Adding Concurrency to Smart Contracts

Thomas Dickerson
Brown University
thomas_dickerson@brown.edu

Paul Gazzillo
Yale University
paul.gazzillo@yale.edu

Maurice Herlihy
Brown University
maurice_herlihy@brown.edu

Eric Koskinen
Yale University
eric.koskinen@yale.edu

ABSTRACT

Modern cryptocurrency systems, such as Ethereum, permit complex financial transactions through scripts called *smart contracts*. These smart contracts are executed many, many times, always without real concurrency. First, all smart contracts are serially executed by *miners* before appending them to the blockchain. Later, those contracts are serially re-executed by *validators* to verify that the smart contracts were executed correctly by miners.

Serial execution limits system throughput and fails to exploit today's concurrent multicore and cluster architectures. Nevertheless, serial execution appears to be required: contracts share state, and contract programming languages have a serial semantics.

This paper presents a novel way to permit miners and validators to execute smart contracts in parallel, based on techniques adapted from software transactional memory. Miners execute smart contracts speculatively in parallel, allowing non-conflicting contracts to proceed concurrently, and "discovering" a serializable concurrent schedule for a block's transactions, This schedule is captured and encoded as a deterministic *fork-join* program used by validators to re-execute the miner's parallel schedule deterministically but concurrently.

Smart contract benchmarks run on a JVM with ScalaSTM show that a speedup of 1.33x can be obtained for miners and 1.69x for validators with just three concurrent threads.

1 INTRODUCTION

Cryptocurrencies such as Bitcoin [20] or Ethereum [8] are very much in the news. Each is an instance of a *distributed ledger*: a publicly-readable tamper-proof record of a sequence of events. Simplifying somewhat, early distributed ledgers, such as Bitcoin's, work

like this: *clients* send *transactions*[1] to *miners*, who package the transactions into *blocks*. Miners repeatedly *propose* new blocks to be applied to the ledger, and follow a global consensus protocol to agree on which blocks are chosen. Each block contains a cryptographic hash of the previous block, making it difficult to tamper with the ledger. The resulting distributed data structure, called a *blockchain*, defines the sequence of transactions that constitutes the distributed ledger[2].

Modern blockchain systems often interpose an additional software layer between clients and the blockchain. Client requests are directed to scripts, called *smart contracts*, that perform the logic needed to provide a complex service, such as managing state, enforcing governance, or checking credentials. Smart contracts can take many forms, but here we will use (a simplified form of) the Ethereum model [8].

A smart contract resembles an object in a programming language. It manages long-lived *state*, which is encoded in the blockchain. The state is manipulated by a set of *functions*, analogous to *methods* in many programming languages. Functions can be called either directly by clients or indirectly by other smart contracts. Smart contract languages are typically Turing-complete. To ensure that function calls terminate, the client is charged for each computational step in a function call. If the charge exceeds what the client is willing to pay, the computation is terminated and rolled back.

When and where is smart contract code executed? There are two distinct circumstances. Each smart contract is first executed by one or more *miners*, nodes that repeatedly propose new blocks to append to the blockchain. When a miner creates a block, it selects a sequence of user requests and executes the associated smart contract code for each Ethereum transaction in sequence, transforming the old contract state into a new state. It then records both the sequence of transactions and the new state in the block, and proposes it for inclusion in the blockchain.

Later, when the block has been appended to the blockchain, each smart contract is repeatedly re-executed by *validators*: nodes that reconstruct (and check) the current blockchain state. As a validator acquires each successive block, it replays each of the transactions' contract codes to check that the block's initial and final states match. Each miner validates blocks proposed by other miners, and older block are validated by newly-joined miners, or by clients querying the contract state. Code executions for validation vastly exceed code executions for mining.

[1]Following blockchain terminology, a transaction is a payment or set of payments, not an atomic unit of synchronization as in databases or transactional memory.
[2]This description omits many important issues, such as incentives, forking, and fork resolution.

Existing smart contract designs limit throughput because they admit no concurrency. When a miner creates a block, it assembles a sequence of transactions, and computes a tentative new state by executing those transactions' smart contracts serially, in the order they occur in the block. A miner cannot simply execute these contracts in parallel, because they may perform conflicting accesses to shared data, and an arbitrary interleaving could produce an inconsistent final state. For Bitcoin transactions, it is easy to tell in advance when two transaction conflict, because input and output data are statically declared. For smart contracts, by contrast, it is impossible to tell in advance whether two contract executions will conflict, because the contract language is Turing-complete.

Miners are rewarded for each block they successfully append to the blockchain, so they have a strong incentive to increase throughput by parallelizing smart contract executions. We propose to allow miners to execute contract codes in parallel by adapting techniques from Software Transactional Memory (STM) [12]: treating each invocation as a speculative atomic action. Data conflicts, detected at run-time, are resolved by delaying or rolling back some conflicting invocations. Treating smart contract invocations as speculative atomic actions dynamically "discovers" a *serializable* concurrent schedule, producing the same final state as a serial schedule where the contract functions were executed in some one-at-a-time order.

