Optimizing Blockchain Execution for High Contention

by

François Ezard

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday June 30, 2025 at 14:00.

Student number:	5112648	
Project duration:	November 11, 2024 – June 30, 2025	
Thesis committee:	Dr.-Ing. Jérémie Decouchant,	TU Delft, supervisor
	Dr.-Ing. Sebastian Proksch,	TU Delft
	Dr. Can Umut Ileri,	IOTA Foundation

TU Delft

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 four core innovations: (i) a greedy commit rule for transactions using only owned objects; (ii) refined handling of dependencies to reduce re-executions; (iii) the use of incomplete but statically derivable read/write hints to guide execution; and (iv) a priority-based scheduler that favors transactions that unblock others. Through simulated execution experiments, we demonstrate that NEMO significantly reduces redundant computation and achieves higher throughput than representative approaches. For example, with 16 workers when running on the Delft High Performance Center (DHPC) supercomputer, 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 PCC baseline used.

Preface

This thesis marks the end of my academic journey at TU Delft, which began nearly six years ago with my bachelor's degree. Looking back, it is remarkable how much I have learned and more importantly, how much I have grown as a person. I am excited to present my thesis on Optimizing Blockchain Execution for High Contention.

Before delving into the thesis, I would like to acknowledge those who supported me throughout this project. First, I would like to thank my supervisor Dr. Jérémie Decouchant for his constant availability and guidance, especially during the early stages of the project. I am also grateful to Dr. Can Umut Ileri for regularly meeting with me and challenging me to deepen my understanding of the topic. Thank you for allowing me to use the contention simulator, which was essential to this research. I would also like to thank Artjom Pugatsov, my fellow student under Dr. Decouchant's supervision, for your invaluable insights. A special thank you to Dr. Sebastian Proksch for being willing to serve on my thesis committee.

Lastly, I want to thank my family, friends, and my girlfriend for their unwavering support throughout my academic journey.

François Ezard
Delft, June 2025

Contents

List of Figures

List of Tables

1 Introduction

Blockchain technologies have been rapidly evolving since Bitcoin [30], which demonstrated that financial services can be implemented in a decentralized manner [8, 40]. Another key milestone was the introduction of the Ethereum Virtual Machine (EVM) [21], which allows transactions to invoke code specified as a *smart contract* [14, 34]. Smart contracts extended the range of possible uses of blockchain technologies [28]. 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 [28, 39]. For example, Bitcoin [30] and Ethereum [21] can, respectively, only commit 7 and 30 transactions per second (TPS). In comparison, Visa can process up to 65,000 per second [28]. In addition, traditional blockchains like Bitcoin and Ethereum execute transactions sequentially to ensure that all validators end up in the same state, which significantly limits their execution throughput, as it fails to efficiently utilize modern multicore CPU architectures [33, 39].

Since consensus was the primary performance bottleneck in early blockchains such as Bitcoin and Ethereum [32], 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 [2, 7, 17, 38], the performance bottleneck has been shifted to transaction execution [22, 24, 31, 32, 33, 36]. As a consequence, modern blockchains [42, 43] have adopted a modular software architecture where consensus is decoupled from execution following an *Order-Execute* model [13]. These *lazy* blockchains [41] decouple components for independent optimization.

1.1. Research Questions

This thesis explores the topic of blockchain execution, which currently represents the primary bottleneck in overall blockchain performance. We look to improve the performance for high contention workloads since blockchain workloads are inherently contended [22, 27, 31, 33]. To do so we will look to maximize parallel execution to take advantage of modern CPU architectures. This leads us to the following main research question:

How can parallelism be used to maximize the throughput of blockchain execution for high contention workloads?

To answer that question we must first look at the following sub-questions:

RQ_1 What are the characteristics of high contention workloads?

RQ_2 Where does the current state of the art fall short? What are their limitations?

RQ_3 How can we maximize parallel execution whilst managing conflicts? And how can we effectively manage these conflicts?

1.2. Contributions

This thesis looks to answer the research questions devised. The main contribution of this work can be found in Chapter 4 that delves into the design of the NEMO, our novel protocol. The main contributions of this thesis are the following:

- Research the current blockchain execution protocols, discuss their strength & weaknesses, and identify a gap in the state of the art.
- Design NEMO a novel protocol to address the gap in the state of the art.
- An implementation of the NEMO protocol written in Rust.
- A thorough evaluation of NEMO, which includes various different settings and different levels of contention. NEMO is compared to the current state of the art as well as to sequential execution, to evaluate its potential impact on the performance of blockchains.

1.3. Thesis Outline

Chapter 2 gives some background information needed to fully understand the rest of the thesis. Chapter 3 goes over related work and also compares the features of the state of the art to that of our NEMO protocol. Chapter 4 goes over how the NEMO protocol works and the intuition behind its design as well as its overall philosophy. Chapter 5 describes our implementation of the protocol. Chapter 6 goes over how the protocol was evaluated, the results obtained and the main takeaways from those results. Finally, Chapter 7 concludes this thesis and explores future work.

2

Background

This chapter provides some background information needed to understand the topic. It covers what a blockchain is, their development leading to the state of modern blockchain technology, and dives into the specifics of the Sui blockchain as our solution targets their data model. The chapter also provides background on the different types of concurrency control, and concludes with a clear description of the problem this work looks to solve.

2.1. Blockchains

A blockchain maintains a trustable distributed ledger, structured as a deterministic replicated state machine [40]. It allows validators to agree on a sequence of transactions submitted by clients on the network [28]. Transactions on a blockchain look to modify the state in some way. During execution, a transaction will read certain values and write certain values to the ledger. The *read set* comprises of all values that a transaction reads and the *write set* comprises of all values that a transaction writes. These sets are usually not available prior to execution, as they depend on the global state at the time of execution. While some assume that every value written must also be read, we do not make that assumption. Instead, for a value to be both read and written it must explicitly be included in both sets. This allows transactions to write values without having to read them first.

A blockchain is said to be *public* if anyone can join as a validator, whereas a *permissioned* blockchain only grants access to authorized participants. Public blockchains focus on security and reliability, whilst permissioned blockchains focus on throughput [15].

Most modern blockchains have adopted a modular architecture where the consensus is decoupled from the execution following the *Order-Execute* paradigm [13]. In this paradigm, consensus first validates the transactions, orders them and propagates them to all the validators. All the validators then individually execute the transactions [6]. For this to work, the execution step has to yield the same result for all validators, which is why Bitcoin [30] and Ethereum [21] opted to sequentially execute the transactions in order. This heavily limits their throughput as it fails to efficiently utilize the multiple cores available in modern computers [33], resulting in a throughput of around 7 transactions per second (TPS) for Bitcoin and around 30 TPS for Ethereum, both drastically less than the 65,000 TPS Visa is capable of [28].

Parallel execution looks to address this issue by parallelizing the workload in order to increase the throughput. Doing so comes with a new set of challenges however, as it must ensure that execution yields the same result for all honest validators. This requirement is called *deterministic serializability*, and states that the effect of the parallel execution should be the same as sequentially executing all transactions according to a given order [24, 33, 35]. Modern blockchains like Aptos [43] and Sui [42] use parallel execution, yielding substantial improvements compared to Bitcoin and Ethereum but are still unable to reach the required throughput for widespread adoption. While consensus had historically been the bottleneck on performance [22, 24, 31, 32, 33, 36], this has now been shifted onto the execution layer thanks to recent advances in consensus [2, 7, 17, 38]. As a result, there has been a renewed interest in improving the throughput of blockchain execution.

2.2. Concurrency Control

Blockchain workloads are naturally highly contended [22, 27, 31, 33] requiring some mechanism to ensure that conflicts are dealt with in order to satisfy deterministic serializability. Conceptually there are two main approaches, either execute conflicting transactions in order or detect that they were executed out of order. The former is *pessimistic concurrency control (PCC)* (also called static parallelism), while the latter is *optimistic concurrency control (OCC)* (also called dynamic parallelism).

Pessimistic concurrency control (PCC) aims to execute conflicting transactions in order, ensuring that the system ends up in the same state as if it had sequentially executed all the transactions. This is usually done through the use of locks that a transaction must acquire before being able to execute. These locks ensure that the transaction has exclusive access to the values it needs, preventing data races. Other conflicting transactions have to wait until the locks are released before they can acquire them and execute. The order in which the locks are granted is critical, and must be the same for every validator.

The problem with this approach is that for a highly contended workload it will lead to a lot transactions being executed sequentially with the extra overhead of acquiring and releasing locks. This fails to utilize all the available resources, and leads to a lot of waiting around. Another problem with this approach in the context of blockchains, is that it requires knowing ahead of time what transactions are conflicting. This means that the read and write sets of all transactions must be known before execution, which is hardly ever possible [29, 33].

Optimistic concurrency control (OCC) assumes that there are no conflicting transactions and that they can all be executed in parallel. This often results in conflicting transactions being executed out of order which is why transactions have to go through an extra validation step before being committed. If this validation step detects a conflict then the transaction must be re-executed. This can potentially happen multiple times. OCC performs great on low contention workloads, as transactions are executed in parallel with minimal re-executions. However, in a high contention scenario it will lead to a lot of re-execution which can lead to cascading aborts, where aborting a transaction leads to more transactions being aborted.

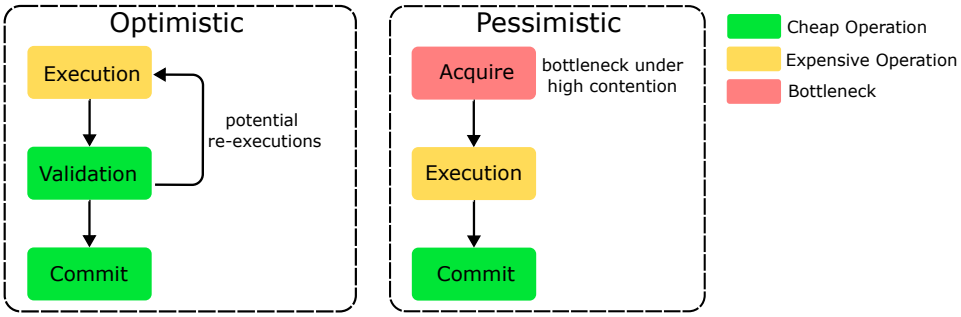


Figure 2.1: Difference between Optimistic and Pessimistic Concurrency Control

Figure 2.1 shows the difference between OCC and PCC in the context of blockchains. Whilst both approaches experience a loss in performance under high contention, it is for different reasons. OCC has the problem of cascading aborts, whereas PCC ensures every transaction is only executed once but is bottlenecked by the acquire phase.

2.3. Data Model

The underlying data model of a blockchain has a big impact on the nature of transactions, greatly affecting the performance of the blockchain. Bitcoin uses the unspent transaction output (UTXO) model where coin transactions are the main way of storing data. Ethereum uses the account data model, giving smart contracts their own account address which can be used to enable buying and selling digital tokens among other things [37]. In the account model a global state is maintained, and transactions can modify that state but do not carry full information about the end state [12]. This puts a bigger burden on validators, as they need to verify that transactions do not conflict with one another potentially causing a bottleneck [3]. This also limits the potential for concurrency as it increases the frequency of conflicts.

More recent blockchains have looked to use other data models to enable more parallelism. Aptos [43] and Sui [42] are both built on top of the Move virtual machine [9] (VM) , which enables the definition of custom resource types. Sui has leveraged this to develop their innovative object data model, where the basic unit of storage is an object with its own global address. This reduces the number of conflicts between transactions and makes it easier to detect conflicts. We will discuss the the object data model more in depth in Section 2.4.2.

At a high level, a transaction looks to perform some action on the global state. This usually requires reading certain values and writing some results. The underlying data model affects how the values are stored. In every model there is some potentially empty set of values read by the transaction called the read set and some potentially empty set of values written by the transaction called the write set. A conflict between two transactions is when the write set of one transaction overlaps with either the read or the write set of another transaction. In such a situation the order in which these transactions are executed matters as it can affect the final state of the system. The data model used affects how the global state of the blockchain is represented and how transactions interact with that global state, which has an effect on the frequency of conflicts.

2.4. Sui

Sui [42] is a decentralized smart contract platform, which focuses on low-latency and is maintained by a permissionless set of validators [10]. It follows the *Order-Execute* paradigm and does so using a modular approach which decouples execution from consensus [31]. Sui has a native token called *SUI* which is used to delegate stake and pay gas fees [42]. It uses a proof-of-stake consensus mechanism, meaning that validators lock up a certain amount of SUI tokens as collateral and are rewarded with more tokens for processing transactions [19]. The more collateral they offer, the larger their stake and thus the larger their voting power. The key feature of Sui is its object-centric data model in which the global state is represented by the state of all objects. This data model enables more parallelism, as it makes it easier to detect conflicts and non-conflicting transactions can be executed simultaneously [10].

2.4.1. Object Data Model

In Sui, the basic unit of storage is an object with its own global address. Objects are long-lived and make up the global state, so in order for a validator to replicate the state it must replicate the set of all objects [10].

There are three different ownership rules for objects, the first one are *read-only objects* which as the name indicates, cannot be mutated. Consequently, they can be used in transactions concurrently with no risk [10]. Even though these objects have an owner, it does not affect the authorization to use them [42]. The second type of objects are *owned objects*, which have an owner field that can either be an address representing a public key or the unique identifier of another object. In the case that the owner is an address, transactions can only use and mutate the object if the transaction is signed by the owning address. Access to an owned object can only be granted to one transaction per epoch [10, 42]. This means that there cannot be two transactions conflicting on an owned object. Crucially, the gas budget of a transaction is paid using an owned object which naturally prevents double spending [10]. In the case that the owner is another object, the child object can only be used if the parent object is included in the transaction and the transaction is allowed to use the parent object. The third type of object is *shared objects*, which have no owner and can be mutated by anyone. Consequently, transactions using shared objects go through consensus to establish an order between potentially conflicting transactions.

A transaction in Sui takes in objects as input and produces new or modifies existing objects as output [11]. Transactions are atomic, meaning that if they abort at some point then the transaction has no effect at all and objects remain the same as they were prior to the transaction [42]. A transaction is said to be *simple* if it does not involve shared objects, in which case it can bypass consensus taking a fast path to execution. A transaction is said to be *complex* if it involves shared objects. Complex transactions must go through consensus. Two transactions are said to be conflicting if they have the same shared object as input and at least one of them is mutating that object. Two transactions that are not conflicting can be executed in any order, since executing them in any order will result in the same outcome. This also means that non-conflicting transactions can safely be executed in parallel.

There are two main advantages of the object data model, the first one is that there is a distinction between *owned*, *immutable* and *shared* objects. This reduces the frequency of conflicts since transactions do not need to touch the shared state, increasing the potential parallelism in the workloads. Furthermore, transactions have greater control over the resources they need allowing them to use the bare minimum. The second main advantage is that giving each object its own global address makes it easier to identify the objects that a transaction uses. Using this we can statically derive the set of objects that a transaction can use, which is useful when it comes to concurrency control.

2.4.2. Concurrency Control in Sui

Sui uses PCC to execute conflicting transactions in order. For this to work Sui needs to know the read/write sets of all transactions ahead of time [4]. Unfortunately, accurately identifying the read/write sets of a transaction ahead of time is hardly ever possible [29, 33]. This is because they depend on the logical path taken by the smart contract, which often depends on the application state at the time of execution. A solution that other works have found is to have application developers pre-declare an exhaustive list of resources a transaction can use. Having entries missing results in the transaction being aborted, which often leads to overestimating the resources needed [29, 31]. This results in phantom conflicts reducing the potential parallelism of the workload which negatively affects the performance of the system.

As mentioned in Section 2.4.1, in Sui each object has its own global address. Transactions that want to use an object need to refer to it using this address, which makes it easier to identify what objects a transaction needs without putting the burden on application developers. This approach provides an exhaustive list of all the objects that a transaction may use. During execution it is likely that transactions only use a subset of these objects depending on the logic and global state at the time of execution. This is an area for improvement which we will come back to in Section 4.3.4 when discussing how NEMO handles concurrency control.

2.4.3. Transaction Lifecycle

Now that we have an understanding of the core concepts of Sui, let us dive into the complete lifecycle of a transaction shown in Figure 2.2.

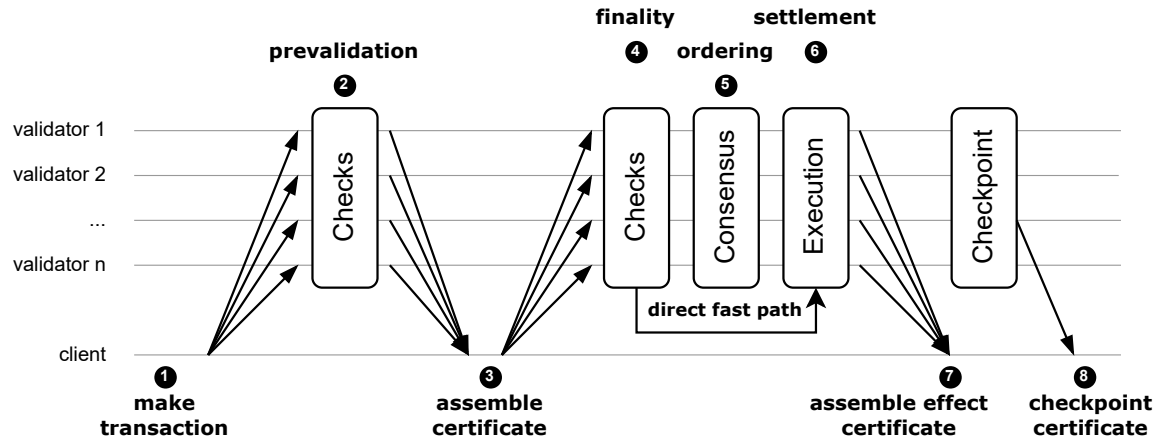


Figure 2.2: Overview of all the steps in one Sui epoch, taken from [10]

Step ❶ is initiated by the user when they create and sign a transaction to mutate objects. This transaction is then broadcasted to all validators by the client, which in turn perform some initial validation (step ❷) to prevent things such as transaction spamming or double spending. In this step, the validator will also look to acquire a mutex for every owned-object transaction input, ensuring that this is the first valid transaction seen using this input and enforcing that only one transaction use a given owned object per epoch. If the transaction is valid and the validator was able to aquire the mutexes, then the validator will return the signed transaction to the client. In step ❸ the client collects signatures from a quorum of validators to form a transaction certificate. The client then broadcasts the transaction to all validators again (with the certificate). In step ❹ the validators check the validity of the certificate and respond to

the client. If the client receives a quorum of responses it ensures *transaction finality*. This means that the transaction will eventually be executed, meaning that it will show up on the ledger but that its effects are not yet known. It could be for example that the transaction execution exceeds the gas budget, in which case this constitutes a successful execution with a failure message resulting in the transaction showing up in the ledger with a failure code. Step ❺ of the process is the consensus stage that orders the certificates. *Simple* transactions can take the fast path and skip this step altogether.

