Forerunner: Constraint-based Speculative Transaction Execution for Ethereum (Full Version)

Yang Chen
yachen@microsoft.com
Microsoft Research

Zhongxin Guo
Zhongxin.Guo@microsoft.com
Microsoft Research

Runhuai Li*
lirunhuai@zju.edu.cn
Zhejiang University
Microsoft Research

Shuo Chen
shuochen@microsoft.com
Microsoft Research

Lidong Zhou
lidongz@microsoft.com
Microsoft Research

Yajin Zhou
Yajin_zhou@zju.edu.cn
Zhejiang University

Xian Zhang
zhxian@microsoft.com
Microsoft Research

Abstract

Ethereum is an emerging distributed computing platform that supports a decentralized replicated virtual machine at a large scale. Transactions in Ethereum are specified in smart contracts, disseminated through broadcast, accepted into the chain of blocks, and then executed on each node. In this new Dissemination-Consensus-Execution (DiCE) paradigm, the time interval between when a transaction is known (during the dissemination phase) to when the transaction is executed (after the consensus phase) offers a window of opportunity to accelerate transaction processing through speculative execution. However, the traditional speculative execution, which hinges on the ability to predict the future accurately, is inadequate because of DiCE's *many-future* nature.

Forerunner proposes a novel constraint-based approach for speculative execution on Ethereum. In contrast to the traditional approach of predicting a single *future* and demanding it to be perfectly accurate, Forerunner speculates on multiple futures and can leverage speculative results based on imperfect predictions whenever certain constraints are satisfied. Under these constraints, a transaction execution is substantially accelerated through a novel multi-trace *program specialization* enhanced by a new form of *memoization*. The fully implemented Forerunner is evaluated as a node connected to the worldwide Ethereum network. When processing 13 million transactions live in real time, Forerunner achieves an effective average speedup of 8.39× on the transactions that it hears during the dissemination phase, which accounts for 95.71% of all the transactions. The end-to-end speedup over all the transactions is 6.06×. The code and data sets are publicly available.

*The work was done during an internship at Microsoft Research.

CCS Concepts: • **Software and its engineering → Compilers**; • **Computer systems organization → Dependable and fault-tolerant systems and networks**; • **General and reference → Performance**.

Keywords: speculative execution, blockchain, Ethereum, transaction throughput

ACM Reference Format:
Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. 2021. Forerunner: Constraint-based Speculative Transaction Execution for Ethereum (Full Version). In *ACM SIGOPS 28th Symposium on Operating Systems Principles(SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3477132.3483564

1 Introduction

By introducing the concept of blockchain-based smart contract [99], Ethereum [109] represents an emerging decentralized computing paradigm that implements a new form of a replicated state machine in an open network with no centralized trust. In this new model, participating *nodes* first broadcast their transaction requests to the network in the *dissemination* phase. The nodes then decide the next chained blocks of transactions in the *consensus* phase. Each node in the network executes the transactions following this same chain of blocks in the *execution* phase. Transaction execution is guaranteed to be deterministic in this model, which ensures that all nodes in the network remain consistent when executing the same sequence of transactions from the same initial state. There are inherent sequential dependencies in this Dissemination-Consensus-Execution (DiCE) model: a node can execute a block of transactions only after the completion of the consensus phase for this block; a node cannot start participating in a new consensus phase until it has completed executing the transactions in the current block. Transaction execution is on the critical path of creating the next block because it does the real work in the state machine to compute the resulting state of the current block, which is needed both for verifying the current block and for constructing the next block [111]. If this bottleneck can be alleviated, the system can be configured to process more transactions within the same time window for each execution phase (i.e., to pack more transactions into each block without increasing the block interval), resulting in throughput increase. Note that, this approach does not require any change to the consensus phase/algorithm, or require increasing the latency (i.e. block interval).

Also inherent in the DiCE model, transactions are known in the dissemination phase, but only executed after the consensus phase, when the context of its execution has been determined. This gap creates an opportunity for speculative pre-execution before the execution phase in order to accelerate the actual execution in the phase. Such speculative pre-execution also effectively introduces parallelism in the otherwise sequential execution of transactions because speculation for different transactions can run concurrently. The effectiveness of traditional speculative execution hinges on the ability to predict the *future* accurately. Accurate prediction is particularly challenging in the *many-future* environment of the DiCE model, where nodes across a wide-area network communicate asynchronously, leading to inconsistent observations of the system, and the result of the consensus phase is highly non-deterministic by its decentralized nature. For example, in Ethereum, some of the parallel futures can be directly observed as temporary forks on the blockchain[1].

Although speculative execution has been applied to many other transaction processing systems [3, 39, 40], the unique *many-future* challenge in DiCE systems calls for new speculative execution techniques. In this paper, we present *Forerunner*, which takes a radically different *constraint-based* approach to speculative transaction execution in the DiCE model. Rather than predicting a single *future*, Forerunner speculates on multiple futures. Rather than relying on the prediction to be perfectly accurate, Forerunner only requires that the reality satisfies the same set of constraints with a predicted future so that a *specialized fast-path program*, synthesized based on that future, can be used to produce the same execution results as the original transaction in the execution phase, but do so significantly faster. The key to constraint-based speculative execution in Forerunner is a novel trace-based *speculative program specialization* [30, 56, 76] technique, augmented with a form of *memoization* [29, 36, 62]. In particular, in the speculation phase, Forerunner executes a transaction on speculated *contexts* to obtain traces, simplifies the program significantly based on the execution path of each trace, gathers all the constraints that need to be satisfied for the transaction to follow exactly the same path, adds the shortcuts to skip memoized computation, and merges all the paths into a structure called *Accelerated Program*, or AP. In the execution phase, an AP first checks which of the constraint sets is satisfied, and then executes the corresponding fast-path program. Both the constraint checking and the fast-path execution can take the *shortcuts* to skip most of the computation. For the small chance of non-satisfaction, the original transaction execution is triggered.

We have fully implemented Forerunner for Ethereum and developed speculative program specialization on Ethereum VM (EVM) bytecode. We evaluated Forerunner as a node connected to the worldwide Ethereum network, processing real-time live traffic. This means not only that all the transactions and blocks were real and complete, but also the timings were real (i.e., all speculative pre-executions were done in real-time between the dissemination and the execution of the transactions). The evaluation results showed that Forerunner achieved an effective average speedup of 8.39× in the execution phase over the transactions it heard during the dissemination phase. These transactions accounted for 95.71% of the transactions actually packed into the official blocks. Taking unheard transactions into account, Forerunner's end-to-end speedup was 6.06×. This accelerated execution capability in the execution phase will give a realistic opportunity to increase Ethereum's throughput in its foundational layer.

2 Background and Motivation

Ethereum implements a replicated state machine in Ethereum Virtual Machine (EVM) [111] on a blockchain. Transactions on Ethereum can invoke code specified as a *smart contract*, which runs in EVM and makes changes to Ethereum's *world state* [111] (or *state* for brevity). Ethereum's smart contracts are quasi-Turing-complete [111] and can (recursively) send messages to other contracts to invoke their code during the execution of a single transaction. This enables user-defined stateful applications, which can interact with each other, to run on Ethereum. One transaction can therefore trigger arbitrarily complex code execution as long as the total amount of computation, measured in the unit of gas [111], does not exceed a specified limit[2] in the Ethereum protocol.

The DiCE distributed computing model. Ethereum exemplifies a new paradigm of distributed computing that we refer to as Dissemination-Consensus-Execution (DiCE). In a dissemination phase, any Ethereum *node* can propose transaction requests and broadcast the requests to the Ethereum's

[1]As of April 6 of 2021, 8.4% of all the successfully mined and propagated blocks are on those temporary forks [110].Note that many more potential forks (futures) did not have the chance to finish their mining process and cannot be directly observed.

[2]The gas limit of a block is collectively controlled by all the miners in a decentralized way. Each transaction also has a sender-specified transaction gas limit to guard against run-away code or unexpected long execution.

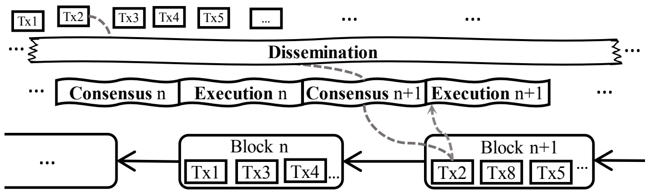

Figure 1. The DiCE (Dissemination-Consensus-Execution) distributed computing model.

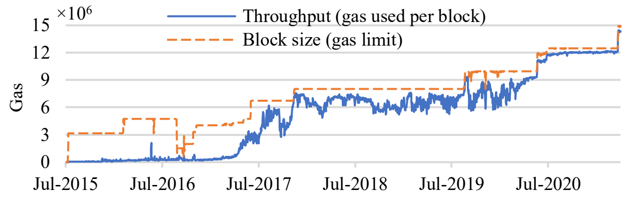

Figure 2. Ethereum's block size and throughput

P2P network. Each Ethereum *node* in the network propagates the requests and caches them in its local pending transaction pool. In a consensus phase, any node, acting as a *miner*, can validate and pack[3] an ordered list of pending transactions into a new block, and run the Proof-of-Work (PoW) [69, 111] consensus algorithm, competing to add the new block to the chain. Once a new block is created and broadcast to the network, all the other nodes validate the block and execute the transactions in the block, before they can start working on the next block.

Transaction execution on the critical path. Although dissemination, consensus, and execution of transactions happen in a decentralized and concurrent fashion, the critical path of the Ethereum system consists of repeated cycles of *consensus* and *execution*, where (i) execution of transactions in a block must follow the creation of this block in the consensus phase, (ii) a new consensus phase for the next block can start only after the execution of the current one completes, and (iii) the transactions in a block must also be executed in sequence to ensure the sequential semantics of the blockchain state changes (Figure 1). In other words, in the DiCE model, no matter what happens (e.g., PoW [111] or even after the transition to PoS [107]) in each consensus phase, the actual transaction processing happens only in the limited time window of each *execution* phase. Thus, the number of transactions that can be executed in each window determines the system throughput.

Accelerated transaction execution can enable the block size to be increased for higher throughput. Historically, as the performance of the commodity hardware (e.g., CPU and SSD) and the global Internet improve over time, Ethereum has gradually raised its block size (gas limit) to allow more transactions to be executed within each block-time interval. However, as Figure 2 shows, the increased block size (i.e., the dashed line) is saturated by the fast-growing throughput (i.e., the solid line), suggesting a strong demand for further throughput increase. Although higher throughput can be obtained by trading latency or security strength [106], changing the consensus assumption [4, 7] or the transaction processing model [31], transaction execution acceleration addresses the core performance problem orthogonal to all these aspects, and therefore a fundamental technology with full backward-compatibility. The throughput gains can be multiplied together if these approaches are combined properly.

The opportunity for speculation. On Ethereum, *dissemination* happens continuously and asynchronously alongside the *consensus* and *execution* cycles. Transactions are known to Ethereum nodes in the *dissemination* phase, but only executed after the *consensus* phase, when the context for their sequential execution is determined. On today's Ethereum, the time interval between hearing a transaction and executing it on a node typically ranges from several seconds to several minutes. It creates an opportunity for speculative pre-computation[4], off the critical path, before the execution phase in order to accelerate the actual computation[5], on the critical path, in the execution phase. Note that the length of a single consensus cycle is usually not a limiting factor for pre-computation because the dissemination-to-execution window for many transactions span multiple consensus-execution cycles[6] and speculation for independent transactions can run in parallel.

The curse of many-future. Any transaction on Ethereum executes in a *context*, which is determined only after the transaction is packed into a block. A traditional speculative execution technique, such as CPU's branch prediction, predicts one execution context, pre-computes the code against this context, and can accelerate the execution when the prediction is accurate.