But what about later validators? Existing STM systems are *nondeterministic*: if a later validator simply mimics the miner by re-running the same mix of speculative transactions, it may produce a different serialization order and a different final state, causing validation to fail incorrectly. Treating contract invocations as speculative transactions improves miners' throughput, but fails to support deterministic re-execution as required by validators.

Notice, however, that the miner has already "discovered" a serializable concurrent schedule for those transactions. We propose a novel scheme where the miner records that successful schedule, along with the final state, allowing later validators to replay that same schedule in a concurrent but deterministic way. Deterministic replay avoids many of the the miner's original synchronization costs, such as conflict detection and roll-back. Over time, parallel validation would be a significant benefit because validators perform the vast majority of contract executions. Naturally, the validator must be able to check that the proposed schedule really is serializable.

This paper makes the following contributions.

- A way for miners to speculatively execute smart contracts in parallel. We adapt techniques from *transactional boosting* [11] to permit non-conflicting smart contracts to execute concurrently.
- A way for miners to capture the resulting parallel execution in the form of a *fork-join* [1] schedule to be executed by validators, deterministically, verifiably, and in parallel.
- A prototype implementation, built on the Java virtual machine and ScalaSTM [21]. An evaluation using smart contract examples drawn from the Solidity documentation yields an overall speedup of 1.33x for miners, and 1.69x for validators with three concurrent threads of execution.

```
1  contract  Ballot {
2    mapping(address => Voter) public voters ;
3    // more state   definitions
4    function vote(uint proposal) {
5      Voter sender = voters[msg.sender];
6      if (sender.voted)
7        throw;
8      sender.voted = true;
9      sender.vote = proposal;
10     proposals[proposal].voteCount += sender.weight;
11   }
12   // more operation   definitions
13 }
```

Figure 1: Part of the Ballot contract

2 BLOCKCHAINS AND SMART CONTRACTS

In Bitcoin and similar systems, transactions typically have a simple structure, distributing the balances from a set of input accounts to a set of newly-created output accounts. In Blockchains such as Ethereum, however, each block also includes an explicit *state* capturing the cumulative effect of transactions in prior blocks. A Transaction is expressed as executable code, often called a *smart contract*, that modifies that state. Ethereum blocks thus contain both transactions' smart contracts and the final state produced by executing those contacts.

The contracts themselves are stored in the blockchain as byte-code instructions for the Ethereum virtual machine (EVM). Several higher-level languages exist for writing smart contracts. Here, we describe smart contracts as expressed in the Solidity language [22].

Listing 1 is part of the source code for an example smart contract that implements a ballot box [23]. The owner initializes the contract with a list of proposals and gives the right to vote to a set of Ethereum addresses. Voters cast their votes for a particular proposal, which they may do only once. Alternatively, voters may delegate their vote. The contract keyword declares the smart contract (Line 1).

The contract's persistent state is recorded in *state variables*. For Ballot , the persistent state includes fields of scalar type such as the owner (omitted for lack of space). State variables such as voters (declared on Line 2) can also use the built-in Solidity type mapping which, in this case, associates each voter's address with a Voter data structure (declaration omitted for brevity). The keys in this mapping are of built-in type address, which uniquely identifies Ethereum accounts (clients or other contracts). These state variables are the persistent state of the contract.

Line 4 declares contract *function*, vote, to cast a vote for the given proposal. Within a function there are transient *memory* and *stack* areas such as sender. The function vote first recovers the Voter data from the contract's state by indexing into the voters mapping using the sender's address msg.sender. The msg variable is a global variable containing data about the contract's current invocation. Next, the sender.vote flag is checked to prevent multiple votes. Note that sequential execution is critical: if this code were naïvely

run in parallel, it would be vulnerable to a race condition permitting double voting. Ethereum contract functions can be aborted at any time via **throw**, as seen here when a voter is detected attempting to vote twice. The **throw** statement causes the contract's transient state and tentative storage changes to be discarded. Finally, this Ballot contract also provides functions to register voters, delegate one's vote, and compute the winning proposal. The complete Ballot example is available elsewhere[3].

Execution Model: Miners and Validators.   When a miner prepares a block for inclusion in the blockchain, it starts with the ledger state as of the chain's most recent block. The miner selects a sequence of new transactions, records them in the new block, and executes them, one at a time, to compute the new block's state. The miner then participates in a consensus protocol to decide whether this new block will be appended to the blockchain.

To ensure that each transaction terminates in a reasonable number of steps, each call to contract bytecode comes with an explicit limit on the number of virtual machine steps that a call can take. (In Ethereum, these steps are measured in "gas" and clients pay a fee to the miner that successfully appends that transaction's block to the blockchain.)

After a block has been successfully appended to the blockchain, that block's transactions are sequentially re-executed *by every node in the network* to check that the block's state transition was computed honestly and correctly. (Smart contract transactions are deterministic, so each re-execution yields the same results as the original.) These *validator* nodes do not receive fees for re-execution.