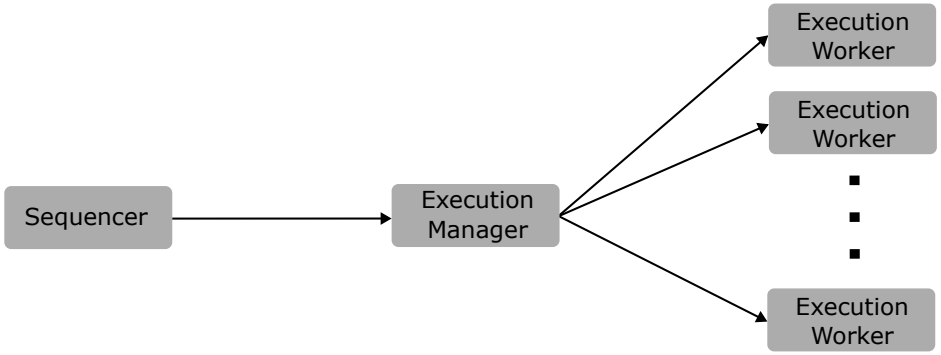


Figure 2.3: Components of Sui Execution

Step ❻ is the execution step and can actually be broken down further, as shown in Figure 2.3. Sequencing takes the set of transactions outputted by consensus and is tasked with defining an order. The key is that it is only looking to enforce a causal order (happens-before relationship) between conflicting transactions [42], ensuring consistency across validators whilst still giving room for parallelization. This is done through the use of version numbers, with the object key (ObjKey) being made up of the (ObjId, Version) tuple [10]. The sequencer goes through all transactions and assigns each object in its read set a version number, which defines what version they must take in as input. The version of an object is only increased if a transaction mutates it, creating a causal relationship between conflicting transactions. This means that transactions reading a certain version of an object are independent of one another, but must be executed before a transaction that would mutate the object to its next version [10]. As discussed in Section 2.4.2, this requires complete prior knowledge of the transactions' read-/write sets, which imposes some constraints which we will come back to in Chapter 4. The execution manager is then tasked with keeping track of the objects and their different versions, as well as scheduling transactions for execution. A transaction is only ready for execution once all the object versions it needs are available, ensuring that the causal order defined by the sequencer is followed. Once a transaction is ready, the execution manager will delegate execution to one of the available execution workers. This worker then runs the transaction using the Move VM [9] and returns the output objects along with their updated version numbers. Once a quorum of validators has executed a transaction, its effects are final.

In step ❼, the client can assemble the signatures from validators to create an *effect certificate* if these validators represent a stake quorum for the epoch in which the transaction is valid. This certificate can be used as a proof of the effects of a transaction. Finally, step ❽ is for the validator to create a checkpoint.

An epoch finishes when a quorum of stake votes for the epoch to end, at which point reconfiguration is triggered. This includes registration of new validators, stake recalculation, preparation of the new committee, checkpointing and handover. Handover terminates the epoch dropping all the lock stores, meaning that OwnedLock is only released here ensuring that owned objects are only referenced by one transaction per epoch.

2.5. Problem Description

As discussed, blockchains have changed a lot since their inception showing a lot of promise, but still struggle to find mainstream adoption. One of the main reasons for that is their throughput fails to match the requirements for widespread adoption. Recent advances [2, 7, 17, 38] have been able to improve the state of the art, but have mostly focused on the consensus layer as it has historically been the

bottleneck. Thanks to these advances the bottleneck has now been shifted onto the execution layer [22, 24, 31, 32, 33, 36].

Improving the throughput of the execution layer is a difficult task and is usually done through parallelism. The problem is that blockchain workloads are inherently contended [22, 27, 31, 33] which limits the degree to which the workload can be parallelized. Concurrency control then becomes key to ensure deterministic serializability while achieving good performance. Some work has been done to improve the throughput of execution, but performance drops off massively for highly contended workloads.

Our work will focus on improving the throughput of the execution layer for high contention workloads. To do that we will focus on efficiently exploiting the limited parallelism available to the fullest, building on top of the state of the art and incorporating features from related fields. Whilst the resulting throughput numbers might be less eye popping than those of other works that optimize for low to medium contention workloads, the resulting protocol should have greater practical applications.

3

Related Work

Blockchains remain a hot topic of research with the state of the art constantly changing. This chapter surveys the current state of the art research in the field of blockchain as well as adjacent fields that can provide inspiration for our work. It also emphasizes the current state of the art blockchain execution engines, highlighting their capabilities as well as their shortcomings. Finally, it provides a clear gap in the current state of the art that this work looks to address.

3.1. Consensus

The papers discussed in this section relate to consensus, and are included to discuss the recent progress made which has shifted the bottleneck onto the execution layer.

Narwhal and Tusk

Narwhal and Tusk [17] is an efficient byzantine fault tolerant (BFT) consensus algorithm that separates the task of transaction dissemination from transaction ordering. One of its key features is that consensus is achieved on metadata making blocks much smaller, greatly improving performance. It consists of two main parts, first Narwhal is a DAG-based mempool protocol that guarantees optimal throughput under asynchrony and second Tusk which is a asynchronous wait-free consensus protocol that when combined with Narwhal only needs to order block digest certificates, which are much smaller than the complete block. Narwhal and Tusk is one of the recent advances which has helped lift the bottleneck imposed by consensus, as it was able to reach up to 160,000 TPS with about 3 second latency.

Narwhal blocks consist of hashes of batches rather than transactions, where a batch is a group of transactions created by a worker node. This allows for scaling out, as there can be multiple workers per validator creating batches and sharing the hash of that batch with the primary for inclusion in a block. As a result Narwhal is able to increase its throughput linearly with the number of workers each validator has without increasing latency.

Narwhal can be used by any partially synchronous consensus algorithms (such as Hotstuff [2]) to improve their throughput, but works best when combined with Tusk. Tusk is an asynchronous consensus algorithm presented in the paper that improves upon DAG-Rider [25] turning it into an implementable system. Tusk receives the partially ordered DAG that was created by Narwhal and creates a total order without additional communication.

Mysticeti

Mysticeti [7] is a family of DAG-based Byzantine consensus protocols that looks to combine high throughput with low-latency. It introduces two protocols, the first is MYSTICETI-C which is a low-latency consensus protocol, and the second is MYSTICETI-FPC which extends it with a fast path for transactions that can forego consensus. Mysticeti has been adopted in production by Sui replacing Bullshark [38].

The main problem that Mysticeti looks to solve is that of latency. A lot of DAG-based consensus protocols scale well in terms of throughput, but come with high latency (around 2-3 seconds). This is problematic as it leads to poor user experience and prevents low-latency applications from adopting blockchains. The main reason for this latency is that these protocols rely on certified DAGs, which uses consistent broadcast. Mysticeti uses uncertified DAG and as a result it able to commit blocks within 3 message rounds, as opposed to 6 for Bullshark.

Mysticeti operates in a sequence of logical rounds. For every round, each honest validator proposes a unique signed block. A block includes references to blocks from prior rounds, as well as new transactions. Once a block contains references to at least $2f + 1$ blocks from previous rounds, the validator signs it and sends it to other validators. The first link should be to the block from the same author in the previous round. Mysticeti protocols then operate by interpreting the structure of the DAG to reach decisions, and they do so by identifying two main patterns the skip pattern, and the certificate pattern. The skip pattern occurs when at least $2f + 1$ blocks in the following round do not support a block. The certificate pattern is the opposite, in that it is when at least $2f + 1$ blocks in the subsequent round support a block, which can then be certified. These patterns allow for implicitly obtaining certificates that hold the same guarantees as Narwhal.

The evaluation of Mysticeti showed that it was able to reach higher throughput with lower latency when compared to state of the art consensus protocols like Bullshark [38] and Narwhal [17] used in combination with HotStuff [2]. The evaluation showed that under ideal conditions Mysticeti was able to provide sub-second latency with up to 400,000 TPS, which was significantly better than the other protocols. A similar trend could be found when evaluated in the presence of faults. The performance of Mysticeti was also seen in production where it lowered the latency by 4x.

3.2. Deterministic Databases

The papers included in this section represent the state of the art in the field of deterministic databases. While the workloads are very different, deterministic databases also require deterministic serializability and follow the *Order-Execute* architecture. These papers have been included, as improvements made to deterministic database execution can serve as inspiration for improvements to blockchain execution.

DOCC

Deterministic and Optimistic Concurrency Control (DOCC) [20] is a scalable protocol for deterministic databases. It looks to improve the performance of deterministic databases by lazily enforcing the predetermined order. It does so by optimistically executing transactions in parallel and constraining the validation order of dependent transactions. Concretely, it means that a transaction can only perform validation once all previous conflicting transactions have been committed. If the validation step fails for a transaction, then it is immediately retried and can directly be committed after it is done executing. DOCC does not need any prior knowledge about the transactions, instead opting to validate the results of execution. This helps improve the generality of deterministic databases.

One optimization that DOCC makes is to use a multi-version design. Instead of updating a record in place, transactions create a new record for each version. This makes it possible to remove conflicts between a read and a later write, since the older version can always be read.

Another optimization that DOCC makes is to speed up retries through data pre-fetching. Retrying transactions introduces a bottleneck which hurts performance. In order to speed up the process, DOCC introduces record placeholders containing a pointer to the latest record. This helps reduce the time needed to access records, lowering the cost of retrying a transaction.

Gria

Gria [44] is an efficient deterministic concurrency protocol for deterministic databases. Gria organizes transactions in batches, and uses an innovative auto-scaling batch size to adapt to different workloads. It also uses a multi-version structure to avoid write-after-write conflicts. Putting it altogether it is able to achieve good performance, without the need for prior knowledge of the read/write sets.

Gria groups transactions into batches and transactions within a batch are split up into different groups. Each group is executed independently by a worker thread, and transactions in the same group run

sequentially. When all groups have finished executing, each transaction performs cross-group conflict validation. This approach aims at limiting the amount of failing validations by executing large portions sequentially whilst utilizing all the available resources. The initial batch size is configured by the user, but following that the batch size is based on the number of committed transactions in the previous batch. The idea being that a large batch size is more suitable for low-contention workloads, whilst a small batch size works better for high-contention workloads. Having the batch size be dynamic allows it to adapt to the different scenarios.

Gria uses a multi-version structure using a lock-free version chain, to avoid write-after-write conflicts. Each record maintains a version chain organizing versions from old to new through a doubly linked list, with the first version coming from the last snapshot. Versions must be traversed one by one from old to new, and so there is no need for a lock on the whole version chain but only on the previous/next pointer of each version. Since transactions update data to different versions, it removes write-after-write conflicts that can happen when transactions overwrite each other in the commit phase.

Dodo

Dodo [45] is a deterministic concurrency control algorithm for distributed databases. Dodo looks to improve scalability in multi-node and multi-core settings, without the need for any prior knowledge. It does so by processing transactions in batches and applying some optimistic optimizations. Dodo also uses a multi-version scheme (similar to that of Gria [44]) to eliminate the write-write conflict between transactions.

Each batch is divided into three phases: execution, validation and commit. In the first phase worker threads will pop transactions from the global transaction queue and execute them. In the second phase, threads are assigned a validation task in a round-robin fashion. The validation task verifies the transaction for read-write conflicts. In the third phase, transactions that passed validation are committed sequentially until a failed validation is detected. Once that is the case, all later transactions in the batch are aborted and will need to be re-executed in the next batch. Having transactions organized into batches allows Dodo to better make use of the resources available in a multi-core setting.

The first optimistic optimization that Dodo makes is the *lazy decision* mechanism. Dodo assumes that the write set of a transaction will not change from one execution to the next. This means that if a transaction passed validation in the current batch then it can also pass it in the next batch. Dodo uses this assumption to prevent cascading aborts, and only schedules transactions to be validated again in the next batch rather than re-executing the entire chain of aborted transactions.

The second optimistic optimization that Dodo makes is the *early-write visibility* mechanism. Dodo assumes that the access count of records in the write set remains the same between executions. Using this assumption frees up dependent read transactions since they do not need to wait for the transaction to execute. Instead the write is visible to them ahead of time, thus the name of the mechanism.

3.3. Miscellaneous Blockchain Papers

The papers in this section are all related to blockchains, but target different parts of the system. They are all looking to improve performance, but have very different ways of going about it. They have been included as they represent the current state of the art and give an insight into some of the research that is currently being done to improve blockchain performance.

Hyperledger Fabric

Most blockchains have a *Order-Execute* architecture where a consensus protocol orders transactions and propagates them to all peers, which then all individually execute these transactions. Hyperledger Fabric [6] introduces a novel *Execute-Order-Validate* architecture which aims to improve resiliency, flexibility, scalability and confidentiality. This approach is drastically different as it executes transactions before reaching a final agreement on their order, the idea being that it enables concurrent execution of transactions.

One of the main criticisms that hyperledger fabric makes of the *Order-Execute* architecture is sequential execution. In a typical *Order-Execute* architecture, all peers will receive the same block that has been agreed upon and will execute all transactions sequentially to reach the same state. As hyperledger

fabric points out, this is a problem as it introduces a potential performance bottleneck both in terms of latency and in terms of throughput. In order to remove this bottleneck it must be possible to execute transactions in parallel, which is not easily done for smart contracts. This leads hyperledger fabric to look for a different paradigm, where transactions are already executed before consensus.

Hyperledger fabric works as follows, first in the endorsement phase a client sends transactions and each transaction is then executed whilst also recording its output. After that comes the ordering phase which consists of using a consensus protocol to produce blocks, which are then broadcasted to all peers. Each peer then validates the state changes from all transactions and verifies the consistency of the execution. Since validation is deterministic and all peers validate transactions in the same order, they all reach the same conclusion. The advantage of this approach is that not all peers need to execute all transactions, and that transaction execution can be done in parallel. Furthermore, it does not require any prior knowledge about the transactions' read/write sets, as they are discovered during execution. The disadvantage is that if there are a lot of conflicting transactions in the same block it will lead to a lot of transactions being disregarded in the validation step. This means that for high-contention workloads, there would be a lot of transactions executed that end up being disregarded leading to poor throughput. This is an issue as having transactions conflict is much more likely with smart contracts compared to with simple transactions.

Nezha

Nezha [46] is a concurrency control scheme for DAG-based blockchains, that aims to improve on conventional conflict graphs in order to enhance concurrent transaction processing. Having concurrent transaction processing would lead to higher throughput as it could take advantage of modern multi-core architectures. The issue is that processing multiple blocks concurrently will increase the number of conflicting transactions, which causes more problems.

Conflict graphs are often used to represent conflicting transactions, and the partial order between them. The limitation of this approach is that there is still a significant amount of work to go from a conflict graph to a total order on transactions. Nezha uses a different approach mapping transactions to the corresponding address rather than capturing dependencies between pairs of transactions. Using this it creates a new *address-based conflict graph* where edges are now used to represent address dependencies. This graph can then be used to derive a total order between transactions by successively sorting transactions on each address.

OptMe

OptMe [35] is a deterministically orderable concurrency control algorithm, which looks to exploit parallelism among transactions to increase the throughput. Whereas traditional blockchains will execute transactions according to the total order defined by consensus, OptMe instead looks to create a more optimal schedule where as many transactions as possible can run concurrently.

The schedule that OptMe generates maintains the partial order between conflicting transactions that is defined through consensus. In order to do that OptMe must be able to identify conflicting transactions, which it can only do if it knows the transactions' read/write sets. OptMe does not assume that this knowledge is available and instead pre-executes transactions in order to retrieve the read/write sets. Using this information OptMe then constructs a *key-based dependency graph* (KDG), which is essentially the same as the ACG defined by Nezha [46]. The main advantage of using such a graph is that it can be constructed in parallel by using a divide-and-conquer approach. During graph construction, OptMe is able to detect transactions that may violate serializability in an epoch and abort such transactions early. It then uses topological sorting on the graph to come up with the schedule and uses inter-epoch reordering to optimistically reorder the aborted transactions.

OptMe orders transactions into epochs, where transactions from an epoch can only start executing once all transactions from the previous epoch have already been committed. Epochs are made up of multiple so called *sequences*, which are a set of independent transactions. The order of sequences affects the commit order of transactions in the same epoch, whilst transactions of the same sequence can be committed in parallel. The *lifecyle* of a transaction is its start time, which is just its epoch number, and its end time which is defined by both its epoch and sequence number. Two transactions are said to be concurrent if their lifecycles overlap. Consequently, two transactions of the same epoch

are concurrent and transactions from different epochs are not. Dividing the schedule up as such is useful for ensuring that conflicting transactions do not interfere with one another. The main thread can then evenly distribute transactions within the current epoch to multiple threads for concurrent execution. After this execution OptMe checks if the read and write sets of the transaction match the estimate gotten from pre-executing the transaction. If the sets are equal then the transaction is considered valid and can be committed in parallel. If the sets are not equal OptMe will check if the actual sets are disjoint from those of other transactions in the epoch, in which case the transaction can still be committed in parallel. If there are conflicting transactions OptMe will mark the transaction as invalid, and simply executes all invalid transactions in a sequential manner.

Anthemius

Anthemius [31] recognizes that the performance of parallel execution engines largely depends on the nature of the workload. This is the case because conflicting transactions require concurrency control which all experience drops in performance under high contention. As a result of that, a single popular application can bottleneck the execution and limit the throughput. Anthemius is a block construction algorithm that looks to optimize the throughput of parallel execution by constructing blocks that can be executed in parallel efficiently. To achieve this, Anthemius looks to make block assembly sensitive to transaction dependencies and execution complexity, charging clients more for accessing popular resources. Anthemius is modular and can seamlessly be integrated into any blockchain, fitting in right before consensus as shown in Figure 3.1.

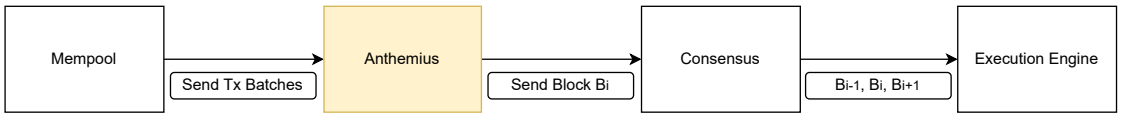


Figure 3.1: Integrating Anthemius, taken from [31]

In order to construct what Anthemius calls ”good blocks” that can be executed in parallel efficiently, Anthemius needs some information about the transactions. These so called *hints*, should give an idea of what resources a transaction will access during execution as well as an estimation of their execution time. This information does not have to be complete however and Anthemius does not assume their correctness either. Incorrect information will degrade the performance of the system, as it will either lead to re-execution if the execution engine is optimistic, or it will lead to aborted transactions if the execution engine is guided.

Anthemius uses two parameters to quantify the execution complexity of a block. The first one is *gas* which represents the computational power needed, similarly to in Ethereum. The second parameter is a concurrency parameter *c* representing the systems ability to parallelize. Combining the two gives that the maximum capacity of each block is $c \cdot gas$.

Anthemius consists of two main elements, the *batch handler* and the *batch scheduler*. The *batch handler* polls batches of transactions from the mempool and gives them to the *batch scheduler*, which in turn tries to include them into the current block. The *batch scheduler* provides feedback about the success rate, which is used to adjust the inclusion policy. This looks to prevent creating blocks that are too small or blocks that are too difficult to schedule.

Scheduling interdependent transactions is an NP-complete problem, but constructing near optimal solutions can be done in polynomial time. This remains too costly to have on the critical path of consensus, as the gains may not outweigh the construction time. Anthemius does not look to achieve a near-optimal scheduler but instead focuses on preventing popular applications from creating a bottleneck. The secondary objective is to delay transactions that access multiple hot resources, as it makes it harder to schedule subsequent transactions. Anthemius is able to do so in linear complexity relative to the number of transactions per block, which experimentally resulted in a speed-up of up to 240% compared to the base parallel execution.

3.4. Blockchain Execution

The papers in this section are the most relevant, as they all relate to blockchain execution. They vary a lot in their approaches, ranging from OCC to sharded execution. These papers were included as they represent the state of the art for blockchain execution engines.

ParBlockchain

ParBlockchain [5] adapts the Order-Execute paradigm to permissioned blockchains to create what they call the *OXII* paradigm. In this paradigm consensus establishes the contents of a block and also creates a dependency graph defining the causal order between conflicting transactions. For this to be possible, ParBlockchain assumes that the read/write sets of transactions are either pre-declared or that they can be obtained via static analysis. Following consensus, the block along with the dependency graph is multicasted to all executor nodes. These nodes are then responsible for executing the transactions, but crucially not every executor needs to execute every transaction.