The key challenge for speculative transaction execution on Ethereum is that, due to the inherently decentralized and non-deterministic nature of the DiCE model, a transaction usually has many possible future contexts. None of them has a dominant probability, thereby making the prediction accuracy impossible to be sufficiently high in reality. For example, in a decentralized system like Ethereum, by design, the consensus algorithm "selects" the miner for a block probabilistically, with no miner having a dominating probability of being chosen. And nodes (miners) receive transaction

[3]Technically speaking, packing a transaction into a block also requires executing it. However, in a Proof-of-Work (PoW) blockchain, except in extreme conditions, the packing efficiency is not the bottleneck of the system, which do not need to happen on the critical path of the miners. We therefore omit such details in the rest of the paper.

[4]Pre-computation does not compete for node resources with mining as the former runs on CPUs and the latter on GPUs or ASICs.

[5]This does not include operations, such as signature verification, which could be done in advance without speculation.

[6]Precomputation can thus not only overlap with consensus, but also with the execution of other transactions.

requests in different orders, make their own, non-uniform, decisions in packing transactions into blocks (e.g., in terms of which transactions to include and in which order), and do so with respect to their own local state (e.g., their local clocks for timestamps), all leading to possibly many different *futures* for each transaction.

3 Constraint-based Speculative Execution

The challenge of many-future in transaction execution on Ethereum, inherent in the DiCE model, calls for a new approach to speculative execution that should no longer depend on *perfect* predictions for acceleration. We therefore propose a new *constraint-based* approach to speculative execution, which proceeds in two phases. In the speculation phase, which is off the critical path, we execute the transaction in one or more future contexts to produce, conceptually, a set of *constraints* and a *specialized fast-path program*. In the transaction execution phase (on the critical path), when the actual context is determined, we check whether the constraints are satisfied in this context, execute the fast-path program when satisfied, and fall back to a full, original execution otherwise.

Leading to constraint-based speculative execution is the key observation that the execution of a transaction in a specific context can be (partially) harvested even when the actual context is different from the one speculated. Constraint-based speculative execution must demonstrate its value by accelerating the actual transaction execution, despite the overhead of checking the constraints, the cost of fast-path execution, and the penalty when constraints are not satisfied. Moreover, generating the constraints and fast paths should be fast enough so that they are available before transactions are executed for real inside the execution windows.

For correctness, constraint-based speculative execution ensures that, when the constraints are satisfied in the actual context, the execution of the specialized fast-path program in the actual context is guaranteed to produce the same result as the original transaction execution in the context (and hopefully do so significantly more efficiently). The effectiveness of constraint-based speculative execution hinges on (i) the probability of the constraints being satisfied and (ii) the efficiency gain of the specialized fast-path program.

Constraint-based speculative execution can be viewed as a generalization of the traditional speculative execution, where the constraints are simply reduced to a perfect match of the context and the fast-path program is simply reduced to committing the pre-computed results. In this generalized form, we allow a spectrum of choices to find an optimal point that balances the probability of a matching and the efficiency of the fast path, opening the door to a highly effective speculative execution solution for Ethereum.

Forerunner's design highlights. Forerunner provides an existence proof that such a generalized form of speculative execution can be profitable in a real system such as Ethereum.

The key design choice is to identify constraints that are highly likely to be satisfied by the actual contexts (i.e., coverage), while offering opportunities for specialization and acceleration (i.e., speedup). Forerunner chooses constraints that assert equivalence about the control flow and the data dependencies in transaction execution, which we name *CD-Equiv*. More precisely, under the CD-Equiv constraints, a transaction executes the same sequence of instructions. And for the instructions that access data in variable locations or of variable sizes, their cross-instruction data dependencies are also the same.

In this case, Forerunner can run a significantly more efficient specialized fast-path program, compared to the original program, because the CD-Equiv constraints of executing a fixed path with known data dependencies make effective code *specialization* and optimizations possible. It also enables the analyses and transformations to be light-weight enough to finish within the time budget. By relaxing the criteria from a perfect match on the context to that of ensuring CD-Equiv, Forerunner increases the probability of leveraging speculation.

Forerunner further incorporates a series of optimizations to improve efficiency. First, Forerunner adds shortcuts to both the logic of checking the constraints and the specialized fast-path program, using a form of *memoization*, to help improve its efficiency in certain cases. This ensures that some segments of the computation will be skipped appropriately if the values that decide the execution of those segments are the same as seen in the speculation, even in a different context. In the best case where the speculation predicts the whole context perfectly, the constraint-checking and fast-path execution with shortcuts can be almost as efficient as in the traditional speculative execution, with minimal overhead.

Second, Forerunner develops an efficient data structure to make it easy to merge multiple instances of constraint-checking logic into one. Those multiple instances are created from speculations on different speculated contexts, leading to different fast-path executions. This allows Forerunner to create multiple sets of constraints by speculating on multiple future contexts, thereby increasing the probability of one set of constraints being satisfied and some fast-paths being taken, without introducing noticeable additional overhead in constraint checking.

Finally, Forerunner leverages the effect of caching and prefetching from pre-computation. This saves the execution-phase I/O time considerably, because the states accessed and internal data structure constructed during speculation are likely to be the same for the same transaction.

4 Forerunner Architecture and Design

In this section, we describe the architecture of Forerunner and how it is designed to enable constraint-based speculative

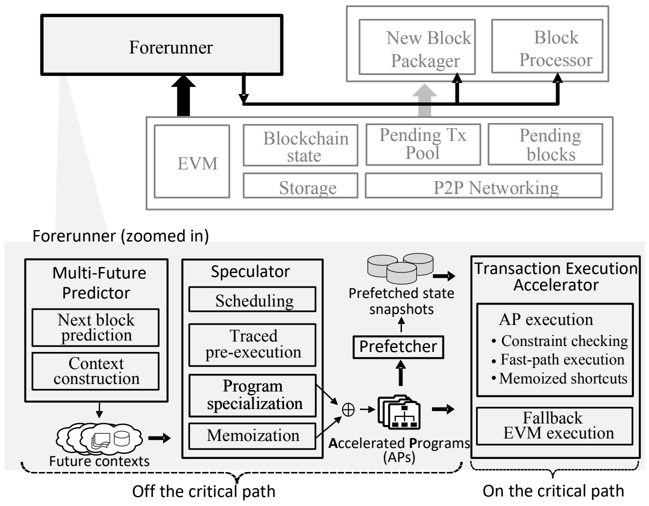

Figure 3. The architecture of Forerunner

execution, with the help of a concrete running example to illustrate the key concepts and mechanisms.

4.1 The Architecture of Forerunner

Forerunner is built as a new component for Ethereum, as shown in Figure 3. It taps into the existing components (shown as light grey boxes) of an Ethereum node to access the blockchain, its state, the pending transactions, and so on. Forerunner takes over the transaction execution responsibility from the block-processing component to implement speculative execution for acceleration.

The lower portion of Figure 3 zooms into Forerunner's internals. It consists of three major components: a *multi-future predictor*, a *speculator*, and a *transaction execution accelerator*. The first two, together with a *state prefetcher*, operate off the critical path. The *multi-future predictor* drives the pipeline: it monitors the blockchain and the pending transaction pool to identify transactions that are likely to be packed into a block in the near future, predicts and constructs possible future contexts for these transactions, and passes the transactions with their contexts to the speculator. The *speculator* pre-executes the transactions in the predicted contexts with runtime tracing to record the context variables read (which form a read set) and written (which form a write set), the dependencies, and the execution traces. Based on each trace, the *speculator* uses program specialization and memoization to synthesize highly optimized fast-path code guarded by constraint checking code to ensure CD-Equiv. We call the result an *accelerated program* (AP).

The read sets are passed into a *prefetcher*, which preloads these variables so that the expensive disk I/O, decoding, and key-value lookup operations can be carried out in advance, off the critical path.

When the execution phase starts, every transaction executes in its actual, fully determined context by the transaction execution accelerator. The accelerator runs the AP corresponding to the transaction. The AP checks whether any set of the guarding constraints is satisfied. If so, the fast-path execution is performed for the corresponding trace. Otherwise, the accelerator falls back to the execution of the general program.

4.2 Example: Price Oracle on Ethereum

For illustration purposes, we construct an example based on *price oracle* [103]. Price oracles play an infrastructural role in the decentralized finance (DeFi) [14] ecosystem, which is one of Ethereum's most popular application domains.

A price oracle is a feed of real-world trading prices of an asset (e.g., the price of ETH, the native cryptocurrency of Ethereum), which allows the information (from the outside world) to be consumed by the decentralized applications on the blockchain in a reliable way. For example, smart contracts, such as those for decentralized exchanges, can query off-chain prices of assets and facilitate a swap of two digital assets at their fair exchange rate on-chain.

```
 s1 contract PriceFeed {
 s2   // persistent state variables of the contract
 s3   uint256 public activeRoundID;
 s4   mapping(uint256 => uint256) public prices;
 s5   mapping(uint256 => uint256) public submissionCounts;
 s6   // method to submit a price for each 5-minute round
 s7   function submit(uint256 roundID, uint256 price) public {
 s8       uint256 curTime = block.timestamp;
 s9       uint256 curRoundID = curTime - curTime % (5 * 60);
s10       if (roundID != curRoundID) {revert();}
s11
s12       if (activeRoundID < roundID) {
s13          activeRoundID = roundID;
s14          prices[roundID] = price;
s15          submissionCounts[roundID] = 1;
s16       } else {
s17          uint256 curPrice = prices[roundID];
s18          uint256 curCount = submissionCounts[roundID];
s19          uint256 newSum = curPrice * curCount + price;
s20          uint256 newCount = curCount + 1;
s21          submissionCounts[roundID] = newCount;
s22          prices[roundID] = newSum / newCount;
s23       }
s24   }
s25   // other methods and state variables...
s26 }
```

Figure 4. The source code of PriceFeed.

PriceFeed: An example smart contract. Figure 4 shows our example smart contract, `PriceFeed`, written in Solidity [104]. It is derived and simplified from a real-world price oracle smart contract [98] deployed on Ethereum. `PriceFeed` models a common type of price oracles that brings the off-chain information about the latest prices of an asset on-chain by aggregating price submissions from independent price observers. In particular, for each round of 300 seconds, `PriceFeed` aggregates all the price submissions to calculate an average observed price. The state variable `prices`, declared in line s4 of Figure 4, stores the mapping from round ID to computed average prices. All state variables are encoded [111] and stored in the persistent storage associated with a deployed instance of the smart contract.

Line s7 declares `submit(...)`, the method of our main interest, with two arguments: `roundID` and `price`. A transaction is created when a price observer calls the method to submit a price (`price`) for a specific round (`roundID`). In addition to the state variables and the arguments, the method can also access the metadata stored in the *header* of the block that carries the transaction calling this method; an example of such metadata is the *timestamp* of the block (`block.timestamp` in line s8), which records the miner's local clock time when packing this particular block.

We include in `PriceFeed` only the representative, core logic for illustration purposes and omit other details, such as rewarding the price submitters to incentivize and compensate for their participation.

Tx_e: An example transaction for price submission. At the top of Figure 5, we show an example transaction, $\mathtt{Tx}_e$, sent by an observer to submit a price. We highlight the relevant properties of $\mathtt{Tx}_e$:

- *sender*: `UserA_Addr`, the observer's Ethereum account. The transaction fee is paid from the account's balance.
- *receiver*: `PriceFeed_Addr`, the Ethereum account that hosts an instance of the smart contract. It serves as a container for the code of the smart contract and the private *storage* to store the persistent state of the contract.
- *data*: a byte array that encodes [111] the ID of the method (`submit`) to be called and the arguments (`roundID`: 3970300 and `price`: 1980) passed into it.
- *gas price*: the amount of transaction fee the sender is willing to pay per unit of *gas*, which measures the amount of computation for the execution of the transaction. The transaction fees are paid to the miners, who tend to prioritize pending transactions with higher prices when packing transactions in order.

