Parallel Block Execution in SoCC Blockchains through Optimistic Concurrency Control

Alex Valtchanov*
Princeton University
Princeton, NJ, USA
alexvaltchanov@princeton.edu

Lauren Helbling*
Montana State University
Bozeman, MT, USA
lauren.helbling@student.montana.edu

Batuhan Mekiker
Montana State University
Bozeman, MT, USA
batuhan.mekiker@montana.edu

Mike P. Wittie
Montana State University
Bozeman, MT, USA
mike.wittie@montana.edu

***Abstract*—The blockchain ecosystem is growing rapidly with new decentralized applications being released constantly, boasting significant growth in functionality and adoption. With accelerating adoption rate, the load on all the existing blockchain networks grows as well, causing increased transaction delays. We seek to improve the performance of blockchain solutions by increasing transaction throughput. This paper presents a novel approach for executing blocks in parallel using optimistic concurrency control and conflict rescheduling. Specifically, we show that our approach improves the performance of Separation of Consensus and Compute (SoCC) blockchains with the potential to reduce smart contract transaction wait times.**

***Index Terms*—blockchain, parallel execution, optimistic concurrency, scheduling, dependency graph**

I. Introduction

The popularity and variety of blockchains is growing rapidly. In fact, the number of digital wallets in all blockchains combined is more than 70 million as of May 2021 and growing [1]. On Ethereum alone, daily transactions reached their peak at approximately 1.7 million transactions per day in May 2021 – an increase of more than 20% from the previous year and a 350% increase from the last 5 years [2]. With the growing volume of transactions, users experience either longer delays or failed transactions. Between April 15th and 21st of 2021, 1.1 million transactions were issued in UniSwap, one of the leading Decentralized Finance (DeFi) applications on Ethereum [3]. Out of the 1.1 mil, roughly 22% failed due to insufficient resources (gas) as users had to compete with one another for transaction inclusion in a block. As blockchain scales with a growing volume of incoming transactions, cases like UniSwap will become more common with blockchain throughput being unable to keep up. Therefore, we believe the future of blockchain depends on the development of higher blockchain throughput.

Existing solutions propose different approaches to increase transaction throughput, one of which is the concurrent execution of transactions within blocks. In that vein, Dickerson et al. propose locking mechanisms, however, locks are not a user-friendly way to handle transaction conflicts due to their pessimistic nature and their introduction of additional delays [4]. Bartoletti et al. explore concurrent transaction execution by introducing static analysis of smart contracts [5]. Yet, static analysis requires pre-execution which reduces the benefits of parallelism. Anjana et al. and Jin et al. leverage optimistic concurrency, dependency graphs, and state rollbacks [6], [7]. However, where Anjana et al. introduce higher complexity with fork-joins, the work of Jin et al. is limited to permissioned blockchains.

Most blockchain models do not distribute network resources efficiently; every node in the network participates in each aspect of finalizing a block, from reaching a consensus to verifying the final state. Flow, a new public blockchain, proposes an approach to increase transaction throughput by addressing this network-level limitation [8]–[10]. In Flow, the selection and ordering of transactions runs independently from their execution by dividing the roles of the network nodes. Transactions are gathered by Collector nodes, ordered and placed into blocks by Consensus nodes, executed by Execution nodes, and verified by Verification nodes. This division of roles allows for multiple blocks to be created nearly simultaneously and executed lazily. The Flow model increases blockchain throughput by eliminating transaction execution delay from the distributed consensus process. Unfortunately, Flow throughput can be limited if the execution cannot keep up. Hereafter, we refer to blockchains that separate consensus from computation, such as Flow, as Separation of Consensus and Compute (SoCC) model blockchains.

We observe an opportunity for parallelism in the SoCC blockchain model that can further increase thrpughput and propose a protocol for parallel execution of blocks using optimistic concurrency control and conflict rescheduling. Transactions are initially placed into blocks optimistically, in the order given by Consensus nodes. These blocks are then executed in parallel and conflicting transactions are dynamically discovered at run-time, creating a *dependency graph* that captures all of the conflicting transactions and the relations among them. Conflicted transactions are *suspended* and put back into the awaiting transaction pool with the new dependency knowledge. Once discovered, we schedule suspended transactions while preserving the order imposed by the consensus mechanism. With already known conflicts, our method guarantees that each transaction is scheduled and executed at most twice.

