Supra Containers as App-Chains for Efficient Parallel Execution of Transactions

Supra Research

September 2024

v2.0

Abstract

By definition, app-chains expect isolation to efficiently serve specific use cases. Currently, Layer 2s, Avalanche's Subnets, Polkadot's Parachains, and Cosmos Zones are commonly used solutions to host app-chains. We observe that in an ecosystem with a high throughput Layer 1 blockchain, it is sufficient to offer this isolation at the application layer via namespaces. We observe that most of the times smart contracts come bundled as they compose and call amongst this bundle. Hence, on Supra, app-chains are offered as Supra Containers, which group smart contracts into namespaces, disallowing atomic function calls across containers.

On a closer investigation we observe that it is not the isolation that is desired in the appchain setting, but a regulated way to access services outside of their appchains to be more precise. These are implemented as cross-(Cosmos)zone, or cross-L2, or cross-(Polkadot)parachain transactions. Supra containers offer container-admins to bake in their custom cross-container access control specification into the runtime so that such services are accessed atomically and efficiently, compared to a multi-network-hop accesses for such accesses in the regular appchains. This completely avoids fragmentation of liquidity across containers.

This container abstraction lends itself naturally to the parallel execution of transactions in batches derived from different containers, as transactions belonging to batches of two different containers are non-conflicting. Through experimentation, we demonstrate the performance impact of Supra's containerized parallel execution by comparing it against Aptos' BlockSTM and Solana's SeaLevel techniques at various parameters. In the presence of cross-container transactions a partial-order based approach towards parallel execution is more applicable.

1 Introduction

Blockchains started off primarily with native currency transfers in the form of the Bitcoin network [16]. Soon Ethereum [9] introduced *smart contracts* capability and provided full Turing-complete on-chain programming, and it was heralded as Blockchain 2.0. This on-chain programming gave birth to transitioning many of the traditional financial instruments on to blockchains and provided the Decentralized Finance (DeFi) space with a clear technological advantage over legacy financial infrastructure. Though blockchains are being adopted in various applications like synthetic assets, forex swaps, supply chain management and validation, etc., DeFi has been the major driving force in the adoption of blockchains.

Over time many blockchains have proliferated, and Turing-complete, on-chain programming has become the norm. With the rise in the adoption of blockchains and the efficiency gains being developed regarding their consensus protocols, soon the challenges of optimizing execution times for transactions came to the fore. The traditional sequential execution employed by various clients of Ethereum seem insufficient at scale, and the need to exploit and apply the advances of parallel execution are now imminent [7].

Blockchains such as Solana [26], Sawtooth [19], Sei [20], and Sui [23] adopted the approach of allowing the clients to specify the read and write set information in the form of the accounts and objects being accessed. This specification helps its runtime to infer the conflicts and

schedule the execution of transactions in such a way as to exploit maximal parallelism and optimize execution time. Without requiring any extra specifications from its clients, Aptos [1] adopted the well-studied *Software Transactional Memory (STM)* [21] approach to the on-chain execution and introduced *Block-STM* [12]. Sui [23] also requires the clients to specify the objects a transaction is going to modify; objects may be shared or have exclusive ownership. Based on the object ownership, it facilitates the identification of independent transactions to execute in parallel [11, 10]. Transactions that interact with owned objects can be executed in parallel, while transactions that access shared objects are executed sequentially since they may lead to conflicting accesses. Moreover, transactions that do not involve a shared object can completely bypass consensus.

In this whitepaper, we introduce *Supra containers* serving as app-chains for the developers on Supra Layer 1 blockchain. We show that Supra containers preserves all the properties / features of app-chains, and in addition provides more benefits. We also study this notion of containers from the perspective of optimizing block execution times and find that they naturally yield to parallel execution of transactions belonging to batches of different containers.

The rest of this paper is organized as follows. Section 2 provides an overview of app-chains. Section 3 introduces and positions Supra containers as Supra's version of app-chains. Section 4 highlights the benefits of Supra containers versus the state-of-art app-chain solutions. Section 5 presents how Supra containers may be leveraged towards optimizing execution times via parallelization. Section 6 discusses the state-of-art in terms of parallel execution algorithms for blockchain transactions: Solana Executor, Aptos Block-STM. Section 7 presents a detailed empirical analysis of the benefits of containers towards execution latency and execution throughput. Finally, section 8 summarizes and concludes this paper.