Transaction $\mathtt{Tx}_e$ executes in a *context* that consists of a block's header fields and the current Ethereum state, which is the result of executing all the previous transactions in the blockchain in sequence. The block header fields accessible through the *context* include metadata of blocks in the blockchain; e.g., the *timestamp* and *block number* of the block into which $\mathtt{Tx}_e$ is packed. The execution of $\mathtt{Tx}_e$ needs to read the value of *timestamp* to determine whether the submission time is within the current round.

Multiple futures of a transaction. Figure 5 shows four of many possible future contexts (`FC1` to `FC4`) for $\mathtt{Tx}_e$. For each future context, the figure shows the corresponding block into which $\mathtt{Tx}_e$ is packed and ordered, as well as the *read set* (values read) and the *write set* (values written) of $\mathtt{Tx}_e$[7].

The four future contexts are constructed to represent the common cases of the many-future reality observed on

[7]For brevity, we omit reads and writes of the balances of the sender and the block miner, used for transferring transaction fees.

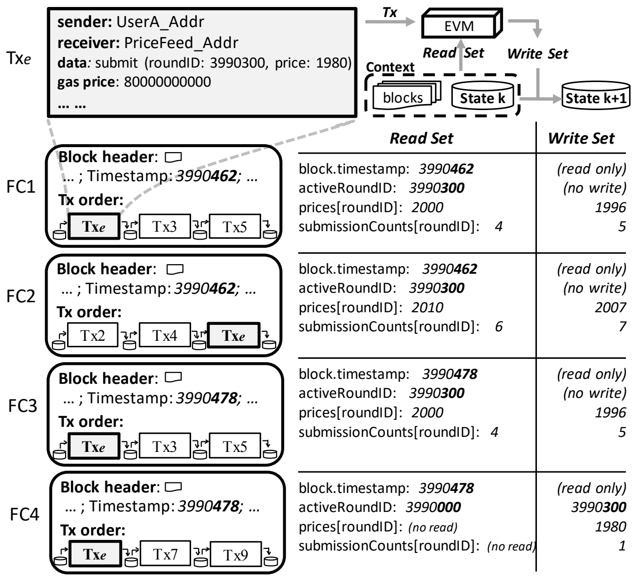

Figure 5. Four possible future contexts for transaction $\mathtt{Tx}_e$

Ethereum. There are two main causes of variations in the context that a transaction ends up executing in: different ordering of inter-dependent transactions and different values in the metadata of the block for the transaction. `FC2` deviates from `FC1` because $\mathtt{Tx}_e$ is ordered differently with respect to other inter-dependent transactions (e.g., other price submission transactions that affect the state variables of `prices` and `submissionCounts` on the same `roundID`). `FC3` differs from `FC1` in the timestamp of the block in which $\mathtt{Tx}_e$ is packed into, whereas `FC4` is different both in the order of inter-dependent transactions and in the block timestamp.

All those variations may happen because (i) the arrivals of inter-dependent transactions are unpredictable; different nodes may observe different sets of those transactions due to the asynchronous nature of a gossip-like protocol [22] in a large, decentralized system; (ii) each node may differ in which transactions to pack and how to order transactions when packing a block. Even if transactions are ordered based on gas prices, ties might be broken differently as their views on the world may differ[8] or they may assign different timestamps based on their local clocks; (iii) The PoW consensus algorithm "selects" the miner for a block probabilistically based on their relative mining power. The security of Ethereum hinges on the fact that no single miner can dictate the choice with a high enough probability.

4.3 Synthesis of Accelerated Programs

The workflow of synthesizing an accelerated program is shown in Figure 6. Given a predicted future context, Forerunner traces the transaction execution with an instrumented

[8]In fact, it is common to have transactions with the same gas price because senders tend to take pricing advice from the same helper tools. And as of this writing, the packing algorithm in the official implementation of the Ethereum nodes order same-price transactions randomly.

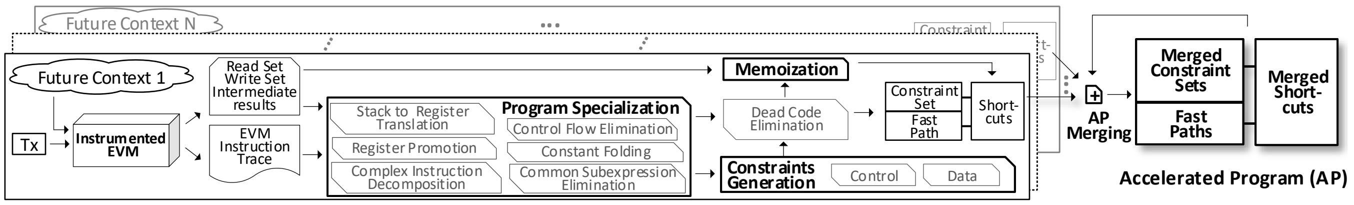

Figure 6. Workflow for synthesizing AP (Accelerated Program) via program specialization and memoization.

b1	PUSH 4	s7	b31	POP	s16	b61	SSTORE	s21
b2	CALLDATALOAD	s7	b32	PUSH 0	s17	b62	SWAP2	s19
b3	PUSH 24	s7	b33	DUP6	s17	b63	SWAP3	s19
b4	CALLDATALOAD	s7	b34	DUP2	s17	b64	POP	s19
b5	TIMESTAMP	s8	b35	MSTORE	s17	b65	SWAP1	s19
b6	PUSH 300	s9	b36	PUSH 1	s17	b66	DUP2	s19
b7	DUP2	s9	b37	PUSH 20	s17	b67	DUP4	s19
b8	MOD	s9	b38	DUP2	s17	b68	MUL	s19
b9	DUP2	s9	b39	DUP2	s17	b69	DUP7	s19
b10	SUB	s9	b40	MSTORE	s17	b70	ADD	s19
b11	PUSH 0	s9	b41	PUSH 40	s17	b71	SWAP1	s19
b12	DUP1	s9	b42	DUP1	s17	b72	DUP1	s22
b13	DUP1	s9	b43	DUP5	s17	b73	DUP3	s22
b14	DUP1	s9	b44	SHA256	s17	b74	PUSH 0	s22
b15	DUP8	s10	b45	SLOAD	s17	b75	DUP11	s22
b16	DUP6	s10	b46	PUSH 2	s18	b76	DUP2	s22
b17	EQ	s10	b47	SWAP1	s18	b77	MSTORE	s22
b18	PUSH [tag] 20	s10	b48	SWAP3	s18	b78	PUSH 1	s22
b19	JUMPI	s10	b49	MSTORE	s18	b79	PUSH 20	s22
b20	JUMPDEST	s12	b50	SWAP1	s18	b80	MSTORE	s22
b21	DUP8	s12	b51	SWAP3	s18	b81	PUSH 40	s22
b22	PUSH 0	s12	b52	SHA256	s18	b82	SWAP1	s22
b23	SLOAD	s12	b53	DUP1	s18	b83	SHA256	s22
b24	LT	s12	b54	SLOAD	s18	b84	SWAP2	s22
b25	ISZERO	s12	b55	SWAP2	s20	b85	SWAP1	s22
b26	PUSH [tag] 21	s12	b56	DUP3	s20	b86	DIV	s22
b27	JUMPI	s12	b57	ADD	s20	b87	SWAP1	s22
b28	JUMPDEST	s16	b58	SWAP1	s21	b88	SSTORE	s22
b29	POP	s16	b59	DUP2	s21	...		
b30	POP	s16	b60	SWAP1	s21			

Figure 7. The EVM instruction trace of $\texttt{Tx}_e$ in FC1 (or FC2 or F3). The corresponding source code line number is annotated on the right (e.g., s7).

EVM, generates the constraints that ensure CD-Equiv to this execution, and applies program specialization and memoization to create fast paths with shortcuts. The same process can be applied to multiple future contexts to improve coverage, with the resulting constraint sets and fast paths with shortcuts merged to form the accelerated program.

CD-Equiv is a good choice for coverage because an Ethereum transaction often traverses the same code path in different contexts, and the number of different paths taken is usually small. For example, $\texttt{Tx}_e$ follows the same path in contexts FC1, FC2, and FC3 (Figure 5). In fact, an execution in any context that satisfies the following two constraints[9]: (i) its `block.timestamp` is within the range from 3990300 to 3990599; (ii) its `activeRoundID` equals `roundID` (3990300) follows the same path, because the two IF-conditions in lines s10 and s12 (Figure 4) both evaluate to false.

CD-Equiv also offers significant opportunities for specialization and acceleration. With CD-Equiv, the execution of a transaction can be reduced to the execution of a sequence of unrolled and inlined instructions with fully-determined data dependencies, making it easy to apply optimizations such as constant folding [67], common sub-expression elimination [16], and register promotion [77]. Moreover, these steps are combined into a one-pass transformation, running fast enough on the fly for the speculative execution scenario.

Program specialization with S-EVM. As a preparation step, when running the transaction on an instrumented EVM in a predicted future context, Forerunner records the sequence of all the executed instructions (i.e., an EVM instruction trace), the intermediate results (i.e., the inputs/outputs of the instructions), and the read/write sets of the execution. Figure 7 is an example EVM instruction trace from pre-executing $\texttt{Tx}_e$ in FC1 (Figure 5), which consists of 88 EVM instructions.

Forerunner then performs *program specialization* on the EVM trace to produce the code for AP as shown in Figure 8. It first converts the stack-based [81] EVM instructions into our register-based intermediate representation called S-EVM. At its core, S-EVM is a highly simplified registered-based version of EVM. It needs to support only a subset of the EVM instructions and a subset of their execution semantics necessary for an accelerated program (AP). S-EVM supports, in a register-adapted form, all the instructions in the categories of arithmetic, comparison, bitwise logic, SHA3, environmental information, block information, storage, logging and system, as classified by the specification of EVM [111]. Each S-EVM instruction fulfills *only one* of the three functionalities: read, write, or compute. Because S-EVM is a register-based VM [81], the EVM instructions dedicated to stack and memory manipulation are no longer needed in S-EVM. All the supported EVM instructions are adapted to read from and write to a global array of registers in S-EVM, instead of accessing stack and (heap) memory in EVM. For example, the ADD instruction in EVM adds the top two values on the stack and replaces them with the result (via push and pop), whereas its counterpart in S-EVM, $v_k = \texttt{ADD}(v_i, v_j)$,[10] adds two values in the i^{th} and j^{th} registers, and puts the result into the k^{th} register. The resulting simplified and explicit execution and data dependencies among the instructions are particularly friendly for the analysis, optimization, transformation (e.g., for memoization), and execution of accelerated programs.

[9]Note that these two constraints are all control constraints as no extra data flow constraints are needed in this simple example.

[10]We do not give new names to these S-EVM instructions as they are otherwise the same as their EVM counterparts.