To summarize, a transaction is executed in two contexts: once by miners before attempting to append a block to the blockchain, and many times afterward by validators checking that each block in the blockchain is honest. In both contexts, each block's transactions are executed sequentially in block order.

3   SPECULATIVE SMART CONTRACTS

This section discusses how miners can execute contract codes concurrently. Concurrency for validators is addressed in the next section.

Smart contract semantics is *sequential*: each miner has a single thread of control that executes one EVM instruction at a time. The miner executes each of the block's contracts in sequence. One contract can call another contract's functions, causing control to pass from the first contract code to the second, and back again. (Indeed, misuse of this control structure has been the source of well-known security breaches [6].) Clearly, even sequential smart contracts must be written with care, and introducing explicit concurrency to contract programming languages would only make the situation worse. We conclude that concurrent smart contract executions must be *serializable*: indistinguishable, except for execution time, from a sequential execution.

There are several obstacles to running contracts in parallel. First, smart contract codes read and modify shared storage, so it is essential to ensure that concurrent contract code executions do not result in inconsistent storage states. Second, smart contract languages

are Turing-complete, and therefore it is impossible in general to determine statically whether contracts have data conflicts.

We propose that miners execute contract codes as *speculative actions*. A miner schedules multiple concurrent contracts to run in parallel. Contracts' data structures are instrumented to detect synchronization conflicts at run-time, in much the same way as mechanisms like transactional boosting [11]. If one speculative contract execution conflicts with another, the conflict is resolved either by delaying one contract until the other completes, or by rolling back and restarting one of the conflicting executions. When a speculative action completes successfully, it is said to *commit*, and otherwise it *aborts*.

Storage Operations.   We assume that, as in Solidity, state variables are restricted to predefined types such as scalars, structures, enumerations, arrays, and mappings. A *storage operation* is a primitive operation on a state variable. For example, binding a key to a value in a mapping, or reading from a variable or an array are storage operations. Two storage operations *commute* if executing them in either order yields the same result values and the same storage state. For example, in the address-to-Voter Ballot mapping in Listing 1, binding Alice's address to a vote of 42 commutes with binding Bob's address to a vote of 17, but does not commute when deleting Alice's vote. An *inverse* for a storage operation is another operation that undoes its effects. For example, the inverse of assigning to a variable is restoring its prior value, and the inverse of adding a new key-value pair to a mapping is to remove that binding, and so on. The virtual machine system can provide all storage operations with inverses.

The virtual machine is in charge of managing concurrency for state variables such as mappings and arrays. Speculation is controlled by two run-time mechanisms, invisible to the programmer, and managed by the virtual machine: *abstract locks*, and *inverse logs*.

Each storage operation has an associated abstract lock. The rule for assigning abstract locks to operations is simple: if two storage operations map to distinct abstract locks, then they must commute. Before a thread can execute a storage operation, it must acquire the associated abstract lock. The thread is delayed while that lock is held by another thread[4]. Once the lock is acquired, the thread records an *inverse operation* in a log, and proceeds with the operation.

If the action commits, its abstract locks are released and its log is discarded. If the action aborts, the inverse log is replayed, most recent operation first, to undo the effects of that speculative action. When the replay is complete, the action's abstract locks are released.

The advantage of combining abstract locks with inverse logs is that the virtual machine can support very fine-grained concurrency. A more traditional implementation of speculative actions might associate locks with memory regions such as cache lines or pages, and keep track of old and versions of those regions for recovery. Such a coarse-grained approach could lead to many false conflicts, where operations that commute in a semantic sense are treated as conflicting because they access overlapping memory regions. In the next section, we will see how to use abstract locks to speed up verifiers.

---

[3]http://solidity.readthedocs.io/en/develop/solidity-by-example.html

[4]For ease of exposition, abstract locks are mutually exclusive, although it is not hard to accommodate shared and exclusive modes.

When one smart contract calls another, the run-time system creates a *nested* speculative action, which can commit or abort independently of its parent. A nested speculative action inherits the abstract locks held by its parent, and it creates its own inverse log. If the nested action commits, any abstract locks it acquired are passed to its parent, and its inverse log is appended to its parent's log. If the nested action aborts, its inverse log is replayed to undo its effects, and any abstract locks it acquired are released. Aborting a child action does not abort the parent, but a child action's effects become permanent only when the parent commits. The abstract locking mechanism also detects and resolves deadlocks, which are expected to be rare.

The scheme described here is *eager*, acquiring locks, applying operations, and recording inverses. An alternative *lazy* implementation could buffer changes to a contract's storage, applying them only on commit.

A miner's incentive to perform speculative concurrent execution is the possibility of increased throughput, and hence a competitive advantage against other miners. Of course, the miner undertakes a risk that synchronization conflicts among contracts will cause some contracts to be rolled back and re-executed, possibly delaying block construction, and forcing the miner to re-execute code not compensated by client fees. Nevertheless, the experimental results reported below suggest that even a small degree of concurrent speculative execution pays off, even in the face of moderate data conflicts.

