimated cost for the instruction at trace step i. Then, based on the cost of each task, calculate an instruction schedule L, which determines the execution order of instructions. Finally, we simulate the execution on n processors and compute the performance gain T_n.

We obtained our cost function from the empirical result of Baird et al. [49]. In their work, they measured the execution time of EVM instructions in segments of one million blocks. The performance model varies with block height because of (1) implementation changes in the EVM client due to protocol updates, performance improvements, and bug fixes, and (2) the costs of storage instructions like `SSTORE` and `SLOAD` that increase with block height.

We then devise a list-scheduling algorithm to compute the schedule L for the tasks. The scheduling problem is known to be NP-complete even with the assumption of zero communication cost and uniform resource distribution [69]. However, Graham [70] showed that any valid list-scheduling heuristic would be within a factor of $2\times$ of the optimal schedule. Even better, it has been shown empirically [71] that a greedy scheduling heuristic using the *critical path* (CP) is within 5 % of the optimal schedule in 90 % of all cases. Hence, for our experiment to obtain task schedules at scale, it is advantageous to employ greedy CP scheduling.

The CP scheduling algorithm always picks the node v with the longest path to the artificial end node (i.e., *End*). Intuitively, the CP scheduling algorithm greedily picks the node that leads to the path with the highest workload. We first calculate the priority function pri() with the help of backward induction. Let $N^+(v)$ denote the out-neighborhood of node v, which is the set of nodes o adjacent from v, i.e., connected by an outgoing edge (v, o). Then, pri(v) denotes the cost from node v to the *End* node:

$$\text{pri}(v) = \begin{cases} 0 & \text{if } v \text{ is the } \textit{End} \text{ node,} \\ f_c(v) + \max(\{\text{pri}(v') : v' \in N^+(v)\}) & \text{else.} \end{cases}$$

After obtaining the priority function, we simulate the bytecode execution on n processors to calculate its expected execution time. Our simulation, as stated in Algorithm 1, employs the dependence graph to identify instructions that have all dependencies met and are thus ready to execute. (For such a *ready* instruction, the corresponding node in the dependence graph has no incoming edges.) Ready instructions are scheduled for execution on the *executor*, a simulated n-way superscalar EVM interpreter that we employ to record the overall execution time of the bytecode instructions. When the executor has completed the execution of an instruction, the node of the corresponding trace step is removed from the dependence graph, which will render subsequent instructions ready for execution. The simulation terminates when all nodes of the dependence graph have been executed.

Our simulator employs priority queues [72], which are sets of elements where each element has a *key* associated. Given a set S, operation Insert(S, x) will add element x to the set. A min-priority queue provides operation ExtractMin(S), which will remove and return the element with the smallest key from S. Likewise, a max-priority queue provides operation ExtractMax(S) to remove and return the element with the largest key.

In lines 7–9 of Algorithm 1, the simulator inserts all nodes from the vertex set $G.V$ of the dependence graph that have an in-degree of 0 into the ready-queue S. The nodes with an in-degree of 0 are those nodes which do not have incoming dependence edges and hence contain instructions that are

Algorithm 1 Bytecode Execution Simulation for n Processors

Input:
pri: priority function
f_c: instruction cost function
G: dependence graph
n: number of processors

Output:
t: execution time utilizing n processors

```
1:  function simulateExecution(pri, f_c, G, n)
2:      t ← 0
3:      S ← ∅        ▷ Ready instructions max-priority queue
4:      Exec ← ∅     ▷ Executor min-priority queue
5:      while G.V ≠ ∅ do
6:          ▷ Add nodes n with no incoming edges in G to
            priority queue S, according to node's priority
            pri(n): ◁
7:          for each n ∈ G.V s.t. deg⁻(n) = 0 do
8:              n.key = pri(n)
9:              Insert(S, n)
10:         ▷ While instructions ready and executor has
            vacant slots: ◁
11:         while S ≠ ∅ and |Exec| ≤ n do
12:             ▷ Get node i with highest priority: ◁
13:             i ← ExtractMax(S)
14:             ▷ Insert instruction i in executor: ◁
15:             i.key = f_c(i)
16:             Insert(Exec, i)
17:         ▷ Execute shortest-running instruction i: ◁
18:         i ← ExtractMin(Exec)
19:         t_i = i.key
20:         t ← t + t_i
21:         ▷ Update costs of instructions in executor: ◁
22:         Temp ← ∅
23:         for each n ∈ Exec do
24:             n.key = n.key − t_i
25:             Insert(Temp, n)
26:         Exec ← Temp
27:         ▷ Update dependence graph G: ◁
28:         remove node i and its outgoing edges in G
29:     return t
```

ready to execute. The loop in lines 11–16 fills executor *Exec* with instructions from the ready queue up to the executor's maximum capacity of n instructions. Because S is a max-priority queue where the key is the priority of a node, the extraction operation in line 13 will always choose the ready instruction with the highest cost towards the *End* node. (As mandated by the CP scheduling algorithm.)

The executor itself is a min-priority queue, where the key is the execution-time cost of instructions (see lines 15–16). The executor will thereby always select the shortest-running instruction for execution (line 18). The executor adds the instruction's execution time t_i to the accumulated time t (lines 19–20). Because execution is n-way parallel, the cost of each of the remaining instructions in the executor is reduced by t_i (lines 22–26). We update the dependence graph by removing node i and its outgoing dependencies (line 28). The simulator iterates until all the vertices in the dependence graph have been processed.

To find out the theoretical maximum speedup that can be obtained from parallel execution of independent instructions, we determine the bytecode execution time for a superscalar EVM interpreter with an infinite number of parallel execution units. In this case, no simulation is needed because the execution time is equivalent to the length of the critical path in the dependence graph G, starting from the artificial root node *Root* to the artificial end node *End*. In what follows, we write T_n for the bytecode execution time achieved by an n-way superscalar EVM interpreter, and T_∞ for the execution the time when utilizing an infinite number of execution units. Note that T_∞ is equivalent to the cost of the critical path in G, which is computed during the scheduling step, i.e., pri(*Root*).

V. REDUNDANT COMPUTATIONS

Let $\texttt{EVM}_i$ be the input state of the EVM at trace step i. The instruction executed at trace step i constitutes a side effect f_i that produces an output state $f_i(\texttt{EVM}_i)$. The instruction at trace step i is considered *redundant* if there exists a previously-executed instruction of the same operation type with side effect f_j that results in $f_i(\texttt{EVM}_i) = f_j(\texttt{EVM}_i)$. Intuitively, this means that there is no need to compute the result of the instruction at trace step i if we can memoize and reuse the result of the instruction at trace step j.

For this work, we are interested in the number of redundant memory and storage instructions, i.e., `MLOAD`, `MSTORE`, `SLOAD` and `SSTORE`. Instructions `MLOAD` and `MSTORE` are used for accessing the EVM memory. They enable features like dynamic data structures and customized data types in high-level smart-contract languages. The `SLOAD` and `SSTORE` instructions are the only means for the user to interact with the storage of an Ethereum contract, and they are the most expensive instructions in terms of gas cost and execution-time overhead [49]. Therefore, we are focusing on those four instructions for investigating redundant computations on the Ethereum blockchain.

We use the runtime dependence graph to determine redundancy. This is done by finding instructions that have the same sequence of data dependencies in the dependence graph. For these metrics, we ignore WaR, WaW, and control dependencies because they do not concern the input data flow of an instruction, but only the order in which the instructions should be executed.

Our redundancy analysis focuses on each contract in isolation, that is, an external contract call will generate its own (separate) metrics. We accomplish this distinction by identifying contract call instructions (e.g., `CALL`, `CALLCODE`) in the dependence graph such that redundancy is only counted within the same contract call.