OXII consists of a set of nodes in an asynchronous distributed network where every node is connected to every other node with a point-to-point connect. There are three types of nodes: clients, orderers and executors. Clients send operations that are to be executed by the blockchain. Orderers are responsible for validating these requests and ordering them into a block. Any consensus protocol can be used here, as ParBlockchain uses a modular architecture. Executors should then execute and validate transactions, update the ledger and blockchain state, and multicast the state after execution.

In ParBlockchain executor nodes are only responsible for one application and only execute transactions related to that application. A transaction can only be executed if all its ancestors in the dependency graph are committed. This means that upon receiving a block an executor node can be in one of three scenarios. The first scenario is that there are no transactions for the application that the node is responsible for, in which case the node will remain in a passive state. In this case it will only listen to commit messages and update the ledger state once it has received sufficient commit messages for a transaction. The second scenario is that there are transactions for the application, but that these transactions only depend on transactions related to the same application. In this case the executor node can simply execute the transactions according to the dependency tree, independent of other executors. In this case, it will multicast a single commit messages once all the transactions are done executing. The third scenario is that there are transactions for the application and they depend on transactions that belong to a different application. When this is the case the execution of these transactions will have to wait on the result of the execution by the other node. To optimize the third scenario executor nodes in ParBlockchain will check if another application depends on the execution result of one of its transactions. If it is the case, the node will multicast a commit message immediately after executing the transaction to minimize how long nodes are left waiting.

The architecture of ParBlockchain closely ties its performance to the way in which transactions are dependent. ParBlockchain will struggle in a scenario where two executors are responsible for different applications, but the dependency tree shows a lot of dependencies between their transaction. That is because the executors will have to communicate a lot with one another and will often be left waiting for messages. This causes a lot of overhead and also leaves executors idle a lot. Another problem is that it could be that one application experiences a big spike in traffic, which will lead to unequal load distribution among executors. This results in the sharding not having the horizontal scalability that was hoped for. Nevertheless, ParBlockchain gives an idea of what sharded execution could look like, the way in which nodes can communicate with each other and the challenges that come with it.

PEEP

PEEP [15] is a parallel execution engine for permissioned blockchains that implements pessimistic concurrency control. It follows the *Order-Execute* architecture, meaning that blocks are only executed after consensus has been achieved. Transactions are executed concurrently through the coordination of the scheduling layer, and updates are then applied to the underlying state trie. The scheduling is deterministic ensuring that different nodes obtain the same result without having to communicate with one another. In order for this to work, PEEP needs to know the read/write sets of all transactions before execution. To this end, PEEP assumes that the read/write sets can be known in advance via execution simulation or static analysis.

In order to achieve parallel execution PEEP contains 2 key components, the lock manager and the thread scheduler. The lock manager is implemented on a single thread and grants locks to transactions in a predefined order. A transaction becomes *active* once it has acquired all the locks it needs and can then be delivered to the buffer of the thread scheduler for execution. The thread scheduler is then tasked with assigning transactions to worker threads. Transaction updates are written into a buffer which is later flushed to the state trie using a parallel update. All locks are released once execution finishes making them available for other transactions.

PEEP has two main limitations, the first one being that it requires complete knowledge of the read/write sets. This is hard to obtain and often involves having to overestimate the sets in practice. The second limitation is that acquiring and releasing locks introduces a significant overhead. For high contention workloads this overhead could nullify any gains made from parallel execution.

Block-STM

Block-STM [23] is a parallel execution engine for smart contracts that uses optimistic concurrency control (OCC). It is currently being used on the Aptos [43] blockchain in production proving its practical efficacy. The result of the parallel execution is equivalent to the sequential execution of transactions as defined by the total order of transactions in the block, but each transaction may have to be executed several times.

One of the biggest selling points of Block-STM is that it can dynamically detect dependencies which improves the generality of the protocol. Block-STM optimistically executes transactions gaining information about the read/write sets. This provides two main advantages over simulated execution, the first one being that if there are no conflicts for the transaction then it will only execute once and the second advantage is that the write-set estimate is going to be more accurate, as it is more up-to-date.

Block-STM starts off by receiving a block containing n transactions, which defines a total order $tx_1 < tx_2 < ... < tx_n$. The goal is to execute these transaction efficiently through parallelism whilst reaching the same final state as sequential execution. In Block-STM a transaction may be executed several times, and each execution is called an *incarnation*. An incarnation is *aborted* if it needs to be re-executed, incrementing the *incarnation number*. A *version* is a tuple consisting of the transaction index and an incarnation number. In order to enable concurrent read and writes, Block-STM uses an in-memory multi-version data structure `MVMemory`, which for each memory location stores the latest value written per transaction as well as the associated transaction version. When executing, a transaction will first look to read values from `MVMemory` written by the highest transaction that appears before it, and will resort to reading from storage if it there is no entry. This means that the first transaction in a block is going to be reading every value from storage. For each incarnation, Block-STM maintains a read set containing the memory locations read with their corresponding version, and a write set containing the updates made by the incarnation as a (memory location, value) tuple. This information is needed to detect conflicts when validating a transaction. Validation happens after execution, with a validation task being created once execution finishes. These validation tasks can also be executed in parallel, but when a transaction fails validation (aborts), then all higher transactions have to be validated again. Validation consists of checking that the current most up to date reads match up with the reads that incarnation used for execution. If the validation task fails then all locations in the write set are marked with `ESTIMATE`, to indicate that the value is out of date and likely to be overwritten. A dependency is indicated by having a read to a memory location marked as `ESTIMATE` by a lower transaction. One optimization that Block-STM implements is to immediately abort a incarnation when such a dependency is detected, since we already know that re-execution will be needed. Using the write set from the last failed incarnation allows Block-STM to discover dependencies between transactions efficiently through optimistic execution. Another optimization is to postpone execution of transactions that read a value marked as `ESTIMATE` until the blocking transaction has been re-executed, avoiding unnecessary re-executions. Once there are no more tasks to run and all transactions have been validated, then Block-STM will commit the whole block at once in order to reduce the amount of synchronization involved.

The scheduling of tasks is managed by the collaborative scheduler, and is crucial to the overall performance of the system. Being that the resulting state should be identical to that of sequential execution, the scheduler will prioritize tasks associated with lower transactions. This does not mean that tasks will be executed sequentially however, as the scheduler will look to utilize all the available threads. In

order to track the tasks that can be scheduled, the scheduler maintains an ordered set V containing validations tasks and an ordered set E containing execution tasks. Upon first receiving a block, the set V will be empty since validation tasks are created as a result of execution and the set E will contain the initial incarnation of all transactions.

To better understand how the protocol works, let us go through an example scenario. Suppose that the block that has been agreed upon by consensus consists of 3 transactions ordered $tx_1 < tx_2 < tx_3$ and that there are two threads available. Initially, we have $V = \emptyset, E = \{(tx_1, 1), (tx_2, 1), (tx_3, 1)\}$ and because the scheduler prioritizes lower transactions it will schedule tx_1 and tx_2 for execution. In this scenario both transactions read from the same memory location A, then because `MVMemory` is initially empty, both transactions have to read from storage. Say that tx_1 writes to location A whilst tx_2 writes to a different location B. In our scenario the execution of tx_2 finishes before that of tx_1, creating a validation task for tx_2 which is scheduled. It also keeps the read set for validation purposes and applies the write set to `MVMemory` (shared memory).

In this scenario, the validation of tx_2 successfully passes before tx_1 has finished execution, and so the scheduler starts executing tx_3 that reads from location C and writes to location C. Both tx_1 and tx_3 finish execution, creating two new validation tasks. Since tx_1 wrote to a location that it previously did not write to (since it is the first incarnation), it also creates a new validation task for tx_2. As mentioned earlier the scheduler prioritizes based on index, so it decides to run the validation tasks for tx_1 and tx_2. The validation task for tx_1 passes since all its reads are up to date, but the validation for tx_2 fails since it read an outdated version of A and so the incarnation is aborted. This means that the location it wrote to in `MVMemory` is marked as `ESTIMATE` in case there are higher transactions reading that value. The scheduler then re-executes tx_2 and runs the validation task for tx_3. The validation task for tx_3 passes since all reads are still up to date and the execution of tx_2 results in the creation of a validation task for tx_2. Since the second incarnation of tx_2 did not write to any new location, there is no need to re-validate tx_3 and so once the validation for tx_2 passes the entire block can be committed. This scenario is summarized in Figure 3.2.

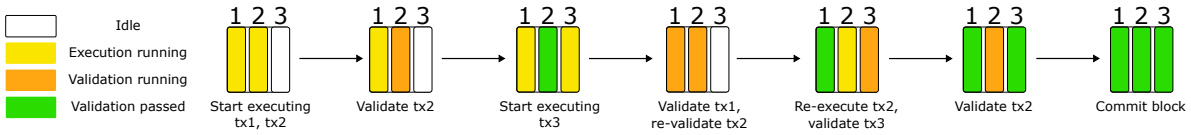


Figure 3.2: Block-STM example scenario

The scenario described above and shown in Figure 3.2 shows the different states incarnations can be in. It also shows how the scheduler works to prioritize tasks associated with a lower transaction. Even with that however, the scenario demonstrates how tasks will not be executed strictly sequentially as task duration is heterogeneous and Block-STM looks to use all of the available resources.

Through this optimistic execution Block-STM is able to achieve outstanding results on a set of independent transactions, as it is able to fully parallelize them and not have to abort any. Under high contention workloads however this could lead to a lot of aborts resulting to a lot of duplicate work being done. Furthermore, this is amplified by cascading aborts in the presence of a long chain of dependencies.

RapidLane

RapidLane [29] is an extension for parallel execution engines aiming to optimize performance for contended workloads. It does so by allowing the engine to capture computations in conflicting parts of transactions and defer their execution until a later time. RapidLane is integrated into Block-STM [23] and deployed into production on the Aptos [43] blockchain.

RapidLane introduces *deferred objects* which are objects expected to be frequently mutated by transactions. They are useful for eliminating read-write conflicts by using a heuristic to predict whether or not a transaction succeeds and logs the deferred computation. Then during the commit phase, the execution engine calculates the correct value and validates the decision. If the decision was correct the transaction can be committed, otherwise it needs to be re-executed.

Another contribution made is the enhancement of Block-STM with a rolling commit mechanism. Block-

STM uses a lazy commit mechanism, where the blockchain state is only updated once per block in order to reduce the synchronization overhead. This does not work well with RapidLane however as values are only revealed in the commit phase, and so if the commit is only done once at the end, revealing a value would require traversing the entire block accumulating deltas. Furthermore, delaying commits as such increases the length of the speculative chain which degrades the predictions. To address this, RapidLane proposes a novel rolling commit mechanism which enables the system to determine when a transaction will no longer be re-executed and can thus already be committed. A transaction tx_i can be committed at time T if tx_{i-1} is committed and the latest validation for tx_i succeeded.

RapidLane showed some promise, but the overall performance of the system heavily relies on the performance of the heuristic, as incorrect decisions lead to sequential re-execution in the commit phase. This means that a poor heuristic could cripple the performance of the system. Another issue is that RapidLane restricts the types that can be wrapped into a deferred object to integer counters and small-sized strings (up to 256 bytes). This reduces their applications and limits the practical impact of RapidLane.

Chiron

Chiron [32] aims to accelerate parallel execution of smart contracts on straggler nodes by using hints extracted during the normal execution at non-straggling nodes. The reason for doing that, is that blockchains like Sui [42] or Aptos [43] only allow stragglers to catch up by downloading signed checkpoints. This is problematic, as it lowers the security of the system by making it such that a single honest party is no longer able to act as a whistle-blower. Chiron looks to make it possible for straggler nodes to catch up through guided parallel execution rather than through signed checkpoints.

Chiron is built on top of Block-STM [23], and so transactions are first executed before later having the execution result validated. In order for straggling nodes to catch up, Chiron execution must be able to consistently outperform that of Block-STM which it does through guided parallel execution. The idea is that straggling nodes can request hints from active nodes that have already executed a block, and leverage these hints to avoid conflicting transactions executing at the same time eliminating all re-executions. These hints do not need to be complete and can also be incorrect. This will not threaten the safety of the system, it will only affect its performance. If a transaction fails validation indicating an incorrect hint, Chiron falls back on Block-STM for the remaining transactions. It will also adjust the trust it has in the node from which it got the incorrect hint, so that over time it can find an honest non-straggling node to extract hints from.

The hints obtained contain the read/write sets of transactions to identify dependencies, along with the gas representing the complexity of the execution and the sum of the gas costs of the chain of parent transactions. Chiron uses this information to build a dependency graph and also maintains two queues, one containing priority transactions and another containing standard transactions. A transaction that has no parents in the dependency graph is ready to be executed and can be added to one of the task queues. If the transaction has children in the graph, indicating that other transactions depend on it, then it is added to the queue for transactions with higher priority. Chiron will first look to execute the transactions of higher priority, and only once that queue is empty will it execute those in the other queue. Once a transaction is executed it resolves the dependencies other transactions have on it. This way Chiron is able to resolve dependencies earlier, increasing the parallelism available in the remainder of the workload. The experimental results showed a speed up of up to 30% when compared to plain Block-STM, but performance is heavily reliant on the quality of the hints.

Shardines

Shardines [18] is an execution engine that looks to achieve horizontal scalability. It identifies three layers of a blockchain: consensus, execution and storage, with the goal to scale each of them independently. Previous work has already been done to scale out consensus horizontally via Narwhal-based quorum store [16] and sharding of the storage layer has also already been deployed. Shardines looks to scale execution horizontally by having multiple execution workers each running Block-STM [23].

The challenge of building a sharded execution protocol revolves around conflicting transactions, as they cannot be executed in parallel. Instead, access to the shared state must be serialized to maintain consistency which imposes a bottleneck on execution. While conflicting transactions on the same node

are taken care of by Block-STM, cross-node conflicts are not and instead require cross-node communication. In order to reduce cross-node communication, Shardines tries to partition transactions across shards in a way that reduces cross-shard communication. This is done via hyper-graph partitioning where each transaction is a node and each resource is a hyperedge connecting all transaction that access it. Minimizing cross-shard communication then reduces to minimizing the number of shards that a hyperedge spans, which is called fanout. Resources with a fanout of 1 do not need cross-shard communication since all the conflicting transactions are dealt with by Block-STM. In order to construct such a hypergraph, Shardines needs to know the read/write sets of all transactions ahead of time, which they get through static analysis. This process runs for each block, meaning that shards are dynamic in that the resources each shard accesses varies from one block to the next. The number of executor shards is configurable but static, meaning that it cannot grow if there is a sudden spike in traffic.

Another vital aspect of Shardines is the novel micro-batching and pipelining they use throughout the entire execution flow. The idea is to split a block into smaller batches and have them pipeline through the various stages of executions. Shardines identifies 5 stages of execution and makes it such that not all 5 batches are ever in the same stage. This helps mitigate the delays caused by serializing/deserializing large data chunks as well as network delays from transmitting large packets.

Shardines is able to achieve impressive results for fully parallelizable workloads, as it is able to scale out horizontally. However, it is not efficient for high contention workloads as it either results in limited sharding opportunities or in a lot of cross-shard communication.

Pilotfish

Pilotfish [26] is a blockchain execution engine that was built specifically for lazy blockchains. It is built in such a way that the state is distributed among several workers, allowing it to be scaled out horizontally by adding more workers. One of its major selling points is that not every machine needs to hold all the data, continuing the separation between data dissemination and consensus which was a major selling point of lazy blockchains. Pilotfish was built for the Sui [42] blockchain, and so it assumes that the read/write set of transactions are known ahead of time. Execution results in the same state as if all transactions had been executed sequentially ensuring that all validators end up in the same state.

The overall architecture is that each validator replicates the state of the system represented as a set of objects, and each validator is composed of multiple workers. One of these workers serves as a primary and participates in consensus. A validator also has several *SequencingWorkers* that fetch and persist client transactions in batches. Consensus can then be achieved on batch digests. A validator also has several *ExecutionWorkers*, where each worker stores a subset of the state and only executes a subset of transactions. These different components can run on dedicated machines or be collocated with other components.

Execution is triggered after consensus with the primary sending the committed sequence to all SequencingWorkers and ExecutionWorkers. The SequencingWorkers then observe the commit sequence and load all batches from storage that they hold. They then parse each transaction of the batch to determine which objects it contains and compose a ProposeMessage for each ExecutionWorker. The ExecutionWorkers also await a ProposeMessage from each SequencingWorker and parse every transaction included in that message to extract all the objects of the read and write sets. This information is used to keep track of all dependencies between transactions. ExecutionWorkers maintain a list of pending transactions for each object for which they are responsible for. A transaction is then ready to be executed once it reaches the head of all the pending lists it is in. The ExecutionWorker will then load all the objects the transaction reference and send them along with a ReadyMessage to the dedicated ExecutionWorker which is responsible for executing the transaction. This dedicated ExecutionWorker is chosen deterministically based on the number of objects of the transaction it already contains, in order to limit the amount of data traveling over the network. Once that worker has received a ReadyMessage from all the relevant ExecutionWorkers it will execute the transaction and send a ResultMessage to all ExecutionWorkers containing the new state of the objects referenced by the transaction. Upon receiving this message a worker will update each object in their local object store, remove all occurrences of the transaction from the pending lists and trigger the next execution.

Pilotfish only supports crash fault tolerance, and requires each validator to dedicate multiple machines

for each ExecutionWorker. This internal replication means that when a crash inevitably occurs, Pilotfish can continue its operation. In order to support the crash of F workers, Pilotfish needs to have $N = 2f + 1$ workers. A *cluster* is then formed of one instance of every worker, whilst a shard is all the instances of the same ExecutionWorker. During normal operation workers only communicate with their own clusters, and inter-cluster communication is only related to checkpointing. However if a worker crashes, then the other workers in the cluster can switch to communicating to the same worker in a different shard. There are two recovery mechanisms, reconfiguration and checkpoint synchronization. In reconfiguration, when a worker crashes the workers that were dependent on reads from it will look for a different worker from that shard and get the reads from them instead. When reconfiguration fails checkpoint synchronization needs to be performed for all workers. In the extreme scenario where all ExecutionWorkers of a cluster crash, the cluster can be recovered by booting a new cluster with the same peers set.

ParallelEVM

ParallelEVM [27] recognizes that blockchain workloads suffer from the *hot spot problem* that results from highly skewed data accesses. To address this, it 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 and has a greater impact for high contention workloads. The issue with this approach is that it relies on the assumption that conflicts only affect a few operations. If that were not the case then the increased parallelism would be very limited. ParallelEVM was developed for Ethereum where this assumption holds, but it remains unclear whether it holds for other blockchains and for the object data model.

3.5. Summary of Related Works

Before going into our contributions its good to get an overview of the related work covered, and see how they compare to one another. Table 3.1 does exactly that, with the last line showing how our protocol (NEMO) compares to the work discussed. The table also contains Sui execution as an entry to highlight its characteristics even though it was not discussed in this chapter, since it was previously discussed in Section 2.4.

