2025 7th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS) | 979-8-3315-5982-3/25/$31.00 ©2025 IEEE | DOI: 10.1109/BRAINS67003.2025.11302922

NEMO: Faster Parallel Execution for Highly Contended Blockchain Workloads

François Ezard*, Can Umut Ileri†, Jérémie Decouchant*
*Delft University of Technology, †IOTA Foundation
francois.ezard@gmail.com, canumut.ileri@iota.org, j.decouchant@tudelft.nl

***Abstract*—Following the design of more efficient blockchain consensus algorithms, the execution layer has emerged as the new performance bottleneck of blockchains, especially under high contention. Current parallel execution frameworks either rely on optimistic concurrency control (OCC) or on pessimistic concurrency control (PCC), both of which see their performance decrease when workloads are highly contended, albeit for different reasons. In this work, we present NEMO, a new blockchain execution engine that combines OCC with the object data model to address this challenge. NEMO introduces three core innovations: (i) a greedy commit rule for transactions that do not use shared objects; (ii) refined handling of dependencies to reduce re-executions; and (iii) the use of incomplete but statically derivable read/write hints to guide execution. Through simulated execution experiments, we demonstrate that NEMO significantly reduces redundant computation and achieves higher throughput than representative approaches. For example, with 16 workers NEMO's throughput is up to 42% higher than the one of Block-STM, the state-of-the-art OCC approach, and 61% higher than the pessimistic concurrency control baseline used.**

***Index Terms*—Blockchain, Execution layer, Optimistic Execution, Scheduling Algorithms**

I. Introduction

Blockchain technologies have been rapidly evolving since Bitcoin [1], which demonstrated that financial services can be implemented in a decentralized manner [2, 3]. However, while the potential of blockchains has been unfolding, they have also been facing adoption issues. In the eye of the public, one of the main limitation of blockchain technologies is their lower performance compared to traditional services [4, 5]. For example, Bitcoin and Ethereum can, respectively, only commit 7 and 30 transactions per second (TPS). In comparison, Visa can process up to 65,000 per second [4].

Since consensus was the primary performance bottleneck in early blockchains such as Bitcoin and Ethereum[6], efficient transaction execution was not a critical concern, and coupling transaction ordering with consensus was a reasonable design choice. However thanks to recent approaches that improve the performance of consensus algorithms [7–10], the performance bottleneck has been shifted to transaction execution [6, 11–15]. As a consequence, modern blockchains [16, 17] have adopted a modular software architecture where consensus is decoupled from execution following an *Order-Execute* model [18]. These *lazy* blockchains [19] decouple components for independent optimization. Parallel execution improves execution throughput while still ensuring that all validators arrive at a consistent final state. More formally, this requirement is called *deterministic serializability*: the results of parallel execution should be identical to sequentially executing all transactions according to the order defined by the consensus layer [11, 15, 20]. Data accesses in blockchain workloads are highly skewed [21] leading to highly contended workloads, which makes it challenging to maintain high performance whilst ensuring deterministic serializability.

Concurrency control is needed to ensure deterministic serializability and comes in two main flavors: *pessimistic* and *optimistic*. Pessimistic concurrency control (PCC) [17, 22], executes possibly conflicting transactions sequentially, which requires prior knowledge of the transactions' read and write sets, often overestimating them to identify and avoid potential conflicts. Optimistic concurrency control (OCC) [6, 23, 24], does not require prior knowledge of the read and write sets. Instead, it optimistically assumes that all transactions are independent and conflicts are detected in the validation step, which may trigger a re-execution of transactions. Both PCC and OCC experience a performance degradation under high contention, though for different reasons. PCC incurs significant overhead due to increased fallback to sequential execution, while OCC experiences frequent transaction re-executions caused by conflicts. Given that blockchain workloads are often highly contended [11–13, 21], preserving the performance gains of parallel execution in such environments is critical.

Another important factor that influences the frequency of conflicts in blockchain systems is the underlying data model. Modern blockchains have developed data models to improve parallelism, reduce contention and improve scalability. Sui [17] developed the *object* data model built on top of the Move virtual machine (VM) [25], which has since been adopted by IOTA [26]. This model treats each object as an independent unit of state with a unique global address, which as we will see later is very powerful.