2 App-Chains - An overview

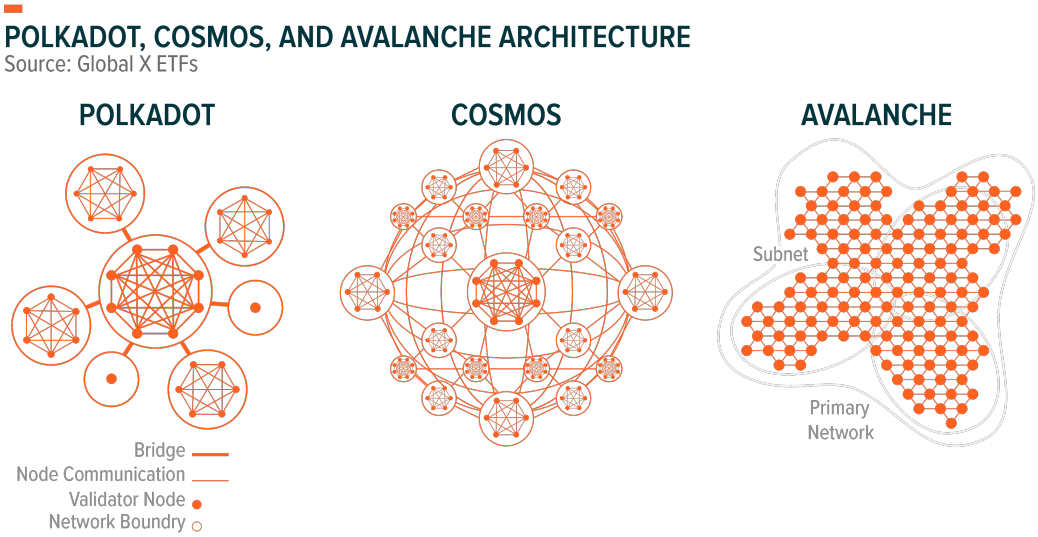

Figure 1 State-of-Art App-chain solutions

When an entire blockchain is reserved for an application or a group of applications then that combination is called an app-chain, or an application specific chain. App-chains have emerged mainly because they grant sovereignty and provide flexibility to the dApp's creators

as well as to its community. A community can employ in their own bespoke business models, release their own tokens, create their own governance mechanisms, and set transaction fees to facilitate smooth and sustainable operations. In a sense, app-chains are becoming digital decentralized gated communities. Some examples of app-chains are dYdX v4 [8] on Cosmos, and DeFi Kingdoms [6] on Avalanche.

Here isolation is the desiderata. The transactions targeting such an application expect that there is no influence from any other transactions meant for another application, be it in the form of escalation of transaction fees, or in any other form. It is often seen in popular public blockchains that a high volume of traffic targeting some contracts increase the transaction fees for the transactions targeting smart contracts that attract low traffic. Users and developers debate over whether this unfairly impacts certain dApps more than others. This is one of the reasons the demand for app-chains has grown stronger as more users enter the space.

Interestingly, we observe that app-chains came into use before public blockchains had become capable of the same high throughput metrics as they are today. Hence Layer 2s, sidechains, Cosmos Zones, Polkadot Parachains, Avalanche Subnets, etc, were favored as solutions to app-chain requirements. Refer Figure 1. However in the case of a high throughput and fast finality Layer 1 blockchain, we believe app-chains can be accommodated and offered inside Layer 1s alone, rather than looking outwardly for solutions in the Layer 2 space. This direction led us to the design of Supra containers.

Though isolation is desired, it is not the end-game for app-chains. They do want some extent of composability, typically for accessing services outside of their app-chain. Usually, such composability requests go through a relay bridge and involve a multi-network-hop process leading towards high latency. We aim to provide such cross-container accesses in an atomic execution resulting with low latencies. As a benefit we also avoid fragmenting liquidity across containers.

3 Supra Containers

Supra containers are basically **namespaces**, partitions of the blockchain's global state (shown in Figure 2), and should not be confused with 'docker' containers. They group multiple related smart contracts under one namespace, regulating any cross namespace function calls. An important thing to note that the cross-container interactions are gated.

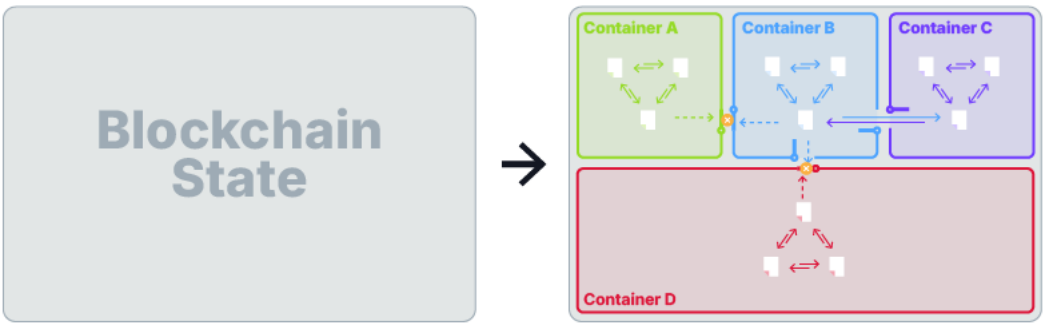

Figure 2 Supra Containers

We also observe that smart contracts are naturally grouped, especially in a setting where no *dynamic dispatch* is used. Ethereum Virtual Machine (EVM) supports dynamic dispatch but the original Move [3] and most of its variants [2, 24] do not support it. However, Aptos

allows a limited version of dynamic dispatch in the way that transactions themselves can contain scripts so that functions across different smart contracts can be dynamically called.

3.1 Cross-container transactions and Access Control Specifications

It is more likely that many of the containers will not be self sufficient, meaning that they may use some assets or some DeFi applications such as a lending module, or an AMM, or some other on-chain component outside of their container. It is also not practical to expect that there will be multiple deployments of the same asset or the same DeFi dApp on multiple containers. Hence cross-container access becomes necessary.

Also if all such accesses are completely allowed then the meaning of Supra container is lost, as all containers collapse into one. So, on a closer observation we see a requirement of a cross-container access control specification by the container adminstrators mandating what modules of which containers are accessible by the users of a given container. Such access control specifications are illustrated in Figure 2 using open and closed gates.

As detailed in our Supra Technology Overview whitepaper[1], the Supra network is amenable to state sharding. So Supra network will have validator-committees called *clans* hosting state-shards and executing transactions related to only the hosted state-shard. In such a scenario a cross-container transaction may be of two kinds: intra-shard or cross-shard. To execute a intra-shard cross-container transaction it is sufficient to have a deterministic context, such as to execute at the beginning of a block. Executing a cross-shard cross-container transaction boils down to the bridging problem. Here we employ a *family* of nodes (set of nodes with at least one honest node) from the Supra network to run Supra's HyperNova [18] interoperability solution to execute such transactions.