	Execution Paradigm	*Transaction Model*	*Application*	*OCC/PCC*	*Efficient under high contention*
DOCC [20], Gria [44], Dodo [45]	OE	-	Deterministic Database	OCC	✔
Hyperledger Fabric [6]	EOV	UTXO	Blockchain Paradigm	OCC	✖
Nezha [46]	OE	Account	Scheduling	Hybrid	-
OptMe [35]	EV	Account	Scheduling	Hybrid	✖
Anthemius [31]	OE	Model Agnostic	Block Construction	Hybrid	-
ParBlockchain [5]	OXII	Account	Blockchain Execution	PCC	✖
PEEP [15]	OE	Account	Blockchain Execution	PCC	✖
Block-STM [23], RapidLane [29]	OE	Resource	Blockchain Execution	OCC	✖
Chiron [32]	OE	Resource	Straggler Execution	Hybrid	✔
Shardines [18]	OE	Resource	Blockchain Execution	Hybrid	✖
Pilotfish [26], Sui [10]	OE	Object	Blockchain Execution	PCC	✖
ParallelEVM [27]	OE	Account	Blockchain Execution	Hybrid	✔
NEMO (this work)	OE	Object	Blockchain Execution	OCC	✔

Abbreviations: OCC = Optimistic Concurrency Control; PCC = Pessimistic Concurrency Control; EOV = Execute-Order-Validate; OE = Order-Execute; EV = Execute-Validate; OXII = Order-Execute-V2; UTXO = Unspent Transaction Output

Table 3.1: Comparison of Related Work

4

Designing NEMO

This chapter presents the NEMO protocol for blockchain execution. It begins with the intuition of the protocol, its philosophy and what it is trying to accomplish. After that it goes into detail about all the optimizations made and their purpose. Finally, it ends with a comparison between the lifecycle of a transaction in NEMO and in Block-STM [23].

4.1. Intuition

Blockchains still lack the necessary throughput to support mainstream adoption, with execution being the current bottleneck [22, 24, 31, 32, 33, 36]. Exploiting the parallelism in smart contract executions is a clear way to increase throughput of execution [22, 33]. Current approaches have looked to do this, but struggle in scenarios of high contention. This is problematic as blockchain workloads are naturally highly contented [22, 27, 31, 33], which limits the practical application of blockchains.

In order to execute transactions in parallel, there must be some sort of concurrency control implemented to deal with conflicting transactions. This is either done by executing conflicting transactions in order using pessimistic concurrency control (PCC) or by validating execution results when using optimistic concurrency control (OCC). As discussed in Section 2.2, both approaches have their limitations when it comes to high contention workloads. The advantage that OCC has however is that it is able to use all of the parallelism available in the workload. This is critical for high contention workloads, as there are limited opportunities to parallelize the workload. In fact, previous work has shown that the performance obtained from lock-free execution is better than that obtained from lock-based execution under scenarios with hotspots [47]. The shortcomings of OCC are that this optimism often leads to frequent re-execution, which results in a lot of duplicate work being done. In order to achieve the best performance under high contention, NEMO must take advantage of every opportunity to parallelize whilst limiting redundant work.

This is the philosophy of NEMO, to be cautiously optimistic in order to avoid the effect of cascading aborts that OCC protocols suffer from under high contention. To achieve this NEMO looks to use all of the available information in order to intelligently schedule transactions, avoiding known conflicts whilst utilizing all of the available resources. NEMO optimizes for high contention scenarios and is willing to trade some performance under low contention for better performance under high contention.

4.2. System Model

NEMO uses Block-STM [23] as a starting point, and as such it has the same system model. Similarly to Block-STM, NEMO is a blockchain execution engine that uses OCC, but is built targeting the object model. NEMO takes in the output of consensus and executes the transactions to update the state. The input that NEMO takes should be an ordered set of transactions. NEMO then executes the transactions in parallel to improve the throughput, but yields the same result as sequential execution.

4.3. NEMO Details

This section explains how the NEMO protocol works, how it optimizes for performance under high contention and what it is looking to achieve.

4.3.1. Overview

NEMO's overall approach is to build on top of Block-STM, but to be less optimistic in order to limit redundant work and avoid cascading aborts. The core architecture remains the same and is depicted in Figure 4.1.

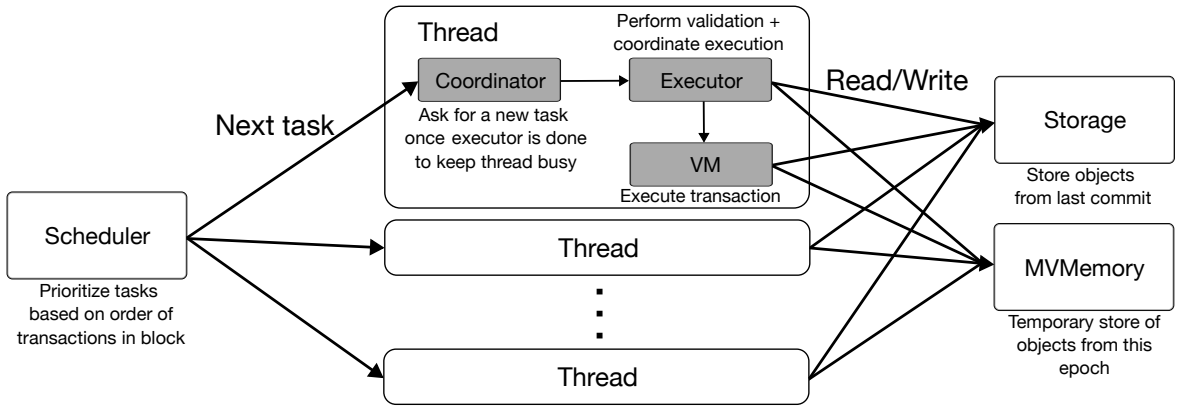


Figure 4.1: Block-STM Core Architecture

The scheduler is shared across all threads and is responsible for giving tasks to the different threads. In Block-STM, it prioritize tasks based on the order of transactions as defined by the block, whilst also trying to keep all the threads busy at all times by assigning them a task as soon as one is available. There are two types of tasks that a thread can be given: a transaction execution task or a transaction validation task, neither of which can be aborted. For an execution task a thread will try to read all inputs from the Multi-Version Memory (`MVMemory`) store, but it could be that there is no entry in which case the thread needs to read the last committed version from storage. All writes are done to `MVMemory` and the final value at the end of the epoch is committed to storage. For validation tasks the thread reads the latest available version of each input, similarly to an execution task, and then compares that version with the version that was read when the transaction was executed, ensuring that all inputs are still up to date. The key is that `MVMemory` is shared across all threads and so when validation fails for a transaction, all entries written by that transaction are marked as `ESTIMATE` to indicate to other transactions that they should not read this value. The scheduler also tries to keep track of dependencies to avoid known conflicts that would likely lead to re-execution.

In order to adapt Block-STM to the object data model changes have to be made in some of the components. The biggest change that has to be made is in `MVMemory` which needs to be altered to now store multiple versions of objects. Similarly, storage also has to be altered to work with the object model and has to support the different kinds of objects. The VM component that actually executes the transactions also needs to be modified to work with objects. The remainder of the logic relating to scheduling and coordination can remain the same.

In the following sections we will describe the different optimizations that NEMO does and what they are trying to achieve. All these optimizations target high contention scenarios and are looking to improve the throughput of the system.

4.3.2. Greedy commit rule

The first optimization that NEMO can make is a direct consequence of using the object model. As discussed earlier in Section 2.4, Sui guarantees that every owned object will only be used by one transaction per epoch. This means that access to a owned object at execution time is guaranteed to be conflict free. Consequently, a transaction that only uses owned objects is guaranteed to be independent of all other transactions which means it will never fail the validation step and only needs to be executed once.

Knowing this we can skip validation altogether which we call the *greedy commit rule*. This means that transactions that do not interact with the shared global state can directly be committed after execution. This provides a true fast-path for such transactions, as Sui already allows such transactions to bypass consensus.

High contention workloads will not contain many such transactions if any, which means that this optimization is unlikely to have a significant impact for such a scenario. Nevertheless, this optimization is essentially ”free” and should have some impact for low contention workloads. Furthermore, this also helps improve the latency of transactions that do not interact with the shared state and also opens up opportunities for future work which we will discuss in Chapter 7.

4.3.3. Limit 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. This is further exacerbated by the fact that tasks cannot be aborted, and occupy a thread for a significant portion of time. The key to performance when using OCC under high contention, is to limit the number of re-executions and to avoid prematurely triggering re-execution if it is likely to fail validation later on.

DOCC [20], which was discussed in Section 3.2, is an OCC protocol for deterministic databases. One of its characteristics is that a transaction is only validated once all previous write transactions it depends on are committed. This is to avoid having validation fail too often which then in turn triggers re-execution. Having this system in place effectively means that a transaction can be executed at most twice assuming that all read/write sets stay the same between different executions of the same transaction.

This logic is very restrictive and can lead to a lot of transactions waiting on a single re-execution. `NEMO` adapts this logic to prevent it from creating a bottleneck, but still limits premature re-executions. One of the issues with limiting when validation can occur, is that it delays finding out that a transaction will require re-execution and is used as a proxy to prevent premature re-executions. `NEMO` looks to validate earlier, in a similar manner to Block-STM, but make it so that a failed validation does not automatically trigger an immediate re-execution. This way we can combine the best of both worlds, and have early knowledge of failed validations without premature re-execution. To achieve this, `NEMO` needs to extract more information about dependencies from execution.

Block-STM is only able to detect dependencies if execution read in an object marked with the `ESTIMATE` flag. This prematurely aborts execution and marks the dependency. The scheduler then ensures that the transaction is not re-executed until the transaction it depends on has successfully been re-executed. This optimization is useful and helps prevent unnecessary re-executions, but does not go far enough.

`NEMO` looks to extract knowledge about the dependencies even when execution succeeds. This information can then be used in case the transaction fails validation and requires re-execution. Prior to triggering this re-execution `NEMO` will check whether a transaction has any dependencies, in which case it will wait for those transactions to be re-executed first. This way `NEMO` is able to avoid prematurely re-executing a transaction that later gets re-executed again. This optimization will have a higher impact in high contention scenarios, as those have a lot more dependencies between transactions which results in more frequent re-execution.

Another optimization that `NEMO` makes in order to limit re-executions, is to only resolve dependencies once the blocking transaction passes validation unlike Block-STM that resolves dependencies after a successful execution. This will make it so that transactions are blocked for longer, but will mean that once a dependency is resolved it is less likely to cause issues later on. This is to avoid having long chains of dependencies re-execute several times, resulting in recurring cascading aborts.

4.3.4. Incomplete hints

One of the main advantages of using the object model is that it makes it easier to identify what resources a transaction uses [10]. This is because every object has its own globally unique identifier which is used by transactions to refer to shared objects [42]. This way it is possible to statically detect the exhaustive sets of objects that a transaction may use. However, it is not possible to know exactly which objects will actually be used during execution as that depends on the transaction logic and the global state

at the time of execution. This leads Sui to use the exhaustive set of objects as the read/write sets of a transaction in order to avoid any potential conflicts. The issue with this approach is that it is overly pessimistic and fails to fully utilize the available parallelism. This is especially problematic for high contention scenarios since there are already limited opportunities for parallelism.

Block-STM goes the other way, assuming it knows nothing about the read/write sets of all transactions and that all transactions are independent. This optimistic approach ensures that Block-STM takes advantage of all the parallelism opportunities present in the workload, but while it may work well for low contention workloads it will lead to a lot of re-execution under high contention. This approach also fails to utilize the information that is statically available when using the object data model.

`NEMO` looks to strike a happy medium between the two, where it is able to leverage partial knowledge about the transactions' read/write sets whilst remaining optimistic and taking advantage of every opportunity to parallelize. To achieve this, `NEMO` assumes that it is possible to statically obtain partial information about the read/write sets thanks to the object model. Unlike Sui's prior knowledge however it should not be exhaustive, and instead should only contain objects that are guaranteed to be used during execution. If no such knowledge is available then `NEMO` will fall back on the logic of Block-STM. Obtaining such partial knowledge should be doable in practice thanks to the object model and `NEMO` can then utilize this information to avoid known conflicts.

To illustrate how the prior knowledge that `NEMO` looks to use differs from that of Sui, let us look at the example logic shown in Figure 4.2.

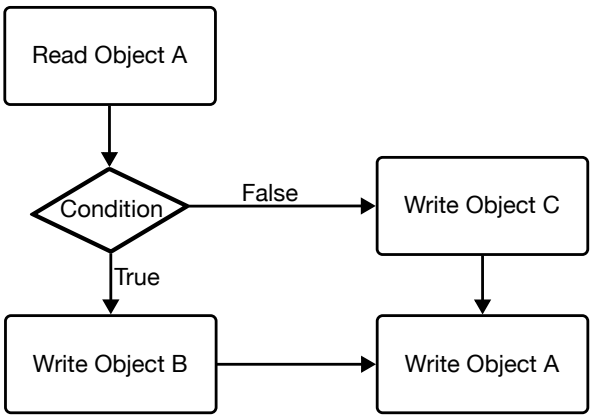


Figure 4.2: Example of Potential Smart Contract Logic

In this example the condition is based on the value read for object A and affects what object the transaction writes to. In such a case, Sui would have statically extracted the following sets: $read = \{A\}$, $write = \{A, B, C\}$ in order to avoid any possible conflicts during execution. The prior knowledge that `NEMO` is looking for on the other hand would be: $read = \{A\}$, $write = \{A\}$, as those do not depend on the condition and are guaranteed to be needed for execution.

`NEMO` then uses this prior knowledge to identify dependencies between transactions before executing any transactions. It does so by pre-processing all the transactions in the block and marking all known write entries as `ESTIMATE` to indicate to later transactions that we know a write will happen here. It then goes through all the objects in the prior knowledge read set and checks if it reads an `ESTIMATE` marker. If it does, then a dependency has been uncovered and the transaction must wait for it to be resolved before it can be executed. This way it is able to execute known conflicting transactions in order. After execution the dependencies are updated using the information extracted (as previously explained), since that information is more up to date than the prior knowledge. This will have the greatest impact for high contention workloads, as optimistically assuming all transactions are independent will result in the most problems for such a workload. Furthermore, pre-processing all transactions has a cost and has to be done sequentially, so in order for it to be worthwhile it must identify a significant amount of conflicts. When this tradeoff becomes worthwhile is hard to tell, and is something we look to answer in Chapter 6.

4.3.5. Priority Scheduling

The Block-STM scheduler prioritizes tasks purely based on the transaction index. This logic is simple and prioritizes earlier transactions to resolve dependencies through execution. The issue with this approach is that it fails to use all the available information. The scheduler could use the information it has about dependencies obtained either through execution or via incomplete hints, to guide execution. This way a later transaction on which other transactions depend could be scheduled before a earlier transaction that no transactions depend on.

Chiron [32], one of the papers discussed in Section 3.4, showed how guided execution built on top of Block-STM could look and managed to achieve up to 30% speedup. Chiron was made to help straggler nodes catch up with the help of hints obtained from validators that are ahead. It then uses these hints to perform guided execution, but falls back on Block-STM execution the moment one transaction fails validation. This means that in order for Chiron to achieve better performance than Block-STM, it requires accurate and complete hints to avoid any validations failing.

NEMO takes inspiration from this approach, but unlike Chiron it never falls back on Block-STM and also uses priority scheduling for both execution and validation tasks. In NEMO, the scheduler maintains a priority queue of tasks where the score associated with a task is the number of direct dependencies. The order of tasks is based on descending score then followed by ascending index in case of a tie. Whenever a thread asks for a new task, the scheduler will pop the task at the front of the priority queue. New tasks have to go through the scheduler, that inserts it into the priority queue. The scheduler also maintains a set of the tasks currently in the queue in order to avoid inserting a duplicate task. It is worth noting that in the case where no prior knowledge is available all transactions will initially have a score of 0, and so the priority queue ordering will behave in the same manner as Block-STM. If there is some knowledge of dependencies available, then this will prioritize executing the blocking transactions first which should make the remainder of the workload more parallelizable.

To illustrate how this priority scheduling can help improve the performance, let us take the scenario depicted in figure 4.3.

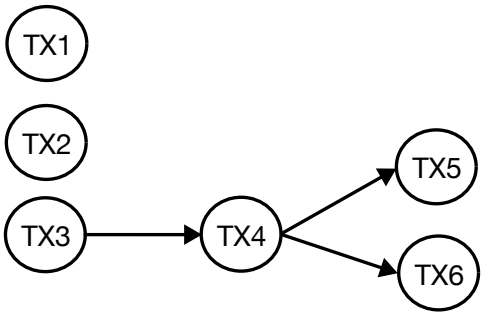
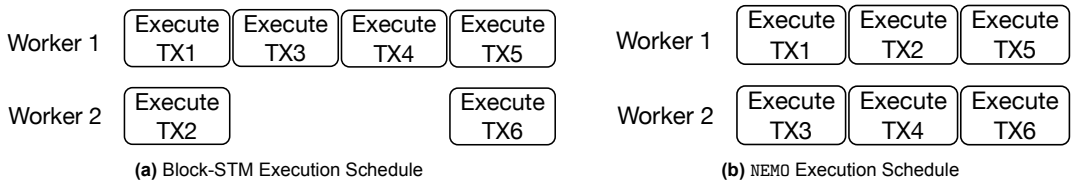


Figure 4.3: Priority Scheduling Example Dependency Graph

In this scenario we have 6 transactions, with the total order based on their transaction index. The dependency graph shows the knowledge we have about the conflicts between transactions. This knowledge may have come from either prior knowledge or previous executions. Let us assume that all transaction take the same time to execute and that there are only two execution workers available. In that case scheduling transactions purely based on their transaction index, as Block-STM does, would result in the execution schedule depicted in Figure 4.4a. As we can see, when the scheduler decides to schedule tx_3 it is not able to schedule anything else concurrently since all remaining transactions depend on it, leaving one worker idle. Whereas using the priority scheduling of NEMO would give the execution schedule depicted in Figure 4.4b. The reason it is able to execute the transactions more efficiently is that it prioritizes the blocking transactions, in this case transactions tx_3 and tx_4, which resolves the dependencies. NEMO still executed tx_1 concurrently to tx_3, in order to keep all the workers busy and avoid having any workers idle. This shows how prioritizing blocking transactions can lead to dependencies being resolved earlier, which helps make the remainding workload more parallelizable.



(a) Block-STM Execution Schedule **(b)** NEMO Execution Schedule

Figure 4.4: Comparison Between Different Execution Schedules

4.4. Transaction States

This section covers the different states a transaction can have in Block-STM and compares it to NEMO. This helps show how the optimizations come together to shape NEMO, and how it is optimized for high contention scenarios.

Before going into the different states a transaction can take in NEMO we first have to understand the different states in Block-STM. Figure 4.5 illustrates the different transaction states in Block-STM and how to transition from one to another.

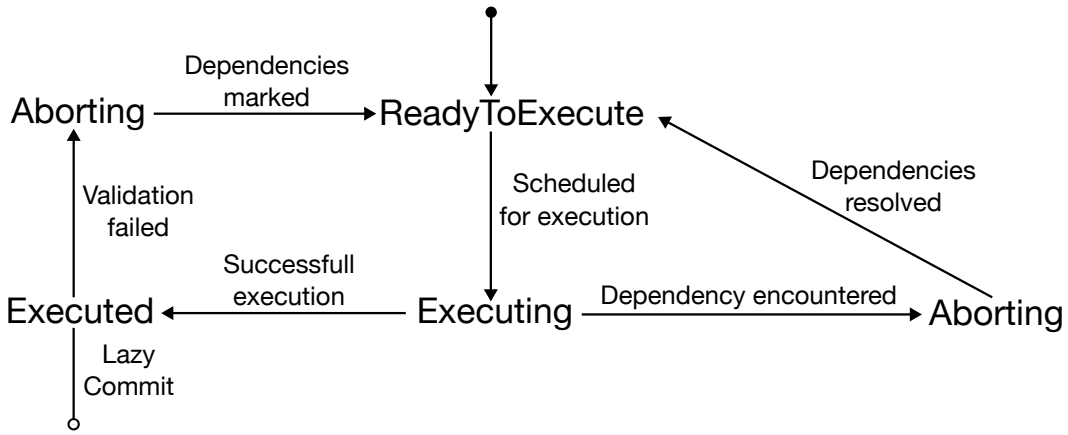


Figure 4.5: Block-STM Transaction States

All transaction are initially in the *ReadyToExecute* state, as Block-STM assumes all transactions are independent. The scheduler then prioritizes transactions based on their transaction index and schedules them for execution, putting them in the *Executing* state. This execution can have one of two outcomes, the first outcome is when an object marked as ESTIMATE is read which indicates that a dependency was uncovered and takes the transaction into the *Aborting* state until the dependency is resolved. In Block-STM a dependency is resolved when the blocking transaction is re-executed. The second possible outcome is a successful execution, in which case the transaction goes into the *Executed* state resolving any dependencies other transactions may have on it. Once a transaction is executed it needs to be validated, remaining in the *Executed* state until its validation task is scheduled. If the transaction passes validation then it remains in the *Executed*, whereas if the validation fails then the transaction goes into the *Aborting* state. This is different from the other way of getting to the *Aborting* state, as it only stays in this state until all entries it wrote are marked with the ESTIMATE flag, which takes very little time. Once that is done the transaction goes back into the *ReadyToExecute* state for the next incarnation, as it needs to be re-executed. Block-STM finishes when all transactions are in the *Executed* state and there are no more tasks left to be scheduled, in which case it proceeds to lazily commit the entire block.