This paper[1] makes the following **contributions**:

- We adapt the state-of-the-art OCC execution engine, Block-STM [23], to support the object data model, introducing a novel fast-path via a greedy commit rule for transactions that use only owned objects.
- We propose new techniques that specifically target performance under high-contention workloads.
- We provide a comprehensive evaluation using realistic blockchain workloads, demonstrating that NEMO achieves sig-

[1]An extended version of this paper is available on arXiv:2510.15122 [27].

nificant performance gains over existing systems.

II. Related Work

Block-STM [23] is a parallel execution engine for smart contracts that uses OCC and is currently used in production in the Aptos [16] blockchain. Block-STM optimistically assumes that all transactions are independent and executes them in parallel recording their read/write sets. Those sets are then used to validate the results of execution, which leads to re-execution if a transaction fails validation. Once all transactions have been validated and there are no more tasks to execute, Block-STM lazily commits the block of transactions.

Sui [28]'s execution engine assigns version numbers to all the objects to enforce a causal order between transactions. Once all object versions a transaction takes in are available, the transaction is ready to be executed and increments the versions of all objects it writes. This approach requires complete prior knowledge of the transactions' read/write sets, which is done by leveraging the object model to obtain the exhaustive set of objects that a transaction could read/write. However, objects in this set are not necessarily used during execution as the transaction logic depends on the global state at the time of execution. This results in an overly cautious approach that fails to take advantage of the limited parallelism opportunities in high contention workloads.

ParallelEVM [21] uses a novel *operation-level* concurrency control algorithm, which handles conflicts at the operation level rather than at the transaction level. This increases the amount of work that can be done in parallel which has a greater impact for high contention workloads. ParallelEVM was developed for Ethereum and relies on the assumption that conflicts affect only a few operations. It is unclear whether this assumption holds for other blockchains and for the object data model.

Related to blockchains, deterministic databases have very different workloads but also require deterministic serializability, and often follow the *Order-Execute* paradigm. Recent works [29–31] have shown that OCC can be used even under high contention and ensure high throughput.

III. Background

In this section, we provide a detailed description of two representative execution layers, namely Block-STM [23] and Sui [17]'s, which respectively follow the OCC and PCC approaches. We also explain why NEMO builds on top of Block-STM to optimize for high contention scenarios.

Block-STM's OCC. Figure 1 illustrates Block-STM's architecture. The scheduler distributes tasks to execution threads. It prioritizes tasks following the order of transactions defined by a block, whilst also trying to keep all the threads busy at all times. There are two types of tasks a thread can be given, a transaction execution task or a transaction validation task, neither of which can be aborted. Transaction execution is done first and creates a validation task to verify the execution result. For an execution task a thread tries to read all input objects from the Multi-Version Memory (MVMemory) store, but ends up reading the last committed version from storage if there is no entry for that object. All writes are done to MVMemory and the final value of an object is committed to storage at the end of an epoch, i.e., when a block has been processed. For validation tasks a thread reads the latest available version of each input object, like for an execution task, and then compares their version number with the version number that was read when the transaction was executed ensuring that the resulting state is correct. Crucially, MVMemory is shared across all threads so that when a transaction fails validation, it marks all entries that it wrote as ESTIMATE to indicate that other transactions should not read their value as they are likely to be overwritten in a near future. The scheduler also tries to keep track of dependencies to avoid known conflicts that would likely lead to additional re-executions.

Sui's PCC. Sui [17] is a decentralized smart contract platform, which focuses on low-latency and is maintained by a permissionless set of validators [28]. The key feature of Sui is its object-centric data model in which the global state is represented by the state of all objects. There are 3 types of objects: i) *immutable objects* can be read by any transactions but cannot be modified; ii) *owned objects* that have a single owner which grants permission to use the object; and iii) *shared objects* that do not have an owner and can be included in anybody's transactions [28]. Critically, while smart contracts can create and mutate objects they do not store them [32], which enables more parallelism as transactions touching different object can safely be executed in parallel. Another key advantage of the object data model, is that each object has its own global address, which allows Sui to provide an exhaustive, but often overestimated, set of objects a transaction may use prior to execution.