This paper offers the following contributions:

1) We develop a mechanism that optimistically executes

∗ Authors contributed equally

blocks in parallel, capturing conflicting transactions in a suspended transaction dependency graph (STDG).

2) We design an efficient scheduling algorithm for conflicted transactions using the STDG to ensure that transactions will be executed at most twice.
3) We compare the performance of parallel blockchain execution to the standard Flow execution model and show a 3.5x speedup on average. We also demonstrate the limited overhead of storing and maintaining the STDG.

Our evaluation demonstrates that executing blocks in parallel with optimistic concurrency control and rescheduling conflicting transactions increases transaction throughput, leading to better performance with minimal changes to the existing blockchain protocol. Our solution could allow for applications to build a better user experience with higher throughput and for throughput heavy applications to explore blockchain solutions.

The rest of the paper is organized as follows. In Section II we provide background on the standard and SoCC blockchain execution model. Section III introduces and compares existing research related to concurrent execution in blockchains. Section IV details our conflict discovery algorithm and scheduler. In Section V we present our evaluation and measurements. Finally, in Section VI we conclude and discuss future work.

II. Background

Blockchain is a decentralized distributed ledger comprised of a network of mutually distrusting nodes. Each block on the blockchain contains a tamper-proof sequence of transactions. The blockchain data structure forms the foundation of modern cryptocurrencies but extends to other applications in which a distributed consensus is needed, such as voting mechanisms. Some blockchains follow an UTXO-based model, such as Bitcoin, in which there are no accounts or wallets maintained, but rather a continual exchange of unspent transactions. On the other hand, account-based blockchains, such as Ethereum, maintain balances and states within accounts. The SoCC model is exclusive to account-based blockchains, where the accounts and states are updated by smart contracts.

In a typical blockchain, a node creates a new block and execute its transactions sequentially. A consensus protocol is then used to determine which new block will be placed on the blockchain. During the verification process, every node serially re-executes every transaction in the new block to validate their execution and the correctness of new account states. This execution model is slow because network dissemination of full blocks takes time and requiring every node to execute every block before forwarding it to other nodes increases dissemination time. The result is long inter-block intervals and decreased throughput in fixed size blocks.

Dapper Labs identified these inefficiencies and considered a novel approach by creating Flow, a new blockchain that divides the work of the network nodes into four roles with sub-tasks: Collector, Consensus, Execution, and Verification [8]–[10]. Collector nodes are responsible for gathering incoming transactions and batching them into collections, which are then transmitted to Consensus nodes. The Consensus nodes agree upon the order of transactions in new blocks but do so on small blocks that only store transaction hashes, which speeds up block dissemination. The resulting blocks are finalized and immutable. Execution nodes, with adequate processing resources, then execute the finalized blocks, while Verification distributes the work of checking execution correctness to attest updates to account states. The division of node roles pipelines block processing while efficiently using network resources among the Consensus nodes and relying on the greater processing power of the Execution nodes, thereby increasing transaction throughput as long as Execution nodes can keep up with Consensus nodes [8].

Since the proposal of blocks occurs independently from block execution in the SoCC blockchain model, multiple blocks may be finalized on the blockchain and not yet executed. In Flow, such blocks would be executed sequentially. We propose that Execution nodes execute multiple blocks in parallel, allowing for higher transaction throughput by speeding up execution.

III. Related Work

Existing research explores blockchain speed-up with novel concurrency protocols for smart contracts. Saraph et al. explore a variety of speculative smart contract concurrency techniques by rerunning historical Ethereum transactions, concluding that parallelism in Ethereum smart contract execution is both possible and beneficial [11]. The strategies follow two phases: first, optimistically executing transactions in parallel, then discarding conflicting transactions and rerunning them sequentially. This model improves transaction throughput when there are few conflicting transactions and many transactions can be executed in parallel. However, their two-phase technique could produce negative speed-up if too many transactions are being rolled back. Regardless, the exploratory study motivated further work on parallel transaction execution.