NEMO introduces two new states for transactions to be in. The first one is the *Waiting* state, which is used to differentiate between the two ways to reach the *Aborting* state in Block-STM. The *Waiting* state in NEMO is for transactions that are waiting on a dependency to be resolved, whereas the *Aborting* state in NEMO is short-lived and used while a transaction is being aborted. The second state that NEMO introduces is the *Validated* state, which is for transactions that have passed validation. This state is needed, because in NEMO a dependency is considered resolved only once the blocking transaction has

been validated and this state makes it possible to quickly check if that is the case. The different states of NEMO and their transitions are shown in Figure 4.6.

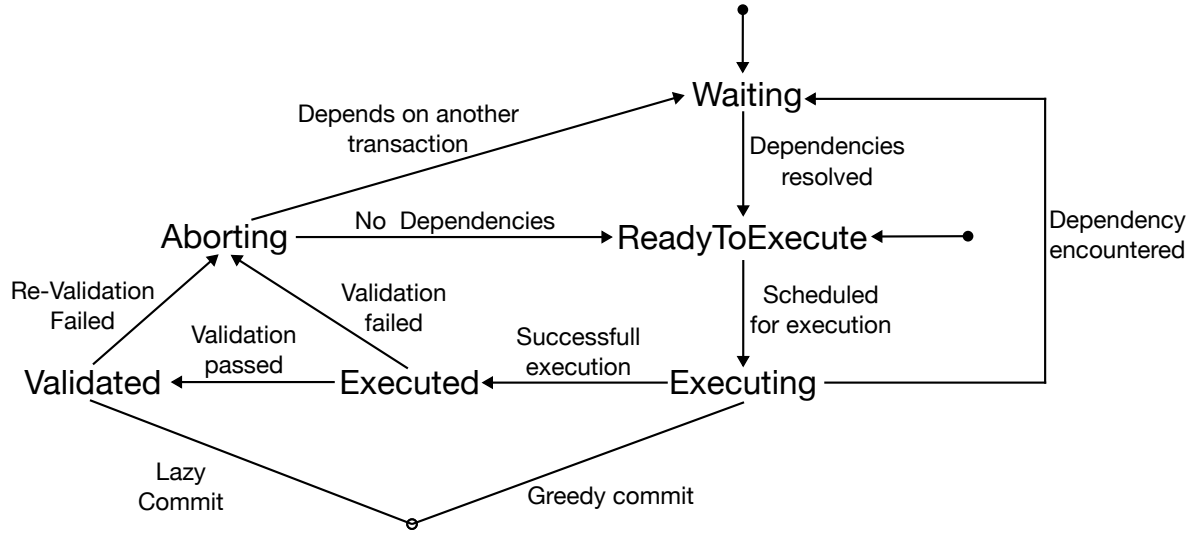


Figure 4.6: NEMO Transaction States

Transactions can either start in the *ReadyToExecute* state if there are no known blocking transactions or in the *Waiting* state if the prior knowledge available shows a dependency. A transaction in the *Waiting* phase remains in that state until all blocking transactions have passed validation. Similarly to Block-STM, once a transaction is scheduled for execution it enters the *Executing* state. If the execution reads an object marked as ESTIMATE, then it marks the dependency and goes into *Waiting*. If the execution succeeds then one of two things could happen, the first option is that the transaction did not use any shared objects in which case it can directly be committed bypassing validation altogether and reaching a terminal state. This was the *greedy commit rule* discussed in Section 4.3.2. The second option is that the transaction did use shared objects thus requiring validation, in which case it goes into the *Executed* state. Transactions in this state wait for the validation task to be scheduled. If the validation passes then the transaction goes into the *Validated* state and resolves any dependencies other transactions may have on it. The *Validated* state is not necessarily a final state, as transactions may be re-validated. If validation or re-validation fails then the transaction enters the *Aborting* state and marks all the objects it wrote with the *ESTIMATE* flag. From there the transaction either goes back to *ReadyToExecute* for the next incarnation if it does not depend on any transactions, or it goes into *Waiting* if it does. The latter is only possible thanks to the extraction of knowledge that NEMO does even when execution is successful, as described in Section 4.3.3. Having the transaction go back into *Waiting* until the dependency is resolved avoids prematurely re-executing it. Once all transactions have either been committed or are in the *Validated* state and there are no more tasks left, NEMO will commit the remaining transactions concluding the epoch.

Comparing the lifecycles of NEMO and Block-STM, we notice that transactions can spend more time blocked by others in NEMO than in Block-STM. This is accordingly with NEMO's philosophy, as it tries to be less optimistic than Block-STM and utilize all the available information to avoid doing unnecessary work. Under high contention this should translate into higher throughput and fewer resource utilization.

5

Implementation Details

This chapter presents the implementation details of the NEMO protocol, written in Rust and incorporated into the contention simulator. As discussed in Chapter 4, NEMO uses Block-STM [23] as a starting point and so the first step was to implement Block-STM. We opted to re-write Block-STM in order to keep the implementation simple and to stick as closely as possible to the pseudo-code from the paper. This implementation also had to be adapted to work with the object data model and NEMO then uses it as a starting point. The code for our implementation is publicly available [1].

5.1. Contention Simulator

The contention simulator is a proprietary piece of software developed by IOTA [2], that makes it possible to simulate blockchain workloads locally. This is extremely practical for developing blockchains, as running a blockchain requires advanced hardware and several machines, making it hard to test if an implementation is working as intended. The contention simulator allows us to generate synthetic workloads according to some parameters, have them go through consensus to output a block of transaction and then simulate the execution of that block.

Transaction execution is problematic when developing software locally, as each transaction is executed in a virtual machine. To avoid this the contention simulator instead allows us to simulate execution. Instead of actually executing the transaction, the thread will simply sleep for a certain amount of time and record the read/write sets of the transaction. For this to work the read/write sets of all transactions have to be determined when the workload is generated, and only made available after execution to simulate the discovery of information through execution. The limitation of this approach is that it forces all incarnations of the same transaction to have the same execution duration and the same read/write sets. This is not realistic, as transaction execution depends on the global state at that time which can change from one incarnation to the next and there is always some randomness involved affecting the duration. Nevertheless, transaction simulation simplifies things a lot and remains fairly realistic. Taking into consideration that NEMO focuses on improving the scheduling of tasks, it makes sense to simulate execution as that is not the focus of our work and should not affect the results in a significant manner.

5.2. Base Block-STM Implementation

The first step of the implementation was to have a working version of Block-STM that worked with the object data model and interfaces properly with the contention simulator. For the Block-STM code we tried to stick as closely as possible to the pseudo-code from the paper, in order to keep the implementation as straightforward as possible. To actually adapt Block-STM to the object data model requires adapting how the global state is represented, but since we simulate execution by just sleeping the thread we only need to change the Storage and `MVMemory` modules from Figure 4.1. In fact, since we are not actually executing the transactions we do not care about the actual values of the objects and

[1] https://anonymous.4open.science/r/nemo-DFC2
[2] https://www.iota.org

simply need to store the version read. This means that we can remove the Storage module altogether, and instead just identify that we would have read from storage when an object is not found in `MVMemory`. The versions of the objects read and written needs to be kept track of, in order for the validation step to have the intended effect. The pseudo-code of Algorithm 1 demonstrates the logic used for simulating execution.

Algorithm 1 Simulated execution

```
1:  function execute(txn)
2:      sleep(txn.duration_ms)                                   ▷ Sleep thread
3:      Scheduler.increment_num_executions()      ▷ Metric: Keep track of number of executions
4:      read_set ← {}
5:      write_set ← {}
6:      for all access ∈ txn.shared_objects do
7:          if access is read then
8:              result ← SharedMemory.read(access.object_id, txn.txn_id)   ▷ Read from MVMemory
9:              if result == ReadError(blocking_txn_index) then
10:                 return (blocking_txn_index, {}, {})                    ▷ Uncovered dependency
11:             else if result == Ok(version) then
12:                 read_set ← read_set ∪ {(access.object_id, version)}
13:             else                                                       ▷ result == NotFound
14:                 read_set ← read_set ∪ {(access.object_id, None)} ▷ None means read from storage
15:             end if
16:         end if
17:         if access is write then
18:             write_set ← write_set ∪ {access.object_id}                 ▷ Always write to MVMemory
19:         end if
20:     end for
        return (None, read_set, write_set)                               ▷ No blocking txn
21: end function
```

After simulating execution (line 2), we go through all the objects accessed by the transaction. This simulates finding out about the read/write sets of a transaction through execution. If a dependency is encountered then the index of the blocking transaction is returned (line 10). Otherwise we keep track of the version of an object that was read, as it will be needed for validation. Crucially, the version *None* indicates that the object was read from storage (line 14). We also keep track of all the objects the transaction writes to. It is worth noting that an object access can be both read and write in which case it is added to both sets.

Adapting the `MVMemory` module to work with the object data model is rather straightforward, because we do not need to store the actual object values since we are simulating execution. Instead of the key of the underlying map being (memory location, txn_index) it should be replaced with (object_id, txn_index). Accessing the data stored in `MVMemory` should be done using the object_id, with the remaining logic staying the same.

5.3. Components of NEMO

The implementation of `NEMO` is composed of several modules, in a similar manner to the core architecture of Block-STM as depicted in Figure 4.1. The first module is *scheduler* which is shared across all threads, and is responsible for giving tasks to the different threads. This is done via the *next_task* method, that worker threads will call once they are free. The *executor* module contains the logic that runs on each worker thread, including the execution simulation. This module has to communicate with the *scheduler* module but also the *mvmemory* module, which is the shared memory from/to which transactions read/write.

The last module is the *nemo* module which interfaces with the contention simulator. Our implementation of `NEMO` was done in such a way to allow turning certain options on and off. So for example, *nemo* takes in a boolean indicating whether priority scheduling should be enabled or not. This allows us to test

out different versions of the NEMO protocol. In order for the protocol to fit with the rest of the contention simulator it must derive the Executor trait, which enforces that it implement the *execute_transactions* method. This method takes in a block of transaction and produces a set result type. The *nemo* module is also responsible for pre-processing and using random sampling to obtain the prior knowledge about transactions. It then spawns the different worker threads and also joins them once they are all done executing. Finally, it verifies that the final state is correct to ensure that the execution worked as intended.

5.4. Failed Ideas

Before going on to the evaluation of the NEMO protocol, let us briefly discuss some ideas that were experimented with but did not prove to be useful.

One idea was to invert the logic around the ESTIMATE marker. So whereas Block-STM adds the ESTIMATE marker when a transaction fails validation, we tried making it such that the marker is added on every write and removed when the transaction passes validation. This was an attempt to be more pessimistic by considering every write to be uncertain until it had passed validation preventing other transactions from using it prematurely. We envisioned that this would yield better performance for high contention scenarios at the cost of performance in low contention scenarios, but instead found that it gave worse performance in all scenarios whilst increasing the code complexity. As a result we decided to remove it altogether.

Another idea was to split the block into several partitions, where execution of one partition could only begin once the previous partition had been committed. The idea was that this would help break up the long dependency chains to avoid cascading aborts. We predicted that having smaller partitions would yield better performance for high contention but found no improvement in performance with a significant increase in code complexity. This also meant that the block was no longer committed all at once, and so the state validation had to be done after every partition which added significant overhead. In the end we decided to remove it altogether.

There was also the idea to extend the partitioning idea with auto-scaling partitions, to fit the contention level. This idea was inspired by the deterministic database paper Gria [44], that was discussed in Section 3.2, and managed to implement such a feature. We never got around to implementing this idea as the partitioning idea failed to be deliver performance.

6 Evaluation

In this chapter we evaluate the performance of NEMO and compare it state of the art. To this end, we devised 6 different scenarios with varying levels of contention to understand how well it performs in different settings and gathered a variety of metrics. The most important metric for our purposes is throughput, as that is the main limiting factor for mainstream blockchain adoption. These experiments look to answer the following questions:

- Q_1 What effect does adding more workers have? Is it the same for all protocols?
- Q_2 What effect does prior knowledge have? How about partial prior knowledge?
- Q_3 What effect does increasing the contention level have on performance? Is it the same for all protocols?
- Q_4 Is NEMO able to outperform the state of the art? If yes, how?

6.1. Setup

Unless stated otherwise, all the evaluations are done locally using the contention simulator described in Section 5.1, running on a MacBook Pro with a Apple M3 Pro chip with 18GB of memory and 11 cores (5 performance and 6 efficiency). All code was written in Rust and ran using release mode. Unless stated otherwise, we ran 10 trials for every data point and computed the mean and standard deviation. For all graphs we plotted the mean with the standard deviation being the error bars.

6.1.1. Metrics

To understand how well our system performs and in order to be able to compare it, we must record certain metrics during execution. The following are the metrics collected:

Duration: This is the main metric we are interested in. We want our protocol to execute all the transactions as quickly as possible. Taking the block size and dividing it by the duration gives the *throughput*, measured in transactions per second (TPS). For NEMO and Block-STM [23] that lazily commit the entire block at once, the duration is a direct measure of latency.

Number of Re-executions: This is only applicable for NEMO and Block-STM, and measures how many times transactions are re-executed. We expect there to be an inverse relation between the number of re-executions and duration, as more re-executions indicates more redundant work. Having fewer re-executions but a longer duration could indicate that some workers are idle which represents an inefficient use of the available resource and would warrant further investigation.

Number of Validations: This is only applicable for NEMO and Block-STM, as those are the only ones with a validation phase. Similarly to the number of re-executions, having more validations indicates more redundant work being done but does not necessarily mean a less efficient system. Being that validation is a lot cheaper than execution, it is going to be less impactful on the overall duration. In fact, it could be worthwhile to favor re-validation over re-execution.

Number of Greedy Commits: This is only applicable for NEMO, as it was one of the optimizations made (refer back to Section 4.3.2), and counts how many times the *greedy commit rule* is used for a given workload. Whilst it is unlikely to be significant for high contention workloads, it may prove valuable in other situations.

6.1.2. Parameters

There are a lot of parameters available to us for generating synthetic workloads with the exact characteristics we are looking for. Let us cover all of the parameters available, their impact and the settings used. A summary of this information can be found in Table 6.2.

Number of Transactions: The first and maybe the most obvious parameter is the number of transactions. In general having more transactions will give more conflicting transactions and also longer dependency chains, both increasing the contention level. However, more transactions also reduces the random variations in measurements that can be caused by outliers. Unless stated otherwise, we set the number of transactions to 5000.

Number of Workers: The number of workers affects how much work can be done in parallel. So if there are 8 workers it means that 8 transactions can be executed simultaneously. Increasing the number of workers can lead to increased throughput as more work can be done in parallel, but can also result in more conflicting transactions being executed out of order leading to more re-executions. There is no default setting as we will be varying the number of workers a lot in the different scenarios to evaluate how it affects the performance of the various protocols. Of course, increasing the number of workers will have no effect on sequential execution.

Computation Cost: As described in Section 5.1, we sleep threads to simulate execution. The computation cost of a transaction is randomly sampled according to some distribution, and determines the number of milliseconds the thread will sleep for. It is important to choose a setting that is realistic, as it has a big impact on the throughput of the system. The available distributions to sample from are: $Constant$ and $LogNormal$, and unless stated otherwise we sample from the $LogNormal(2.0, 0.5)$ distribution. This gives an expected value of around 8.4, a median of around 7.4 and a 90th percentile of around 14.0, which is the amount of milliseconds the thread will sleep for to simulate execution. This gives us a distribution where most transactions have a short duration with some outliers that can last significantly longer, simulating the natural variation between transactions.

Total Object Count: This parameter affects how many shared objects there are. In general having a larger number will naturally mean the workloads are less contended, whereas a smaller number of objects will result in more contention. Of course it is not the only parameter affecting the contention level. Unless stated otherwise this is set to 20 as it offers a good amount of freedom to generate workloads of varying contention levels.

Number of Objects per Transaction: The number of objects per transaction is randomly sampled from a distribution and affects how many shared objects each transaction uses. This represents the number of objects that are a part of the exhaustive set of objects a transaction can use during execution. The available distributions are: $Constant$, $Poisson$ and $LogNormal$. Each transaction randomly samples an integer according to the distribution where a larger number will result in a higher likelihood of conflicts. If the transaction samples 0 then it indicates that it does not touch the shared state and represents a *simple* transaction, as described in Section 2.4.1. The distribution and its parameters have to be chosen carefully in combination with the total object count and are vital in creating scenarios of different contention levels.

Object "Hotness": The "hotness" of an object determines its likelihood to be used by a transaction. A hot object has a high likelihood of being used by a transaction. This likelihood is defined by a distribution and once the number of objects used by a transaction is set, it must sample that many times from this distribution to determine which object it uses (without duplicates). The available distributions are: $Uniform$ and $Zipf$. The distribution and its parameters have a big impact on the nature of the conflicts, going hand in hand with the number of objects per transaction distribution.

Read Frequency: Once a transaction has sampled how many and which object it uses, it must then determine whether it reads the object, writes to the object or both. The read frequency is a decimal

between 0 and 1 indicating the probability that a transaction only reads an object. For every object it uses, it randomly samples to determine if it only reads the object or if it also writes to the object. This affects the frequency of conflicts, as a conflict occurs when a transaction reads and another transaction writes. Unless stated otherwise, we set the read frequency to 0.35 which gives each shared object used a 35% chance of only being read.

Read Frequency Given Write: This parameter only comes into play if an object access was sampled as not being read only. Then it must determine whether it only writes or it also reads. This parameter is a decimal between 0 and 1 that determines how likely it is to read from an object given that it already writes to it. Unless stated otherwise, we this to 0.65 which gives each shared object being written a 65% chance of also being read. Combined with the read frequency this means that every shared object used by a transaction has a 77.25% chance of being read and a 65% of being written to. This information is displayed in Table 6.1.

Event	Expression	Probability
Only Reading	$P(R \cap \neg W)$	0.35
Only Writing	$P(\neg R \cap W)$	0.2275
Both Reading and Writing	$P(R \cap W)$	0.4225
Total Reading	$P(R) = P(R \cap W) + P(R \cap \neg W)$	**0.7725**
Total Writing	$P(W) = P(R \cap W) + P(\neg R \cap W)$	**0.65**

Table 6.1: Probabilities of Reading and Writing Objects

Percentage Actual Access: For each object in the exhaustive read/write set, this parameter determines the likelihood of the transaction using it during execution. It is given as a decimal between 0 and 1, where a smaller number will indicate a greater overestimate by the exhaustive set and setting it to 1 means no overestimate. Unless stated otherwise, we set this parameter to be 0.9 giving each object in the exhaustive read/write set a 90% chance of being used during execution.