The dependence graph we build represents each trace step as an individual node. To find the instructions that share

```
1  contract SimpleStorage_3 {
2      uint storage1;
3      uint storage2;
4      uint init;
5
6      function reset() public {
7          storage1 = init;
8          storage2 = init;
9      }
10 }
```

```
1  PUSH1 0x2    ;address of init
2  SLOAD
3  PUSH1 0x0    ;address of storage1
4  SSTORE       ;storage1 = init
5  PUSH1 0x2    ;address of init
6  SLOAD
7  PUSH1 0x1    ;address of storage2
8  SSTORE       ;storage2 = init
```

FIGURE 10. Smart contract (left) and the EVM bytecode of function `reset` (right).

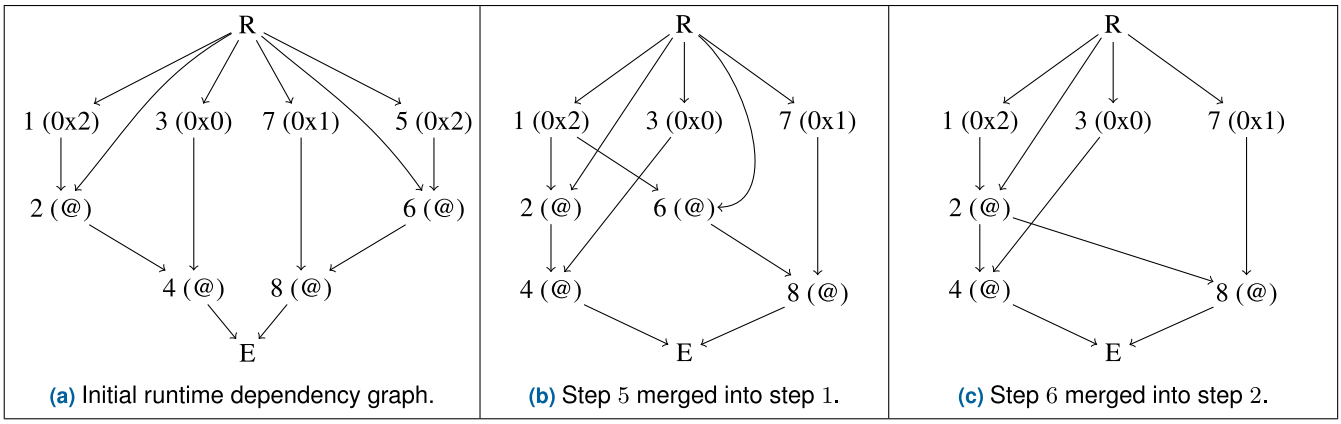


FIGURE 11. Reduction of the runtime dependence graph (a) to its minimal form (c).

the same data dependencies, we reduce the data dependence graph to its *minimal form*, i.e., we merge redundant instructions into a single node.

We classify the data input of instructions into two categories, either *stateless* or *stateful*. Stateless input constitutes input values that are always the same for each execution of a given instruction. E.g., the `PUSH` instruction pushes a constant onto the stack; the value being pushed is statically encoded in the bytecode format and hence stateless. Another example is the `SLOAD` instruction, where one of its input values is the address of the target contract. The context is the current contract address if invoked through the `CALL` instruction. If invoked through the `DELEGATECALL` instruction, the context is the caller's address. This value is predetermined before execution starts and is always the same within the same contract execution; therefore constitutes a stateless input. Our dependence graph records the stateless inputs for each trace step. We use function $f_{\text{stateless}}(i)$ to return an ordered list of stateless inputs for the instruction at trace step i. Likewise, function $f_{\text{RaW}}(i)$ returns an ordered list of inputs from RaW dependencies at trace step i.

A stateful input is an input value that has been created by another instruction and that is dependent on the EVM state (i.e., stack, memory, or storage). Stateful inputs are described by the data dependence graph. For example, the second argument of `SLOAD` is the storage key to load from. This value is always read from the top of the stack.

We can then find redundant instructions and merge them into the same node incrementally. Starting at trace step $i = 1$, for each trace step, an instruction is redundant if there exists another instruction at trace step j such that the following four conditions hold.

(c1) $j < i \wedge$
(c2) $\text{ts}_{\text{op}}(i) = \text{ts}_{\text{op}}(j) \wedge$
(c3) $f_{\text{stateless}}(i) = f_{\text{stateless}}(j) \wedge$
(c4) $f_{\text{RaW}}(i) = f_{\text{RaW}}(j)$.

Intuitively, the instruction at trace step i is redundant if there exists a trace step j prior to step i in the trace, trace steps i and j contain the same instructions, and the instructions agree on the stateless inputs and on the RaW dependencies.

E.g., consider the example program and its corresponding bytecode in Figure 10. The example is similar to the one in Figure 6, with the only difference that the two storage values are reset to another storage value `init`. Figure 11(a) shows the final dependence graph produced by the tracer. All edges represent RaW dependencies; With each trace step i we depict the stateless input $f_{\text{stateless}}(i)$ next to it. E.g., trace step 1 is a `PUSH` instruction associated with the value 0×2, and the `SLOAD` and `SSTORE` instructions are associated with

TABLE 1. Hardware specification for experiments.

Evaluation Environment	
CPU	AMD Ryzen Threadripper 2990WX 32 cores 2 hyperthreads per core
RAM	128 GB
Disks	5 × Seagate ST6000NM0235 6 TB SATA HDD (7,200 RPM, 256 MB cache)
OS	Ubuntu Linux 22.04.01 LTS

the contract address, which is represented by @ (they are all the same in this example). Starting at trace step 1, the first redundancy is found at trace step 5, where the two push instructions `1:PUSH` 0×2 and `5:PUSH` 0×2 are associated with the same stateless input 0×2 and only dependent on the *Root* node. After merging trace step 5 into 1, the next redundancy is found at trace step 6 which coincides with trace step 2, where the two `SLOAD` instructions share the same data dependencies (trace step 1 and *Root*) and are associated with the same contract address. The remaining trace steps 7 and 8 do not yield further redundancies showing that the dependence graph in Figure 11(c) is in its minimal form, and one redundant `PUSH` and one redundant `SLOAD` instruction have been identified.

VI. EVALUATION

The hardware specification of our evaluation platform is stated in Table 1. This platform is an on-premise bare metal server maintained by the authors of this study. To accommodate the substate database and the trace data, an array of five 6 TB Seagate HDDs was used.

Our tracing stage employed 64 worker threads and collected statistics for the initial 12 M blocks of the Ethereum mainnet. The details of the runtime dependence graphs for each segment of 1 M blocks are depicted in Table 2. Tracing finished within three days and resulted in 614 million runtime dependence graphs, which amount to 29.13 TB of data. The number of traces and the size of the dependence graphs increase in the number of blocks (i.e., with block height), which indicates that the number of transactions per block and the complexity of the computations inherent in transactions have been steadily increasing since the inception of Ethereum in 2015.

A. PARALLELISM METRIC

Regarding our parallelism metric we investigate the following research questions:

1) How much parallelism can be leveraged at runtime?
2) How many execution units do we need to obtain a given speedup?
3) Is it worth developing contract-level parallelism for future smart contract execution models?

Table 3 depicts the average speedup for each segment of 1 M blocks. We write $\overline{T_\infty}$ to denote the theoretical maximum speedup from an infinite number of execution units, and $\overline{T_n}$ for the speedup using n execution units. It follows from column "$\overline{T_\infty}$" that the parallelism inherent in the bytecode is similar across all segments. The largest speedup of a factor of 2.29× is observed with the segment 2–3 M, but we note that in this segment, the Ethereum network was facing denial-of-service (DOS) attacks that exploited underpriced EVM instructions to slow down block processing [73]. The 2–3 M segment may thus not be the most representative wrt. the performance of main-stream smart contract workloads. Leaving out this segment, we obtain a geometric mean [74] for the theoretical maximum speedup $\overline{T_\infty}$ of 1.90×. As can be observed from columns "$\overline{T_2}$"–"$\overline{T_8}$", the speedup from increasing the number of execution units levels off quickly. Four execution units already leverage a large part of the inherent parallelism, and the performance with eight execution units is very close to the theoretical maximum speedup.