4 CONCURRENT VALIDATION

The speculative techniques proposed above for miners are no help for validators. Here is the problem: miners use speculation to discover a concurrent schedule for a block's transactions, a schedule equivalent to some sequential schedule, except faster. That schedule is constructed non-deterministically, depending on the order in which threads acquired abstract locks. To check that the block's miner was honest, validators need to reconstruct the same (or an equivalent) schedule chosen by the miner.

Validators need a way to deterministically reproduce the miner's concurrent schedule. To this end, we extend abstract locks to track dependencies, that is, who passed which abstract locks to whom. Each speculative lock includes a *use counter* that keeps track of the number of times it has been released by a committing action during the construction of the current block. When a miner starts a block, it sets these counters to zero.

When a speculative action commits, it increments the counters for each of the locks it holds, and then it registers a *lock profile* with the VM recording the abstract locks and their counter values.

When all the actions have committed, it is possible to reconstruct their common schedule by comparing their lock profiles. For example, consider three committed speculative actions, A, B, and C. If A and B have no abstract locks in common, they can run concurrently. If an abstract lock has counter value 1 in A's profile and 2 in C's profile, then C must be scheduled after A.

A miner includes these profiles in the blockchain along with usual information. From this profile information, validators can construct a *fork-join* program that deterministically reproduces the

---

Algorithm 1 MINEINPARALLEL(T) - Mine in parallel

Require: A set of contract transactions T
Ensure: A serial order S of transactions and a happens-before
    graph H of the locking schedule
1: **function** MINEINPARALLEL(B)
2:    Initialize log L for recording locking operations
3:    Execute all transactions $t \in T$ in parallel, recording
        locking activity in L
4:    Generate happens-before graph H from L
5:    Create the serial ordering S via a topological sort of H
6:    **return** (S, H)
7: **end function**

Algorithm 2 CONSTRUCTVALIDATOR(S, H) - Construct a parallel validator

---

Require: The serial ordering S and happens-before graph H from
    the miner
Ensure: A set of fork-join tasks ensuring parallel execution ac-
    cording to the happens-before graph
1: **function** CONSTRUCTVALIDATOR(B)
2:    Initialize a mapping F from each transaction t to its
        fork-join task f
3:    Create the happens-after graph H' by reversing the
        edges of H
4:    **for all** $t \in S$ **do**
5:        $B \leftarrow$ all transactions $u \in H'$ that happen immediately
            before t, i.e., its outedges
6:        Create a fork-join task f for t that first joins with all
            tasks in B, i.e.,

$$f \leftarrow \textbf{for } (b \text{ in } B) \ \{ \ F.\text{get}(b).\text{join}() \ \} \ \text{execute}(t)$$

7:        Save the new fork-join task in F, i.e., $F.\text{put}(t, f)$
8:    **end for**
9:    **return** the value set of F, the fork-join tasks
10: **end function**

---

miner's original, speculative schedule. Algorithm 1 provides a high-level sketch of the operation of the miner. By logging the locking schedule during parallel execution, the miner generates a happens-before graph of transactions according to the order in which they acquire locks and commit. A valid serial history is produced from a topological sort of this graph. Algorithm 2 constructs the validator by scanning through the list of actions as they appear in the serial history. A fork-join task is created for each action and stored for lookup by its identifier. Each task will first lookup and join any tasks that must precede it according to the locking schedule before executing the action itself.

The resulting fork-join program is not speculative, nor does it require inter-thread synchronization other than forks and joins. The validator is not required to match the miner's level of parallelism: using a work-stealing scheduler [1], the validator can exploit whatever degree of parallelism it has available. The validator does not need abstract locks, dynamic conflict detection, or the ability to roll back speculative actions, because the fork-join structure ensures that conflicting actions never execute concurrently.

To check that the miner's proposed schedule is correct, the validator's virtual machine records a trace of the abstract locks each thread would have acquired, had it been executing speculatively. This trace is thread-local, requiring no expensive inter-thread synchronization. At the end of the execution, the validator's VM compares the traces it generated with the lock profiles provided by the miner. If they differ, the block is rejected.

What is a miner's incentive to publish a block's fork-join schedule? A miner who publishes an incorrect schedule will be detected and its block rejected, but a miner might publish a correct sequential schedule equivalent to, but less parallel than the schedule it discovered. Such a tactic seems ill-advised, however, because that block may be competing with other blocks produced at the same time, and the miner will be rewarded only if the other miners choose to build on that block. Publishing a block with a parallel valiation schedule makes the block more attractive for validation by other miners. Because fork-join schedules are published in the blockchain, their degree of parallelism is easily evaluated.