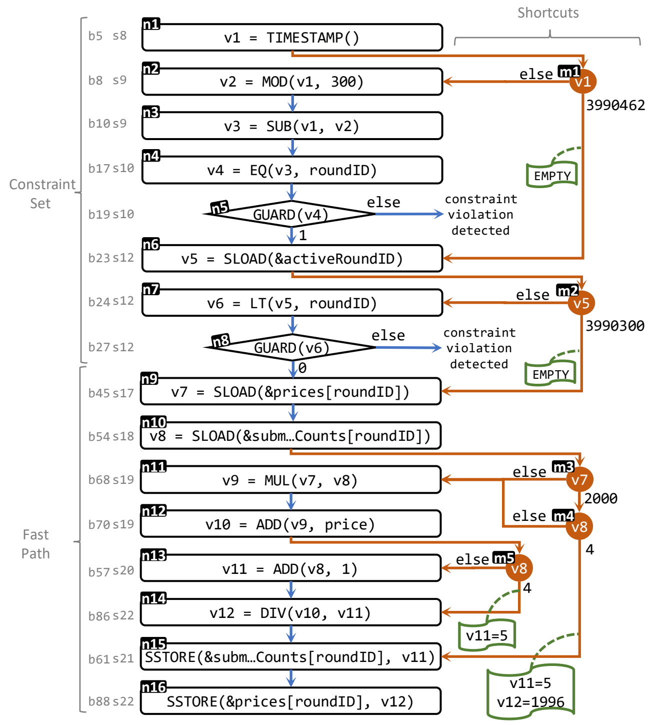
Shortcuts
Constraint Set
Fast Path
b5 s8 n1 v1 = TIMESTAMP()
b8 s9 n2 v2 = MOD(v1, 300)
b10 s9 n3 v3 = SUB(v1, v2)
b17 s10 n4 v4 = EQ(v3, roundID)
b19 s10 n5 GUARD(v4)
b23 s12 n6 v5 = SLOAD(&activeRoundID)
b24 s12 n7 v6 = LT(v5, roundID)
b27 s12 n8 GUARD(v6)
b45 s17 n9 v7 = SLOAD(&prices[roundID])
b54 s18 n10 v8 = SLOAD(&subm…Counts[roundID])
b68 s19 n11 v9 = MUL(v7, v8)
b70 s19 n12 v10 = ADD(v9, price)
b57 s20 n13 v11 = ADD(v8, 1)
b86 s22 n14 v12 = DIV(v10, v11)
b61 s21 n15 SSTORE(&subm…Counts[roundID], v11)
b88 s22 n16 SSTORE(&prices[roundID], v12)
else
constraint violation detected
1
0
else m1 v1
3990462
EMPTY
else m2 v5
3990300
EMPTY
else m3 v7
2000
else m4 v8
4
else m5 v8
4
v11=5
v11=5
v12=1996

Figure 8. Accelerated program of $\mathtt{Tx}_e$ synthesized in `FC1`. All the variables whose names do not start with v, such as `roundID`, `&activeRoundID`, are constants. The corresponding line numbers in EVM trace and the source code are annotated in the left two columns.

The conversion to S-EVM is done in four steps: (i) *Complex instruction decomposition*, which translates the complex EVM instructions with multiple functionalities into multiple S-EVM instructions. For example, the EVM instruction `SHA256` (see `b44`), which reads its input from memory and hashes it, is decomposed into two instructions: a memory read instruction and a register-based hashing instruction. (ii) *Stack to register translation* [21], which analyzes the implicit data dependencies among EVM instructions caused by sharing stack and memory, and translates them into a register-based S-EVM instruction sequence in the single-static-assignment (SSA) form [19], while keeping the data dependencies equivalent. (iii) *Register promotion*, which not only eliminates all memory accesses by allocating registers to store the memory values, but also eliminates redundant context accesses by keeping only the first read from and the last write to each variable in the context. The memory read instruction generated at the first step in the previous `SHA256` example will be eliminated in this step. (iv) *Control flow elimination* removes all the EVM control-flow instructions as accelerated programs are constrained to follow a pre-determined execution path.

After a trace is converted into a sequence of S-EVM instructions, Forerunner can apply further optimizations, where *constant folding* (recursively) removes all the instructions that produce constant results and *common sub-expression elimination* removes instructions that do duplicated computations. Both are trivial to do because the control flow and data dependencies among the instructions are fully determined.

Constraint generation. Accelerated programs must be guarded with constraints for CD-Equiv. The control constraints are assertions to ensure that the execution indeed follows a specific path at each control-flow change point. For example, a control constraint, which asserts that a particular conditional jump is not taken, checks whether the jump condition is false. Similarly, a constraint, which asserts that an unconditional jump with a variable target jumps to a particular position, checks whether the computed jump target equals a particular value. As the code to compute the control-flow-impacting jump conditions, jump targets, and other values (e.g., the target address of an indirect call) is already in the unfolded instruction sequence, constraint checking can be implemented by inserting special *guard* [33, 79] instructions.[11] The guard instructions compare these computed values to specific constants (e.g., true/false or target addresses) and, if not equal, abort the execution to indicate that a constraint is not satisfied. The guard instructions are inserted right after each of the values are computed to detect constraint violation as early as possible. For example, `n5` in Figure 8 is the guard instruction for the first control constraint of our running example. It is inserted right after the instructions corresponding to the IF-condition in line `s10`, which checks whether the value of the register `v4` is 1. If so, the constraint is satisfied, and the next constraint checking instruction (`n6`) can be executed (the downward arrow from `n5` labelled with value 1). Otherwise, a constraint violation is detected, which stops the execution from checking the rest of the constraints (see the rightward `else` arrow from `n5`). In some sense, the guard instructions reintroduce a restricted form of control flow back into the AP.

The data constraints are for instructions that access data in variable locations or of variable sizes. For example, the EVM instructions `MLOAD` and `MSTORE` read and write data at a particular offset of a non-persistent memory [111]. The offsets of such instructions might not be constants, leading to different data dependencies in different contexts. A data constraint asserting the (non-)existence of data dependency between two such instructions makes the dependencies fixed to enable optimizations (e.g., register promotion and common sub-expression elimination). It is implemented by inserting

[11] For transaction fee charging, Ethereum measures the cost of each executed instruction in the unit of `gas` [111]. If the gas consumption of a smart contract invocation exceeds a user-defined budget, the invocation will be terminated with its effects reverted and the control flow transferred back to its caller [111]. Thus, we also insert instructions and guards to implement control constraints that make sure all gas-induced control flow changes are identical across contexts.

instructions to compare the computed offsets (or data sizes) and guard the comparison outcome. Because only simple value comparison operations are introduced for checking both the control and the data constraints, constraint checking in general incurs low overhead.

Dead code elimination. After the guards are inserted, all the instructions whose results do not affect the guards or the fast-path execution results (e.g., instructions that compute values only for a branch not taken by this control-flow path) can be eliminated.

Rollback-free execution. All the instructions that do not affect any of the guards are moved after the last guard into the fast-paths. This is to avoid wasting the execution in the cases where the constraint sets are not satisfied. In particular, register promotion enables all the state writes to be moved into the fast-paths, which makes AP execution *rollback free*; i.e., when a guard is not satisfied, there is nothing to revert before the original transaction can execute from scratch, which saves time on the critical path.

Memoization. We employ a simple variation of memoization [29, 36, 62] to create a shortcut of an AP instruction segment if the read-set variables (registers) of the inputs to the segment match the values seen during the pre-execution. The right part of Figure 8 shows an example of the added shortcuts. The inputs to a code segment are all its referenced registers that are assigned before it. For example, the inputs to the segment from n9 to n14 are stored in registers v7 and v8. They happen to be read-set registers themselves, which store the current price and the submission count read from the context (n9 and n10). Two shortcut nodes, m3 and m4, are added to check whether v7 takes value 2000 and v8 takes value 4, both remembered from the pre-execution in FC1. If so, the whole segment can be skipped (the downward transition arrow from m4 to n15), with the remembered outputs of segment committed to the registers. Otherwise, the segment needs to be executed (the leftward else transition arrows).

The outputs of a segment are all the assigned registers that are referenced by the subsequent instructions. The outputs of the example segment are stored in v11 and v12. Thus, the remembered outputs of the segment, i.e., v11=5 and v12=1996, are attached to the transition arrow (see the curved box). Note that the memoized segments may overlap with each other. For example, there is shortcut node (m5) for the n13-to-n13 segment, which is a subset of the code segment n11-to-n14. It is added because, even if the larger segment cannot be skipped, it is still possible that the smaller can, as it depends on fewer read-set registers.[12] In the best case, if an actual context perfectly matches FC1, all the AP instructions will be skipped except for those reading the read-set variables (n1, n6, n9, n10) and writing the write-set variables (n15, n16).

[12]A simple heuristic is used to identify eligible segments to memoize and limit the total number of memoized segments. More refined memoization heuristics can be interesting future work.

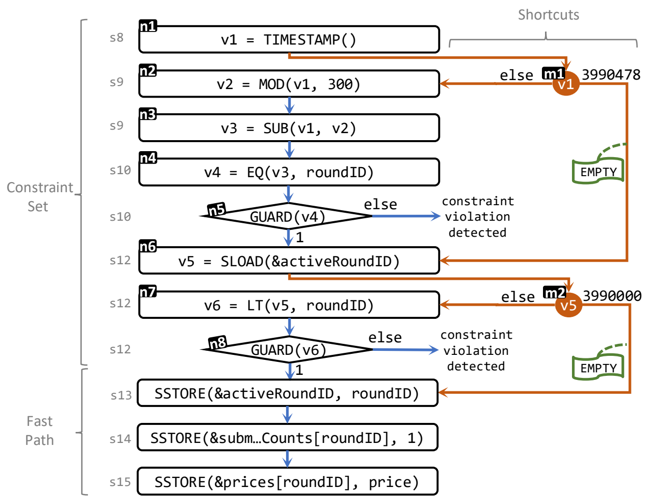
Shortcuts
Constraint Set
Fast Path
s8 n1 v1 = TIMESTAMP()
s9 n2 v2 = MOD(v1, 300)
s9 n3 v3 = SUB(v1, v2)
s10 n4 v4 = EQ(v3, roundID)
s10 n5 GUARD(v4) else 1
constraint violation detected
s12 n6 v5 = SLOAD(&activeRoundID)
s12 n7 v6 = LT(v5, roundID)
s12 n8 GUARD(v6) else 1
constraint violation detected
s13 SSTORE(&activeRoundID, roundID)
s14 SSTORE(&subm…Counts[roundID], 1)
s15 SSTORE(&prices[roundID], price)
else m1 v1 3990478
else m2 v5 3990000
EMPTY
EMPTY

Figure 9. Accelerated program of $\mathtt{Tx}_e$ synthesized in FC4.

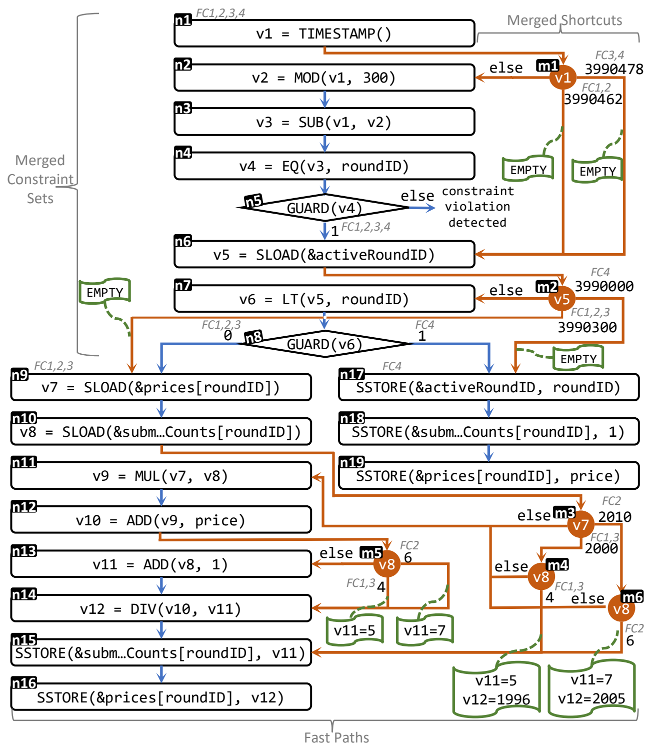
Merged Shortcuts
Merged Constraint Sets
Fast Paths
n1 FC1,2,3,4 v1 = TIMESTAMP()
n2 v2 = MOD(v1, 300)
n3 v3 = SUB(v1, v2)
n4 v4 = EQ(v3, roundID)
n5 GUARD(v4) else 1 FC1,2,3,4
constraint violation detected
n6 v5 = SLOAD(&activeRoundID)
n7 v6 = LT(v5, roundID)
n8 GUARD(v6) FC1,2,3 0 FC4 1
else m1 v1 FC3,4 3990478 FC1,2 3990462
else m2 v5 FC4 3990000 FC1,2,3 3990300
EMPTY
n9 FC1,2,3 v7 = SLOAD(&prices[roundID])
n10 v8 = SLOAD(&subm…Counts[roundID])
n11 v9 = MUL(v7, v8)
n12 v10 = ADD(v9, price)
n13 v11 = ADD(v8, 1)
n14 v12 = DIV(v10, v11)
n15 SSTORE(&subm…Counts[roundID], v11)
n16 SSTORE(&prices[roundID], v12)
n17 FC4 SSTORE(&activeRoundID, roundID)
n18 SSTORE(&subm…Counts[roundID], 1)
n19 SSTORE(&prices[roundID], price)
else m3 v7 FC2 2010 FC1,3 2000
else m4 v8 FC1,3 4 else m6 v8 FC2 6
else m5 v8 FC2 6 FC1,3 4
v11=5
v11=7
v11=5 v12=1996
v11=7 v12=2005

Figure 10. Accelerated Program of $\mathtt{Tx}_e$ merged from the synthesized programs in FC1, FC2, FC3, and FC4, with the source APs of the merged instructions and shortcuts annotated.