Figure 12 shows the distribution of the theoretical maximum speedup. We define the performance ratio as $1/\overline{T_\infty}$

TABLE 2. Experimental results per segment of 1 M blocks.

Segment	# traces (1×10^3)	Avg. size (B)	Throughput (blocks/s)	Data size (TB)
0–1 M	355	2,481	504.37	0.03
1–2 M	975	1,201	471.97	0.02
2–3 M	2,595	1,754	121.16	0.12
3–4 M	7,812	590	113.65	0.10
4–5 M	46,994	1,049	105.20	1.24
5–6 M	68,835	1,431	48.31	2.19
6–7 M	54,012	2,474	45.17	3.21
7–8 M	64,664	1,952	43.46	2.32
8–9 M	73,430	2,070	30.43	2.90
9–10 M	72,799	2,523	33.50	4.09
10–11 M	108,996	2,443	20.55	5.47
11–12 M	118,133	2,901	15.42	7.45

TABLE 3. Average obtainable speedup T_n for an n-way superscalar EVM interpreter, per segment of 1 M blocks.

Segment	$\overline{T_\infty}$	$\overline{T_2}$	$\overline{T_4}$	$\overline{T_8}$
0–1 M	1.93×	1.42×	1.74×	1.89×
1–2 M	1.94×	1.41×	1.76×	1.92×
2–3 M	2.29×	1.27×	1.57×	1.63×
3–4 M	1.93×	1.41×	1.77×	1.91×
4–5 M	1.98×	1.43×	1.82×	1.96×
5–6 M	1.95×	1.43×	1.79×	1.93×
6–7 M	1.90×	1.40×	1.75×	1.88×
7–8 M	1.89×	1.41×	1.77×	1.88×
8–9 M	1.84×	1.39×	1.71×	1.84×
9–10 M	1.85×	1.39×	1.71×	1.84×
10–11 M	1.86×	1.40×	1.72×	1.86×
11–12 M	1.86×	1.40×	1.73×	1.86×

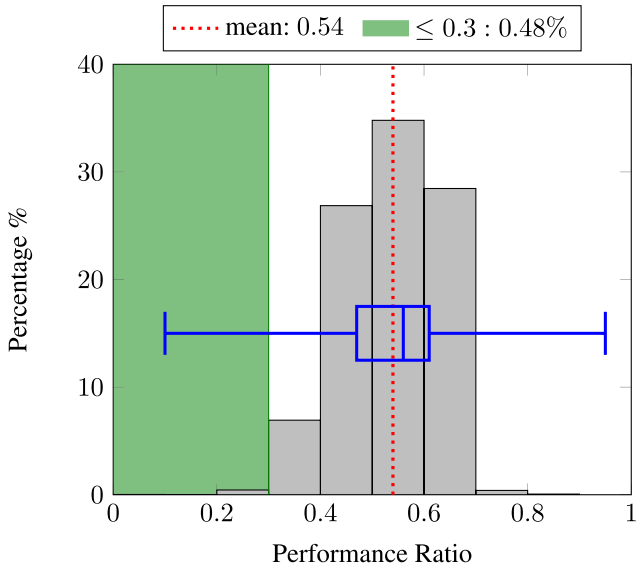


FIGURE 12. Distribution of parallelism potential. Performance ratio = $1/T_\infty$.

to normalize the speedup. E.g., a performance ratio of 0.2 corresponds to a 5× speedup. We observe a median performance ratio of 0.56 at a speedup of 1.79×. The lower and upper quartiles are 0.47 and 0.61, respectively. Half of all transactions thus exhibit a speedup in the range 1.63–2.12×. Only 0.48 % of all transactions have a performance ratio less or equal to 0.3 to achieve a theoretical maximum speedup ≥3.33×.

It is instructive to draw a comparison of these performance figures from the EVM with the instruction-level parallelism (ILP) observed with CPU architectures. The computer architecture field has a 25-year history of research and development into ILP, which has produced comprehensive quantitative data about the performance of various workloads on contemporary, register-based microarchitectures [75], [76], [77], [78], [79]. The EVM, in comparison, is a novel, stack-based virtual machine architecture with smart contracts as its only workload.

Despite these inherent differences, we observe the following striking similarity: the reported speedup that ILP yields from leveraging the parallelism within basic blocks is on average 1.74×, median 1.5× [77]. In comparison, the median speedup obtained in our study is 1.79×. (Note that our use of control dependencies between basic blocks enforces in-order execution of basic blocks.) Although advances in computer architecture (e.g., speculative execution [75], register renaming [80], alias analysis [81], predicated execution [82]) and compilers (e.g., loop unrolling, software pipelining [56]) further increased ILP, the diminishing returns from those measures prevented modern CPU designs from exceeding 8-fold instruction issue [75, Fig. 3.46], and gains have been found diminishing already beyond 4-fold instruction issue [76].

To assess the question as to what extent the ILP in EVM bytecode can be further leveraged by the before-mentioned

TABLE 4. Redundant computations and wasted resources in the initial 12 M blocks of the Ethereum mainnet. For each instruction, we provide the total number of occurrences (non-redundant plus redundant), the number of redundant occurrences (out of the total), and the wasted resources in terms of Ethereum gas and USD. All quantities stated in units of million (M). Gas price and Ethereum price are averaged at 80.50 gwei per gas unit and 2453.65 USD per Ether, using the historical average price [84], [85] from Jan. 2021 until Nov. 2022 (the time of writing).

Instruction	Total (M)	Redundant (M)	Units gas (M)	USD (M)
SLOAD	9,173	3,208	1,284,891	253.79
SSTORE	2,348	0.90	2,780	0.55
MLOAD	21,742	11,858	35,576	7.03
MSTORE	31,190	4,367	13,103	2.59

computer-architectural measures, we note that the EVM meters the execution of bytecode instructions in units of gas, and that any measure that increases the number of executed bytecode instructions (including failed speculative execution across basic blocks due to branch misprediction) will inevitably increase the gas cost of transactions and thus will not be economical on the Ethereum platform. Likewise, the amount of storage used by a smart contract is metered, and measures that may increase the storage size of a smart contract (including duplication of storage locations to eliminate false dependencies) will not be economical. In contrast, compilation techniques such as software pipelining and loop unrolling can potentially be adopted for the economic constraints of the EVM's execution model. We consider this as an area of future research that exceeds the scope of this paper, which focuses on the bytecode deployed on the Ethereum mainnet. The contribution of `EVMTracer` for such work will be the provision of a suitable metric that can readily be computed for a smart contract deployed on a testnet.

To answer our third research question, our experiments suggest that the bytecode already deployed on the Ethereum mainnet contains a non-negligible amount of ILP (1.79×). To make its exploitation profitable, optimizing compilers may be required to further increase the inherent amount of ILP in bytecode. But optimizing-compiler support for the blockchain domain is reported to be immature and lacking optimizations for ILP [41]. Smart contracts have been found to contain a sufficient amount of loops [83], which provides the potential for the before-mentioned optimizations on software pipelining and loop unrolling, as long as the inherent economic constraints of the Ethereum gas cost model can be met. The Ethereum community has been found to anticipate new language and compiler versions quickly [42], but already-deployed contracts are immutable and hence unaffected by future changes in the compiler.

B. REDUNDANT COMPUTATIONS

With our redundancy analysis, we focus on the `MLOAD`, `MSTORE`, `SLOAD` and `SSTORE` instructions to conduct

TABLE 5. Average number of redundant instructions per transaction.