3.2 Features of App-chains / Supra Containers

Localized Fee Markets. A very important requirement for app-chains / containers is the protection against the unnecessary influence on transaction fees from the transactions targeting smart contracts outside of this container. Essentially, the transactions compete for block space. Consequently, when there is a high volume of transactions, the transaction fees naturally increase to disincentivize users from submitting unnecessary additional transactions. If a transaction is assured a place in the block, then there is no impact on the global rate of incoming transactions. Hence, a natural way to provide this protection to the transactions targeting a container is to provide quotas for containers with respect to the block space. Note that the transaction fees may still increase based on the traffic inside containers. Hence the term *localized fee market*.

This opens up an interesting study of auctioning block space by introducing another entity called *block space lessee*, who reserve block spaces by predicting container-wise transaction traffic. This opens up revenue generation options for the container administrators and for their community. The readers are referred to our breakthrough research paper [**?**] for more details.

Customized Fees. The community governing a container may institute its own transaction fees for various kinds of transactions targeting the container. The fees may also be in its own desired asset kind, and not necessarily be in the Supra network's gas token - Supra coin. This may be easily enabled using one of the *account abstraction* methods.

[1] Supra Technology Overview whitepaper will be available soon on Supra.com.

Sovereignty. More generally, a community governing a container gains sovereignty over that namespace. This community may act as a owner or a guardian or a caretaker of its container in various respects. It may regulate the deployment and upgrading of smart contracts in its namespace. This in turn regulates the existence of various assets and their supply for its community. It may place an eligibility criteria for any user to participate and use the services offered by the container. These criteria could be in the form of a possession of a asset (for example, possession of an NFT related to a real-world asset - RWA), or a deposit in a certain vault, or some attributes related to the user (for example, above 18 years of age), etc.

4 Benefits inherited from the Supra Platform

Supra Containers naturally inherit the advantages of Supra's network and its services, and the prominent ones are listed below:

Security The containers are hosted on Supra's Layer 1 blockchain, meaning they share the security guarantees of Supra's network, which are detailed in the Supra Technology Overview whitepaper[1]. There is no exclusive set of validators responsible for containers, but all containers share the same validators set from the Supra network.

Vertically Integrated and Cross-chain Services As detailed in the Supra Technology Overview whitepaper, all the services hosted on Supra network such as DORA [5] for Oracle feeds, VRF [13] for on-chain randomness, zero-block delay Automation, and Supra's IntraLayer for cross-chain services become available for the applications deployed in a container. So this turns any dApp into a *Super-dApp* from the get-go.[2]

State Sharding Supra containers complement state sharding. Traditionally sharding [15] has been the first approach to scale execution. However the complexities of demarcating the states across various shards so that a consistent global state is maintained has proven to be the core challenge. Semantically, sharding and the container abstraction are seemingly the same. Container abstraction provides this required demarcation such that several containers can be packed into a single shard, and as long as the state shards respect container boundaries – by not splitting a container across shards – this challenge is tamed.

DeFi Containers also enable the separation of jurisdictions. Consider the cases of public DeFi and nation state-regulated DeFi. Such cases typically come with a mandate to disallow interactions between the services of one with the other. Privacy-preserving and publicly-visible services can be another class. Here too, container abstraction helps the governance bodies of these classes to structure their offerings without violating any mandates as separation is guaranteed via containerization.

DevX Towards making strides in developer experience (DevX), Supra is working to enable the simple plug-and-play creation of containers or app-chains. Developers will be presented with the options of which applications their containers should be initialized with. Developers may choose lending protocol X from an array of choices, and another Automated Market Maker Y from another set, and build a dApp in their own exclusive container.

[2] Supra zero-block delay Automation whitepaper will be available soon on Supra.com.

5 Leveraging Containers for Parallel Execution

In this section we investigate into the benefits of grouping smart contracts into containers towards optimizing the execution time.

From the perspective of executing transactions, containers may be seen as an access specification. So we posit this study amidst the spectrum of input specifications of storage locations that are read by or written to by the transactions. On one extreme end, transactions come with no specifications – we call *access set oblivious* model, e.g., Ethereum [9] and Aptos [1] transactions. On the other extreme end, transactions come with full specification of read and write sets – we call *access set aware* model, e.g., Solana [26] and Sui [23]. It is well noted by the community that the transaction sizes bloat and occupy more block space with full specifications. We find the specifications from the container abstraction to be in the middle of these extremes. As they do not add any extra information to the transactions themselves, they don't bloat the transactions burdening the network bandwidth to disseminate the transaction data. In the rest of the paper, we show that the execution time benefits yielding from the container abstraction is significant and is comparable with the full specification, and hence this abstraction is a secure and efficient optimization.

We now present *Supra-work-sharing queue* (*Supra-WSQ*) scheduler, a simple algorithm that leverages the container abstraction and achieves maximal parallelism. This algorithm works analogously to the following real-world scenario. Consider that there are many clerks and a stream of incoming clients who require a clerk's signature on a document. The clients form a queue such that whoever is in the front of the queue walks to any available clerk. The clerk gets busy working through the document provided by the client and concludes the session with the client by either signing the document or not. The clerks are never idle as the next client approaches a clerk as soon as the current client finishes his session. This protocol is depicted in Figure 3.