5 CORRECTNESS

Concurrent calls to smart contract functions might leave persistent storage in an inconsistent state not possible after a serial execution. Instead, we must show that every concurrent execution permitted by our proposal is equivalent to some sequential execution. Because miners are free to choose the order in which contracts appear in a block, any sequential execution will do.

Our argument builds on prior proofs that transactional boosting is serializable [11, 16, 17]. A given execution of a contract's function involves a sequence of storage operations on persistent objects. (The Ethereum gas restriction ensures this sequence is finite.) Recall that if two storage operations map to distinct abstract locks, then they commute. If another thread executes another sequence of operations, and there are two operations that do not commute, then both threads will try to acquire the same lock, and one will be delayed until the other completes. (Deadlocks are detected and resolved by aborting one execution.) As proved elsewhere [11, 16, 17], the result is a serializable execution[5].

We cannot guarantee that the schedule published by the miner is the same one that it executed, but we can guarantee the two are equivalent to a common sequential history. Validators replay the concurrent schedule published by the miner, and will detect if the schedule produces a final state different from the one recorded in the block, or if the schedule has a data race (an unsynchronized concurrent access).

6 IMPLEMENTATION

Because the EVM is not multithreaded, our prototype uses the Java Virtual Machine (JVM). Speculative actions are executed by the Scala Software Transactional Memory Library (ScalaSTM [21]).

Examples of smart contracts were translated from Solidity into Scala, then modified to use the concurrency libraries. Each function from the Solidity contract is turned into a speculative transaction by wrapping its contents with a ScalaSTM atomic section. Solidity mapping objects are implemented as boosted hashtables, where

key values are used to index abstract locks. Additionally, solidity struct types were translated into immutable case classes. Methods take a msg field to emulate Solidity contracts' global state, which includes details of the transaction, addresses of participants, and so on. Scalar fields are implemented as a single a boosted mapping.

The Solidity **throw** operation, which explicitly rolls back a contract execution, is emulated by throwing a Java runtime exception caught by the miner.

In our prototype, abstract locks are implemented via interfaces exported by ScalaSTM, relying on ScalaSTM's native deadlock detection and resolution mechanisms.

6.1 Miners and Validators

Miners manage concurrency using Java's ExecutorService. This class provides a pool of threads and runs a collection of callable objects in parallel. A block of transactions in Ethereum is implemented as a set of callable objects passed to the thread pool. To generate locking profiles from the parallel execution, we instrument smart contracts to log when atomic sections start and end, as well as calls to boosted operations. From the log, we can encode the locking schedule as a happens-before graph for the validator. The validator transforms this happens-before graph into a fork-join program. Each transaction from the block is a fork-join task that first joins with all tasks according to its in-edges on the happens-before graph.

7 EXPERIMENTAL EVALUATION

Our goal is to improve throughput for miners and validators by allowing unrelated contracts to execute in parallel. To evaluate this approach, we created a series of benchmarks for sample contracts that vary the number of transactions and their degree of conflict. These benchmarks are conservative, operating on only one or a few contracts at a time and permitting higher degrees of data conflict than one would expect in practice.

Our experiments are designed to answer two questions. (1) For a given amount of data conflict, how does speedup change over increasing transactions? We expect to see more speedup as the number of transactions increases, limited by the number of cores available on the underlying hardware. (2) How does the speedup change as data conflict increases? For low data conflict, we expect our parallel miner to perform better than serial. But as data conflict increases, we expect a drop-off in speedup, limited by core availability.

7.1 Benchmarks

There are four benchmarks, one for each of the example contracts we implemented, Ballot, SimpleAuction, and EtherDoc, as well as the Mixed benchmark containing transactions from all other contracts. For each benchmark, our implementation is evaluated on blocks containing between 10 and 400 transactions with 15% data conflict, as well as blocks containing 200 transactions with data conflict percentages ranging from 0% to 100% data conflict. The data conflict percentage is defined to be the percentage of transactions that contend with at least one other transaction for shared data. As we will see, the impact of data conflict on speedup depends on the contract implementation.

---

[5] Because speculative executions take place entirely within a virtual machine, opacity [10] is not an issue.

These benchmarks are conservative. For all benchmarks besides Mixed, the entire block operates on the same contract, calling only one or two methods. In reality, mined blocks contained transactions on unrelated contracts and accounts. While the theoretical maximum number of transactions per block is currently around 200 transactions[6], we test a wide range from 10 to 400. This maximum increases and decreases over time, as determined by miner preference [9]. In practice, the number of transactions can be far fewer per block, e.g., when there are costly transactions. For testing speedup over number of transactions, we fix the data conflict rate at 15%, though we expect that blocks in practice rarely have very much internal data conflict. While we did not measure data conflict in the existing blockchain, our approach implemented in EVM could be used to collect such data on an existing blockchain. For testing speedup as data conflict increases, we fix the number of transactions per block to 200, the current theoretical maximum.