Blocks (M)	SLOAD		SSTORE		MLOAD		MSTORE	
	Total	Redundant	Total	Redundant	Total	Redundant	Total	Redundant
0–1 M	26.01	11.96	6.44	2.1×10^{-4}	38.10	25.04	120.36	6.80
1–2 M	26.86	8.97	4.81	19×10^{-4}	28.46	19.48	50.97	6.00
2–3 M	8.90	2.46	1.55	39×10^{-4}	8.72	5.46	14.11	1.31
3–4 M	9.22	3.21	2.57	2.6×10^{-4}	14.95	9.32	20.09	2.64
4–5 M	10.69	3.93	3.12	2.0×10^{-4}	22.44	14.69	31.11	4.20
5–6 M	15.57	6.22	4.39	10×10^{-4}	28.41	18.62	41.96	8.55
6–7 M	19.68	7.40	4.83	11×10^{-4}	36.51	22.81	54.97	10.41
7–8 M	18.17	6.91	4.80	17×10^{-4}	36.27	20.49	59.27	8.36
8–9 M	21.13	7.87	5.54	9.2×10^{-4}	44.55	23.36	64.75	9.55
9–10 M	22.95	7.86	4.91	16×10^{-4}	57.59	28.85	85.42	11.09
10–11 M	19.66	5.24	4.99	28×10^{-4}	59.30	30.38	77.63	9.38
11–12 M	16.42	4.29	5.03	23×10^{-4}	61.33	34.89	81.23	10.22

a quantitative analysis to answer the following research questions:

1) What is the amount of redundant computations that smart contracts execute at runtime?
2) What is the cost overhead of redundant computations in terms of gas and USD?

Table 4 summarizes the redundant computations and wasted resources for the initial 12 M blocks of the Ethereum mainnet. Among the surveyed instructions, `SSTORE` exhibits the most efficient utilization with a redundancy rate below 0.1 % of all executed instructions. Next, in terms of efficient utilization is the `MSTORE` instruction, with a redundancy rate of 14 %. Efficiency decreases further with the `SLOAD` and `MLOAD` instructions, of which we find that 34.97 % of all `SLOAD` instructions and 54.54 % of all `MLOAD` instructions are redundant. Consequently, the amount of wasted gas and its monetary equivalent of 253.79 million USD for `SLOAD` and 7.03 million USD for `MLOAD` are substantial. Connecting these findings with quantitative performance data reported for the Ethereum blockchain [49], we note that with the later blocks in the cited study (i.e., the range 5–8 M), the EVM on average had to spend more than 75 % of the overall bytecode interpretation time on `SLOAD` instructions. Therefore, the execution-time overhead of the `SLOAD` instructions that `EVMTracer` detected to be redundant can be expected to be non-negligible and must be tackled to reach the transaction throughput goals of Ethereum (cf. Section I).

We attribute the low redundancy rate of `SSTORE` to its prohibitively high gas cost. The Ethereum specification as of Oct. 2022 [24] charges 2900 units gas for an `SSTORE` instruction (category G_{sreset}), compared to 3 units gas for `MSTORE` ($G_{verylow}$). Programmers of smart contracts thus always had a high incentive to eliminate redundant storage write operations already in the source code, e.g., by deferring updates to the storage until after the entire computation has been accomplished. Related, a study from the year 2020 already found the Solidity compiler `solc` [46] to cache the most frequently accessed storage data in memory [41]. The historical data that `EVMTracer` collected for redundant instructions *per transaction* as depicted in Table 5 supports these facts: the overall number of `SSTORE` instructions (column "SSTORE.Total") per transaction does not significantly change across the 1-million segments of blocks on the Ethereum mainnet. However, we observe a steady increase in the number of `MSTORE` instructions (column "MSTORE.Total").

We present two contributors for the high redundancy rate of `MSTORE` instructions. First, compilers of smart contracts will cache storage data in memory, as mentioned above, which will convert redundant `SSTORE`s into redundant `MSTORE`s. Second, it has been reported on the Solidity GitHub repository that the Solidity compiler will generate redundant `MSTORE` instructions under certain circumstances. In particular, we have found two issues related to redundant `MSTORE`s, and both can be detected by our analysis. Issue 12211 [86] mentions that redundant `MSTORE` instructions are issued when copying struct objects. Issue 10755 [87] outlines a possible improvement of the compiler so that `MSTORE` will no longer store zero values in locations that are known to be zero already.

The high number of redundant `MLOAD` instructions are expected as they are related to the EVM's stack model of execution. Interpreters that employ a stack instead of registers to hold temporary results during the evaluation of a program are attractive for their compact bytecode representation, which spurred the network computing ecosystem of Java [88], Microsoft's .NET common language runtime [89], and several VM infrastructures for sensor networks [90], [91], [92], [93]. Stack-based interpreters found renewed interest with blockchains where the small code size of smart contracts reduces the on-chain storage requirements, e.g., with the EVM [24], TVM [94], and WebAssembly [95]. Stack code

is compact because instructions only encode the operation (op-code), while operands are implicitly consumed from the stack and results produced on the stack. Operands in registers can be used several times, but a stack operation consumes its operands. To reuse a value computed on the stack at a later time, the compiler will have to generate code that spills the value into a temporary local variable in memory from where it can be later reloaded onto the stack (facilitated by the EVM's `MSTORE` and `MLOAD` instructions). Stack access is more efficient than access to local variables, which spurred research into optimizations that convert local variable accesses to stack accesses [96], [97], [98], [99], [100]. But their adoption for the Solidity compiler has not been confirmed yet [41], [46]. The EVM instruction set provides the bytecode instructions `SWAP`$\langle n \rangle$ and `DUP`$\langle n \rangle$, which allow to directly address the 16 top-most stack slots and hence facilitate such optimizations. However, the margin of profit is narrow: the `MSTORE` and `MLOAD` instructions are in the same gas-cost category as the before-mentioned stack manipulation instructions ($G_{verylow}$). Storing and later reloading a temporary value will thus cost six units gas, and any optimization that caches the temporary value on the stack must stay below this limit to be profitable.

In our metric, if a value is referenced more than once in the program, subsequent loads will result in data redundancies in the trace. Therefore, the high redundancy observed with `MLOAD` instructions is an artifact caused by the EVM architecture. Even for a highly-optimized, register-based interpreter the code size overhead has been found to be 26 % [101], which means that the stack-based architecture of the EVM is likely to stay and the before-mentioned optimizations, as well as design changes in the EVM instruction set that benefit such optimizations are necessary means to mitigate the runtime overhead. As mentioned with the parallelism metric, the contribution of `EVMTracer` can be the provision of a metric that is readily computable to guide the design and development effort.

Finally, it follows from Table 4 that 34.97 % of all `SLOAD` instructions have been identified as redundant. Unlike `MLOAD`, where the current infrastructure does not have a more economical solution to cache a repeatedly used value, storage values can indeed be cached in memory or on the stack to reduce gas consumption and improve performance—storage instructions are much more expensive compared to memory and stack instructions. E.g., as of Oct. 2022 [24], loading a value from storage for the first time during contract execution costs 2100 units of gas ($G_{coldsload}$). For subsequent loads ($G_{warmaccess}$), the cost is 100 units of gas. Those costs are substantially higher than the costs of the `MLOAD` and `MSTORE` instructions and thus make caching a profitable target for further performance optimizations.

VII. RELATED WORK

A. ILP IN COMPUTER ARCHITECTURE

Research in computer architecture has a 25-year history in ILP, which has produced comprehensive quantitative data about the performance of a variety of workloads on contemporary, register-based microarchitectures [75], [76], [77], [78], [79]. In comparison, the EVM is a stack-based virtual machine architecture implemented in software, with operations for persistent storage, and smart contracts as the sole workload. Our parallelism metric models all dependencies of the EVM runtime environment, including dependencies related to the EVM stack and to persistent storage. By restricting our metrics to EVM bytecode, they are applicable to all smart contracts deployed on the Ethereum mainnet, and they can be readily applied with testnet-based development environments. As discussed in Section VI-A, the cost model of the EVM charges a nominal gas fee per executed bytecode instruction, which renders many ILP mechanisms from computer architecture infeasible on the EVM. Prior work in the area of ILP in computer architecture does not support our redundancy metric.