Figure 3 Supra-Work-Sharing Queue (Supra-WSQ) Algorithm

Recall that the container abstraction requires the transactions to target a single container. We term an ordered list of transactions targeting a single container as a *batch of container transactions*, or simply a batch, when the context is clear. Now we apply the aforementioned real-world scenario to our execution model by mapping clerks to the cores and the clients to the batches of container transactions.

We present the pseudocode of Supra-work-sharing queue scheduler in Algorithm 1. The procedure *init* sets all the cores (note that clerks are analogous to cores) to the available state. The procedure *Supra-work-sharing-queue* is the main procedure. It takes the batch which is in the front of the batch queue (given by $head(batches)$) and assigns to an available core (given by $getAvailableIndex(cores)$). Note that 'getAvailableIndex' needs to be an atomic operation. If no cores are available, the algorithm waits until at least one becomes

Algorithm 1 Supra-Work-Sharing Queue Scheduler for Containerized Batch Execution.

```
7  Procedure init():
8      forall i: cores[i] = available
9  Procedure Supra-work-sharing queue(cores[], batches[]):
10     while batches not empty do
11         execute(head(batches), getAvailableIndex(cores));
12 Procedure execute(b, ci):
       // The core cores[ci] starts executing the batch b.
13     cores[ci] = busy;
14 Procedure complete(ci):
15     b = batchBeingWorked(ci);
16     remove(b, batches);
17     cores[ci] = available; // The core cores[ci] finished executing b and is available.
18 Procedure getAvailableIndex(cores):
19     wait if no ci: cores[ci] = available;
20     return ci: cores[ci] = available;
```

available. Then, the assigned core initializes the execution of its batch. Upon completion of the execution, the core becomes available for execution of subsequent batches in the queue. Because the Supra-work-sharing queue algorithm uses multiple cores for the sequential execution of batches, it is faster than executing all the batches sequentially on a single core when multiple batches from different containers are to be executed.

Consider an approach in which we utilize a scheduling algorithm or other technique that utilizes all the cores to execute a single batch. After the execution of a batch, the next batch is also executed in a similar fashion. Since we do not preclude the presence of conflicts in the execution of transactions inside a batch, naturally and necessarily these techniques need to account for such conflicts and facilitate its resolution using *aborts* and *retries*.

As there are no aborts and retries in the Supra-work-sharing queue algorithm, it executes even faster than these techniques in the presence of many conflicts among transactions. The important thing to note is that in these techniques are not exploiting the specification given by the container abstraction so it is not truly *apples-apples* comparison. Nevertheless, because the Supra-work-sharing queue algorithm leverages this specification based on container abstraction, it executes faster.

To reiterate, it is not the superiority of the Supra-work-sharing queue algorithm that gives an advantage in execution times, but rather the facility of the specification given by the container abstraction.

6 Related Work

In this section we review the state of the art parallel execution techniques pertaining to blockchain transactions and posit our containerized parallel execution technique.

In the access set oblivious model, the transactions do not come with any specification on the accounts or the storage locations that they may read or write. This is so in the case of Ethereum, Aptos and many other chains. Ethereum uses a sequential execution mechanism to process the transactions. However, Aptos utilizes parallel execution of transactions on multi-core processors by innovating a speculative Software Transactional Memory (STM) approach, called Block-STM [12].

In Block-STM, there are two tasks for each transaction: the execution task and the validation task, prioritizing tasks for transactions lower in the preset serialization order.

A transaction may be executed and validated several times by the *scheduler*, the term i^{th} *incarnation* refers to the i^{th} execution of a transaction. An incarnation is terminated when the scheduler determines that a subsequent re-execution with an increased incarnation number is required. For each incarnation i of a transaction T_k, Block-STM maintains two transaction local buffers: a read-set$_k^i$ and a write-set$_k^i$. The account and respective versions that are read during the execution of the incarnation are contained in the read-set$_k^i$. The update is represented by (account, value) pairs and is stored in the write-set$_k^i$. For an incarnation i of T_k, the write-set$_k^i$ is stored in the multi-version in-memory data structure. Further, an incarnation of a transaction must pass validation once it executes. The validation compares the observed versions after re-reading (double-collecting) the read-set$_k^i$. Intuitively, a successful validation indicates that the write-set$_k^i$ is still legitimate and all reads in the read-set$_k^i$ are consistent, while an unsuccessful validation implies that the incarnation i must be aborted. When a transaction T_k is aborted, it implies that all higher transactions in preset order after T_k can be committed only if they get successfully validated afterwards.

By executing transactions according to a preset serialization order, Block-STM dynamically detects dependencies and avoids conflicts during execution, further ensuring that the outcome is consistent with sequential execution.

In the access set aware model, the transactions come up with a specification of all the accounts or the storage locations that they may read or write. Consider Solana [22]. The following is from the best of our understanding of Solana's documentation and code. We are happy to revise our paraphrasing of Solana's approach if we are made aware known of any errors or discrepancies. The client's wallet packs a transaction with all the accounts that are read from and written to by it, and sends it to the current and the next leader (leaders are determined in advance) via an RPC node. This read-write set specification helps in parallelizing the execution as illustrated by a simplified algorithm run by the leader (mimicking Solana's parallel execution) given in Algorithm 2. Figure 4 (taken from [25]) also depicts this scheme visually.