Reijsbergen et al. study the conflict rates of real transaction data from Bitcoin and Ethereum blockchains and the potential of concurrency to speed-up execution time [12]. They argue that group concurrency, such as executing blocks in parallel, offers more potential speed-up than single-transaction concurrency.

Dickerson et al. first proposed a method of executing transactions within a block concurrently using locking and inverse logs [4]. Miners speculatively execute transactions in parallel and develop a happens-before graph of transaction dependencies, which is provided to the validators. With this knowledge, validators can execute transactions in parallel without conflict. In this way, the performance of both execution and verification improves with parallelism. Since there is not a division of roles, a fork-join approach must be used to allow for concurrent execution. While functional, by the nature of pessimistic concurrency control, this approach exhausts computational power and time by anticipating conflicts with locking for every transaction.

Bartoletti et al. expand upon Dickerson's idea, but argue for static analysis of smart contracts rather than locking [5]. In their protocol, transactions are safely swapped prior to execution in order to parallelize without conflicts. This paper assumes that

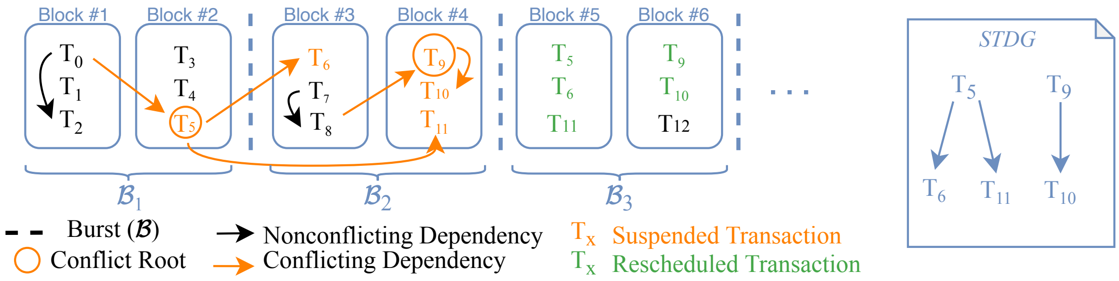

Fig. 1: Protocol overview. Each block within a burst is executed in parallel and dependencies are learned at execution time. Conflict roots such as T_5 and T_9 initiate a cascading series of conflicted transactions which must be suspended. The Suspended Transaction Dependency Graph contains suspended transactions which are later rescheduled according to their dependency structure.

the analysis of read/write keys to detect the swap-ability of transactions is possible but does not propose a specific method of doing so with real smart contracts.

Other protocols [6], [7] eliminate locking concurrency control by optimistically executing and developing dependency graphs to roll back transactions when needed. Similar to Saraph et al. rolled back transactions are then executed sequentially [11].

Jin et al. propose a protocol for dividing the roles of proposal and validation [7]. Their work is specifically for permissioned blockchain, since it does not rely on a consensus algorithm such as proof of work, and aims to run both execution and validation in parallel. Instead, our protocol applies to all public SoCC blockchains, dividing nodes efficiently using the Flow framework to optimize execution.

Our protocol is unique in executing entire blocks in parallel rather than individual transactions within a block. We also guarantee that every transaction will be executed in a parallel block while being scheduled at most twice.

IV. Parallel Execution Paradigm

To achieve parallel execution of blocks, we extend the SoCC model by having Consensus nodes finalize an ordered set of blocks all at once, rather than one block at a time. We refer to such a set as a *burst*; this set of blocks maintains a well-defined ordering, thereby preserving the agreed-upon order of transactions. Each burst $\mathcal{B}_t$ defines a time-step t; blocks within a burst are executed in parallel on different Execution nodes while each burst is executed sequentially. In this model of computation, the transactions within each block are still executed sequentially though on different Execution nodes. Thus our approach is orthogonal to solutions that execute transactions in parallel within a block [4], [12]

When introducing parallelism to transaction execution, the notion of *dependencies* between transactions is critical in maintaining a consistent computational state. Contrary to parallel execution in conventional blockchains, our mechanism must keep track of dependencies across blocks since blocks themselves are executed in parallel. Moreover, we must also keep track of dependencies across bursts because blocks are finalized in advance of their execution within SoCC blockchains. We interpret dependencies in the form of a *happens-before* graph, where transactions are vertices and a directed edge from T_a to T_b indicates that T_b depends on T_a (T_b either reads or writes to accounts written to by T_a) and must be executed after T_a.