Percentage Prior Knowledge: This controls the likelihood of an object access being known ahead of time. This is given as a number between 0 and 100, where 0 means that there is no prior knowledge and 100 means that there is complete prior knowledge of all the read/write sets. For every object access we sample a number between 0 and 100, and if it is less than the percentage prior knowledge parameter, then the object access is added to the set of prior knowledge. This way we can simulate having partial knowledge of a transaction's read/write sets ahead of time. We also experimented with having complete prior knowledge for some transactions and no prior knowledge for others, where the percentage affects how likely a transaction is to have complete knowledge, but felt like that was less realistic. As described in Section 4.3.4, having partial knowledge of transactions' read/write sets is realistic when using the object data model and this parameter helps us figure out the impact that it has on performance.

Table 6.2 contains a summary of all the parameters discussed.

Parameter	Effect	Default Setting
Num. Transactions	Block Size	5000
Num. Workers	Amount of Transactions Executed Simultaneously	Varying
Computation Cost	Simulated Execution Duration	$LogNormal(2.0, 0.5)$
Total Object Count	Amount of Shared Objects Available	20
Num. Objects per Transaction	Number of Shared Objects used per Transaction	Scenario Dependent
Object "Hotness"	Likelihood of using an Object	Scenario Dependent
Read Frequency	Likelihood of only Reading Shared Objet	0.35
Read Frequency Given Write	Likelihood of Reading & Writing Shared Object	0.65
Percentage Actual Access	Likelihood of Using Object from Exhaustive Sets	0.9
Percentage Prior Knowledge	Likelihood of Having Prior Knowledge About Shared Object Access	Varying

Table 6.2: Summary of Parameters

6.1.3. Baselines

Alongside the results of NEMO we also plot the results for sequential execution, Block-STM and a PCC baseline. We chose to compare it to sequential execution to get an idea of how much work is being done in parallel. The ideal scenario would be for the throughput to scale linearly with the number of workers which is unlikely to happen, but we can see how far away we are from that. We compare NEMO to Block-STM as it is currently the state of the art OCC execution engine that NEMO looks to build upon. We also plot the PCC baseline to get an idea of how well NEMO does compared to Sui execution.

We chose not to implement Sui execution in its entirety since there is no pseudocode available and so it would require diving deep into the Sui production code. Instead we implemented the PCC baseline, which uses the exhaustive read/write sets to enforce the causal order between conflicting transactions. Unlike NEMO, this can be done without having to sleep threads to simulate execution and the execution duration can be determined from adding all the transaction durations that were scheduled on the same worker. This way of determining the execution duration is practical since it takes a lot less time, but means that the implementation has zero overhead. As a result, the performance of the PCC baseline is likely superior to that of Sui execution. Similarly to Sui execution, a transaction is only ready to execute after all its dependencies have been executed. The scheduling logic is different however, as Sui relies on the tokio [1] library to schedule tasks, which it does in a non-deterministic manner. The PCC baseline schedules transactions based on when they first became ready and assigns them to the next worker that is free. We do not expect this difference in scheduling to have a drastic impact on performance, and so PCC baseline should give a good idea of how Sui execution would perform.

We would expect sequential execution, Block-STM and PCC baseline to have the same performance regardless of the percentage prior knowledge, since none of them makes use of this prior knowledge. As such, in an experiment where we vary the percentage prior knowledge we would expect the baselines to give a perfectly horizontal line where any variation is purely noise that comes from random sampling. When evaluating the throughput of NEMO it is then more telling to compare it to the baselines, rather than looking at the numbers in isolation. For graphs showing the number of re-executions we will not include the PCC baseline and sequential execution they never re-execute transactions.

The code for all the baseline implementations along with the code for NEMO is publicly available [2].

6.2. Scenarios

In order to evaluate the performance of NEMO under different conditions, we devised 6 different scenarios. Each scenario specifies the parameters it uses, and if a parameter is not specified then it means that the default value listed in Table 6.2 was used.

6.2.1. Single Worker

The first scenario consists of using a single worker to prevent any sort of parallelism and get an insight into the overhead of the different protocols. For this scenario we set the prior knowledge percentage to always be 0. In order to ensure that none of the transactions make use of the *greedy commit rule*, we set the number of objects used per transaction to always be 1. The parameters for this scenario can be found in Table 6.3.

Parameter	Setting
Num. Transactions	1000
Num. Workers	1
Num. Objects per Transaction	$Constant(1)$
Object “Hotness”	$Uniform$
Percentage Prior Knowledge	0

Table 6.3: Parameters for Single Worker Scenario

We opted to lower the number of transactions to 1000 in order to speed up the experiments without

[1] https://crates.io/crates/tokio
[2] https://anonymous.4open.science/r/nemo-DFC2

having a noticeable impact on the results. For this scenario we only care about the throughput since no transactions fail validation which means that the number of re-executions is always going to be 0. The results can be found in Table 6.4.

Protocol	Mean (TPS)	Standard Deviation
Sequential Execution	127.21	1.90
Block-STM	106.36	1.62
PCC Baseline	127.21	1.90
NEMO	106.09	1.42
NEMO w/o priority scheduling	106.63	1.79

Table 6.4: Results for Single Worker Scenario

Looking at the results in Table 6.4, we see that NEMO does introduce some overhead compared to sequential execution, but that the overhead is similar to that introduced by Block-STM. So while a 16.6% overhead is significant, the gains from being able to parallelize execution will outweigh it and NEMO is on par with the state of the art. As expected, the PCC baseline does not introduce any overhead.

6.2.2. Fully Parallelizable

The fully parallelizable scenario is the best case scenario for Block-STM, as its very optimistic assumption holds. The goal of this experiment is to see how well NEMO is able to scale in ideal circumstances, even if those are not realistic.

Parameter	Setting
Num. Workers	Varies
Num. Objects per Transaction	$Constant(0)$
Object "Hotness"	$Uniform$
Percentage Prior Knowledge	0

Table 6.5: Parameters for Fully Parallelizable Scenario

The parameters for this scenario can be found in Table 6.5. To ensure that the workload is fully parallelizable we set the number of objects per transaction to always be 0. Once that is set the object hotness has no effect. Similarly the percentage prior knowledge also has no effect. The only setting that varies for this scenario is the number of workers, for which we tried 4, 8, 12 and 16. The only metric we measured was throughput, since the number of re-executions is always 0 and every transaction is greedily committed.

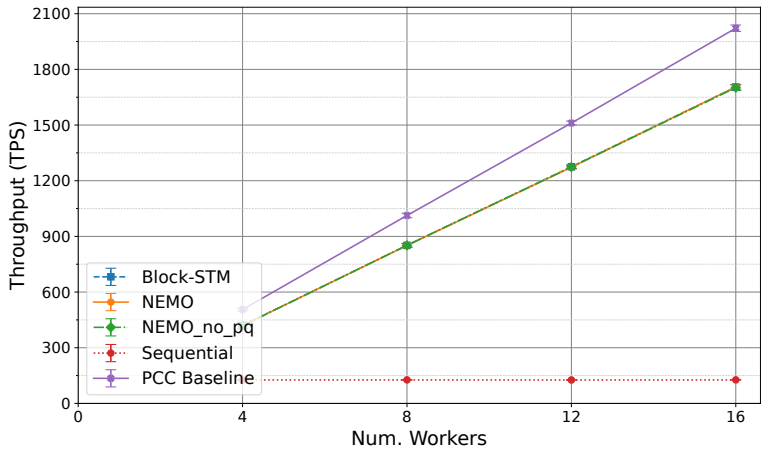


Figure 6.1: Fully Parallelizable Scenario Results

Looking at the results shown in Figure 6.1, we see that NEMO performs identically to Block-STM for fully

parallelizable workloads but worse than the PCC baseline. This was expected as PCC baseline does not have any overhead. Comparing NEMO to Block-STM we see that both scale linearly (with overhead) with the number of workers, and since NEMO uses the *greedy commit rule* it should have lower latency.

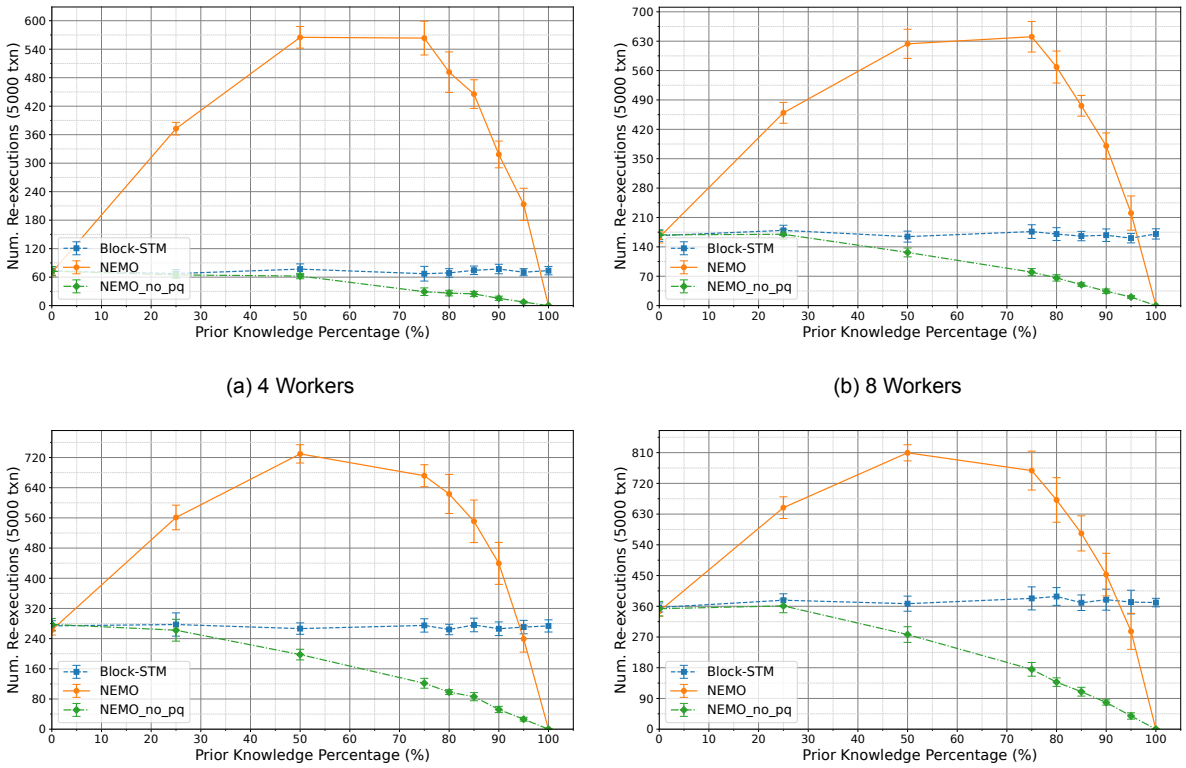
6.2.3. Low Contention

This scenario looks to simulate a realistic baseline contention where there is some limited contention. This is to evaluate the performance of NEMO under regular conditions to ensure that we did not over-optimize for high contention workloads.

Parameter	Setting
Num. Workers	Varies
Num. Objects per Transaction	$LogNormal(0.25, 0.25)$
Object "Hotness"	$Uniform$
Percentage Prior Knowledge	Varies

Table 6.6: Parameters for Low Contention Scenario

The parameters for the low contention scenario can be found in Table 6.6. We chose to sample the number of objects per transaction (N) from $N \sim LogNormal(0.25, 0.25)$, giving us an expected value of around 1.33, a median of around 1.28 and a 90th percentile of around 1.77. The object hotness (H) samples from $H \sim Uniform$ to give all shared objects the same likelihood of being used. Combining these two settings should give little contention that is evenly spread across the shared objects. We then varied the number of workers (4, 8, 12 & 16) as well as the prior knowledge percentage (0%, 25%, 50%, 75%, 80%, 85%, 90%, 95% & 100%).



(a) 4 Workers

(b) 8 Workers

(c) 12 Workers

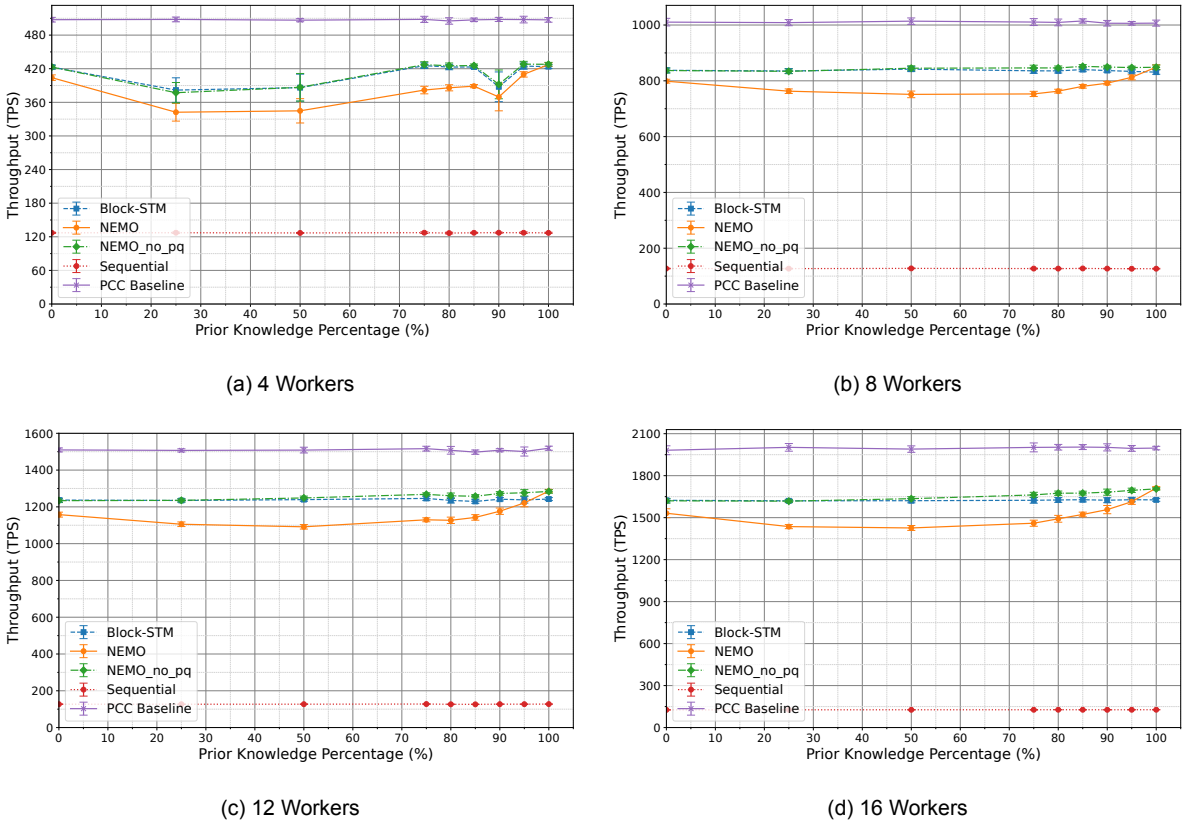
(d) 16 Workers

Figure 6.2: Low Contention Scenario: Number of Re-executions vs % Prior Knowledge

Looking at the number of re-executions, shown in Figure 6.2, we see that having partial knowledge can actually cause the number of re-executions to rise when using priority scheduling. Whilst this may be surprising at first it does make sense, as the priority scheduling is based on the number of

known dependencies. This means that a later transaction that is known to block another one will be scheduled before a earlier transaction where no dependencies are known. This is the intended behavior, but in a setting with limited prior knowledge it often means that the earlier transaction does block other transactions but that we are not aware of it. This results in a lot of conflicting transactions being executed out of order which explains the increase in the number of re-executions. On the other hand, when priority scheduling is not activated we see that having partial prior knowledge is successful in reducing the number of re-executions.

Another interesting point is that increasing the number of workers tends to increase the number of re-executions overall. This makes sense, as having more transactions executing at the same time means it is more likely for transactions to be executed out of order. This effect is more pronounced for Block-STM than for `NEMO`.



(a) 4 Workers

(b) 8 Workers

(c) 12 Workers

(d) 16 Workers

Figure 6.3: Low Contention Scenario: Throughput vs % Prior Knowledge

Looking at the resulting throughput, shown in Figure 6.3, we see that the PCC baseline achieves the best throughput. Increasing the prior knowledge leads to a marginal improvement in the throughput of `NEMO`, such that it is able to outperform Block-STM but not the PCC baseline. We can also see that in order for priority scheduling to work effectively, it is crucial to have complete prior knowledge as that is the only setting for which it outperforms Block-STM.

The results show that there is an inverse relation between the number of re-executions and throughput, although it is not very pronounced. This is because in a low contention setting, the number of re-executions is low overall as the optimistic assumption that all transactions are independent is closer to holding. In fact, we notice that as the number of workers increases the benefit of partial prior knowledge grows. This is because when keeping all other parameters the same, increasing the number of workers leads to more out of order execution requiring more re-executions.

6.2.4. Medium Contention

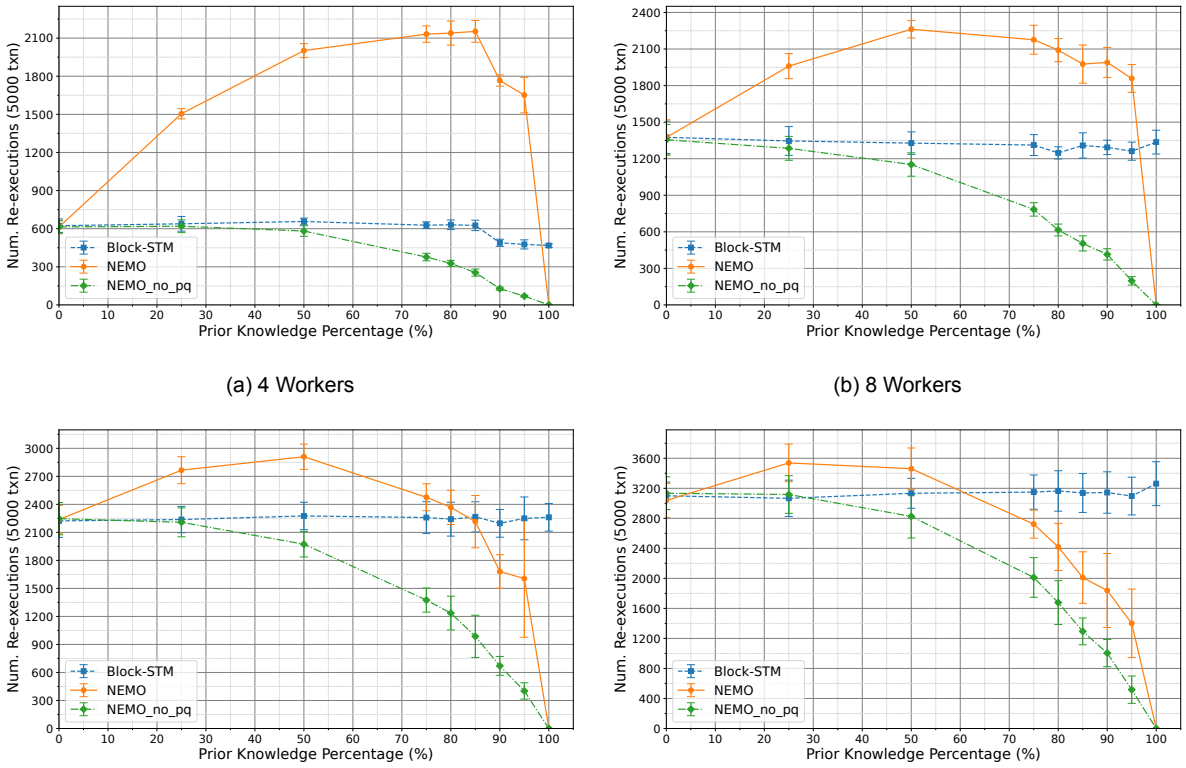
For the medium contention scenario we want to increase the congestion level, and simulate a scenario where some objects are more commonly used than others. To do this we kept all the parameters the

same except for the number of objects per transaction and the object hotness.