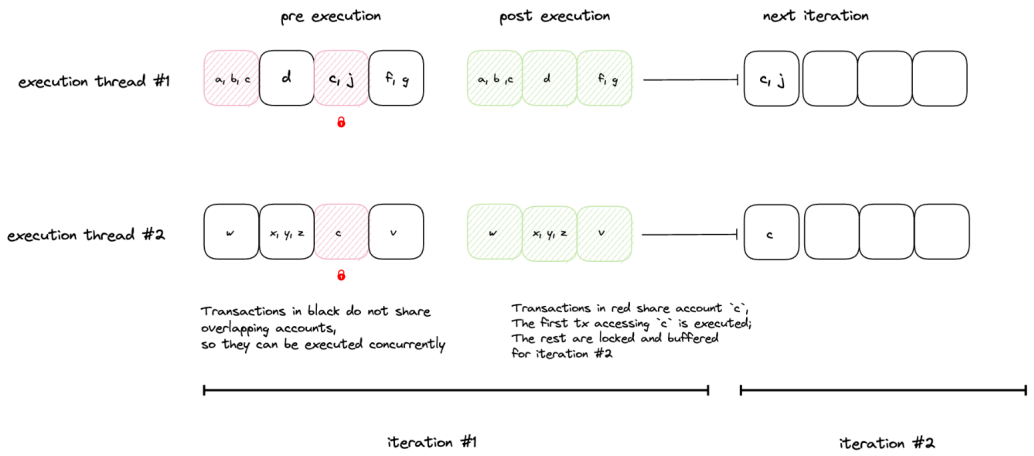

Figure 4 Solana's Parallel Execution Scheme

Let us consider the question of maximum number of iterations. As there are n cores, there are n lists. Given a list, we denote k to be the *contention factor*, meaning the maximum number of transactions in that list contending for a common account. Let $k_1, k_2, \ldots, k_n$ be the contention factors of each of the lists. Then the number of iterations will be

Algorithm 2 `SeaLevel()` scheduler for Solana parallel execution.

Data: Let n denote the number of cores.
Procedure `SeaLevel(`*txs[]*`)`**:**
  Transactions are partitioned into n ordered lists.
  Account Locking phase. The transactions in the lists try to lock their specified access (read and write) accounts respecting the order of the list. If a lock on an account is already held, then the locking fails. If the locking of any of the specified access accounts fails then that corresponding transaction is marked for next iteration.
  Execution. Execute the unmarked transactions in the list obeying its order.
  Form new ordered lists by including the marked transactions.
  Repeat from **Account Locking phase**, for the new lists.

$max(k_i), 1 \leq k \leq n$.

Theoretically speaking, Algorithm 2 performs the best when all transactions are executed in a single iteration. By increasing the number of iterations, the algorithm degrades in performance.

As mentioned earlier, the containers abstraction falls in between the access-set-oblivious model and the access-set-aware model and the rest of the paper shows the benefits from such an abstraction.

7 Experiments

In this section we study the execution latency and execution throughput of the aforementioned Supra-work-sharing queue (Supra-WSQ) approach by executing on multiple cores.

As discussed in Section 5, containerization groups transactions into different batches, each of which is executed sequentially, while batches of different containers are executed in parallel. The sequential execution of transactions within each batch will overcome the abort and re-execution overhead of speculative or optimistic execution. Hence, we can expect that batching will outperform speculative execution when the workload is inherently sequential, meaning transactions inside each batch are sequential by nature and form a chain of conflicts.

Experimental Setup and Implementation

We ran our experiments on a single-socket AMD machine consisting of 32 cores, 64 vCPUs, 128 GB of RAM, and running Ubuntu 18.04 operating system with 100 GB of SSD.

We design these experiments to evaluate containerized parallel execution with state-of-the-art approaches like Aptos's Block-STM [12], Solana's SeaLevel [27] and Ethereum's baseline sequential execution [9]. The experiments are performed on the benchmarking setup [4] of the Block-STM simulating in-memory (no access to persistent storage) read and write operations on Move-VM.

The input test workloads for all three approaches are the same. The Supra-work-sharing queue approach processes the transactions by grouping into batches and executing them in parallel across batches. The transactions from those batches are concatenated to form a list of transactions (preset serialization order) and given as input to Block-STM and SeaLevel.

Construction of Synthetic Tests

We now discuss how workloads are constructed and are grouped into multiple non-conflicting batches.

Table 1 Synthetic workload

Workload	Oprs/Tx	Txs/Block	Batches/Block	Keys/Batch
High Conflict ($\mathbf{W}_h$)	**20**	1000	100	1000
Moderate Conflict ($\mathbf{W}_m$)	**2**	1000	100	1000
Low Conflict ($\mathbf{W}_l$)	2	1000	100	**100000**

Synthetic workload means the workload of fake or constructed transactions. We analyzed historical transactions (≈30 million) of Solana Mainnet to better understand the account access patterns in order to construct a synthetic workload. We observe that transactions have a tail distribution, meaning that the majority of accesses in a block are made by a few accounts that are responsible for the majority of conflicts (read-write, write-write, and write-read), while other accounts have just a few accesses.

We construct a block to be made up of several batches. A batch is a group of transactions, and each transaction operates (reads and writes) on keys randomly chosen (from the batch assigned keys) following a log-normal distribution [14]. Here, we considered four parameters to generate the workloads: operations per transaction, transactions in the block, batches in the block, and the number of keys per batch. These parameters are interrelated, and changing them can increase or decrease conflicts in the block.