IV. NEMO

This paper introduces NEMO a novel blockchain execution framework that is optimized for high contention workloads.

A. Combining the object-model and OCC

NEMO uses the OCC paradigm as it has been shown that the performance obtained with a lock-free approach is higher than the one obtained with a lock-based approach under scenarios with hotspots [33], which are common for blockchains [21]. NEMO then adopts the object model as it provides the greatest potential for parallelism and provides more prior knowledge about the read/write sets. NEMO combines the object model with OCC to take advantage of every opportunity to parallelize the workload whilst avoiding cascading aborts. NEMO is implemented on top of Block-STM and thus shares the same core architecture shown in Figure 1. We believe that NEMO is the first protocol to combine OCC with the object data model and it is also among the first protocols to optimize blockchain execution for the high contention scenarios.

B. Greedy commit rule

Having adapted Block-STM [23] to the object data model allowed us to take advantage of it. A transaction that does

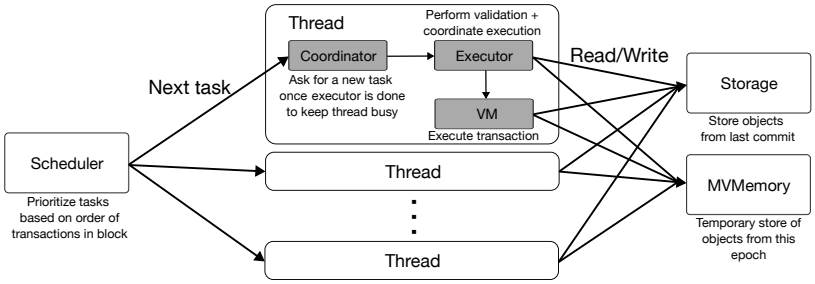


Fig. 1: Block-STM's architecture

not use any shared objects will be independent of all other transactions in that epoch, and can safely be executed at any time. Such a transaction can also skip the validation step and be greedily committed immediately after execution. This *greedy commit rule* offers a true fast-path for transactions that do not interact with the shared global state, and builds upon the existing fast-path that allows such transactions to bypass consensus [28].

C. Limiting re-executions

Transaction execution is the costliest operation in terms of time, so limiting the number of times a transaction is re-executed is crucial for performance. The key to obtaining good performance when using OCC is to avoid prematurely triggering re-execution which likely will result in further re-execution. `NEMO` looks to validate early, but to have transactions enter the *Waiting* phase for as long as they have some unresolved dependencies. This limits the redundant work being done and protects against cascading aborts. For this to work `NEMO` has to extract information about dependencies even when execution succeeds, unlike Block-STM that only does so when execution fails. We expect this to have a higher impact for high contention workloads, as there are more dependencies.

To limit re-executions, `NEMO` resolves dependencies once the blocking transaction passes validation, unlike Block-STM that resolves dependencies after a successful execution. Transactions are therefore blocked for longer, but once a dependency is resolved it is less likely to cause issues later on.

D. Incomplete hints

Block-STM [23] assumes no prior knowledge of transaction read/write sets and that all transactions are independent. Under high contention this results in a lot of failing validations and thus a lot of re-executions. As mentioned earlier, in the object model each object has its own globally unique identifier [28]. This allows one to statically extract the exhaustive set of objects a transaction may access. However, this exhaustive set is often overly pessimistic, as it includes all possible objects a transaction might touch depending on the execution path and global state, which limits parallelism.

`NEMO` seeks a middle ground by leveraging partial information about the read/write sets of transactions to limit unnecessary re-executions, whilst remaining optimistic enough to exploit parallelism. This partial information should only include objects that are guaranteed to be accessed during execution. `NEMO` assumes that this partial information can be obtained statically by leveraging the object data model, as it exposes unique identifiers and explicit references in transaction logic. If no such information is available, `NEMO` falls back to the same assumptions as Block-STM.