B. CONCURRENCY CONTROL MECHANISMS IN SOFTWARE

A large body of work [102], [103], [104], [105], [106], [107], [108] focuses on execution schemes that facilitate the execution of multiple smart contract transactions in parallel. This is different from exploiting parallelism within the contract itself. Muchhala et al. [109] purpose a system that allows multiple nodes to execute smart contracts using a MapReduce approach with a focus on Big Data applications. Other, more generic approaches in the virtual machine community enable parallelism by redesigning the execution model of Python interpreters [110], [111], adding specialized instructions to access data and compute resources in parallel for Java [112], enabling thread-safe built-in collections [113], and facilitating accelerator-assisted garbage collection [114], [115]. However, no work has been done yet in the context of smart contract virtual machines. Our study is the first to investigate the parallelism inherent in Ethereum transactions, and at scale.

C. REDUNDANT COMPUTATIONS

Our redundancy metric determines the number of redundant computations that occur in a smart contract at runtime. Many optimizations exist that aim at eliminating redundant computations. They can be categorized into static and dynamic approaches. Common subexpression elimination and partial redundancy elimination are static optimizations that are based on control flow analysis [51], [56], [116]. *Memoization* [117], [118], [119] is a dynamic redundancy elimination technique that trades memory space for performance. For each executed function call, the input argument values and computed results are cached in a lookup table. If the function is subsequently called with the same argument values, the cached results will be returned to avoid re-computation. For the stack model of execution, a large body of work has been conducted on compiler optimizations that replace local variable accesses by stack accesses [96], [97], [98], [99], [100]. In [120], dynamic

instruction scheduling is performed to reduce the stack usage of a JVM. In the process of improving the compiler optimization pipeline, the Ethereum community has identified several optimization opportunities regarding redundant operations, as evident in issues 10690, 10755, 12211, 12460, 12735, and 12755 from the Solidity GitHub repository [121]. Those issues are open and waiting to be fixed.

D. TRACING

Tracing has found various applications, including coverage analysis [122], performance profiling [123], and validation [124]. On the topic of blockchain, Chen et al. [125] introduce Forerunner, which pre-executes transactions and generates constrained-based, highly-optimized program representations. The technique is similar to a tracing just-in-time compiler [126] and relies heavily on tracing to collect runtime information for future optimization. Different from `EVMTracer`, their system is an optimizer and works in real-time to speed up the transaction throughput through speculative execution. Ding et al. [127] introduce a prototype system, SCMon, for monitoring smart-contract runtime behavior such as function execution time, function call graphs and gas consumption. The system uses an instrumented tracing technique that requires the source code of a smart contract. A similar shadow stack is implemented to track function calls. Their experimental results are restricted to a set of synthetic smart contracts. In contrast, our system does not require the smart-contract source code and is able to provide at-scale statistics of all historical transactions on the Ethereum blockchain. Because the metrics of `EVMTracer` require more fine-grained statistics to construct runtime dependence graphs, we had to implement a complete set of shadow data structures, including stack, memory, and storage.

VIII. CONCLUSION

We have introduced `EVMTracer`, an offline tracing framework to obtain runtime dependence graphs during transaction execution on the EVM. From the runtime dependence graphs, `EVMTracer` computes two valuable metrics: (1) contract-level parallelism and (2) redundant computations. We used `EVMTracer` to collect the runtime dependence graphs and compute both metrics for the initial 12 M blocks on the Ethereum mainnet. We found that Ethereum smart contracts include a non-negligible amount of contract parallelism, with a geometric mean of 1.90× theoretical maximum speedup. Wrt. redundant computations, we found that transaction execution is affected by a high number of redundant `MLOAD` and `SLOAD` instructions. The redundant `MLOAD` instructions are related to the EVM's stack model of execution and further work in compiler optimizations is required to mitigate this overhead. Redundant `SLOAD` instructions, by their high gas costs, have already caused significant economic damage and should be treated as a priority. Overall, we have shown that `EVMTracer` is capable of performing large, at-scale runtime tracing of Ethereum transactions and we have used the obtained runtime dependence graphs for the computation of two metrics to shed light on the design of future blockchain engines and smart contract compilers.

AVAILABILITY

The source code of the `EVMTracer` infrastructure is publicly available at https://github.com/verovm/evmtracer.

REFERENCES

[1] Y. Chang, E. Iakovou, and W. Shi, "Blockchain in global supply chains and cross border trade: A critical synthesis of the state-of-the-art, challenges and opportunities," *Int. J. Prod. Res.*, vol. 58, no. 7, pp. 2082–2099, Apr. 2020, doi: 10.1080/00207543.2019.1651946.

[2] Y. Yang and A. Irrera. (2022). *Clearinghouse's Blockchain-Based Settlement System Goes Live*. Accessed: Dec. 1, 2022. [Online]. Available: https://www.bloomberg.com/professional/blog/clearinghouses-blockchain-based-settlement-system-goes-live

[3] Y. Wang, Z. Su, J. Ni, N. Zhang, and X. Shen, "Blockchain-empowered space-air-ground integrated networks: Opportunities, challenges, and solutions," *IEEE Commun. Surveys Tuts.*, vol. 24, no. 1, pp. 160–209, 1st Quart., 2022.

[4] T. M. Fernandez-Carames and P. Fraga-Lamas, "A review on the use of blockchain for the Internet of Things," *IEEE Access*, vol. 6, pp. 32979–33001, 2018.

[5] B. Liu, L. Xiao, J. Long, M. Tang, and O. Hosam, "Secure digital certificate-based data access control scheme in blockchain," *IEEE Access*, vol. 8, pp. 91751–91760, 2020.

[6] L. Wei, J. Wu, and C. Long, "Blockchain-enabled trust management in service-oriented Internet of Things: Opportunities and challenges," in *Proc. 3rd Int. Conf. Blockchain Technol.*, Mar. 2021, pp. 90–95, doi: 10.1145/3460537.3460544.

[7] L. Marchesi, K. Mannaro, M. Marchesi, and R. Tonelli, "Automatic generation of Ethereum-based smart contracts for Agri-food traceability system," *IEEE Access*, vol. 10, pp. 50363–50383, 2022.

[8] A. Irrera. (2017). *Northern Trust Uses Blockchain for Private Equity Record-Keeping*. Accessed: Jan. 8, 2022. [Online]. Available: https://www.reuters.com/article/nthern-trust-ibm-blockchain/northern-trust-uses-blockchain-for-private-equity-record-keeping-idUSL1N1G61TX

[9] M. del Castillo. (2022). *$1.5 Trillion Asset Manager Northern Trust Creates Digital Assets Group to Meet Growing Demand*. Accessed: Jun. 30, 2022. [Online]. Available: https://www.forbes.com/sites/michaeldelcastillo/2022/06/23/15-trillion-asset-manager-northern-trust-creates-digital-assets-group-to-meet-growing-demand/?sh=59949bca774c

[10] Federal Reserve Bank of Boston. (2019). *Beyond Theory: Getting Practical With Blockchain*. Accessed: Jul. 21, 2022. [Online]. Available: https://www.bostonfed.org/publications/fintech/beyond-theory-getting-practical-with-blockchain/building-an-ethereum-blockchain-proof-of-concept.aspx

[11] A. A. Mamun, A. Al Mamun, S. R. Hasan, S. R. Hasan, M. S. Bhuiyan, M. S. Bhuiyan, M. S. Kaiser, M. S. Kaiser, M. A. Yousuf, and M. A. Yousuf, "Secure and transparent KYC for banking system using IPFS and blockchain technology," in *Proc. IEEE Region 10 Symp. (TENSYMP)*, Jun. 2020, pp. 348–351.

[12] Y. Chen and C. Bellavitis, "Blockchain disruption and decentralized finance: The rise of decentralized business models," *J. Bus. Venturing Insights*, vol. 13, Jun. 2020, Art. no. e00151.