- Naturally, the number of conflicts in a batch increases as we decrease the number of keys. Further, increasing the number of operations per transaction increases the number of conflicts in a batch. We construct the *high conflict workload (W_h)* by allocating just two keys and performing 20 operations per transaction.We include 10 transactions per batch and 100 such batches in a block. This results in a workload with high-conflicts (more conflicts) pushing the batch towards sequential execution. The transactions in every batch operate by accessing a key space of size 10^3 randomly (following the log-normal distribution).
- When 10^3 keys are allocated per batch with 2 operations per transaction we find that the conflicts reduce, and so we generate a *moderate conflict workload (W_m)*. Here we include 10 transactions per batch and 100 such batches in a block.
- When we increase the key space allotted per batch to 10^5 with a uniform random key distribution for transactions to access them, the number of conflicts decreases further. We construct the *low conflict workload (W_l)*, by fixing 2 operations per transaction, 10 transactions per batch, and 100 such batches in the block, where each batch is allocated 10^5 keys.

All the above mentioned workloads are succinctly represented in Table 1.

Assumptions

Our experiments and evaluation are under the following assumptions:

A1 There are more batches in the block than the number of cores available to start with.

A2 The batches contain transactions targeted for a single container, meaning there are no cross-container transactions. We plan to consider and analyze cross-container transactions in the next version of this paper.

A3 The workload is homogeneous meaning batches contain equal number of transactions.

Evaluation

For each workload (from Table 1), we execute blocks for Supra-work-sharing queue (containerized batching), Block-STM, SeaLevel, and sequential execution. The sequential execution serves as a baseline. The histogram plots in Figure 5 − Figure 7 show an average execution *throughput (tps)* on the Y-axis, while the number of threads varies on the X-axis from 2 to 32. The experiments were carried out 52 times; the first 2 executions are left as warm-up runs, and each data point in the histograms is an average of 50 executions.

The histogram plots for the average tps on high conflict workload W_h can be seen in Figure 5. As shown, Supra-work-sharing queue outperforms as the number of threads increases since there is no speculative execution overhead of abort and re-execution. Due to increased contention at the shared queue, there is a decline in performance for Supra-work-sharing queue at 32 threads. Although throughput for both Block-STM and SeaLevel increases with threads, the Block-STM performance falls short of Supra-work-sharing queue due to increased aborts and re-execution (re-validation) overhead; similarly, in SeaLevel, there will be more iterations with fewer transactions. The optimal throughput for Supra-work-sharing queue is $\approx 98k$ tps at 24 threads. Block-STM and SeaLevel reach a peak throughput of $\approx 54k$ and $\approx 52k$ tps at 32 threads, respectively.

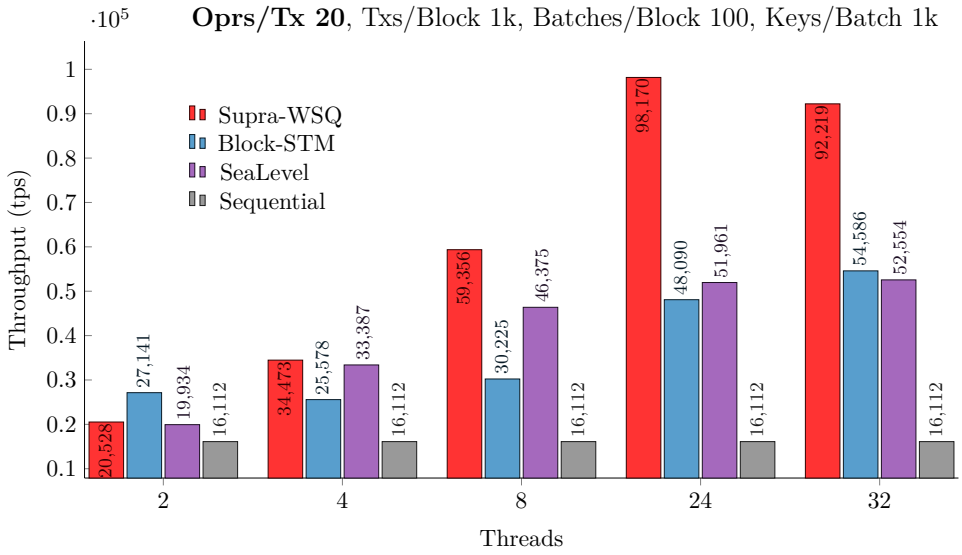

Figure 5 Synthetic workload W_h: high conflicts within each batch

The workload W_m has fewer conflicts than W_h. Figure 6 shows that all three of the parallel execution approaches outperform sequential execution in this case and perform neck-to-neck with one another. The execution overhead in Block-STM is moderate because there aren't many conflicts; likewise, SeaLevel will have a moderate number of iterations with multiple transactions. Block-STM performs more effectively with an optimal tps of $\approx 102k$ at 24 threads. At 32 threads, the maximum throughput of Supra-work-sharing queue is $\approx 91k$, while SeaLevel reaches $\approx 77k$ tps.

Figure 7 shows the histogram plot for W_l with low conflicts in the batches. The plots demonstrate how the performance of Block-STM, Supra-work-sharing queue, and SeaLevel increases with the number of threads, peaking at $\approx 127k$, $\approx 100k$, and $\approx 114k$, respectively. All three approaches have the maximum throughput because blocks have the high concurrency in this workload. The Block-STM outperforms other approaches as it utilizes the threads in