Ballot. This contract is an example voting application from the Solidity documentation [23] and is described in Section 2. For all benchmarks, the contract is put into an initial state where voters are already registered. All block transactions for this benchmark are requests to vote on the same proposal. To add data conflict, some voters attempt to double-vote, creating two transactions that contend for the same voter data. 100% data conflict occurs when all voters attempt to vote twice.

SimpleAuction. This contract, also from the Solidity documentation [23] implements an auction. There is a single owner who initiates the auction, while any participant can place bids with the `bid()` method. A mapping tracks how much money needs to be returned to which bidder once the auction is over. Bidders can then `withdraw()` their money. For the benchmarks, the contract state is initialized by several bidders entering a bid. The block consists of transactions that withdraw these bids. Data conflict is added by including new bidders who call `bidPlusOne()` to read and increase the highest bid. The rate of data conflict depends on how many bidders are bidding at the same time, thus accessing the same highest bidder. 100% data conflict happens when all transactions are `bidPlusOne()` bids.

EtherDoc. EtherDoc[7] is a "Proof of Existence" decentralized application (DAPP) that tracks per-document metadata including hashcode and owner. It permits new document creation, metadata retrieval, and ownership transfer. For the benchmarks, the contract is initialized with a number of documents and owners. Transactions consist of owners checking the existence of the document by hashcode. Data conflict is added by including transactions that transfer ownership to the contract creator. As with SimpleAuction, all contending transactions touch the same shared data, so we expect a faster drop-off in speedup with increased data conflict than Ballot. 100% data conflict happens when all transactions are transfers.

Mixed. This benchmark combines transactions on the above smart contracts in equal proportions, and data conflict is added the same way in equal proportions from their corresponding benchmarks.

---

[6]A transaction costs 21,000 gas plus the gas for the computation [25]. The gas limit on block 3,110,235 (latest as of writing) was 4,005,875, a maximum close to 200.
[7]https://github.com/maran/notareth

7.2 Results

We ran our experiments on a 4-core 3.07GHz Intel Xeon W3550 with 12 GB of memory running Ubuntu 16. All of our experiments run on the Java Virtual Machine (JVM) with JIT compilation disabled. Parallel mining and validation are run with a fixed pool of three threads, leaving one core available for garbage collection and other system processes/threads.

For each benchmark, blocks were generated for each combination of the number of transactions and data conflict percentage. Each block is run on the parallel miner, the validator, and a serial miner that runs the block without parallelization. The serial results serve as the baseline for computing speedup. The running time is collected five times and the mean and standard deviation are measured. All runs are given three warm-up runs per collection.

Figure 2 shows the speedup of the parallel miner and validator relative to the serial miner for all. (The running times with mean and standard deviation can be found in Appendix A.) The left charts plot the speedup over the number of transactions in the block at a fixed data conflict percentage of 15%. The speedup for all benchmarks follows roughly the same pattern. For low numbers of transactions, there is no speedup and even some slowdown. This is likely due to data conflict as well as the overhead of multithreading. For over around 50 transactions, there is a speedup that increases to about 2x, in line with expectations from a thread pool of size three. EtherDoc is an exception, seeing less than 1.5x speedup. The validator generally has a higher speedup than the parallel miner. This is because the parallel miner has done the hard work of finding data conflict and produced a locking schedule for the validator to follow.

The right-hand charts of Figure 2 plot the speedup as the data conflict percentage increases for fixed blocks of 200 transactions. As data conflict increases, the miner's speedup reduces from 2x to close to serial as many transactions touch shared data. The validator also starts at around 2x with no data conflict, but goes down to about 1.5x, again benefiting from the work of the parallel miner.

Ballot's parallel mining hovers around 1.5x speedup, suffering little from the extra data conflict. Data conflict in SimpleAuction and EtherDoc, however, has an expectedly higher impact, because each contending transaction touches the same data. The Mixed benchmark provides a more realistic view of a block by combining transactions from unrelated contracts. Even though EtherDoc reduces parallelism under high data conflict, when mixed with other transactions, the parallel miner can still gain a substantial speedup.

The average of speedups of all benchmarks is 1.33x for the parallel miner and 1.69x for the validator. Table 1 shows the average speedups for each benchmark.