[13] P. Baker. (Jul. 29, 2020). *Soaring DeFi Usage Drives Ethereum Contract Calls to New Record*. Accessed: Jan. 7, 2022. [Online]. Available: https://www.coindesk.com/soaring-defi-usage-drives-ethereum-contract-calls-to-new-record

[14] F. Schär, "Decentralized finance: On blockchain- and smart contract-based financial markets," *Review*, vol. 103, no. 2, pp. 153–174, 2021, doi: 10.20955/r.103.153-74.

[15] Y. Liu, Q. Lu, G. Yu, H.-Y. Paik, and L. Zhu, "Defining blockchain governance principles: A comprehensive framework," *Inf. Syst.*, vol. 109, Nov. 2022, Art. no. 102090, doi: 10.1016/j.is.2022.102090.

[16] F. Lumineau, W. Wang, and O. Schilke, "Blockchain governance—A new way of organizing collaborations?" *Org. Sci.*, vol. 32, no. 2, pp. 500–521, Mar. 2021, doi: 10.1287/orsc.2020.1379.

[17] A. Norta, "Designing a smart-contract application layer for transacting decentralized autonomous organizations," in *Proc. Int. Conf. Adv. Comput. Data Sci.* Cham, Switzerland: Springer, 2016, pp. 595–604.
[18] Q. Wang, R. Li, Q. Wang, and S. Chen, "Non-fungible token (NFT): Overview, evaluation, opportunities and challenges," 2021, *arXiv:2105.07447*.
[19] B. White, A. Mahanti, and K. Passi, "Characterizing the OpenSea NFT marketplace," in *Proc. Companion Web Conf.*, Apr. 2022, pp. 488–496, doi: 10.1145/3487553.3524629.
[20] L. Ante. (Jun. 6, 2021). The Non-Fungible Token (NFT) Market and Its Relationship With Bitcoin and Ethereum. [Online]. Available: https://ssrn.com/abstract=3861106
[21] Y. Wang, Z. Su, N. Zhang, R. Xing, D. Liu, T. H. Luan, and X. Shen, "A survey on metaverse: Fundamentals, security, and privacy," *IEEE Commun. Surveys Tuts.*, vol. 25, no. 1, pp. 319–352, 1st Quart., 2023.
[22] T. R. Gadekallu, T. Huynh-The, W. Wang, G. Yenduri, P. Ranaweera, Q.-V. Pham, D. B. da Costa, and M. Liyanage, "Blockchain for the metaverse: A review," 2022, *arXiv:2203.09738*.
[23] S. Malwa. *DeFi Locked Value Falls to Yearly Low, $27b Lost Over the Weekend*. Accessed: May 10, 2022. [Online]. Available: https://www.coindesk.com/markets/2022/05/09/defi-locked-value-falls-to-yearly-low-27b-lost-over-the-weekend/
[24] G. Wood. (Oct. 24, 2022). *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. Berlin Version Beacfbd. Accessed: Oct. 24, 2022. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf
[25] G. A. Oliva, A. E. Hassan, and Z. M. Jiang, "An exploratory study of smart contracts in the Ethereum blockchain platform," *Empirical Softw. Eng.*, vol. 25, no. 3, pp. 1864–1904, May 2020, doi: 10.1007/s10664-019-09796-5.
[26] B. Marques. (Apr. 2022). *Ethereum Surpassed Visa in 2021 Concerning the Sums Traded*. Accessed: Nov. 25, 2022. [Online]. Available: https://www.cryptodefinance.com/ethereum-surpassed-visa-in-2021
[27] K. Dukovski. (2022). *If Solana Becomes the Visa of Crypto, Where Does Visa Fit in*. Accessed: Sep. 11, 2022. [Online]. Available: https://www.finder.com/if-solana-becomes-visa-of-crypto-what-about-visa#
[28] P. Zheng, Z. Zheng, X. Luo, X. Chen, and X. Liu, "A detailed and real-time performance monitoring framework for blockchain systems," in *Proc. 40th Int. Conf. Softw. Eng., Softw. Eng. Pract.*, May 2018, pp. 134–143. http://dl.acm.org/citation.cfm?doid=3183519.3183546
[29] 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 *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Jul. 2021, pp. 565–579. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/kim-yeonsoo
[30] P. Wackerow, J. Douglas, S. A. Green, M. Havel, and C. Smith. (2022). *Proof-of-Stake (POS)*. Accessed: Sep. 11, 2022. [Online]. Available: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/,ethereum.orgEthereumcommunity
[31] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. 4th Annu. IEEE Int. Workshop Workload Characterization. (WWC)*, Dec. 2001, pp. 3–14.
[32] J. Bucek, K.-D. Lange, and J. V. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Proc. Companion ACM/SPEC Int. Conf. Perform. Eng.*, Apr. 2018, pp. 41–42, doi: 10.1145/3185768.3185771.
[33] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.
[34] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proc. OOPSLA 21st Annu. ACM SIGPLAN Conf. Object-Oriented Programing, Syst., Lang., Appl.* New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.
[35] B. Varghese, N. Wang, D. Bermbach, C.-H. Hong, E. D. Lara, W. Shi, and C. Stewart, "A survey on edge performance benchmarking," *ACM Comput. Surv.*, vol. 54, no. 3, pp. 1–33, Apr. 2021, doi: 10.1145/3444692.
[36] A. Das, S. Patterson, and M. Wittie, "EdgeBench: Benchmarking edge computing platforms," in *Proc. IEEE/ACM Int. Conf. Utility Cloud Comput. Companion (UCC Companion)*, Dec. 2018, pp. 175–180.
[37] S. Baset, M. Silva, and N. Wakou, "SPEC Cloud IaaS 2016 benchmark," in *Proc. 8th ACM/SPEC Int. Conf. Perform. Eng.*, Apr. 2017, p. 423, doi: 10.1145/3030207.3053675.
[38] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson, "Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0," in *Proc. 1st Workshop Cloud Comput. Appl.*, vol. 8, 2008, p. 228.
[39] J. Li, "SPEChpc 2021 benchmark suites for modern HPC systems," in *Proc. Companion ACM/SPEC Int. Conf. Perform. Eng.*, Jul. 2022, pp. 15–16, doi: 10.1145/3491204.3527498.
[40] N. Ajienka, P. Vangorp, and A. Capiluppi, "An empirical analysis of source code metrics and smart contract resource consumption," *J. Softw., Evol. Process*, vol. 32, no. 10, Oct. 2020, Art. no. e2267, doi: 10.1002/smr.2267.
[41] T. Brandstatter, S. Schulte, J. Cito, and M. Borkowski, "Characterizing efficiency optimizations in solidity smart contracts," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, Nov. 2020, pp. 281–290.
[42] A. Pinna, S. Ibba, G. Baralla, R. Tonelli, and M. Marchesi, "A massive analysis of Ethereum smart contracts empirical study and code metrics," *IEEE Access*, vol. 7, pp. 78194–78213, 2019.
[43] P. Hegedűs, "Towards analyzing the complexity landscape of solidity based Ethereum smart contracts," in *Proc. 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, May 2018, pp. 35–39.
[44] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Feb. 2017, pp. 442–446.
[45] Etherscan. *Ethereum Daily Verified Contracts Chart*. Accessed: May 19, 2020. [Online]. Available: https://etherscan.io/chart/verified-contracts
[46] Solidity Development Community. *Solidity Online Documentation*. Accessed: Nov. 10, 2022. [Online]. Available: https://docs.soliditylang.org
[47] C. Dannen, *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*, 1st ed. New York, NY, USA: Apress, 2017.
[48] Vyper Development Community. *Vyper Online Documentation*. Accessed: Nov. 10, 2022. [Online]. Available: https://vyper.readthedocs.io/en/stable/
[49] K. Baird, S. Jeong, Y. Kim, B. Burgstaller, and B. Scholz, "The economics of smart contracts," 2019, *arXiv:1910.11143*.
[50] J. K. Hollingsworth, B. P. Miller, and J. Cargille, "Dynamic program instrumentation for scalable performance tools," in *Proc. IEEE Scalable High Perform. Comput. Conf.*, May 1994, pp. 841–850.
[51] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Cham, Switzerland: Springer, 2010.
[52] P. Cousot, "Abstract interpretation," *ACM Comput. Surv.*, vol. 28, no. 2, pp. 324–328, 1996.
[53] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *ACM SIGPLAN Notices*, vol. 25, no. 6, pp. 246–256, Jun. 1990.
[54] B. P. Miller and J.-D. Choi, "A mechanism for efficient debugging of parallel programs," *ACM SIGPLAN Notices*, vol. 23, no. 7, pp. 135–144, Jul. 1988.
[55] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.
[56] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 1998.
[57] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2006.
[58] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proc. 8th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 1981, pp. 207–218, doi: 10.1145/567532.567555.
[59] V. Sarkar, "Automatic partitioning of a program dependence graph into parallel tasks," *IBM J. Res. Develop.*, vol. 35, no. 5.6, pp. 779–804, Sep. 1991.
[60] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," in *Proc. 1st ACM SIGSOFT/SIGPLAN Softw. Eng. Symp. Practical Softw. Develop. Environments (SDE)*, 1984, pp. 177–184, doi: 10.1145/800020.808263.