`NEMO` then uses this information to schedule conflicting transactions in order and avoid unnecessary re-executions. This is done by identifying known dependencies during pre-processing and starting blocked transactions in the `Waiting` phase.

V. Performance Evaluation

A. Implementation

We implemented `NEMO` on top of Block-STM [23] in Rust using around 3,500 lines of code [2]. We use simulated execution instead of actually executing transactions, as the improvements made by `NEMO` do not target the execution of transactions inside the virtual machine. We simulate execution by making a thread sleep for a certain amount of time and record the read/write sets of a transaction. We randomly sampled this simulated execution duration given in milliseconds following a $LogNormal(2.0, 0.5)$ distribution in order to have some variation. This gave us an expected execution time per transaction of around 8.4 ms, a median of around 7.4 ms and a 90th percentile of around 14.0 ms.

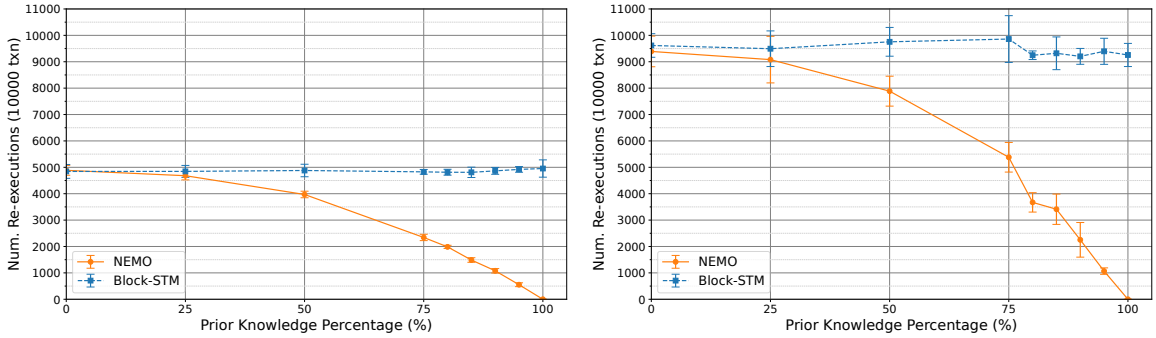
To accurately simulate the execution of transactions, the exhaustive read/write sets of objects that a transaction can use is determined when generating the workload. Each object then has a 90% chance of actually being used by the transaction during execution. We then vary the likelihood of an object being known ahead of time to simulate partial prior knowledge.

B. Workloads

To evaluate `NEMO` under high contention, we consider a synthetic scenario with 50 shared objects where each transaction accesses a number of objects that is sampled from $LogNormal(0.5, 0.5)$. For each object accessed we then sample from $Zipf(50, 2.0)$ which is a skewed distribution to simulate hot objects causing contention between transactions.

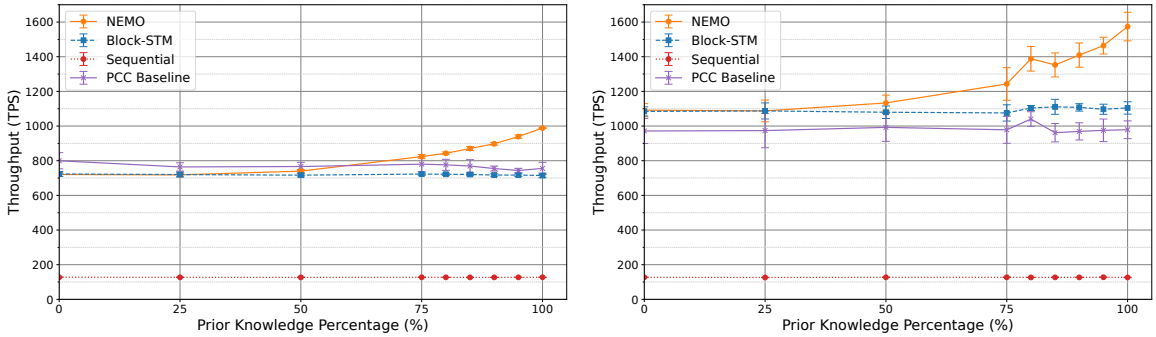
The main metric used to evaluate performance is the duration taken to execute a block. Taking the block size and