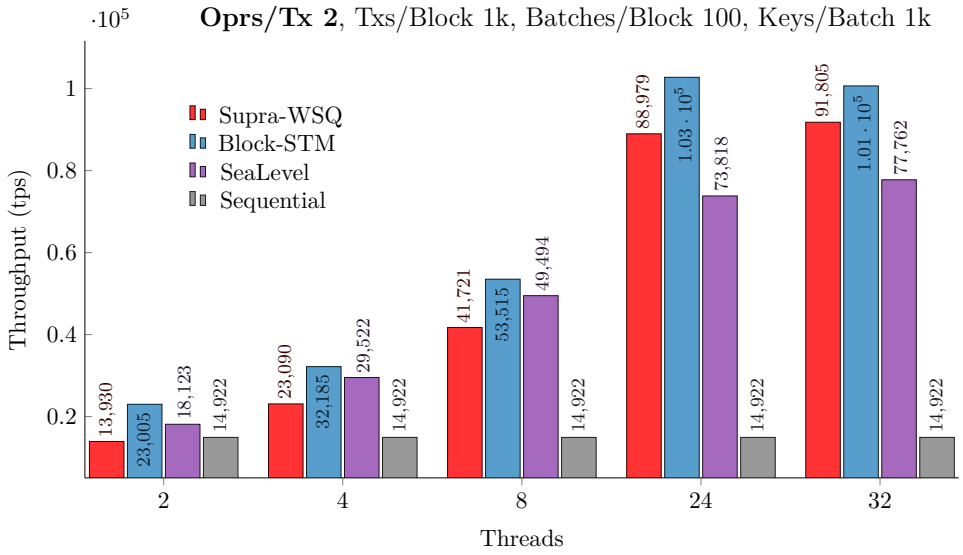

Figure 6 Synthetic workload W_h: moderate conflicts within each batch

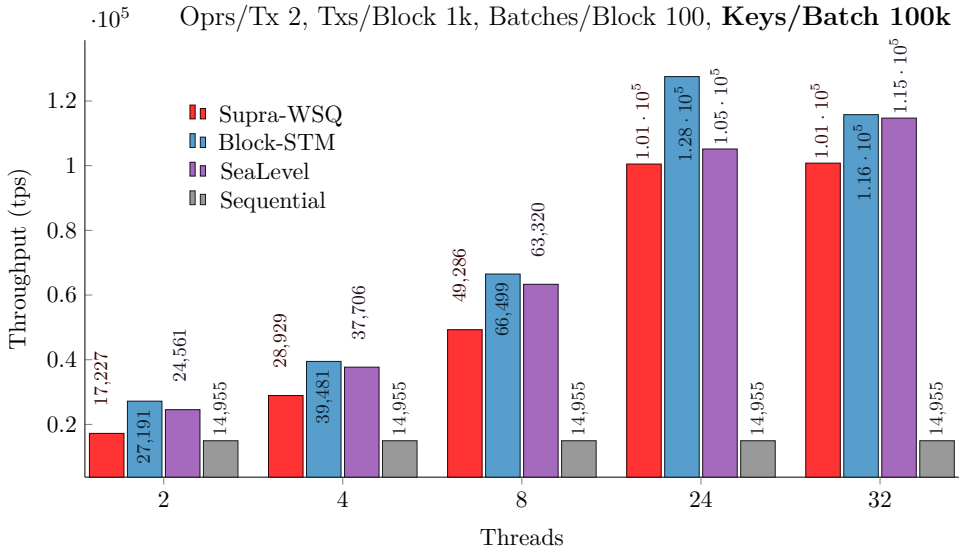

Figure 7 Synthetic workload W_l: low conflicts within each batch

a more balanced way than Supra-work-sharing queue and SeaLevel.

Since there are no aborts and re-executions in containerized batching, in the presence of many conflicts among batch transactions (as we have seen in Figure 5), it outperforms other techniques. Moreover, when there are moderate (Figure 6) or fewer conflicts (Figure 7), it matches with the performance of other parallel execution approaches. The proposed Supra-work-sharing queue approach achieve $\approx 6\times$ speedup over sequential execution under all synthetic workloads.

Testing on Real-World transactions

We downloaded ≈30 million historical transactions from publicly available 15k blocks (205465000 – 205480000) from Solana [26] using the `getBlock()` JSON-RPC Solana API

Table 2 Solana Historical non-voting transaction workload (W_{ht})

Block Number	Oprs/Txs				No. of Batches	Maximum Txs in a Batch	Total Txs	Keys
	Average		Maximum					
	Reads	Writes	Reads	Writes				
205465004	5	7	23	37	50	118	382	1007
205465000 - 205465049	6	7	34	41	171	1321	5626	5354

of Quicknode [17]. We chose Solana because transactions are made up of account access information, an array of accounts to read from (*read-set*) or write to (*write-set*) [22]. This information aids in the construction of the batches for containerized execution. The real-world block transactions show heterogeneity in the workload, meaning when transactions are batched they are not necessarily of the same sizes. Transactions may also not all have the same number of reads and writes unlike in our synthetic tests.

We show the result of two experiments here: one on a randomly selected block - Block number 205465004 of Solana blockchain, another on a block constructed by concatenating consecutive 50 blocks from Block number 205465000 to 205465049 of Solana blockchain. The Table 2 show more details of these tests. We remove the Solana voting transactions from these blocks. The containerized batches are constructed by placing conflicting transactions in different batches. For any two transactions i and j if either: $\textit{read-set}_i \cap \textit{write-set}_j$ or $\textit{read-set}_j \cap \textit{write-set}_i$ or $\textit{write-set}_i \cap \textit{write-set}_j$ is non-empty then they are placed in different batches.