[61] Etherscan. *Ethereum Full Node Sync (Archive) Chart*. Accessed: May 19, 2022. [Online]. Available: https://etherscan.io/chartsync/chainarchive

[62] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The Art Multiprocessor Programming*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2021.

[63] B. Schmidt, J. Gonzalez-Martinez, C. Hundt, and M. Schlarb, *Parallel Programming: Concepts and Practice*. San Mateo, CA, USA: Morgan Kaufmann, 2017.

[64] T. Rauber and G. Rünger, *Parallel Programming*. Cham, Switzerland: Springer, 2013.

[65] M. D. McCool, A. D. Robison, and J. Reinders, *Structured Parallel Programming: Patterns for Efficient Computation*. Waltham, MA, USA: Morgan Kaufmann Publishers, 2012.

[66] C. Lin and L. Snyder, *Principles of Parallel Programming*, 1st ed. Reading, MA, USA: Addison-Wesley, 2008.

[67] M. J. Quinn, "Parallel programming," *TMH CSE*, vol. 526, p. 105, Jan. 2003.

[68] V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[69] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan, "Optimization and approximation in deterministic sequencing and scheduling: A survey," *Ann. Math.*, vol. 5, pp. 287–326, Jan. 1979.

[70] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell Syst. Tech. J.*, vol. 45, no. 9, pp. 1563–1581, Nov. 1966.

[71] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. ACM*, vol. 17, no. 12, pp. 685–690, Dec. 1974, doi: 10.1145/361604.361619.

[72] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[73] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in *Proc. 6th Int. Conf. Princ. Secur. Trust*, vol. 10204. New York, NY, USA: Springer-Verlag, 2017, pp. 164–186, doi: 10.1007/978-3-662-54455-6_8.

[74] R. Jain, *The Art of Computer Systems Performance Analysis Techniques for Experimental Design, Measurement, Simulation, and Modeling* (Wiley Professional Computing). Hoboken, NJ, USA: Wiley, 1991.

[75] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach* (The Morgan Kaufmann Series in Computer Architecture and Design), 6th ed. Amsterdam, The Netherlands: Elsevier, 2017.

[76] K. Olukotun and L. Hammond, "The future of microprocessors: Chip multiprocessors' promise of huge performance gains is now a reality," *Queue*, vol. 3, no. 7, pp. 26–29, Sep. 2005, doi: 10.1145/1095408.1095418.

[77] D. W. Wall, "Limits of instruction-level parallelism," in *Proc. 4th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS-IV)*, 1991, pp. 176–188, doi: 10.1145/106972.106991.

[78] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream parallelism is greater than two," in *Proc. 18th Annu. Int. Symp. Comput. Archit. (ISCA)*, 1991, pp. 276–286, doi: 10.1145/115952.115980.

[79] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," in *Proc. 3rd Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS-III)*, 1989, pp. 290–302, doi: 10.1145/70082.68209.

[80] D. Sima, "The design space of register renaming techniques," *IEEE Micro*, vol. 20, no. 5, pp. 70–83, Sep. 2000.

[81] S. Debray, R. Muth, and M. Weippert, "Alias analysis of executable code," in *Proc. 25th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 1998, pp. 12–24, doi: 10.1145/268946.268948.

[82] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M.-W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 2, pp. 138–150, May 1995, doi: 10.1145/225830.225965.

[83] B. Mariano, Y. Chen, Y. Feng, S. K. Lahiri, and I. Dillig, "Demystifying loops in smart contracts," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Dec. 2020, pp. 262–274, doi: 10.1145/3324884.3416626.

[84] Etherscan. *Ethereum Average Gas Price Chart*. Accessed: Nov. 20, 2022. [Online]. Available: https://etherscan.io/chart/gasprice

[85] Etherscan. *Ether Daily Price (USD) Chart*. Accessed: Nov. 20, 2022. [Online]. Available: https://etherscan.io/chart/etherprice

[86] Ethereum.org Ethereum Community. *Solidity GitHub Repository Issue 12211*. Accessed: Nov. 16, 2022. [Online]. Available: https://github.com/ethereum/solidity/issues/12211

[87] Ethereum.org Ethereum Community. *Solidity GitHub Repository Issue 10755*. Accessed: Nov. 16, 2022. [Online]. Available: https://github.com/ethereum/solidity/issues/10755

[88] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*, 1st ed. Reading, MA, USA: Addison-Wesley, 2014.

[89] J. Richter, *CLR Via C#*, 4th ed. Unterschleissheim, Germany: Microsoft Press, 2012.

[90] J. Koshy and R. Pandey, "VMSTAR: Synthesizing scalable runtime environments for sensor networks," in *Proc. 3rd Int. Conf. Embedded Netw. Sensor Syst.*, Nov. 2005, pp. 243–254, doi: 10.1145/1098918.1098945.

[91] R. Müller, G. Alonso, and D. Kossmann, "A virtual machine for sensor networks," *ACM SIGOPS Operating Syst. Rev.*, vol. 41, no. 3, pp. 145–158, Mar. 2007, doi: 10.1145/1272998.1273013.

[92] K. Hong, J. Park, S. Kim, T. Kim, H. Kim, B. Burgstaller, and B. Scholz, "TinyVM: An energy-efficient execution infrastructure for sensor networks," *Softw., Pract. Exper.*, vol. 42, no. 10, pp. 1193–1209, Oct. 2012, doi: 10.1002/spe.1123.

[93] K. Hong, J. Park, T. Kim, S. Kim, H. Kim, Y. Ko, J. Park, B. Burgstaller, and B. Scholz, "TinyVM, an efficient virtual machine infrastructure for sensor networks," in *Proc. 7th ACM Conf. Embedded Netw. Sensor Syst.*, Nov. 2009, pp. 399–400, doi: 10.1145/1644038.1644121.

[94] N. Durov, "Telegram open network virtual machine," Open Netw., White Paper, Mar. 2020. [Online]. Available: https://ton-blockchain.github.io/docs/tvm.pdf

[95] WebAssembly Community Group. *WebAssembly (WA)*. Accessed: Nov. 11, 2022. [Online]. Available: https://webassembly.github.io/spec/core/intro/overview.html

[96] J. Park, J. Park, W. Song, S. Yoon, B. Burgstaller, and B. Scholz, "Treegraph-based instruction scheduling for stack-based virtual machines," *Electron. Notes Theor. Comput. Sci.*, vol. 279, no. 1, pp. 33–45, Dec. 2011, doi: 10.1016/j.entcs.2011.11.004.

[97] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java bytecode using the Soot framework: Is it feasible?" in *Proc. 9th Int. Conf. Compiler Construct.*, vol. 1781. Cham, Switzerland: Springer, 2000, pp. 18–34.