AP merging. The APs synthesized from different pre-executions of the same transaction are composable. When we further pre-execute $\mathtt{Tx}_e$ in FC2, FC3, and FC4, three more APs are synthesized. As the APs for FC2 and FC3 differ from the AP for FC1 only in the memoization nodes, we put them in Appendix A due to the space limitation. Figure 9 shows the AP for FC4, which traverses a different code path from the other three.

As shown in Figure 10, the four APs for $\texttt{Tx}_e$ pre-executed in FC1 to FC4 can be merged into one AP. The merged AP consists of merged constraint sets (i.e., the constraint checking code), merged shortcuts, and a collection of corresponding fast-paths. The constraint checking code of different APs can be merged because any two of them have a non-empty common prefix. They only diverge at guard instructions, which correspond to the control flow splitting points. For example, the AP for FC1 (Figure 8) and the AP for FC4 (Figure 9) diverge at the guard instruction at n8 (corresponding to the IF-statement in line s12). The n8 guard in FC1 requires that the ELSE-branch be taken, while the same guard in FC4 instead requires that the IF-branch be taken. The two n8 guards are merged into one as shown in Figure 10. The merged n8 guard serves a dual purpose of checking and case-branching the constraint sets (e.g., FC1,2,3 vs. FC4). If the guarded register (e.g., v6) does not take any of the guarded values (e.g., 0 or 1), a constraint violation is detected, in which case a full execution of the original transaction in EVM will be triggered.

When a branch is taken, the subsequent AP execution will switch to check the remaining constraints, if any, in the selected sets, and do further case-branching if needed, until either a violation is detected or a fast-path program is reached and executed. In our example, if v6 equals 0, the constraint set derived from FC1 (same as FC2 or FC3) will be selected and n9 will be executed next (the left transition arrow labelled with 0), which is the first instruction of the corresponding fast-path. If v6 equals 1, the fast-path program synthesized in FC4 (i.e., n17 to n19) will be executed instead. The case-branching functionality of the guard instructions is crucial for performance. It makes the time complexity of executing the merged AP, which combines N individual APs, independent of N, thus can handle the many-future situation nearly as efficiently as a single future.

The shortcut nodes of multiple APs can also be merged. Similar to a merged guard, a merged shortcut node serves a dual purpose of memoization and case-branching of constraint sets. For example, shortcut node n5 for the n7-to-n8 segment in AP for FC1 (Figure 8) and its counterpart in the AP for FC4 (Figure 9) are merged into one in Figure 10. It determines which of the two AP branches (split at n8) to take based on the value (e.g., 3990000 or 3990300) of the read-set register v5. Note that the shortcuts are independent from each other. For example, it is possible that an actual execution takes the FC3/FC4 branch at the m1-shortcut, and the FC2 branch at the m3-shortcut. Therefore, the correct parts of several predicted contexts can be stitched together to achieve acceleration.

4.4 Predictor and Prefetcher

Multi-future predictor. Forerunner's *predictor* leverages the knowledge about Ethereum to pick transactions likely to be packed next from the pending pool and construct probable future contexts for those transactions. The resulting transactions and their contexts serve as inputs to the Speculator. The predictor consists of the *next-block predictor* and the *context constructor*. The *next-block predictor* simulates how miners pack blocks to predict which pending transactions are likely packed into the next block. It uses two heuristics of the miners: 1) transactions with higher gas prices (transaction fees) are usually packed into blocks earlier, 2) some miners prioritize the transactions from themselves. The strategy is to err on the side of recall over precision to increase the chance that transactions are pre-executed before being packed into blocks, compensated by a capping mechanism to avoid overwhelming the node with too many pre-executions. It also collects simple statistics about the block generation process to predict the block's header information (e.g., timestamp and coinbase). The *context constructor* examines the dependencies among the pending transactions to group those with dependencies into the same group. The ordered list, denoted as `Tx order` in Figure 5, that affects a transaction's context is therefore just a list within the transaction's own group. The *context constructor* simulates miner preferences when ordering transactions and exposes inherent non-determinism by enumerating all the possible contexts in a random order (or does random sampling if there are too many).

The predictor's strategy is tailored to the packing behaviors dictated and incentivized by the Ethereum protocol. As will be shown later, we have demonstrated that it is feasible to construct a practical and effective predictor for the latest version of the protocol as of this writing. Fully addressing more adversarial situations, such as DoS attacks against the predictor, may require further work. Preferably, the predictor and revisions to the protocol can be co-designed in the future if Forerunner gets adopted into the public Ethereum.

State prefetcher. Ethereum's state consists of key-value pairs cryptographically indexed by a Merkle-Patricia trie [99]. Transaction execution on the critical path accesses the blockchain's state via StateDB objects, each corresponding to a snapshot of Ethereum's state. Looking up the value associated with a key involves iteratively loading StateDB objects from the disk to decode all the serialized intermediate nodes from the trie's root down to the leaf node, which contains the value. The values and the decoded intermediate nodes are cached to expedite future lookups. Our prefetcher utilizes this caching mechanism to reduce substantially the number of lookups on the critical path by pre-creates StateDBs to populate the internal caches of the StateDBs with the values likely accessed on the critical path.

5 Implementation and Evaluation

We have implemented Forerunner in Go [101] Ethereum, a.k.a. Geth, v1.9.9 [100]. It consists of 34,815 lines of code: 2,482 for the multi-future predictor, 24,836 for the speculator,

Table 1. Datasets used in our evaluations

Tag[†]	Duration	Block number range	Block Count [‡]	Tx count	% of heard txs	%(weighted*) of heard txs
L_1	03/12/2021-03/22/2021	12024033-12087152	66173	13079242	93.31%	95.71%
R_1	03/12/2021-03/22/2021	12024033-12087152	66173	13079242	92.47%	93.28%
R_2	02/01/2021-02/03/2021	11767001-11780202	13845	2776899	92.24%	91.45%
R_3	01/01/2020-01/05/2020	11566001-11596913	32274	5932909	95.73%	96.92%
R_4	12/09/2020-12/11/2020	11416001-11430726	15397	2672694	97.22%	98.15%
R_5	10/01/2020-10/04/2020	10967001-10991575	25795	4028889	97.59%	97.91%

[†] L_n: run in live mode; R_n: run in replay mode.
[‡] Including blocks that are on temporary forks of the main chain.
* The percentage weighted by transaction's baseline execution time.

5,028 for the transaction execution accelerator, 558 for the prefetcher, and the rest 1,911 lines for an emulator used in evaluation. The source code and all the data sets described below are publicly accessible at https://github.com/microsoft/Forerunner.

We leverage Ethereum's openness to run Forerunner as an Ethereum node to process *live traffic*. It means not only that the real and complete traffic is used to measure how Forerunner would perform on the current public Ethereum workload, but also all the timings are real so that APs must be generated in time to achieve any speedups. Our evaluation of Forerunner focuses on the following aspects: (1) correctness validation, (2) the speedup and constraint-set satisfaction rate achieved on real Ethereum traffic, (3) how much each individual technology contributes to the overall result, and (4) the stability of the result over multiple time periods.

5.1 Testbed and Datasets

Testbed. Our testbed consists of Azure Dedicated Host [97] DSv3-Type1 physical servers, with 2.3 GHz Intel® Xeon® E5-2673 v4 (Broadwell) (64 vCPUs, 256 GB memory), running Ubuntu 16.04.6 LTS. Each server hosts only one Azure Standard D64s v3 VM to avoid performance interference. To run an evaluation task, we use four identical VMs with identical configurations. Every evaluation result we report is the average of the four independent results, with low variance: all the coefficients of variation are below 0.06.

Datasets. We run Forerunner as a normal Ethereum node observing and processing real-time transactions on Ethereum. The main evaluation was conducted in a 10-day period of March 12–22 (2021), as shown in the L_1 row of Table 1.

To make our results reproducible and to evaluate the latest version of Forerunner in periods of historical traffic for comparison, we develop a recorder/emulator that can record and replay the live traffic of Ethereum. The recorder is a dedicated and live Geth node modified to capture all the pending transactions and the blocks it receives from the Ethereum network in a given period with precise timings. The emulator is a Geth node with Forerunner integrated, which is modified to not connected to the public network. Instead, it takes a period of recorded traffic and a copy of the local blockchain database, resets the state to where the traffic starts, and replays the traffic faithfully, making sure the relative arrival timings of the transactions and blocks are accurately respected, to reproduce the results of a live Forerunner node in this traffic.

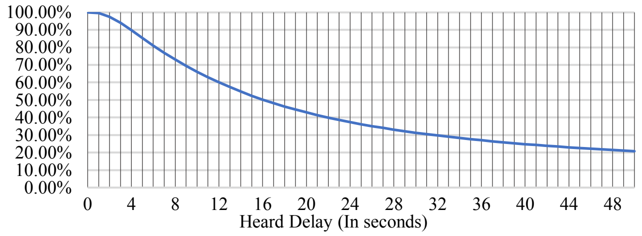

Figure 11. Reverse CDF of heard delay (y percentage of heard transactions whose delay exceed x seconds).

We recorded the real traffic in five time periods, shown as $R_1 - R_5$. The period R_1 is the same as L_1 for validation of the recorder/emulator. R_1 has a different heard rate than L_1 because our traffic recorder is in an independent Ethereum node, which had different p2p connections with the rest of the Ethereum network. The other four periods represent our unbiased random sampling of traffic across 5 months, offering extra datasets for evaluating Forerunner against the natural evolution of Ethereum's traffic characteristics.

Opportunity for speculative execution. Our hypothesis that leads to speculative execution is that a substantial number of transactions can be *heard* during the dissemination phase in the DiCE paradigm. This is confirmed by column %Heard txs of Table 1, which shows that the Forerunner nodes indeed heard 92.24% ∼ 97.59% of the transactions (91.45% ∼ 98.15% if weighted by execution time) before they were mined into blocks. Figure 11 shows the reverse CDF of the delay between when a pending transaction is heard and when it needs to be executed, which is the time window for speculative pre-execution. It shows that, for more than 90% of the heard transactions, the time window exceeds 4 seconds.