7.3 Discussion

These results show that speculative concurrent execution speeds up mining when threads are occupied and the data conflict rate is not too high. Data conflicts among transactions in the same block is likely to be infrequent over the long term. (Miners could also choose transactions so as to reduce the likelihood of conflict, say by including only those contracts that operate on disjoint data sets.) Due to limited hardware, our experiments used only three concurrent threads, but even this modest level of concurrency showed a
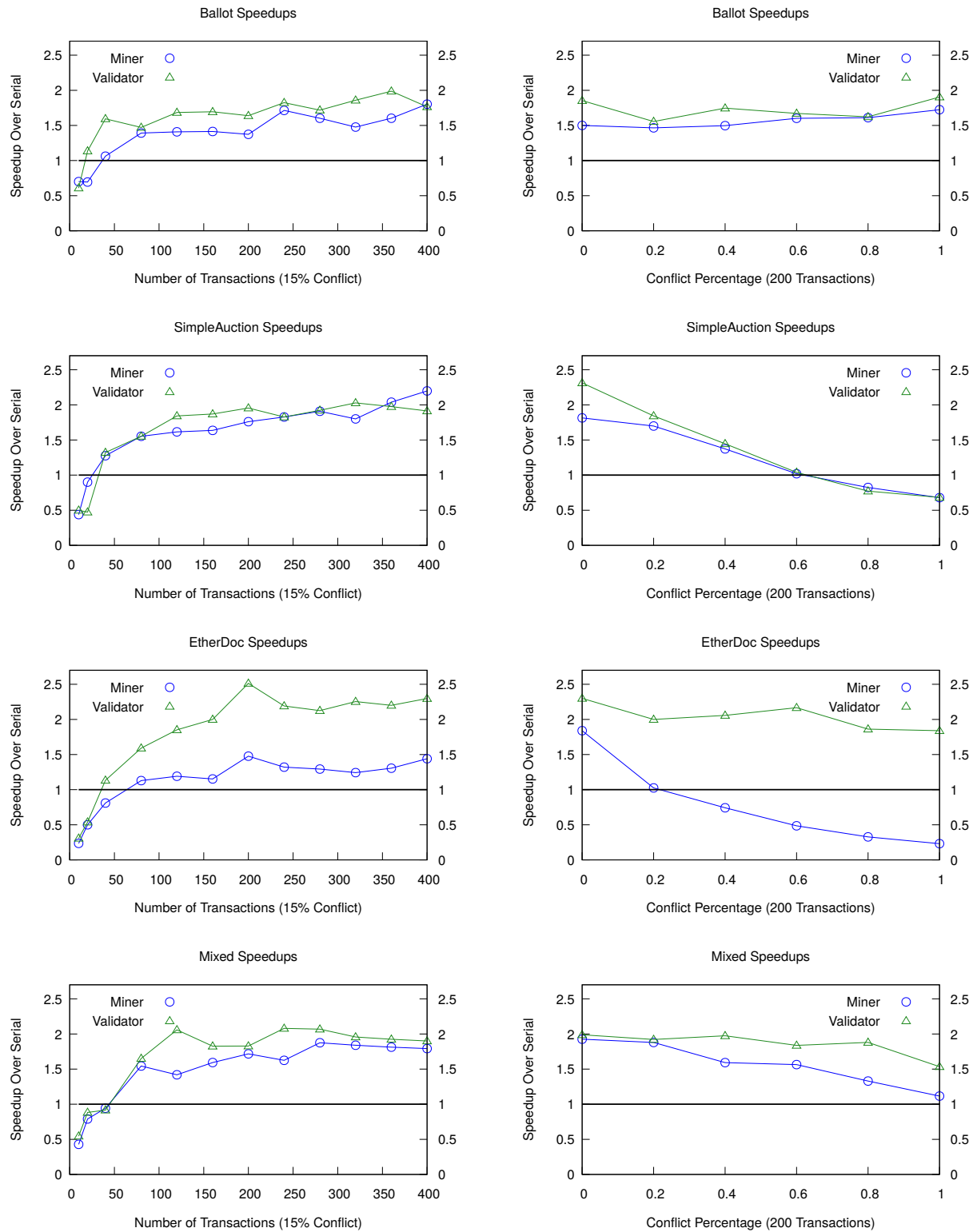
Figure 2: **The speedup of the miner and validator versus serial mining for each benchmark. The left charts are the speedup as block size increases, while the right are the speedup as data conflict increases.**

	SimpleAuction		Ballot		EtherDoc		Mixed	
	Conflict	BlockSize	Conflict	BlockSize	Conflict	BlockSize	Conflict	BlockSize
Miner	1.23x	1.58x	1.57x	1.35x	0.78x	1.09x	1.57x	1.45x
Validator	1.35x	1.60x	1.73x	1.58x	2.04x	1.75x	1.86x	1.64x

Table 1: The average speedups for each benchmark.

benefit. Concurrent hardware has proved effective for speeding up solutions to proof-of-work puzzles, and now similar investments could speed up smart contract execution and validation.

Concurrent smart contract execution speeds up miners by enabling them to construct blocks faster before appending them to the chain. But in permissionless blockchains, the bulk of miner time is spent computing proof-of-work for the block after construction. Concurrent smart contracts, however, still provide a big win for validators. Validators spend much time executing transactions, while validating the proof-of-work by miners is fast. Ethereum is transitioning to proof-of-stake to reduce the computational burden imposed by proof-of-work. Additionally, permissioned blockchains eschew proof-of-work, so concurrent smart contract execution provides even more of a boost to throughput to these blockchains.