We say that a transaction is *conflicted* if (a): it depends on at least one other transaction that is executed in parallel, or (b): it depends on at least one conflicted transaction. Transactions of the type (a) are referred to as *conflict roots*.

We propose that Consensus nodes schedule transactions optimistically and discover conflicted transactions using the dependency structure learned by the Execution nodes. Then, Consensus nodes *re-add* any conflicted transactions on newly finalized bursts using the discovered dependencies in a way that avoids conflicts among re-added transactions. This approach, outlined in Figure 1, allows our mechanism to guarantee every transaction will be executed at most twice; transactions are initially executed in parallel optimistically, while any conflicted transaction will be re-executed under a conflict-free schedule. The Execution nodes execute transactions within a burst based on the state of the blockchain disseminated among the Execution nodes at the end of the previous burst.

A. Suspended Transaction Dependency Graph

A transaction which is known to be conflicted but has yet to be re-executed is referred to as *suspended*. We must keep track of suspended transactions and their dependencies so that any dependent transactions are accounted for (i.e. discovered to be conflicted). Furthermore, the dependencies of suspended transactions can be used to schedule conflicted transactions a second time without conflict. A transaction is no longer suspended once it has been re-executed.

We define the *Suspended Transaction Dependency Graph* (STDG), notated $\mathcal{G}_t$, as the happens-before graph of all transactions which are currently suspended at time-step t. This graph captures the exact set of conflicted transactions that need to be re-added to the chain while maintaining their dependency structure. To update the STDG, all dependencies learned from executing transactions in a burst are temporarily added to the STDG, while any re-executed transactions and their associated dependencies are removed from the STDG. Then, a second pass is performed through the STDG to remove any dependencies which do not contribute to transaction conflicts. Algorithm 1 details the process of updating the STDG at each time-step.

Procedure UpdateDependencyGraph takes the STDG of the previous time-step $\mathcal{G}_{t-1}$ and the current burst $\mathcal{B}_t$ as input. UpdateDependencyGraph relies on the Execute, Prune, NewConflictRoots, and TraverseConflicts sub-procedures, and returns the updated STDG $\mathcal{G}_t$.

The Execute sub-procedure returns a happens-before graph $\mathcal{G}_E$ which contains newly discovered dependencies within and between the current burst $\mathcal{B}_t$ and the currently suspended transactions $V(\mathcal{G}_{t-1})$. Next, the Prune sub-procedure removes any re-executed transactions in $\mathcal{B}_t$ from $\mathcal{G}_{t-1}$ and returns $\mathcal{G}_P$ as the pruned graph; as long as the scheduling algorithm adheres to the learned dependencies, re-executed transactions cannot be conflicted again and can no longer contribute to any further conflicts.

The NewConflictRoots sub-procedure takes the newly discovered dependencies $\mathcal{G}_E$ and performs a Breadth-First-Search (BFS) traversal of $\mathcal{G}_E$ from the set of transactions in the current burst $V(\mathcal{B}_t)$. This process discovers any edges spanning different blocks in order to collect and return any conflict roots $\mathcal{C}$ in the current burst.

At this point, Algorithm 1 has collected new dependencies in $\mathcal{G}_E$ and pruned re-executed transactions from the old STDG in $\mathcal{G}_P$. Together, the combined graph $\mathcal{G}_E \cup \mathcal{G}_P$ contains both conflicting dependencies as well as nonconflicting dependencies. To remove the nonconflicting dependencies, the TraverseConflicts sub-procedure performs a BFS traversal of $\mathcal{G}_E \cup \mathcal{G}_P$ from both the currently suspended transactions $V(\mathcal{G}_P)$ and the new conflict roots $\mathcal{C}$; any vertex visited by this traversal is dependent on a conflict root or a suspended transaction and is therefore conflicted. The subgraph which is visited by this traversal is returned as the updated STDG $\mathcal{G}_t$.