5.2 Correctness Validation

Forerunner has been designed rigorously and implemented carefully to ensure correctness of speculative execution, including taking into account the peculiarities of Ethereum, such as the implicit gas-limit checking.

Forerunner's correctness has been validated extensively as a by-product of our evaluation. This is because Ethereum maintains its state as a Merkle tree. Two states are identical if and only if the values of their Merkle roots are equal. Every

Table 2. Effective speedup

	Speedup	% satisfied	% (weighted*)
Baseline	1x	N/A	N/A
Forerunner	8.39x	99.16%	98.41%
Perfect matching	2.11x	68.81%	51.40%
Perfect matching + multi-future prediction	5.13x	87.59%	84.64%

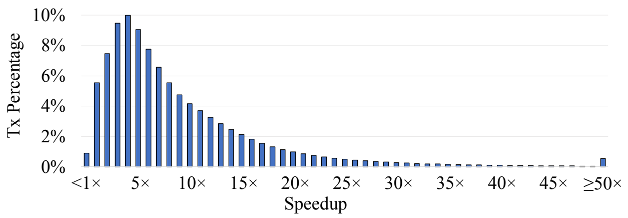

Figure 12. Speedup distribution across all heard transactions

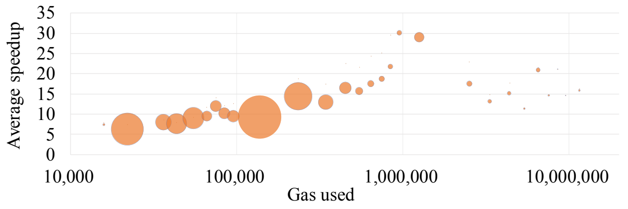

Figure 13. The correlation between gas used and average speedup achieved over effectively predicted transactions (bubble size proportional to transaction count)

block carries a value of the state's Merkle root after all the transactions in the block are executed. As shown in Table 1, Forerunner has managed to process a total number of 121,210 blocks carrying 22,557,724 transactions, always resulting in a matching value of the Merkle root with others in every block, despite the use of speculative execution.

5.3 Speedup with Breakdowns

To measure the effectiveness of speculative execution, we report the *effective speedup* as the average speedup observed on all transactions where speculative execution is applied. Table 2 shows the results on the L_1 live traffic. Forerunner achieves a 8.39× speedup compared to the official Go-Ethereum (Geth) 1.9.9, which is our baseline. The percentage of the transactions satisfying AP's constraint-sets (hench accelerated by APs) is 99.16%. Because transactions' execution times vary substantially, ranging from sub-milliseconds to tens of milliseconds, with an average of 0.56 milliseconds baseline execution time, we also report a *weighted percentage* of 98.41%, calculated with each transaction weighted by its baseline time. This means that 98.41% of time that spent by Ethereum for processing transactions can be accelerated by APs.

To better understand the 8.39× speedup, we further show the distribution of speedup in Figure 12: most are showing a speedup between 2 and 20; only 0.88% of the transactions are not accelerated, with 0.53% accelerated by more than 50×. We even observe some over 1000×. As shown in Figure 13, Forerunner generally achieve higher speedups on more complex transactions as measured by the amount of gas used. It suggests that, if the future workload of Ethereum consists of a larger portion of complex transactions, we may expect a better overall speedup from Forerunner.

Table 3. Breakdown by prediction outcome

		% txs	% (weighted*)	Speedup
satisfied	perfect	87.19%	83.84%	11.33×
	imperfect	11.96%	14.58%	4.55×
unsatisfied	missed	0.85%	1.59%	1.21×

* The percentage weighted by transaction's baseline execution time.

Table 2 further provides two reference points to understand the value of Forerunner's innovative approach. We estimate the effectiveness of using a traditional speculative execution approach that demands perfect matching with speculation: the resulting speedup is only 2.11×, and the weighted percentage of perfectly predicted transactions is 51.40%. Even with the addition of multiple futures, we observe a speedup of 5.13× with the weighted percentage reaching 84.64%. This result shows that the multi-future prediction is beneficial for speculative execution in Ethereum, but the constraint-based approach is significantly more effective. This more sophisticated approach is crucial for breaking through the speedup limit imposed by hard-to-predict transactions. We believe that such a superior prediction effectiveness may enable even higher speedups with further engineering efforts.

Breakdown by prediction outcomes. Table 3 shows the breakdown for the "Forerunner" row in Table 2 by three prediction outcomes: perfect prediction, imperfect prediction, and missed prediction. *Perfect prediction* means that a transaction's context is identical to one of the speculated contexts. *Imperfect prediction* means that a transaction's context satisfies a constraint-set of an AP, but is not identical to any speculated context[13]. *Missed prediction* refers to the remaining situation. The average speedup is 11.33× for a perfect prediction and 4.55× for an imperfect prediction. Even for missed predictions, a majority of transactions still get the benefit from prefetching, so the average speedup is 1.21×.

End-to-end speedup. The effective speedup excludes unheard transactions and the end-to-end speedup, which includes unheard transactions, drops to 6.06× with the weighted percentage at 94.19% for dataset L_1. To understand why, we observe the speedup on unheard transactions at 0.81 (i.e., a modest slowdown), due to the overhead of Forerunner's current implementation.

5.4 Emulation using recorded datasets

We conduct the same experiments using the emulator on all recorded datasets, shown in Figure 14. The emulation result on R_1 is sufficiently close to the real experimental

[13] For example, if an actual execution has `v1=3990555` and `v5=3990000`, then the AP in Figure 9 is considered an imperfect prediction, because `m1` needs to take the `else` transition, but the guard at `n5` can still be satisfied.

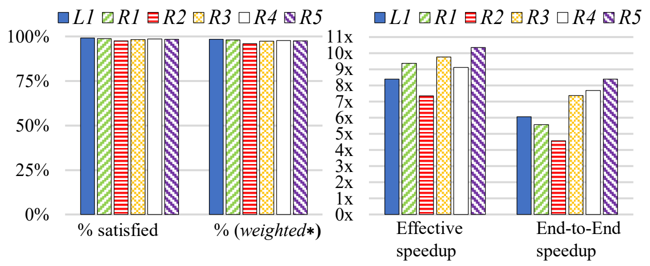

Figure 14. Evaluations using L_1 and recorded datasets

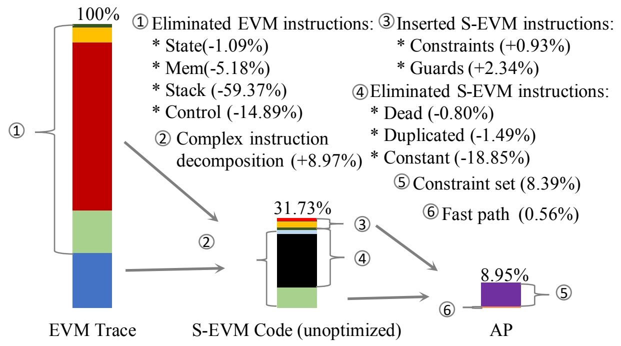

Figure 15. Code reduction during AP synthesis.

result on L_1 to validate our emulator. We observe that the percentage and weighted percentage of heard transactions satisfying constraint-sets are above 95% across the board. The end-to-end speedups vary from 4.56× to 8.38×.

5.5 AP Synthesis and Execution

Figure 15 shows detailed measurements on how the AP synthesis process in Figure 6 converts a recorded EVM trace (the first column) to S-EVM code (the second column) and then optimizes it into a highly compact AP program (the third column). The results shown are the average of all the APs synthesized in L_1 and are normalized by setting the first column as 100%. In this subsection, the symbol "%" means "percentage point". This figure shows the effects of each individual optimization or conversion. For example, the stack-related EVM instructions eliminated by the stack-to-register conversion account for 59.37% of the original EMV trace. The number of S-EVM instructions eliminated by constant folding equals 18.85% of the original EVM trace. Complex instruction decomposition adds 8.97% extra S-EVM instructions. The net result is that the synthesized AP path, including all the constraint checking code and the fast path program, is only 8.95% of the size of the original EVM trace. This confirms that the CD-Equiv constraints indeed offer significant specialization and acceleration opportunities. Each AP path's average S-EVM instruction count is 351.

Among the transactions, 82.2% have one AP path synthesized, 13.5% have two distinct AP paths synthesized and merged, and 2.4% have three. The rest 1.9%, which have more than three, on average, have 14 paths. For AP synthesis, 63.4% of the transactions have been pre-executed in one distinct future context, 4.0% in two distinct contexts, 1.0% have been in three. The rest 31.4%, in more than three contexts, on average, have been pre-executed in 47 distinct future contexts. In addition to code size reduction, an average number of 311 shortcuts (nodes and edges) are added to each AP path via memoization. They enable 80.92% of the S-EVM instructions to be skipped in the APs executed on the critical path.

5.6 Overhead Off the Critical Path

While the primary goal of this work is to understand the performance potential of constraint-based speculative execution on the critical path, we also measure the performance off the critical path. For the speculator, which is the major component off the critical path, the end-to-end time to pre-execute a transaction in a context and synthesize an AP takes on average 12.19× the time to execute the transaction, without being optimized. We further report the (unoptimized) resource consumption of the current implementation of Forerunner, covering the cost of the multi-future predictor, the speculator, the prefetcher, and the cache. Running on the VMs in our testbed, the average memory utilization of Forerunner is 67.05 GB, the average utilization of all 64 vCPUs is 23.84%. In comparison, for the baseline (i.e., the official geth), the former is 19.15 GB, and the latter is 5.51%. In other words, with Forerunner, the CPU utilization is 3.33× higher than the baseline, and the memory consumption 2.50× higher.

6 Related work

Speculative execution. Speculative execution has a wide range of successful applications across all the layers of computer systems [10, 37, 45, 52, 63, 70, 72, 75, 90, 112, 113, 115]. In most of these approaches, if a perfect matching on inputs is not found, a full or partial [91] re-execution is triggered.The rest of them use execute-verify models [3, 39], which trigger a full re-execution when replicas disagree on the speculative results. Forerunner introduces a new paradigm by generalizing perfect matching to constraint satisfaction, and pre-computed result commitment to fast-path execution.

Program specialization & memoization. Program specialization [17, 30, 66, 68, 76, 80, 92, 94] is a way to speed up a program when some inputs are known in advance. Forerunner differs from the existing approaches by further restricting the execution of the specialized code to specific control flow and data dependencies via constraints, so that more optimizations can be applied. The tradeoff is that the specialized program supports only a subset of all its possible inputs; i.e., those satisfy the constraints. Forerunner shares some core ideas with tracing just-in-time compilation [6, 8, 9, 12, 15, 25, 32, 33, 35, 50, 65, 86], i.e., trace based optimization with special guards inserted to detect execution divergence. Forerunner's data guards enable optimizations that make sure the AP can be synthesized in a register-only IR, which simplifies the implementation of memoization. Memoization [1, 2, 18, 23, 29, 36, 46, 57, 59, 62, 78, 87, 88, 93, 96] is a widely applicable optimization that saves execution time by returning cached results of an expensive function (or other unit of code execution) if it is called with a repeated input. Forerunner's shortcuts can cross the artificial boundaries created by high-level program structures.