[98] T. Vandrunen, A. L. Hosking, and J. Palsberg. (2000). *Reducing Loads and Stores in Stack Architectures*. [Online]. Available: www.cs.ucla.edu/~palsberg/draft/vandrunen-hosking-palsberg00.pdf

[99] M. Maierhofer and M. A. Ertl, "Local stack allocation," in *Proc. 7th Int. Conf. Compiler Construct.*, vol. 1383, K. Koskimies, Ed. Cham, Switzerland: Springer, 1998, pp. 189–203, doi: 10.1007/BFb0026432.

[100] P. J. Koopman, "A preliminary exploration of optimized stack code generation," *J. Forth Appl. Res.*, vol. 6, pp. 1–6, Jan. 1994.

[101] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual machine showdown: Stack versus registers," *ACM Trans. Archit. Code Optim.*, vol. 4, no. 4, Jan. 2008, doi: 10.1145/1328195.1328197.

[102] 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 *Proc. 44th Int. Conf. Softw. Eng.*, May 2022, pp. 2315–2326, doi: 10.1145/3510003.3510086.

[103] W. Yu, K. Luo, Y. Ding, G. You, and K. Hu, "A parallel smart contract model," in *Proc. Int. Conf. Mach. Learn. Mach. Intell.*, Sep. 2018, pp. 72–77, doi: 10.1145/3278312.3278321.

[104] V. Saraph and M. Herlihy, "An empirical study of speculative concurrency in Ethereum smart contracts," in *Tokenomics*, vol. 71. Wadern, Germany: Schloss Dagstuhl Leibniz-Zentrum Für Informatik, 2019, pp. 4:1–4:15.

[105] P. S. Anjana, H. Attiya, S. Kumari, S. Peri, and A. Somani, "Efficient concurrent execution of smart contracts in blockchains using object-based transactional memory," in *Networked Systems*, C. Georgiou and R. Majumdar, Eds. Cham, Switzerland: Springer, 2021, pp. 77–93.

[106] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," *Distrib. Comput.*, vol. 33, nos. 3–4, pp. 209–225, Jun. 2020, doi: 10.1007/s00446-019-00357-z.

[107] M. J. Amiri, D. Agrawal, and A. El Abbadi, "ParBlockchain: Leveraging transaction parallelism in permissioned blockchain systems," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 1337–1347.

[108] H. Eedi and P. A. Khan, "Execution of smart contacts concurrently using fine-grain locking and asynchronous functions," in *Smart Computing Techniques and Applications*, S. C. Satapathy, V. Bhateja, M. N. Favorskaya, and T. Adilakshmi, Eds. Singapore: Springer, 2021, pp. 773–784.
[109] Y. Muchhala, H. Singhania, S. Sheth, and K. Devadkar, "Enabling MapReduce based parallel computation in smart contracts," in *Proc. 6th Int. Conf. Inventive Comput. Technol. (ICICT)*, Jan. 2021, pp. 537–543.
[110] R. Meier and T. R. Gross, "Reflections on the compatibility, performance, and scalability of parallel Python," in *Proc. 15th ACM SIGPLAN Int. Symp. Dyn. Lang.*, Oct. 2019, pp. 91–103, doi: 10.1145/3359619.3359747.
[111] R. Meier, A. Rigo, and T. R. Gross, "Virtual machine design for parallel dynamic programming languages," *Proc. ACM Program. Lang.*, vol. 2, pp. 1–25, Oct. 2018, doi: 10.1145/3276479.
[112] J. Fumero, A. Stratikopoulos, and C. Kotselidis, *Running Parallel Bytecode Interpreters on Heterogeneous Hardware*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 31–35, doi: 10.1145/3397537.3397563.
[113] B. Daloze, A. Tal, S. Marr, H. Mössenböck, and E. Petrank, "Parallelization of dynamic languages: Synchronizing built-in collections," *Proc. ACM Program. Lang.*, vol. 2, pp. 1–30, Oct. 2018, doi: 10.1145/3276478.
[114] M. Maas, K. Asanovic, and J. Kubiatowicz, "A hardware accelerator for tracing garbage collection," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 138–151.
[115] M. Maas, P. Reames, J. Morlan, K. Asanović, A. D. Joseph, and J. Kubiatowicz, "GPUs as an opportunity for offloading garbage collection," *ACM SIGPLAN Notices*, vol. 47, no. 11, pp. 25–36, Jun. 2012, doi: 10.1145/2426642.2259002.
[116] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2007.
[117] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. Cambridge, MA, USA: MIT Press, 1996.
[118] U. A. Acar, G. E. Blelloch, and R. Harper, "Selective memoization," *ACM SIGPLAN Notices*, vol. 38, no. 1, pp. 14–25, Jan. 2003, doi: 10.1145/640128.604133.
[119] D. Michie, "'Memo' functions and machine learning," *Nature*, vol. 218, pp. 19–22, Apr. 1968.
[120] W. Munsil and C.-J. Wang, "Reducing stack usage in Java bytecode execution," *ACM SIGARCH Comput. Archit. News*, vol. 26, no. 1, pp. 7–11, Mar. 1998.
[121] Ethereum.Org Ethereum Community. *Solidity GitHub Repository*. Accessed: Nov. 16, 2022. [Online]. Available: https://github.com/ethereum/solidity/
[122] S. Thummalapenta, J. D. Halleux, N. Tillmann, and S. Wadsworth, "DyGEN: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces," in *Proc. Int. Conf. Tests Proofs*. Cham, Switzerland: Springer, 2010, pp. 77–93.
[123] J. Caubet, J. Gimenez, J. Labarta, L. DeRose, and J. Vetter, "A dynamic tracing mechanism for performance analysis of OpenMP applications," in *Int. Workshop OpenMP Appl. Tools*. Cham, Switzerland: Springer, 2001, pp. 53–67.
[124] E. S. Andreasen, A. Møller, and B. B. Nielsen, "Systematic approaches for increasing soundness and precision of static analyzers," in *Proc. 6th ACM SIGPLAN Int. Workshop State Art Program Anal.*, Jun. 2017, pp. 31–36, doi: 10.1145/3088515.3088521.
[125] Y. Chen, Z. Guo, R. Li, S. Chen, L. Zhou, Y. Zhou, and X. Zhang, "Forerunner: Constraint-based speculative transaction execution for Ethereum," in *Proc. SOSP*, Oct. 2021, pp. 570–587. [Online]. Available: https://www.microsoft.com/en-us/research/publication/forerunner-constraint-based-speculative-transaction-execution-for-ethereum/
[126] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, "Tracing the meta-level: PyPy's tracing JIT compiler," in *Proc. 4th workshop Implement., Compilation, Optim. Object-Oriented Lang. Program. Syst.*, Jul. 2009, pp. 18–25.
[127] Y. Ding, C. Wang, Q. Zhong, H. Li, J. Tan, and J. Li, "Function-level dynamic monitoring and analysis system for smart contract," *IEEE Access*, vol. 8, pp. 229161–229172, 2020.

XIAOWEN HU received the B.S. degree in computer science from The University of Sydney, Australia, in 2020, where he is currently pursuing the Ph.D. degree in computer science. His research interests include the theory and implementation of programming languages and the optimization of smart contract virtual machines.

BERND BURGSTALLER received the Ph.D. degree from the Vienna University of Technology in 2005. He was a postdoctoral researcher at the University of Sydney until 2007. He is currently a professor in the Department of Computer Science at Yonsei University. His research interests include programming languages, parallel computing on multicore architectures, and embedded systems. Before pursuing an academic career, he spent three years as a software engineer and architect at Philips Consumer Electronics, Vienna.

BERNHARD SCHOLZ received the Ph.D. degree from the Vienna University of Technology, in 2001. He is currently a Full Professor with the School of Computer Science, The University of Sydney, Australia. He is on leave and setting up a research laboratory with Fantom's Blockchain Foundation. His research interests include programming languages and systems research.