B. Scheduling Suspended Transactions

With an STDG $\mathcal{G}_t$ given by the iterative application of Algorithm 1 at each time-step t, Consensus nodes are kept up to date on which transactions need to be re-added to the chain along with all dependencies among those transactions.

To reschedule suspended transactions without conflict, we first perform a BFS traversal of the STDG to obtain the weakly connected components. Because suspended transactions are pruned at the time of execution, rather than at the time of scheduling, the BFS traversal begins with any already scheduled transaction marked as visited. Each connected component represents a set of transactions whose dependencies are completely contained within those transactions–every transaction within a connected component must be executed in sequence. For each burst, we schedule each connected component on its own block, with the original ordering of transactions maintained, prioritized by the oldest transaction in each connected component. We continue to schedule connected components on their own block within a burst until there are no more connected components or until each block is full. This approach guarantees that rescheduled transactions are never conflicted; connected components never span across blocks within a burst.

Algorithm 1 Suspended Transaction Dependency Graph

```
Input: Current STDG $\mathcal{G}_{t-1}$, burst $\mathcal{B}_t$
Output: Updated STDG $\mathcal{G}_t$
procedure UpdateDependencyGraph($\mathcal{G}_{t-1}$, $\mathcal{B}_t$)
    $\mathcal{G}_E \leftarrow$ Execute($\mathcal{B}_t, V(\mathcal{G}_{t-1})$)
    $\mathcal{G}_P \leftarrow$ Prune($\mathcal{G}_{t-1}, V(\mathcal{B}_t)$)
    $\mathcal{C} \leftarrow$ NewConflictRoots($\mathcal{G}_E, V(\mathcal{B}_t)$)
    $\mathcal{G}_t \leftarrow$ TraverseConflicts($\mathcal{G}_E \cup \mathcal{G}_P, V(\mathcal{G}_P) \cup \mathcal{C}$)
    return $\mathcal{G}_t$
end procedure
function Prune($\mathcal{G}$, $V$)
    $\mathcal{G}_P \leftarrow (\emptyset, \emptyset)$
    $V(\mathcal{G}_P) \leftarrow V(\mathcal{G}) \setminus V$
    $E(\mathcal{G}_P) \leftarrow E(\mathcal{G}) \setminus \{(v, w) \in E(\mathcal{G}) : v \in V\}$
    return $\mathcal{G}_P$
end function
function NewConflictRoots($\mathcal{G}$, $V$)
    $\mathcal{C} \leftarrow \emptyset$
    for $(v, w) \in$ BFS($\mathcal{G}, V$) do
        if $w \in V \wedge$ Block$_{\mathrm{ID}}(v) \neq$ Block$_{\mathrm{ID}}(w)$ then
            $\mathcal{C} \leftarrow \mathcal{C} \cup \{w\}$
        end if
    end for
    return $\mathcal{C}$
end function
function TraverseConflicts($\mathcal{G}$, $V$)
    $\mathcal{G}_t \leftarrow (\emptyset, \emptyset)$
    for $(v, w) \in$ BFS($\mathcal{G}, V$) do
        $V(\mathcal{G}_t) \leftarrow V(\mathcal{G}_t) \cup \{v, w\}$
        $E(\mathcal{G}_t) \leftarrow E(\mathcal{G}_t) \cup \{(v, w)\}$
    end for
    return $\mathcal{G}_t$
end function
```

Consensus nodes must wait until all suspended transactions have been scheduled before including fresh transactions; this is so that the size of the STDG cannot grow arbitrarily large. By prioritizing suspended transactions over fresh transactions, with the oldest suspended transactions granted the highest priority, we can ensure that transactions will not be starved or delayed indefinitely while limiting the size of the STDG.

The size of the STDG is correlated with the gap between the most recently executed burst and the most recently scheduled burst; if blocks are finalized further in advance of block execution, the STDG must account for farther-reaching dependencies. To mitigate this issue, we impose a limit on the number of bursts that Consensus nodes can schedule in advance of execution – we refer to this limit as the *burst limit*.