ShortCut [26] is a highly related work, which shares a common theme of accelerating "similar" executions via specialization. ShortCut aims to accelerate "mostly-deterministic code regions" for those expected to be executed many times. It is demonstrated on server and desktop applications. The key insight is that these different executions are "similar" because a large subset of the inputs remains unchanged. ShortCut proposes a novel *partial memoization*, which generates executable slices for the subset of likely-variant inputs identified offline. The specialization employed, which tolerates more divergences than allowed by CD-Equiv, is to make the slicing manageable, without applying aggressive optimizations to shorten the code as Forerunner does. The slicing happens offline without the stringent time constraints under which Forerunner has to operate. In contrast, Forerunner aims to use *speculative execution* to accelerate blockchain transactions, each of which is expected to be executed in the near future. Unlike ShortCut, our notion of "similarity" is about CD-Equiv. It increases the coverage of pre-executions, because CD-Equiv does not (necessarily) require any inputs to be identical. By strictly adhering to CD-Equiv, Forerunner can use highly efficient program specialization to generate an AP (constraints + fast-path) program, which is already faster than the original transaction without (necessarily) memoizing any concrete input value. Furthermore, the role and design of memoization in Forerunner are different from those in ShortCut. In Forerunner, memoization is not the top-level goal, but a further optimization for the already optimized AP code. The design does not use slices, but jumps based on individual variable values.

Blockchain performance. Existing techniques for improving blockchain's throughput can be broadly categorized into two classes. This first class aims to enable bigger blocks by developing technologies such as parallel transaction execution [24, 116, 117], JIT [108], and hardware [54], to process more transactions within the same limited time window of each block. Forerunner achieves the same goal with a different and complementary approach, i.e., by allowing transactions to be speculatively executed in advance to speed up execution on the critical path. The other class of techniques aims to increase the block generation frequency by (i) employing DAG-based chain structures [48, 49, 83–85], by (ii) utilizing heterogeneity [5, 28] to allow certain blocks to be generated faster, proposing alternative consensus protocols [11, 13, 34, 41, 43, 60, 64, 73, 89] to reduce consensus time between two blocks, by (iii) developing sharding solutions [44, 47, 55, 71, 95, 107, 118, 119] or other parallel-chain techniques [58, 114] to introduce a collection of partially independent sub-chains each of which can generate blocks in parallel, or by (iv) layer-2 solutions [20, 27, 38, 42, 51, 53, 61, 74, 82, 102, 105], to offload certain workloads to external systems.

7 Conclusion

Forerunner's constraint-based approach makes speculative execution profitable in the DiCE paradigm. Every constraint-set generated in a pre-executed transaction defines a class of possible executions of the transaction that are equivalent regarding their control flows and data dependencies. This makes an actual execution highly likely (e.g., about 98.41% likelihood in our main dataset) to benefit from pre-execution. On the other hand, the constraint set provides enough specialization power so that the instruction sequence of a transaction can be aggressively optimized to reduce the code size by one order of magnitude. Memoization complements the power of constraint-sets to further reduce the amount of actually-executed code. With these techniques, Forerunner achieves an effective speedup of 8.39× and an end-to-end speedup of 6.06×. This capability opens up realistic opportunities (e.g, expanding the block size) toward a significant throughput increase in Ethereum's foundational layer.

Acknowledgments

We thank Jiayi Sun, Shiqi He, and Chen Chen for their valuable contributions during the early stage of the project. We are grateful to the anonymous reviewers and our shepherd, Sarah Meiklejohn, for their insightful comments and guidance. Runhuai Li and Yajin Zhou were partially supported by the Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (No. 2018R01005).

References

[1] Umut A Acar, Guy E Blelloch, and Robert Harper. 2003. Selective memoization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 14–25.

[2] Carlos Alvarez, Jesus Corbal, and Mateo Valero. 2005. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.* 54, 7 (2005), 922–927.

[3] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.

[4] L. M. Bach, B. Mihaljevic, and M. Zagar. 2018. Comparative analysis of blockchain consensus algorithms. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 1545–1550.

[5] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. 2019. Prism: Deconstructing the Blockchain to Approach

Physical Limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 585–602.

[6] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 1–12.

[7] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. 2019. SoK: Consensus in the age of blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 183–198.

[8] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G Siek, and Sam Tobin-Hochstadt. 2015. Pycket: a tracing JIT for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 22–34.

[9] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. 2010. SPUR: a trace-based JIT compiler for CIL. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 708–725.

[10] Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M Frans Kaashoek, and Nickolai Zeldovich. 2020. Efficiently Mitigating Transient Execution Attacks using the Unmapped Speculation Contract. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1139–1154.

[11] Iddo Bentov, Charles Lee, Alex Mizrahi, and Meni Rosenfeld. 2014. Proof of activity: Extending bitcoin's proof of work via proof of stake [extended abstract] y. *ACM SIGMETRICS Performance Evaluation Review* 42, 3 (2014), 34–37.

[12] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. 18–25.

[13] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. 2020. Combining GHOST and Casper. *arXiv preprint arXiv:2003.03052* (2020).

[14] Yan Chen and Cristiano Bellavitis. 2020. Blockchain disruption and decentralized finance: The rise of decentralized business models. *Journal of Business Venturing Insights* 13 (2020), e00151.

[15] Lin Cheng, Berkin Ilbeyi, Carl Friedrich Bolz-Tereick, and Christopher Batten. 2020. Type freezing: exploiting attribute type monomorphism in tracing JIT compilers. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 16–29.

[16] John Cocke. 1970. Global common subexpression elimination. In *Proceedings of a symposium on Compiler optimization*. 20–24.

[17] Charles Consel, Julia L. Lawall, and Anne-Françoise Le Meur. 2004. A tour of tempo: a program specializer for the C language. *Science of Computer Programming* 52, 1 (2004), 341–370.

[18] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. 2010. Stable Deterministic Multithreading through Schedule Memoization.. In *OSDI*, Vol. 10. 10.

[19] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[20] Sourav Das, Vinay Joseph Ribeiro, and Abhijeet Anand. 2018. YODA: Enabling computationally intensive contracts on blockchains with Byzantine and Selfish nodes. *arXiv preprint arXiv:1811.03265* (2018).

[21] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. 2003. The case for virtual register machines. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*. 41–49.

[22] Christian Decker and Roger Wattenhofer. 2013. Information propagation in the bitcoin network. In *IEEE P2P 2013 Proceedings*. IEEE, 1–10.

[23] Luca Della Toffola, Michael Pradel, and Thomas R Gross. 2015. Performance problems you can fix: a dynamic analysis of memoization opportunities. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 607–622.

[24] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2019. Adding concurrency to smart contracts. *Distributed Computing* (2019), 1–17.

[25] Stefano Dissegna, Francesco Logozzo, and Francesco Ranzato. 2014. Tracing compilation by abstract interpretation. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 47–59.

[26] Xianzheng Dou, Peter M Chen, and Jason Flinn. 2019. ShortCut: accelerating mostly-deterministic code regions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 570–585.

[27] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. 2018. General state channel networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 949–966.

[28] Ittay Eyal, Adem Efe Gencer, Emin Gun Sirer, and Robbert Van Renesse. 2016. Bitcoin-NG: A Scalable Blockchain Protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*.

[29] Richard A Frost and Barbara Szydlowski. 1996. Memoizing purely functional top-down backtracking language processors. *Science of Computer Programming* 27, 3 (1996), 263–288.

[30] Yoshihiko Futamura. 2013. Generalized partial computation. In *Proceedings of the 2013 conference on Internet measurement conference*. ACM, 127–140.

[31] Enrique Fynn and Fernando Pedone. 2018. Challenges and Pitfalls of Partitioning Blockchains. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 128–133.

[32] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, et al. 2009. Trace-based just-in-time type specialization for dynamic languages. *ACM Sigplan Notices* 44, 6 (2009), 465–478.

[33] Andreas Gal, Christian W Probst, and Michael Franz. 2006. HotpathVM: An effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*. 144–153.

[34] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 51–68.

[35] Shu-yu Guo and Jens Palsberg. 2011. The essence of compiling with traces. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 563–574.

[36] Berthold Hoffmann. 1992. Term rewriting with sharing and memoization. In *International Conference on Algebraic and Logic Programming*. Springer, 128–142.

[37] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, Taebum Kim, and Byung-Gon Chun. 2019. Speculative Symbolic Graph Execution of Imperative Deep Learning Programs. *Operating Systems Review* 53, 1 (2019), 26–33.

[38] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. 2018. Arbitrum: Scalable, private smart contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium*. USENIX Association, 1353–1370.

[39] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about eve: Execute-verify replication for multi-core servers. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 237–250.

[40] Bettina Kemme, Fernando Pedone, Gustavo Alonso, and André Schiper. 1999. Processing transactions over optimistic atomic broadcast protocols. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No. 99CB37003)*. IEEE, 424–431.

[41] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*. Springer, 357–388.

[42] Julia Koch and Christian Reitwiessner. 2018. A Predictable Incentive Mechanism for TrueBit. *arXiv preprint arXiv:1806.11476* (2018).

[43] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *25th USENIX Security Symposium (USENIX Security 16)*. 279–296.

[44] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy*. 583–598.

[45] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 45–58.

[46] Wonchan Lee, Elliott Slaughter, Michael Bauer, Sean Treichler, Todd Warszawski, Michael Garland, and Alex Aiken. 2018. Dynamic tracing: Memoization of task graphs for dynamic task-based runtimes. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 441–453.

[47] Derek Leung. 2018. *Vault: Fast bootstrapping for cryptocurrencies*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[48] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. 2015. Inclusive Block Chain Protocols. In *Financial Cryptography and Data Security*.

[49] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. 2020. A Decentralized Blockchain with High Throughput and Fast Confirmation. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*.

[50] HeuiChan Lim and Saumya Debray. 2021. Automated bug localization in JIT compilers. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 153–164.

[51] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. 2019. Teechain: A Secure Payment Network with Asynchronous Blockchain Access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 63–79. https://doi.org/10.1145/3341301.3359627

[52] Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2019. Accelerating sequential consistency for Java with speculative compilation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2019*. ACM Press, Phoenix, AZ, USA, 16–30. https://doi.org/10.1145/3314221.3314611

[53] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. 2019. Fast and Secure Global Payments with Stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 80–96. https://doi.org/10.1145/3341301.3359636

[54] Tao Lu and Lu Peng. 2020. BPU: a blockchain processing unit for accelerated smart contract execution. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[55] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A Secure Sharding Protocol For Open Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 17–30.

[56] Renaud Marlet. 2012. *Program Specialization*. John Wiley & Sons, Ltd.

[57] Simon Marlow. 2011. Parallel and concurrent programming in Haskell. In *Central European Functional Programming School*. Springer, 339–401.

[58] Will Martino, Monica Quaintance, and Stuart Popejoy. 2018. Chainweb: A proof-of-work parallel-chain architecture for massive throughput. *Chainweb Whitepaper* 19 (2018).

[59] James Mayfield, Tim Finin, and Marty Hall. 1995. Using automatic memoization as a software engineering tool in real-world AI systems. In *Proceedings the 11th Conference on Artificial Intelligence for Applications*. IEEE, 87–93.

[60] David Mazières. 2015. The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus.

[61] Patrick McCorry, Surya Bakshi, Iddo Bentov, Sarah Meiklejohn, and Andrew Miller. 2019. Pisa: Arbitration outsourcing for state channels. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 16–30.

[62] Donald Michie. 1968. "Memo" functions and machine learning. *Nature* 218, 5136 (1968), 19–22.

[63] James W Mickens, Jonathan R Howell, Jacob R Lorch, Jeremy E Elson, and Edmund B Nightingale. 2012. Network application performance enhancement using speculative execution. US Patent 8,140,646.

[64] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 31–42.

[65] James George Mitchell. 1970. *The design and construction of flexible and efficient interactive programming systems*. Carnegie Mellon University.

[66] Torben Ae Mogensen. 1989. Separating binding times in language specifications. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture (FPCA)*. 12–25.

[67] Steven Muchnick et al. 1997. *Advanced compiler design implementation*. Morgan kaufmann.

[68] G. Muller, R. Marlet, E.-N. Volanschi, C. Consel, C. Pu, and A. Goel. 1998. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No.98CB36183)*. 240–249.