8 RELATED WORK

The notion of smart contracts can be traced back to an article by Nick Szabo in 1997 [24]. Bitcoin [20] includes a scripting language whose expressive power was limited to protect against non-terminating scripts. Ethereum [8] is perhaps the most widely used smart contract platform, employing a combination of a Turing-complete virtual machine protected from non-termination by charging clients for contract running times. Solidity [22] is the most popular programming language for programming the Ethereum virtual machine.

Luu *et al.* [18] identify a number of security vulnerabilities and pitfalls in the Ethereum smart contract model. Luu *et al.* [19] also identify perverse incentives that cause rational miners sometimes to accept unvalidated blocks. Delmolino *et al.* [7] document common programming errors observed in smart contracts. The Hawk [15] smart contract system is designed to protect the privacy of participants.

As noted, many of the speculative mechanisms introduced here were adapted from *transactional boosting* [11], a technique for transforming thread-safe linearizable objects into highly-concurrent transactional objects. Boosting was originally developed to enhance the concurrency provided by software transactional memory systems [12] by exploiting type-specific information. Other techniques that exploit type-specific properties to enhance concurrency in STMs include *transactional predication* [3] and *software transactional objects* [13].

There are other techniques for deterministically reproducing a prior concurrent execution. See Bocchino *et al.* [2] for a survey.

Cachin et al. discuss non-deterministic execution of smart contracts in the context of BFT-based permissioned blockchains [4].

9 CONCLUSION

We have shown that one can exploit multi-core architectures to increase smart contract processing throughput for both miners and validators. First, miners execute a block's contracts speculatively and in parallel, resulting in lower latency whenever the block's contracts lack data conflicts. Miners are incentivized to include in each block an encoding of the serializable parallel schedule that produced that block. Validators convert that schedule into a deterministic, parallel fork-join program that allows them to validate the block in parallel. Even with only three threads, a prototype implementation yields overall speedups of 1.33x for miners and 1.69x for validators on representative smart contracts.

Although our discussion has focused on "permisionless" systems where anyone can participate, the mechanisms proposed here would also be useful for "permissioned" systems, such as Hyperledger [14], where participants are controlled by an authority such as an organization or consortium. For example, in a permissioned blockchain based on Practical Byzantine Fault-Tolerance (PBF) [5], the leader might use speculative execution to discover a concurrent schedule for a block, while particpants in the PBFT protocol would use the concurrent schedule to validate the block before voting.

Future work includes adding support for multithreading to the Ethereum virtual machine, in much the same way as today's Java virtual machines. Our proposal for miners only is compatible with current smart contract systems such as Ethereum, but our overall proposal is not, because it requires including scheduling metadata in blocks and incentivizing miners to publish their parallel schedules. It may well be compatible with a future "soft fork" (backward compatible change), a subject for future research.

In addition to a multithreaded VM, we see room for advancement in programming language support for smart contracts. Designing a language that lends itself to finer-grained concurrency will increase the success of speculative execution thereby increasing throughput. It would also be useful for the language to provide better control of concurrency, helping the smart contract developer maximize throughput while avoiding concurrency pitfalls.

REFERENCES

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[2] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

[3] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. Transactional predication: High-performance concurrent sets and maps for stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 6–15, New York, NY, USA, 2010. ACM.

[4] C. Cachin, S. Schubert, and M. Vukolic. Non-Determinism in Byzantine Fault-Tolerant Replication. In P. Fatourou, E. Jiménez, and F. Pedone, editors, *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, volume 70 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[5] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[6] DAO. Thedao smart contract. Retrieved 8 February 2017.

[7] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi. *Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab*, pages 79–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[8] Ethereum. https://github.com/ethereum/.

[9] Ethereum design Rationale. http://github.com/ethereum/wiki/wiki/Design-Rationale#gas-and-fees.

[10] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP'08)*, pages 175–184, New York, NY, USA, 2008. ACM.

[11] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM.

[12] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[13] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 31:1–31:16, New York, NY, USA, 2016. ACM.

[14] Hyperledger white paper. http://www.the-blockchain.com/docs/Hyperledger%20Whitepaper.pdf.

[15] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy*, 2015.

[16] E. Koskinen and M. J. Parkinson. The push/pull model of transactions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15), Portland, OR, USA*. ACM, 2015.

[17] E. Koskinen, M. J. Parkinson, and M. Herlihy. Coarse-grained transactions. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*, pages 19–30. ACM, 2010.

[18] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 254–269, 2016.

[19] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena. Demystifying incentives in the consensus computer. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 706–719, New York, NY, USA, 2015. ACM.

[20] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009.

[21] Scala STM Expert Group. Scalastm. web. Retrieved from http://nbronson.github.com/scala-stm/, 20 November 2011.

[22] Solidity documentation. http://solidity.readthedocs.io/en/latest/index.html.

[23] Solidity documentation: Solidity by example. http://solidity.readthedocs.io/en/develop/solidity-by-example.html.

[24] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[25] G. Wood. Ethereum: A secure decentralised generalised transaction ledger.

A   MEAN AND STANDARD DEVIATION OF BENCHMARK RUNNING TIMES