Parameter	Setting
Num. Workers	Varies
Num. Objects per Transaction	$LogNormal(0.5, 0.5)$
Object "Hotness"	$Zipf(20, 1.1)$
Percentage Prior Knowledge	Varies

Table 6.7: Parameters for Medium Contention Scenario

Table 6.7, contains the parameters for this scenario. The number of objects per transaction (N) now samples from $N \sim LogNormal(0.5, 0.5)$, giving an expected value of around 1.87, a median of around 1.65 and the 90th percentile is around 3.13. The object hotness (H) is sampled from $H \sim Zipf(20, 1.1)$, where the 20 comes from the total number of objects default value. The second parameter affects how skewed the distribution is. This gives us that the hottest object has around a 31% chance of being sampled, the second hottest around a 15% chance and the third around a 9% chance of being sampled. This is already quite skewed, which is why we only slightly increased the expected number of shared objects used compared to the low contention scenario, as the skewed object hotness will already result in higher contention.



(a) 4 Workers

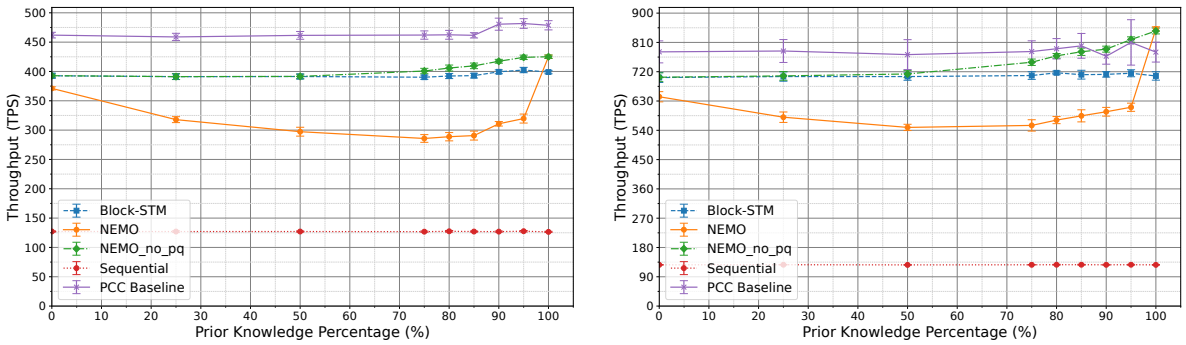
(b) 8 Workers

(c) 12 Workers

(d) 16 Workers

Figure 6.4: Medium Contention Scenario: Number of Re-executions vs % Prior Knowledge

Looking at the number of re-executions, shown in Figure 6.4, we find a similar trend as in the low contention scenario but with a higher number of re-executions across the board. This advantages `NEMO` which is able to halve the number of re-executions when given 80% prior knowledge. With priority scheduling, we see that it starts outperforming Block-STM at 75% prior knowledge for 16 workers. This indicates that it performs better relative to Block-STM when there are higher number of re-executions and is promising for the high contention scenario.



(a) 4 Workers

(b) 8 Workers

(c) 12 Workers

(d) 16 Workers

Figure 6.5: Medium Contention Scenario: Throughput vs % Prior Knowledge

Looking at the resulting throughput, shown in Figure 6.5, we see that the PCC baseline performs the best for 4 workers but that it does not do as well as the others when the number of workers increases, to the point that it performs the worse than Block-STM for 16 workers. Increasing the prior knowledge leads to a significant increase in throughput for `NEMO` when priority scheduling is disabled. When priority scheduling is enabled, we see that complete prior knowledge is required to outperform Block-STM. This explains why Chiron [32] opted to fall back on Block-STM after one failed validation, as that indicates incomplete or incorrect prior knowledge. When operating with full prior knowledge we see that priority scheduling is beneficial and achieves a higher throughput than when it is disabled.

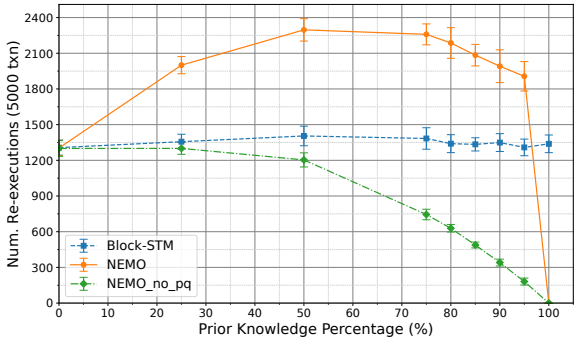
6.2.5. High Contention

This scenario is the most important, as it was the scenario for which `NEMO` was optimizing for. This scenario looks to simulate the *hot spot problem*, a defining feature of blockchain workloads in which highly skewed data accesses lead to a highly contended workload [27]. The parameters used for this scenario are the same as for the medium contention scenario except for a more skewed object hotness, and can be found in Table 6.8.

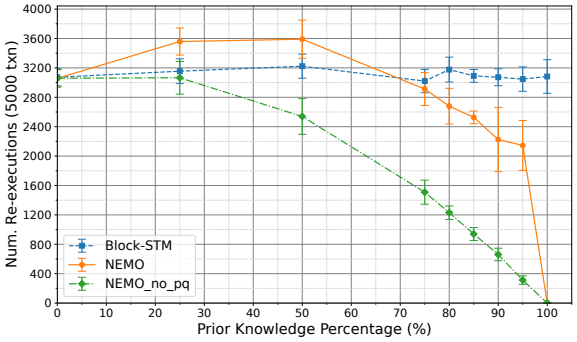
Parameter	Setting
Num. Workers	Varies
Num. Objects per Transaction	$LogNormal(0.5, 0.5)$
Object "Hotness"	$Zipf(20, 2.5)$
Percentage Prior Knowledge	Varies

Table 6.8: Parameters for High Contention Scenario

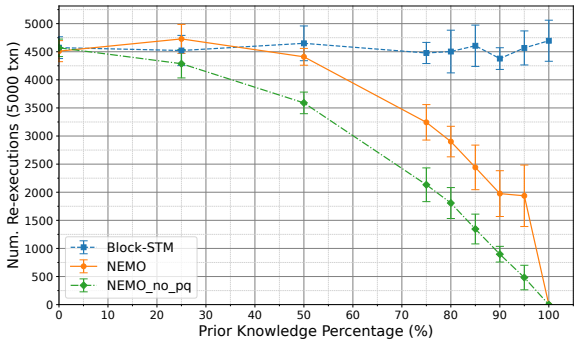
The object hotness (H) now samples from $H \sim Zipf(20, 2.5)$ which is more skewed than previously. This gives us that the hottest object has around a 75% chance of being sampled, the second hottest around a 13% chance and the third around a 5% chance of being sampled.



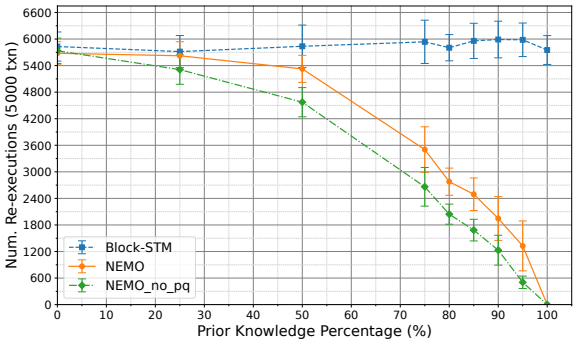
(a) 4 Workers



(b) 8 Workers

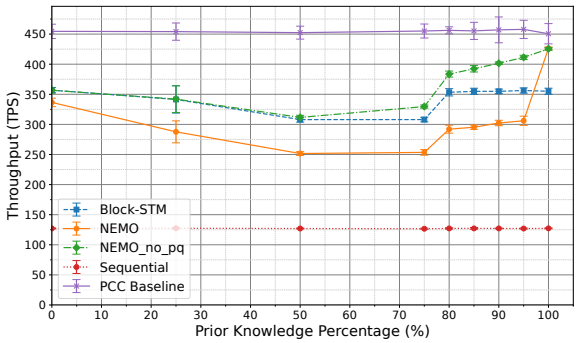


(c) 12 Workers

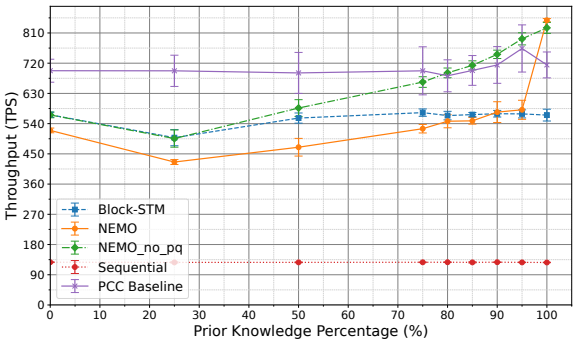


(d) 16 Workers

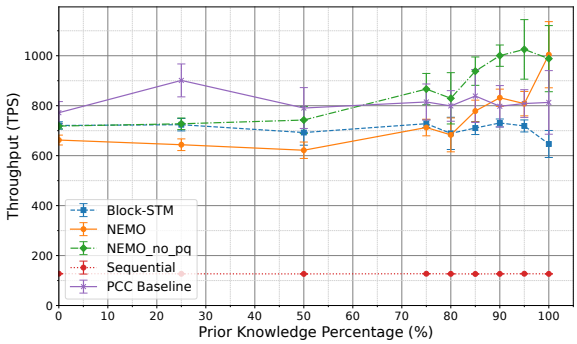
Figure 6.6: High Contention Scenario: Number of Re-executions vs % Prior Knowledge



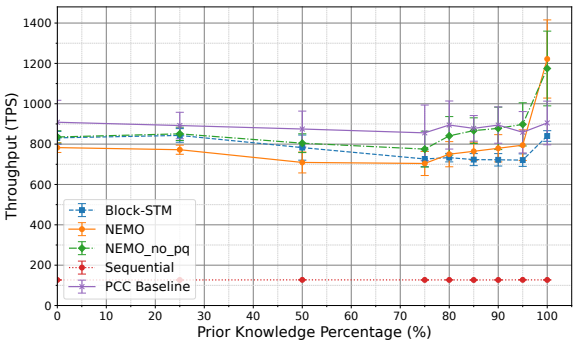
(a) 4 Workers



(b) 8 Workers



(c) 12 Workers

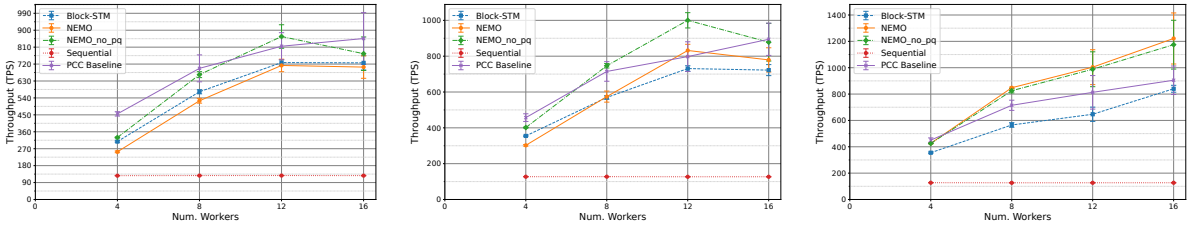


(d) 16 Workers

Figure 6.7: High Contention Scenario: Throughput vs % Prior Knowledge

Looking at the number of re-executions, shown in Figure 6.6, we find a similar trend as we did for the medium contention scenario only more pronounced. We find that NEMO with priority scheduling performs a lot better relative to Block-STM as it did for the previous scenarios.

Looking at the resulting throughput, shown in Figure 6.7, we see that increasing prior knowledge leads to increased throughput and that even partial knowledge can lead to a significant performance gain. This is a similar trend to that observed for the medium contention scenario, but even more pronounced as for 8 workers NEMO without priority scheduling already significantly outperforms Block-STM at 75% prior knowledge with a throughput of 664 TPS compared to 573 TPS for Block-STM which represents a 15.9% improvement in throughput. As was the case for medium contention, we observe an inverse relation between the number of re-executions and throughput. We also find that NEMO with priority scheduling is able to reach the best peak performance, as it is able to reach the best performance with full prior knowledge. Whilst it still suffers a severe drop in performance when the prior knowledge is incomplete, that is less pronounced as for the medium and low contention scenarios. With full prior knowledge we find that for 8 workers NEMO reaches 848 TPS compared to 565 TPS for Block-STM which is a 50.1% improvement in throughput. This significantly outperforms the 30% improvement that Chiron [32] was able to achieve using hints. When comparing NEMO to the PCC baseline we find see that it is only able to outperform it when there is a high percentage of prior knowledge available. When given full prior knowledge with 16 workers NEMO is able to outperform the PCC baseline by 35.0%.



(a) 75% Prior Knowledge (b) 90% Prior Knowledge (c) 100% Prior Knowledge

Figure 6.8: High Contention Scenario: Throughput vs Num. Workers

We were also interested in seeing if adding more workers could lead to more throughput. Figure 6.8 plots the throughput of the various protocols against the number of workers for different prior knowledge percentage levels. From these graphs we can see that adding more workers would not yield higher throughput except if there is full prior knowledge available. For the 75% and 90% prior knowledge setting we can see that throughput has actually already plateaued since having 12 workers.

6.2.6. Delft High Performance Computing (DHPC)

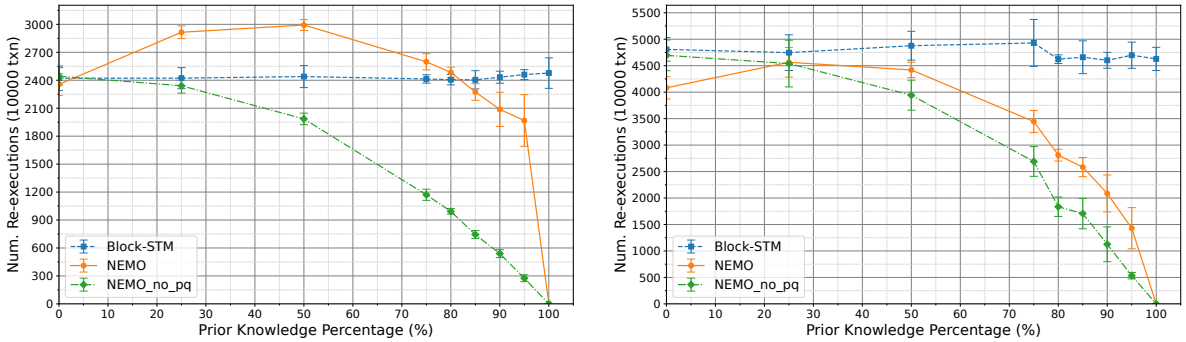
In order to reduce the overhead experienced by NEMO we also ran experiments on a more powerful machine. We made use of the Delft High Performance Computing (DHPC) center [1] with 2GB of memory per CPU. The scenario ran was in between the medium and high contention scenario, and the full list of parameters can be found in Table 6.9. For this experiment we opted to only run 5 trials per data point instead of the usual 10, due to the limited access we had to the DHPC center.

Parameter	Setting
Num. Transactions	10000
Num. Workers	Varies
Num. Objects per Transaction	$LogNormal(0.5, 0.5)$
Total Object Count	50
Object "Hotness"	$Zipf(50, 2.0)$
Percentage Prior Knowledge	Varies

Table 6.9: Parameters for DHPC Scenario

We chose to increase the number of objects to 50 and to change the object hotness to lower the contention level slightly to compensate for increasing the number of transactions to 10,000. This results

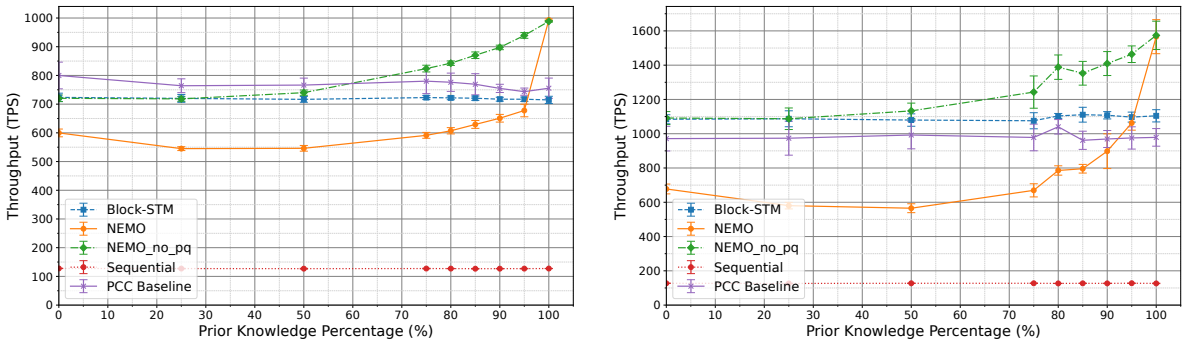
in the hottest object having around a 63% chance of being sampled, the second hottest around a 15% chance and the third around a 7% chance of being sampled. Lowering the contention level ever so slightly allows us to experiment with larger numbers of workers which the DHPC can handle.



(a) 8 Workers (b) 16 Workers

Figure 6.9: Number of re-executions depending on assumed proportion of prior knowledge

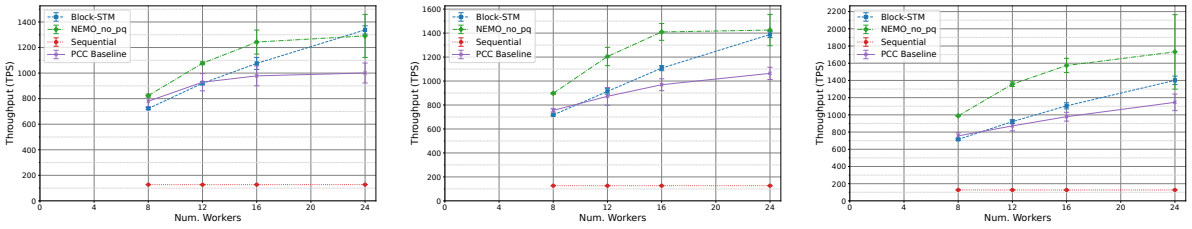
Looking at the number of re-executions, shown in Figure 6.9, we see similar results to that found for the medium and high contention scenarios. This indicates that the amount of redundant work done by `NEMO` and Block-STM is of the same order as it was when we were experimenting locally.



(a) 8 Workers (b) 16 Workers

Figure 6.10: Execution throughput depending on assumed proportion of prior knowledge

Looking at the throughput, shown in Figure 6.10, we see similar results to those found for the medium and high contention scenarios except that the PCC baseline performs a lot worse relative to `NEMO` and Block-STM. In fact, we see that for 16 workers Block-STM is able to outperform the PCC baseline. This is because the PCC baseline experiences zero overhead because of its implementation, but that advantage is reduced when running experiments on the DHPC. This gives us a better indication of the performance of the PCC baseline, and we can see that for 16 workers `NEMO` achieves a 27.1% higher throughput at 75% prior knowledge and a 60.8% higher throughput for 100% prior knowledge.