[2] https://github.com/AirWinter/nemo



(a) 8 Workers (b) 16 Workers

Fig. 2: Number of re-executions depending on assumed proportion of prior knowledge



(a) 8 Workers (b) 16 Workers

Fig. 3: Execution throughput depending on assumed proportion of prior knowledge

dividing it by the duration then gives us the *throughput* measured in transactions per second (TPS). NEMO and Block-STM both use a lazy commit except that NEMO also has the *greedy commit rule*, which means that if NEMO is able to achieve a better throughput it would also mean it achieved a better latency than Block-STM. We also measured the number of re-executions to track how much redundant work has been done, because we designed NEMO to minimize this metric in hopes that it would lead to an increase in performance.

All experiments were run using the Delft High Performance Computing (DHPC) center [34] with 2GB of memory per CPU. All code was ran using release mode. For every data point collected, we plot the mean and standard deviation of 5 runs. In our experiments we compare NEMO to sequential execution, Block-STM, and a PCC baseline. The PCC baseline looks to simulate Sui Lutris [28] execution by using the exhaustive read/write sets to prevent potentially conflicting transactions from executing in parallel.

C. Number of re-executions

Figure 2 shows the number of re-executions with 8 and 16 workers for Block-STM and NEMO. Other baselines, i.e., sequential execution and PCC baseline, never re-execute transactions and are therefore not shown. More prior knowledge effectively reduces the number of re-executions for NEMO, down to 0 with full knowledge. NEMO is able to halve the number of re-executions when given 75% prior knowledge for both 8 and 16 workers. Increasing the number of workers increases the number of re-executions for both protocols.

D. Throughput

Figure 3 shows the throughput of the protocols. We can see that increasing prior knowledge leads to increased throughput and that even partial knowledge can lead to a significant performance gain. For 16 workers we can see that NEMO already significantly outperforms Block-SMT at 75% prior knowledge with a throughput of 1243 TPS compared to 1076 TPS for Block-STM which represents a 15.5% improvement in throughput, and even more at 90% with a throughput of 1409 TPS compared to 1108 TPS for Block-STM which is a 27.1% improvement.

VI. Conclusion

This work presented NEMO, a novel blockchain execution engine that integrates the object data model with optimistic concurrency control (OCC) to achieve high throughput under highly contended workloads. Our evaluation demonstrates that even partial prior knowledge of transaction object access patterns significantly reduces redundant re-executions and improves throughput. NEMO consistently outperforms existing baselines under high contention, with up to 42% improvement in throughput compared to Block-STM and 61% compared to the PCC baseline.

References

[1] S. Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: (May 2009).

[2] S. Baheti, A. Parwat Singh, S. Peri, and Y. Simmhan. "DiPETrans: A framework for distributed parallel execution of transactions of blocks in blockchains". In: *Concurrency and Computation: Practice and Experience* 34 (Jan. 2022).

[3] S. Su, J. Hu, Y. Govil, and S. Kohli. "Concerto: Transaction-Parallel EVM". In: *Stanford University* (2024).

[4] G. Mitenkov. "Metering the Meter, or How to Efficiently and Deterministically Charge the Execution of Smart Contracts". Master Thesis. ETH Zurich, Oct. 2023.

[5] S. Sridhar, A. Sonnino, and L. Kokoris-Kogias. *Stingray: Fast Concurrent Transactions Without Consensus*. 2025. arXiv: 2501.06531.

[6] R. Neiheiser, A. Babaei, G. Alexopoulos, M. Kogias, and E. K. Kogias. *CHIRON: Accelerating Node Synchronization without Security Trade-offs in Distributed Ledgers*. 2024. arXiv: 2401.14278.

[7] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman. "Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus". In: *EuroSys*. 2022.

[8] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias. "Bullshark: DAG BFT Protocols Made Practical". In: *CCS*. 2022.

[9] I. Abraham, G. Gueta, and D. Malkhi. "Hot-Stuff the Linear, Optimal-Resilience, One-Message BFT Devil". In: *CoRR* abs/1803.05069 (2018). arXiv: 1803.05069.