[69] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[70] Naveen Neelakantam, David R. Ditzel, and Craig Zilles. 2010. A real system evaluation of hardware atomicity for software speculation. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems (ASPLOS XV)*. Association for Computing Machinery, Pittsburgh, Pennsylvania, USA, 29–38. https://doi.org/10.1145/1736020.1736026

[71] Lan N Nguyen, Truc DT Nguyen, Thang N Dinh, and My T Thai. 2019. Optchain: optimal transactions placement for scalable blockchain sharding. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 525–535.

[72] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. 2005. Speculative execution in a distributed file system. *ACM SIGOPS Operating Systems Review* 39, 5 (Oct. 2005), 191–205. https://doi.org/10.1145/1095809.1095829

[73] Rafael Pass and Elaine Shi. 2017. Hybrid Consensus: Efficient Consensus in the Permissionless Model. In *31st International Symposium on Distributed Computing (DISC 2017)*. 39:1–39:16.

[74] Joseph Poon and Vitalik Buterin. 2017. Plasma: Scalable autonomous smart contracts. *White paper* (2017), 1–47.

[75] Junqiao Qiu, Lin Jiang, and Zhijia Zhao. 2020. Challenging Sequential Bitstream Processing via Principled Bitwise Speculation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, Lausanne, Switzerland, 607–621. https://doi.org/10.1145/3373376.3378461

[76] Thomas Reps and Todd Turnidge. 1996. Program specialization via program slicing. In *Partial Evaluation*. Springer, 409–429.

[77] A. V. S. Sastry and Roy D. C. Ju. 1998. A New Algorithm for Scalar Register Promotion Based on SSA Form. *SIGPLAN Not.* 33, 5 (May 1998), 15–25. https://doi.org/10.1145/277652.277656

[78] Eric Schnarr and James R Larus. 1998. Fast out-of-order processor simulation using memoization. *ACM SIGPLAN Notices* 33, 11 (1998), 283–294.

[79] David Schneider and Carl Friedrich Bolz. 2012. The efficient handling of guards in the design of RPython's tracing JIT. In *Proceedings of the sixth ACM workshop on Virtual machines and intermediate languages*. 3–12.

[80] Ulrik P Schultz, Julia L Lawall, and Charles Consel. 2003. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2003), 452–499.

[81] Yunhe Shi, Kevin Casey, M Anton Ertl, and David Gregg. 2008. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization (TACO)* 4, 4 (2008), 1–36.

[82] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Kathleen Ruan, Parimarjan Negi, Lei Yang, Radhika Mittal, Giulia Fanti, and Mohammad Alizadeh. 2020. High Throughput Cryptocurrency Routing in Payment Channel Networks. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 777–796.

[83] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. 2017. SPECTRE : Serialization of Proof-of-work Events : Confirming Transactions via Recursive Elections.

[84] Yonatan Sompolinsky and Shai Wyborski. 2020. PHANTOM and GHOSTDAG: A Scalable Generalization of Nakamoto Consensus.

[85] Yonatan Sompolinsky and Aviv Zohar. 2015. Secure High-Rate Transaction Processing in Bitcoin. In *Financial Cryptography and Data Security*.

[86] Gregory T Sullivan, Derek L Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. 2003. Dynamic native optimization of interpreters. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*. 50–57.

[87] Arjun Suresh, Erven Rohou, and André Seznec. 2017. Compile-time function memoization. In *Proceedings of the 26th International Conference on Compiler Construction*. 45–54.

[88] Arjun Suresh, Bharath Narasimha Swamy, Erven Rohou, and André Seznec. 2015. Intercepting functions for memoization: A case study using transcendental functions. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 2 (2015), 18–1.

[89] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J. Fischer, and Bryan Ford. 2017. Scalable Bias-Resistant Distributed Randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*. 444–460.

[90] Chen Tian, Min Feng, and Rajiv Gupta. 2010. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. Association for Computing Machinery, Toronto, Ontario, Canada, 62–73. https://doi.org/10.1145/1806596.1806604

[91] Chen Tian, Changhui Lin, Min Feng, and Rajiv Gupta. 2011. Enhanced speculative parallelization via incremental recovery. *ACM SIGPLAN Notices* 46, 8 (2011), 189–200.

[92] Valentin F Turchin. 1986. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1986), 292–325.

[93] Georgios Tziantzioulis, Nikos Hardavellas, and Simone Campanoni. 2018. Temporal approximate function memoization. *IEEE Micro* 38, 4 (2018), 60–70.

[94] Philip Wadler. 1988. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*. Springer, 344–358.

[95] Jiaping Wang and Hao Wang. 2019. Monoxide: Scale out Blockchains with Asynchronous Consensus Zones. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[96] David S Warren. 1999. Programming in tabled prolog (very) draft. *Unfinished book on XSB programming* (1999).

[97] Website. [n.d.]. Azure Dedicated Host. https://azure.microsoft.com/en-us/services/virtual-machines/dedicated-host/.

[98] Website. [n.d.]. Chainlink: Blockchain Oracles for Connected Smart Contracts. https://chain.link/.

[99] Website. [n.d.]. Ethereum White Paper. https://ethereum.org/en/whitepaper/.

[100] Website. [n.d.]. Geth version 1.9.9. https://github.com/ethereum/go-ethereum/releases/tag/v1.9.9.

[101] Website. [n.d.]. The Go Programming Language. https://golang.org/.

[102] Website. [n.d.]. Optimistic-Rollups for Ethereum. https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/optimistic_rollups/.

[103] Website. [n.d.]. Price Oracle - A Must Have Infrastructure. https://www.huobidefilabs.io/en/Insight/1313163603254243330.

[104] Website. [n.d.]. The Solidity Contract-Oriented Programming Language. https://soliditylang.org/.

[105] Website. [n.d.]. State Channels for Ethereum. https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/state-channels/.

[106] Website. [n.d.]. Toward a 12 second block time. https://blog.ethereum.org/2014/07/11/toward-a-12-second-block-time/.

[107] Website. 2018. The Eth2 upgrades | ethereum.org. https://ethereum.org/en/eth2/. [Online; accessed Apr-2021].

[108] Website. 2018. Ethereum flavored WebAssembly (ewasm). https://github.com/ewasm.

[109] Website. 2020. Ethereum Homepage. https://www.ethereum.org/.

[110] Website. 2020. Etherscan. https://etherscan.io/.

[111] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151 (2014), 1–32.

[112] Yang Xia, Peng Jiang, and Gagan Agrawal. 2020. Scaling out Speculative Execution of Finite-State Machines with Parallel Merge *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 160–172. https://doi.org/10.1145/3332466.3374524

[113] Xinan Yan, Arturo Pie Joa, Bernard Wong, Benjamin Cassell, Tyler Szepesi, Malek Naouach, and Disney Lam. 2018. SpecRPC: A General Framework for Performing Speculative Remote Procedure Calls. In *Proceedings of the 19th International Middleware Conference on*. 266–278.

[114] Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. 2020. Ohie: Blockchain scaling made simple. In *2020 IEEE Symposium on Security and Privacy (SP)*. 90–105.

[115] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. 2020. Speculative Data-Oblivious Execution: Mobilizing Safe Prediction For Safe and Efficient Speculative Execution. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 707–720. https://doi.org/10.1109/ISCA45697.2020.00064

[116] Lian Yu, Wei-Tek Tsai, Guannan Li, Yafe Yao, Chenjian Hu, and Enyan Deng. 2017. Smart-contract execution with concurrent block building. In *2017 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE, 160–167.

[117] Wei Yu, Kan Luo, Yi Ding, Guang You, and Kai Hu. 2018. A Parallel Smart Contract Model. In *Proceedings of the 2018 International Conference on Machine Learning and Machine Intelligence*. 72–77.
[118] S. M. Zamani, M. Movahedi, and Mariana Raykova. 2018. RapidChain : A Fast Blockchain Protocol via Full.
[119] Jianting Zhang, Zicong Hong, Xiaoyu Qiu, Yufeng Zhan, Song Guo, and Wuhui Chen. 2020. SkyChain: A Deep Reinforcement Learning-Empowered Dynamic Blockchain Sharding System. In *49th International Conference on Parallel Processing - ICPP*.

A APs synthesized in FC2, FC3.

The AP synthesized in FC2 is shown in Figure 16. The AP synthesized in FC3 is shown in Figure 17.

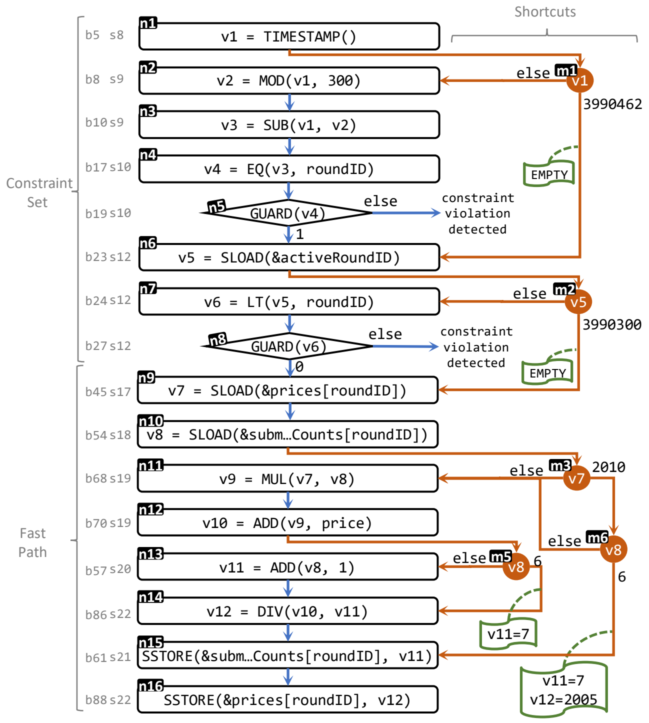

Figure 16. Accelerated program of Tx_e synthesized in FC2. All the variables whose names do not start with v, such as roundID, &activeRoundID, are constants. The corresponding line numbers in EVM trace and the source code are annotated in the left two columns.

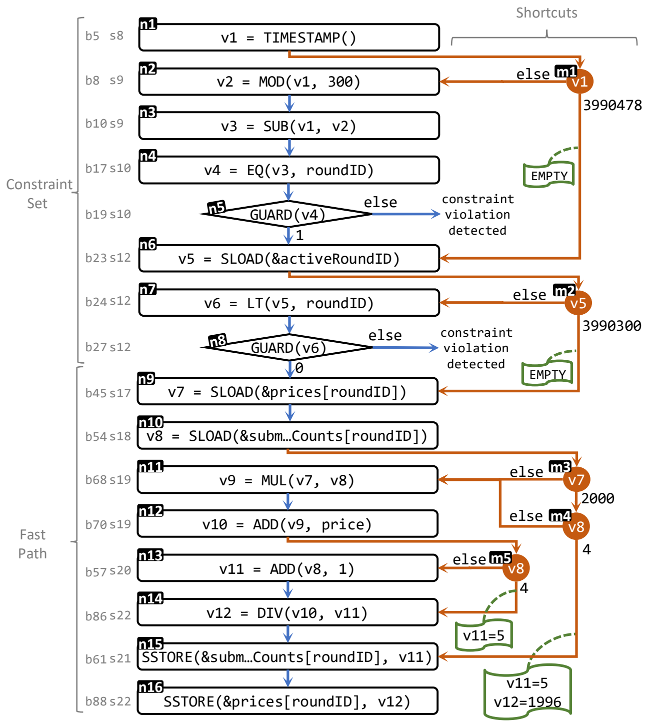

Figure 17. Accelerated program of Tx_e synthesized in FC3. All the variables whose names do not start with v, such as roundID, &activeRoundID, are constants. The corresponding line numbers in EVM trace and the source code are annotated in the left two columns.