(a) 75% Prior Knowledge (b) 90% Prior Knowledge (c) 100% Prior Knowledge

Figure 6.11: DHPC: Throughput vs Num. Workers

We also experimented with having more than 16 workers to see if adding more workers could increase throughput further. Looking at the results in Figure 6.11, we see that throughput mostly stagnated when increasing the number of workers from 16 to 24 except for the case when there is full prior knowledge. This indicates that having more workers would not increase throughput and that having 16 workers is better suited for this workload.

6.3. Results Discussion

Based on these evaluations we can answer the questions devised at the start of this chapter.

Q_1: What effect does adding more workers have? Is it the same for all protocols?

Regarding Q_1, throughout the experiments we observed that increasing the number of workers leads to an increase in throughput except for sequential execution. Across the different scenarios we saw that all other things being equal, increasing the number of workers led to increased contention. This was most noticeable when looking at the number of re-executions for NEMO and Block-STM. This meant that increasing the number of workers did not yield a linear increase in throughput and eventually performance plateaued. We can conclude that the number of workers should be chosen wisely to maximize performance without having unnecessary costs.

Q_2: What effect does prior knowledge have? How about partial prior knowledge?

For Q_2 we can say that prior knowledge leads to an increased throughput, but we did uncover that partial prior knowledge has a negative impact on NEMO when using priority scheduling. From that we can conclude that priority scheduling should only be used if there is a high amount of prior knowledge available. When using NEMO without priority scheduling, we found that partial prior knowledge was effective at reducing the number of re-executions and that it translated to throughput.

Q_3: What effect does increasing the contention level have on performance? Is it the same for all protocols?

Regarding Q_3, the evaluations showed that increasing the contention level decreased the throughput across the board. Comparing NEMO to the baselines we see that relative to the protocols, NEMO performs a lot better under high contention. This is due to the increased impact that prior knowledge has on performance. We also found that the PCC baseline was the most negatively affected by contention, as it had the best throughput for low contention but was outperformed by Block-STM and NEMO for high contention.

Q_4: Is NEMO able to outperform the state of the art? If yes, how?

Regarding Q_4, we can say that yes NEMO was able to outperform the state of the art and that it was most significant for high contention workloads. Comparing NEMO to Block-STM we saw that NEMO was able to reduce the number of re-executions across all scenarios, which translated to increased throughput. The largest difference was found in the high contention scenario with 8 workers where NEMO achieved a 30.9% higher throughput than Block-STM when given 90% prior knowledge and a 50.1% higher throughput when given full prior knowledge. Comparing NEMO to the PCC baseline, we saw that NEMO had a lower throughput for low contention workloads but a higher throughput for high contention. As mentioned in Section 6.1.3, the implementation of the PCC baseline has zero overhead. This effect was minimized when using the DHPC center to run the experiments. This gave us that for 16 workers NEMO was able to achieve a 45.4% higher throughput than the PCC baseline for 90% prior knowledge and a 60.8% higher throughput when given full prior knowledge.

Overall the evaluation showed that NEMO was able to perform well in a variety of scenarios, and was able to outperform state of the art protocols by using partial prior knowledge. We also found that while priority scheduling was able to achieve the best performance when full prior knowledge is available, its performance dropped massively when there was only partial prior knowledge available. This indicates that priority scheduling should only be used if full prior knowledge is available, and explains why Chiron [32] is only used for straggler execution. We did find however that NEMO without the priority scheduling was able to benefit greatly from partial prior knowledge which is encouraging because as explained in Section 4.3.4, obtaining partial prior knowledge is definitely realistic when using the object model.

7

Conclusion

Throughput remains the limiting factor for mainstream blockchain adoption, with execution being the current bottleneck. The root of the issue is the skewed data access that is typical of blockchain workloads and leads to increased contention levels. This limits the parallelism available in the workload and requires concurrency control to ensure that all validators end up in the same state. The purpose of this thesis was first to research the current state of the art blockchain execution protocols, understanding their strengths and weaknesses. From there we designed NEMO to build upon the state of the art, specifically targeting high contention workloads. We also implemented the NEMO protocol and evaluated it for a variety of workloads, comparing it to the current state of the art. The results indicate that NEMO effectively advances the state of the art for high contention workloads.

Let us circle back to the beginning of this thesis and address the research questions. Below are the research questions asked, and a summary of their answers.

RQ_1: What are the characteristics of high contention workloads?

Blockchains have highly skewed data accesses which creates contention between transactions. Having such *hot spots* results in a high number of conflicting transactions that require concurrency control. This limits the parallelism available in the workload. Consequently, both optimistic and pessimistic concurrency control experience drops in performance for high contention workloads, limiting the throughput of blockchains.

RQ_2: Where does the current state of the art fall short? What are their limitations?

The current state of the art blockchain execution protocols are able to use parallel execution, but struggle for high contention workloads. They are either too pessimistic, failing to exploit the limited parallelism available or they are too optimistic, resulting in a lot of redundant work being done. Optimistic approaches have shown more promise, but fail to use all of the available information.

RQ_3: How can we maximize parallel execution whilst managing conflicts? And how can we effectively manage these conflicts?

This thesis demonstrated how optimistic concurrency control could be enhanced with knowledge about dependencies to avoid known conflicts. This relies on the use of the object data model that increases parallelism and enables obtaining prior knowledge about the transactions' read/write sets. Leveraging this information allows NEMO to remain optimistic whilst avoiding cascading aborts that typically plague optimistic protocols under high contention.

Ultimately, we have concluded that intelligent parallel execution can be used to maximize the throughput of blockchain execution for high contention workloads, with the design and implementation of NEMO contributing to the state of the art.

7.1. Future Work

As with all research, there are limitations to this work that warrant further investigation. The most obvious continuation of this work, is to build on the evaluation of NEMO using it on real workloads. This could then be extended to using a variant of the NEMO protocol in production depending on the results. This however consists more of engineering work than research.

The following points provide ideas for future research:

- One of the main contributions of NEMO is the use of partial prior knowledge to reduce redundant work. While we are confident that such knowledge can be extracted statically, there is currently no work doing so. Researching how static analysis can be used to extract the partial prior knowledge used by NEMO is an idea for future work. This work would likely have a large emphasis on engineering however.
- Our evaluation showed that while NEMO performed the best for high contention workloads, it was the PCC baseline that had the best performance for low contention workloads. Having some mechanism to identify the contention level of a workload and automatically adapt the execution protocol could be an area of research. This could involve tweaking the parameters of the protocol, similar to how Gria [44] auto-scales the batch size, or could mean changing protocols altogether.
- One of the main limitations of NEMO is its reliance on shared memory. This is problematic as it limits the number of workers and prevents it from scaling out horizontally. Research has already been done on sharded execution, but approaching it with the knowledge of NEMO and focusing on high contention workloads could yield interesting results. This could involve clustering conflicting transactions to have them all on the same machine and/or postponing highly conflicting transactions to create isolated clusters.
- NEMO focuses on improving the scheduling of execution and relied on simulated execution for the performance evaluation. Transaction execution for blockchains is a field ripe for research. One idea could be to use the information gained from a previous execution of a transaction to speed up re-execution. This was something explored by DOCC [20], which used data pre-fetching to speed up transaction re-execution for deterministic databases. Similar concepts could be used to enhance NEMO by speeding up transaction re-execution.
- NEMO introduced the *greedy commit rule*, identifying that *simple* transactions will always be independent. Currently, these transactions are able to take a fast-path to be executed as soon as possible. Research could be done to see if these transactions could instead go to a buffer when all workers are busy with *complex* transactions. The transactions in the buffer could then be executed when there are idle workers. This would increase the latency for simple transactions in order to have better resource utilization.
- The result of NEMO execution is equivalent to sequentially executing the output of consensus, restricting the scheduler's ability to parallelize the workload. Developing a protocol that is deterministic, but that effectively utilizes transaction reordering to maximize throughput is a potential avenue for research.
- Finally, a paper surveying the field of blockchain execution could be interesting to gather all of the information in one place. This could also include benchmarking the work surveyed to compare the performance of the different approaches and to establish the current state of the art performance. This field is rapidly evolving with new papers being published all the time, and yet there has been no survey paper written as of now.

Although we have highlighted several potential directions, the scope for future work remains wide and goes beyond the ideas listed.

References

[1] Delft High Performance Computing Centre (DHPC). *DelftBlue Supercomputer (Phase 2)*. https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2. 2024.

[2] Ittai Abraham, Guy Gueta, and Dahlia Malkhi. “Hot-Stuff the Linear, Optimal-Resilience, One-Message BFT Devil”. In: *CoRR* abs/1803.05069 (2018). arXiv: 1803.05069. url: http://arxiv.org/abs/1803.05069.

[3] “All About Objects”. In: *Sui Foundation* (2023). url: https://blog.sui.io/all-about-objects/.

[4] “All About Parallelization”. In: *Sui Foundation* (2024). url: https://blog.sui.io/parallelization-explained/.

[5] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. “ ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems ”. In: *ICDCS*. 2019.

[6] Elli Androulaki et al. “Hyperledger fabric: a distributed operating system for permissioned blockchains”. In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys ’18. ACM, Apr. 2018. doi: 10.1145/3190508.3190538. url: http://dx.doi.org/10.1145/3190508.3190538.

[7] Kushal Babel et al. *Mysticeti: Reaching the Limits of Latency with Uncertified DAGs*. 2024. arXiv: 2310.14821. url: https://arxiv.org/abs/2310.14821.

[8] Shrey Baheti et al. “DiPETrans: A framework for distributed parallel execution of transactions of blocks in blockchains”. In: *Concurrency and Computation: Practice and Experience* 34 (Jan. 2022). doi: 10.1002/cpe.6804.

[9] Sam Blackshear et al. “Move: A Language With Programmable Resources”. In: 2019. url: https://api.semanticscholar.org/CorpusID:201681125.

[10] Sam Blackshear et al. *Sui Lutris: A Blockchain Combining Broadcast and Consensus*. 2024. arXiv: 2310.18042. url: https://arxiv.org/abs/2310.18042.

[11] Micah Casella. “Diving Into Sui”. In: *Messari* (May 2023). url: https://messari.io/report/diving-into-sui.

[12] Aldar C-F. Chan. *UTXO in Digital Currencies: Account-based or Token-based? Or Both?* 2021. arXiv: 2109.09294. url: https://arxiv.org/abs/2109.09294.

[13] Junchao Chen et al. *Thunderbolt: Concurrent Smart Contract Execution with Nonblocking Reconfiguration for Sharded DAGs*. 2025. arXiv: 2407.09409. url: https://arxiv.org/abs/2407.09409.

[14] Yang Chen et al. “Forerunner: Constraint-based Speculative Transaction Execution for Ethereum”. In: *SOSP*. 2021.

[15] Zhihao Chen et al. “PEEP: A Parallel Execution Engine for Permissioned Blockchain Systems”. In: *DASFAA*. 2021.

[16] Brian Cho and Alexander Spiegelman. “Quorum Store: How Consensus Horizontally Scales on the Aptos Blockchain”. In: (May 2023). url: https://medium.com/aptoslabs/quorum-store-how-consensus-horizontally-scales-on-the-aptos-blockchain-988866f6d5b0.

[17] George Danezis et al. “Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus”. In: *EuroSys*. 2022.

[18] Manu Dhundi et al. *Shardines: Aptos’ Sharded Execution Engine Blazes to 1M TPS*. Feb. 2025. url: https://aptoslabs.medium.com/shardines-aptos-sharded-execution-engine-blazes-to-1m-tps-71c5f9b8bf60.

[19] Sui Documentation. *Sui Tokenomics*. 2025. url: https://docs.sui.io/concepts/tokenomics.

[20] Zhi-Yuan Dong et al. “Optimistic Transaction Processing in Deterministic Database”. In: *J. Comput. Sci. Technol.* 35.2 (Mar. 2020), pp. 382–394. issn: 1000-9000. doi: 10.1007/s11390-020-9700-5. url: https://doi.org/10.1007/s11390-020-9700-5.

[21] “Ethereum: a secure decentralised generalised transaction ledger”. In: 2019. url: https://api.semanticscholar.org/CorpusID:261284440.

[22] Péter Garamvölgyi et al. “Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts”. In: *ICSE*. 2022.

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

[24] Yaron Hay and Roy Friedman. “ Batch-Schedule-Execute: On Optimizing Concurrent Deterministic Scheduling for Blockchains ”. In: *SRDS*. 2024.

[25] Idit Keidar et al. “All You Need is DAG”. In: *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. PODC’21. Virtual Event, Italy: Association for Computing Machinery, 2021, pp. 165–175. isbn: 9781450385480. doi: 10.1145/3465084.3467905. url: https://doi.org/10.1145/3465084.3467905.

[26] Quentin Kniep et al. *Pilotfish: Distributed Transaction Execution for Lazy Blockchains*. 2024. arXiv: 2401.16292. url: https://arxiv.org/abs/2401.16292.

[27] Haoran Lin et al. “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. isbn: 9798400711961. doi: 10.1145/3689031.3696063. url: https://doi.org/10.1145/3689031.3696063.

[28] George Mitenkov. “Metering the Meter, or How to Efficiently and Deterministically Charge the Execution of Smart Contracts”. Master Thesis. ETH Zurich, Oct. 2023. doi: 10.3929/ethz-b-000638680.

[29] George Mitenkov et al. *Deferred Objects to Enhance Smart Contract Programming with Optimistic Parallel Execution*. 2024. arXiv: 2405.06117. url: https://arxiv.org/abs/2405.06117.

[30] Satoshi Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System”. In: (May 2009). url: http://www.bitcoin.org/bitcoin.pdf.

[31] Ray Neiheiser and Eleftherios Kokoris-Kogias. *Anthemius: Efficient & Modular Block Assembly for Concurrent Execution*. 2025. arXiv: 2502.10074. url: https://arxiv.org/abs/2502.10074.

[32] Ray Neiheiser et al. *CHIRON: Accelerating Node Synchronization without Security Trade-offs in Distributed Ledgers*. 2024. arXiv: 2401.14278. url: https://arxiv.org/abs/2401.14278.

[33] Xiaodong Qi, Jiao Jiao, and Yi Li. “Smart Contract Parallel Execution with Fine-Grained State Accesses”. In: *ICDCS*. 2023. doi: 10.1109/ICDCS57875.2023.00068.

[34] Ankit Ravish et al. *Unleashing Multicore Strength for Efficient Execution of Transactions*. 2024. arXiv: 2410.22460. url: https://arxiv.org/abs/2410.22460.

[35] Donghyeon Ryu and Chanik Park. “Toward High-Performance Blockchain System by Blurring the Line between Ordering and Execution”. In: *SC*. 2024.

[36] Rizwan Shahid. “Parallel Transaction Execution in Public Blockchain Systems”. MA thesis. University of Waterloo, 2024.

[37] Kiarash Shamsi et al. “Chartalist: Labeled Graph Datasets for UTXO and Account-based Blockchains”. In: *NeurIPS*. 2022.

[38] Alexander Spiegelman et al. “Bullshark: DAG BFT Protocols Made Practical”. In: *CCS*. 2022.

[39] Srivatsan Sridhar, Alberto Sonnino, and Lefteris Kokoris-Kogias. *Stingray: Fast Concurrent Transactions Without Consensus*. 2025. arXiv: 2501.06531. url: https://arxiv.org/abs/2501.06531.

[40] Stephen Su et al. “Concerto: Transaction-Parallel EVM”. In: *Stanford University* (2024). url: https://www.scs.stanford.edu/24sp-cs244b/projects/Concerto_Transaction_Parallel_EVM.pdf.

[41] Ertem Nusret Tas et al. “Light Clients for Lazy Blockchains”. In: *FC*. 2025.

[42] The MystenLabs Team. *The Sui Smart Contracts Platform*. https://docs.sui.io/paper/sui.pdf. 2022.

[43] *The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure*. https://aptosfoundation.org/whitepaper/aptos-whitepaper_en.pdf. 2022.

[44] Xinyuan Wang, Yun Peng, and Hejiao Huang. “Gria: an efficient deterministic concurrency control protocol”. In: *Frontiers of Computer Science* 18 (2023), pp. 1–13. url: https://api.semanticscholar.org/CorpusID:266344805.

[45] Xinyuan Wang et al. “Dodo: A scalable optimistic deterministic concurrency control protocol”. In: *Future Generation Computer Systems* 159 (2024), pp. 15–26. issn: 0167-739X. doi: https://doi.org/10.1016/j.future.2024.05.004. url: https://www.sciencedirect.com/science/article/pii/S0167739X24002139.

[46] Jiang Xiao et al. “Nezha: Exploiting Concurrency for Transaction Processing in DAG-based Blockchains”. In: *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. 2022, pp. 269–279. doi: 10.1109/ICDCS54860.2022.00034.

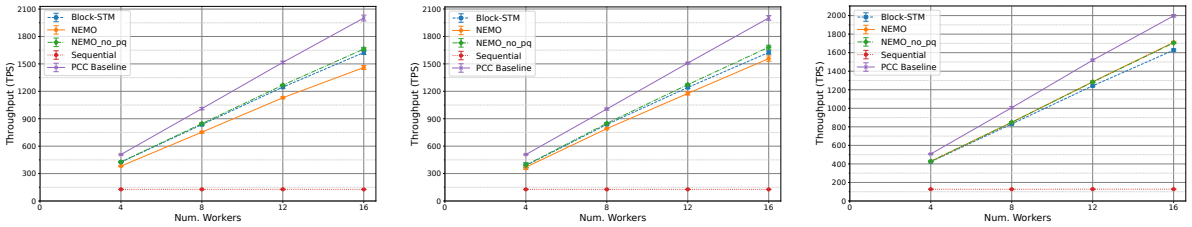
[47] Jianting Zhang et al. *Optimal Sharding for Scalable Blockchains with Deconstructed SMR*. 2024. arXiv: 2406.08252. url: https://arxiv.org/abs/2406.08252.

A

Additional Graphs

This appendix contains some graphs that were made, but that we did not discuss in the evaluation section due to their minimal significance. Instead we include them in this appendix for completeness.

Low Contention Scenario

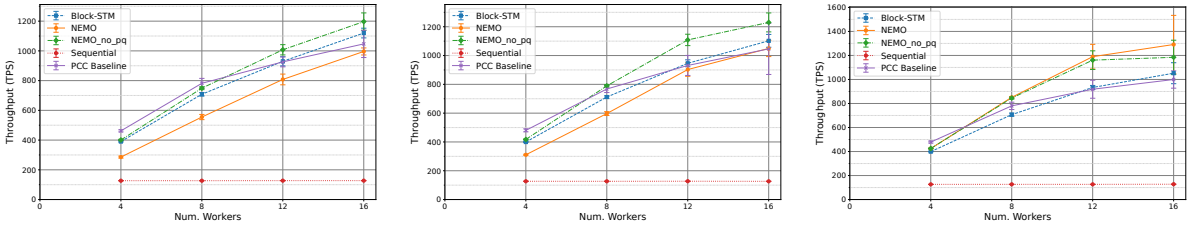


(a) 75% Prior Knowledge (b) 90% Prior Knowledge (c) 100% Prior Knowledge

Figure A.1: Low Contention Scenario: Throughput vs Num. Workers

Figure A.1 shows that for this scenario all protocols (except sequential execution) would benefit from having more workers. There is no indication of a plateau and throughput appears to scale linearly with the number of workers.

Medium Contention Scenario



(a) 75% Prior Knowledge (b) 90% Prior Knowledge (c) 100% Prior Knowledge

Figure A.2: Medium Contention Scenario: Throughput vs Num. Workers

Figure A.2 shows that for this scenario increasing the number of workers would likely increase the throughput for all protocols (except sequential execution). However, we can already see a decline in the benefit of adding more workers indicating that a plateau is approaching.