With a block size of T transactions, a burst size of B blocks, and a burst limit of L bursts, we can achieve a simple bound on the number of suspended transactions S_t at any time-step t:

$$\max_t(S_t) \leq TBL \tag{1}$$

The RHS of Eq. 1 captures the maximum number of transactions which can be scheduled in advance of execution. In the worst case, all of these transactions are conflicted and every such transaction would become suspended upon execution. However, because the scheduling algorithm waits until every suspended transaction is re-scheduled, no additional transactions will be scheduled and consequently suspended.

V. Evaluation

To evaluate the efficacy of our parallel execution paradigm, we simulate the process of finalizing and executing blocks under our protocol with synthetic transactions and dependencies, which we peg to the dependency frequency observed by Reijsbergen et al. [12]. We vary the *block size* (number of transactions per block), *burst size* (number of blocks per burst), and *burst limit* (number of bursts finalized ahead of execution) to study the effect of our protocol parameters on the performance blockchain execution.

We use the *speed-up factor* as a metric for our evaluation. We define the speed-up factor as the ratio between the number of time-steps required to (re-)execute all transactions under our protocol and the number of time-steps required under sequential execution. When calculating the number of time-steps required during sequential execution, we assume that every block is filled to capacity for a given block size. Additionally, we show that the resources consumed by our protocol, in terms of computation time and memory, are negligible.

A. Experiment Setup

As a basis for our synthetic dependencies, we use an estimate of the available parallelism among smart contracts; in their study of the number of conflicts among Ethereum transactions, Reijsbergen et al. reported that 60% of transactions within a block were conflicted [12]. Reijsbergen et al. assess conflicts based on the sender/receiver addresses of each smart contract, including internal transactions.

Unlike concurrency protocols for typical blockchains, our protocol must keep track of dependencies across blocks. To generate synthetic dependencies across blocks, we first sample the number of dependencies each transaction has from five different distributions:

$$\begin{array}{ll} \mathcal{D}_1 \sim \mathcal{P}(\infty, 1) & \mathbb{E}(\mathcal{D}_1) = 1 \\ \mathcal{D}_2 \sim \mathcal{P}(11, 1) & \mathbb{E}(\mathcal{D}_2) = 1.1 \\ \mathcal{D}_3 \sim \mathcal{P}(6, 1) & \mathbb{E}(\mathcal{D}_3) = 1.2 \\ \mathcal{D}_4 \sim \mathcal{P}(3, 1) & \mathbb{E}(\mathcal{D}_4) = 1.5 \\ \mathcal{D}_5 \sim \mathcal{P}(2, 1) & \mathbb{E}(\mathcal{D}_5) = 2 \end{array}$$

where $\mathcal{P}(\alpha, \lambda)$ is the *Pareto* distribution with a probability density function given by $f(x) = \frac{\alpha\lambda^\alpha}{x^{\alpha+1}};\ \lambda, \alpha > 0, x \geq \lambda$. Then, for each transaction with dependency sampled from $\mathcal{D}_i$, we sample a dependency *distance* from $\mathcal{P}(\alpha, 1)$ which assigns each dependency to a paired transaction. For each $\mathcal{D}_i$ and block size, we tune the parameter α so that the average conflict rate *within* a block is close to 60% as reported by Reijsbergen et al. [12]. The progression from $\mathcal{D}_1$ to $\mathcal{D}_5$ captures an increasing number of total dependencies.