[10] K. Babel et al. *Mysticeti: Reaching the Limits of Latency with Uncertified DAGs*. 2024. arXiv: 2310.14821.

[11] X. Qi, J. Jiao, and Y. Li. "Smart Contract Parallel Execution with Fine-Grained State Accesses". In: *ICDCS*. 2023.

[12] P. Garamvölgyi, Y. Liu, D. Zhou, F. Long, and M. Wu. "Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts". In: *ICSE*. 2022.

[13] R. Neiheiser and E. Kokoris-Kogias. *Anthemius: Efficient & Modular Block Assembly for Concurrent Execution*. 2025. arXiv: 2502.10074.

[14] R. Shahid. "Parallel Transaction Execution in Public Blockchain Systems". MA thesis. University of Waterloo, 2024.

[15] Y. Hay and R. Friedman. " Batch-Schedule-Execute: On Optimizing Concurrent Deterministic Scheduling for Blockchains ". In: *SRDS*. 2024.

[16] *The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure*. 2022.

[17] T. M. Team. *The Sui Smart Contracts Platform*. 2022.

[18] J. Chen, A. Sonnino, L. Kokoris-Kogias, and M. Sadoghi. *Thunderbolt: Concurrent Smart Contract Execution with Non-blocking Reconfiguration for Sharded DAGs*. 2025. arXiv: 2407.09409.

[19] E. N. Tas, D. Tse, L. Yang, and D. Zindros. "Light Clients for Lazy Blockchains". In: *FC*. 2025.

[20] D. Ryu and C. Park. "Toward High-Performance Blockchain System by Blurring the Line between Ordering and Execution". In: *SC*. 2024.

[21] H. Lin, H. Feng, Y. Zhou, and L. Wu. "ParallelEVM: Operation-Level Concurrent Transaction Execution for EVM-Compatible Blockchains". In: *Proceedings of the Twentieth European Conference on Computer Systems*. EuroSys '25. Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 211–225.

[22] Z. Chen, X. Qi, X. Du, Z. Zhang, and C. Jin. "PEEP: A Parallel Execution Engine for Permissioned Blockchain Systems". In: *DASFAA*. 2021.

[23] R. Gelashvili et al. "Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing". In: *PPoPP*. 2023.

[24] G. Mitenkov et al. *Deferred Objects to Enhance Smart Contract Programming with Optimistic Parallel Execution*. 2024. arXiv: 2405.06117.

[25] S. Blackshear et al. "Move: A Language With Programmable Resources". In: 2019.

[26] IOTA Foundation. *IOTA Rebased: Fast Forward*. 2024. URL: https://blog.iota.org/iota-rebased-fast-forward.

[27] F. Ezard, C. U. Ileri, and J. Decouchant. *NEMO: Faster Parallel Execution for Highly Contended Blockchain Workloads (Full version)*. 2025. arXiv: 2510.15122 [cs.DC].

[28] S. Blackshear et al. *Sui Lutris: A Blockchain Combining Broadcast and Consensus*. 2024. arXiv: 2310.18042.

[29] Z.-Y. Dong et al. "Optimistic Transaction Processing in Deterministic Database". In: *J. Comput. Sci. Technol.* 35.2 (Mar. 2020), pp. 382–394.

[30] X. Wang, Y. Peng, and H. Huang. "Gria: an efficient deterministic concurrency control protocol". In: *Frontiers of Computer Science* 18 (2023), pp. 1–13.

[31] X. Wang, Y. Peng, H. Huang, and X. Li. "Dodo: A scalable optimistic deterministic concurrency control protocol". In: *Future Generation Computer Systems* 159 (2024), pp. 15–26.

[32] "All About Objects". In: *Sui Foundation* (2023).

[33] J. Zhang, Z. Luo, R. Ramesh, and A. Kate. *Optimal Sharding for Scalable Blockchains with Deconstructed SMR*. 2024. arXiv: 2406.08252.

[34] D. H. P. C. C. (DHPC). *DelftBlue Supercomputer (Phase 2)*. https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2. 2024.