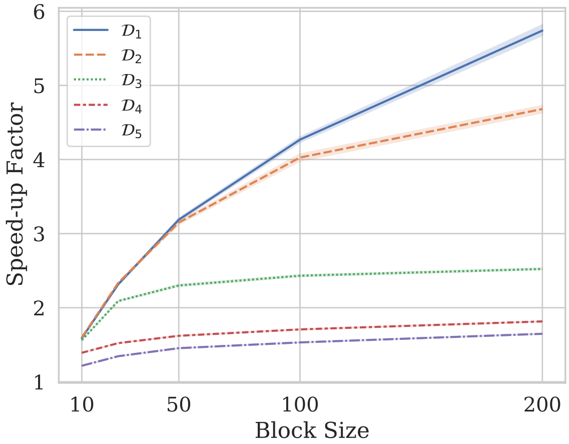

Fig. 2: Speed-up Factor versus Block Size

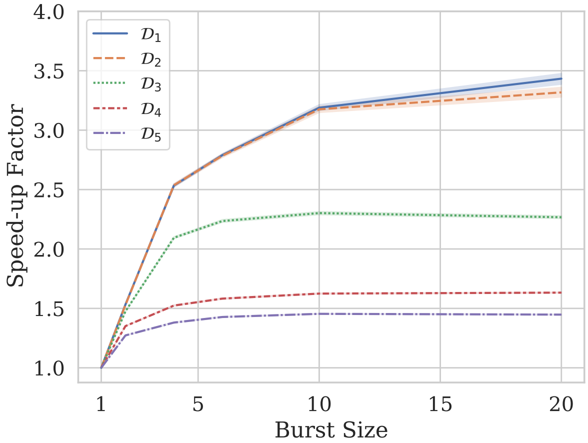

Fig. 3: Speed-up Factor versus Burst Size

Each “blockchain” that we simulate processes 100k transactions and each simulation is performed 40 times for a given set of parameters. The figures plot the averages with a 97% confidence intervals.

B. Speed-up Factor

Figure 2 and Figure 3 show the speed-up factor of the various dependency distributions, ranging between $\mathcal{D}_1$ and $\mathcal{D}_5$. In Figure 2, the x-axis shows block size in number of transactions, while in Figure 3 the x-axis shows burst size in terms of number of blocks. In both figures, the y-axis marks the speed-up factor as measured by the ratio of time-steps required to execute transactions in our protocol and under sequential execution. Both share a similar trend where we observe a decaying increase in speed-up factor with increasing block size or burst size. Increase in speed-up factor is directly associated with the extended capacity of a single burst and therefore correlated with increased parallelism. As block size and burst size increase the growth of the speed-up factors diminishes because of the increased number of conflicts among the transactions within a burst, which requires more transactions to be rescheduled in following bursts. Finally, we observe that as the number of transaction dependencies increases from $\mathcal{D}_1$

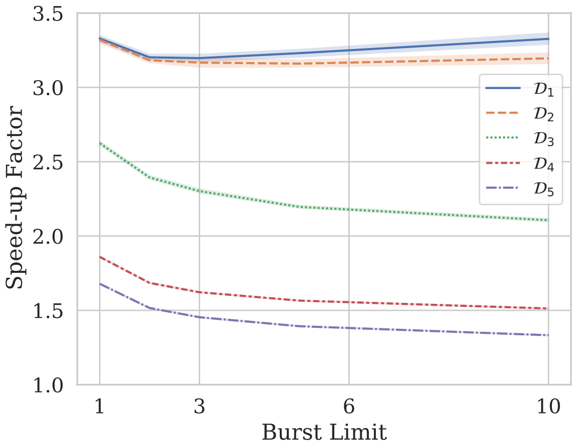

Fig. 4: Speed-up Factor versus Burst Limit

to $\mathcal{D}_5$ the opportunities for parallelization diminish as does the speed-up factor.

Figure 4 illustrates the speed-up factor with varying dependency distributions where the x-axis shows burst limit in number of bursts scheduled ahead of execution and the y-axis marks the speed-up factor as defined above. In general, we observe a decrease of the speed-up factor with increased burst limit. With more transactions scheduled in advance, dependencies must be kept track of across a larger number of bursts resulting in a greater number of conflicts. Interestingly, we observe an eventually rising trend in speed-up factor with increasing burst limit for transaction dependencies sampled from $\mathcal{D}_1$ and $\mathcal{D}_2$. When the conflict rate is low, there is less of a chance that new transactions in later bursts conflict with suspended transactions from previous bursts. In low conflict scenarios, the optimistic concurrency control realizes the opportunities for parallelization, while still rescheduling conflicted transactions as needed.

C. Efficiency of the Dependency Graph

Recall that Eq. 1 bounds the size of the STDG in terms of the block size, burst size, and burst limit. For a more practical understanding of the size of this graph, we test our protocol under the worst case dependency scenario with a block size of 50, a burst size of 5, a burst limit of 3, and 10k transactions. The worst case occurs when every transaction depends on every other transaction within the bound given by Eq. 1; any greater dependency distance cannot contribute to a conflict and therefore can never be included in the STDG. With the above parameter configuration, this means that most of the 10k transactions have $50 \cdot 5 \cdot 3 = 750$ dependencies.

Even with this worst-case dependency structure, our protocol processes 10k transactions in 66.4 seconds averaged over 5 trials on an 11$^{\text{th}}$ generation Intel i3 quad-core CPU. The maximum amount of memory taken up by the STDG was 22.97 MB. For comparison, when we draw dependencies from $\mathcal{D}_5$, as opposed using the worst case dependency structure, our protocol processes the 10k transactions in less than 1 sec with a graph size of 0.24 MB. These results show that the size of the STDG needed for transaction rescheduling is negligible in practice.

VI. Conclusions and Future Work

This paper presented a mechanism for parallel execution of blocks in SoCC model blockchains, which provide a finalized order of transactions and blocks, whose execution may then be parallelized independently of the consensus mechanism. Our mechanism optimistically executes blocks in parallel, discovers conflicting transactions, and generates a Suspended Transaction Dependency Graph (STDG). Moreover, we introduced a scheduling algorithm for rescheduling conflicted transactions using the STDG. We evaluated our parallel execution solution and scheduling algorithm with synthetically generated transactions that reflect observed transaction conflict rates, achieving 3.5 times faster transaction execution in SoCC model blockchains on average, even with a conflict rate as high as 60%.

In the future, we plan to test our approach using real transaction data. This approach would add complexity to our experiment by requiring the execution of smart contracts, but would provide more insight into the exact dependencies among transactions. It would also be profitable to implement this approach in a running blockchain, such as Flow, to study how parallel execution and conflict scheduling fits into the blockchain network, taking into consideration multi-core processors and the division of execution nodes.

References

[1] "Number of Blockchain wallet users worldwide from November 2011 to June 14, 2021," Accesed on Jul. 2021 [Online]. Available: https://tinyurl.com/StatistaNumWallets.

[2] "Ethereum Transactions Per Day," Accesed on Jul. 2021 [Online]. Available: https://tinyurl.com/ETHTxPerDay.

[3] "Technology Analysis: Cryptocurrency Ethereum is flourishing but risks linger," May 2021, [Online]. Available: https://tinyurl.com/ReutersETHAnalysis.

[4] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," in *ACM Symposium on Principles of Distributed Computing*, Jul. 2017.

[5] M. Bartoletti, L. Galletta, and M. Murgia, "A true concurrent model of smart contracts executions," in *Coordination Models and Languages*, Jun. 2020.

[6] P. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "An efficient framework for optimistic concurrent execution of smart contracts," in *Parallel, Distributed and Network-Based Processing*, Feb. 2019.

[7] C. Jin, S. Pang, X. Qi, Z. Zhang, and A. Zhou, "A high performance concurrency protocol for smart contracts of permissioned blockchain," *IEEE Transactions on Knowledge and Data Engineering*, Feb. 2021.

[8] A. Hentschel, D. Shirley, and L. Lafrance, "Flow: Separating consensus and compute," Sep. 2019, [Online]. Available: https://arxiv.org/abs/1909.05821.

[9] A. Hentschel, Y. Hassanzadeh-Nazarabadi, R. Seraj, D. Shirley, and L. Lafrance, "Flow: Separating Consensus and Compute – Block Formation and Execution," Feb. 2020, [Online]. Available: https://arxiv.org/abs/2002.07403.

[10] A. Hentschel, D. Shirley, L. Lafrance, and M. Zamski, "Flow: Separating consensus and compute – execution verification," [Online]. Available: https://arxiv.org/abs/1909.05832.

[11] V. Saraph and M. Herlihy, "An empirical study of speculative concurrency in Ethereum smart contracts," in *Blockchain Economics, Security and Protocols*, May 2019.

[12] D. Reijsbergen and T. Dinh, "On exploiting transaction concurrency to speed up blockchains," in *IEEE Distributed Computing Systems (ICDCS)*, Nov. 2020.