The histogram plots for the throughput in average tps on block 205465004 from Solana are shown in Figure 8. As shown, the containerized abstraction proves more valuable as the number of threads increases since there is no speculative execution overhead of abort and re-execution. Although throughput for both Block-STM and SeaLevel increases with threads, the Block-STM performance falls short of Supra-work-sharing queue due to increased aborts and re-execution (re-validation) overhead; similarly, in SeaLevel, there will be more iterations with fewer transactions. The optimal throughput for Supra-work-sharing queue is $\approx 3k$ tps. Block-STM and SeaLevel reach a peak throughput of $\approx 1k$ and ≈ 800 tps,

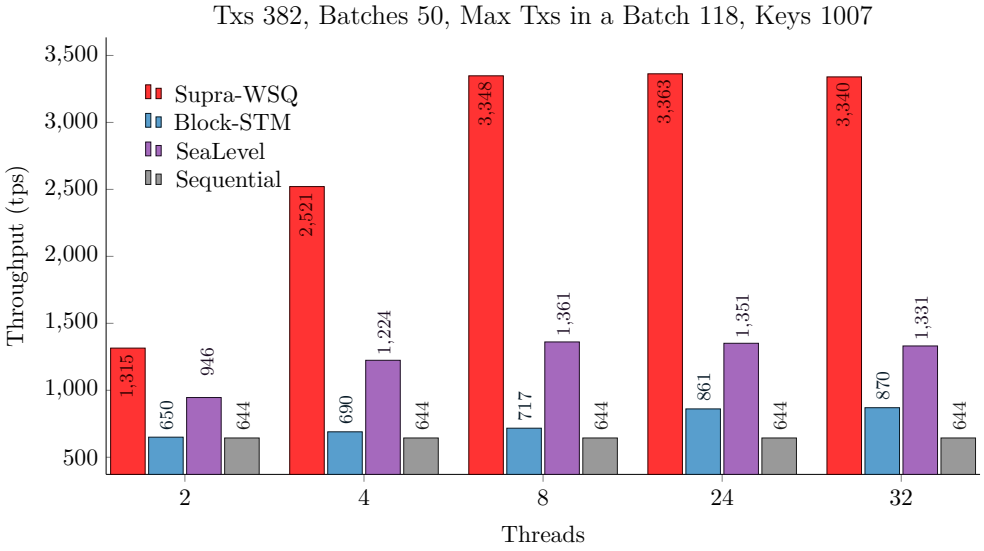

Figure 8 Historical workload W_{ht}: Block 205465004

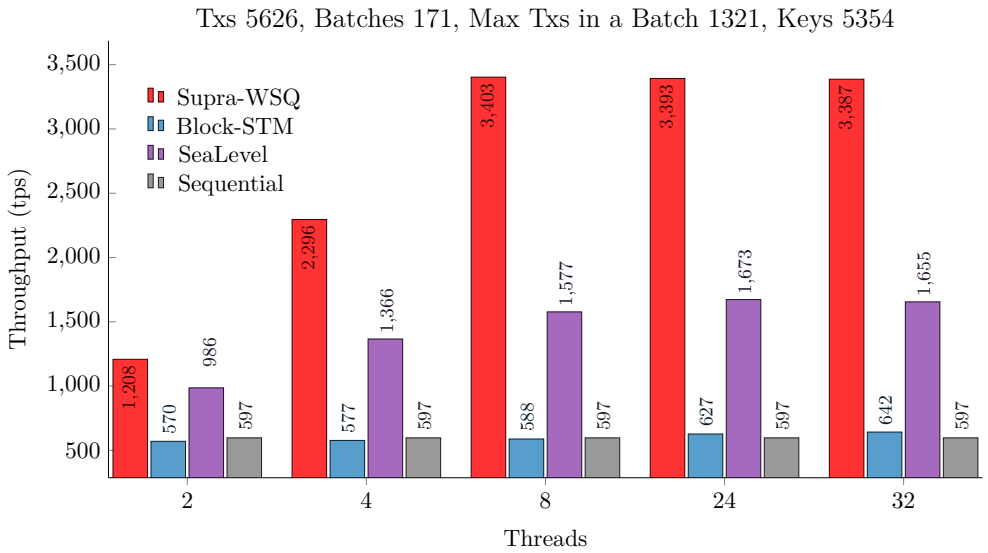

Figure 9 Historical workload W_{ht}: Block 205465000 - 205465049

respectively. Similarly, on the concatenated Solana blocks: from 205465000 to 205465049, the performance trends remain the same for Supra-work-sharing queue and Block-STM, reach a peak throughput of $\approx 3.4k$ and ≈ 642 tps, respectively, as shown in Figure 9. However, the SeaLevel performance has seen upward trends and achieves an optimal throughput of $\approx 1.5k$ tps, due to the increased number of transactions per iteration.

In summary, the analysis of real-world transactions of the Solana blocks shows that the container abstraction provides more value in parallel execution than existing state-of-art parallelization techniques.

8 Conclusion

We introduced Supra containers as an answer to the requirements of app-chains by creating namespaces inside blockchain state.

We observe that the container abstraction technique may be used as access specification that can be leveraged towards executing transactions in parallel. Through our experiments we demonstrate that such access specification gives a boost in performance comparable to standard parallel execution techniques and in some scenarios outperforms them too.

In the next version of this paper, we plan to include studies on cross-container transactions, parallelized execution inside a containerized batch, and ways to utilize adaptive schedulers after assessing the input block of transactions.

References

1 The aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure. `https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf`, August 2022. [August 11, 2022].

2 Move on aptos. `https://aptos.dev/move/move-on-aptos/`. [Online: accessed 22 Sep 2023].

3 Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. Move: A language with programmable resources. *Libra Assoc*, page 1, 2019.

4 Block-stm implementation. https://github.com/danielxiangzl/Block-STM. [Online: accessed 15 January 2024].

5 Prasanth Chakka, Saurabh Joshi, Aniket Kate, Joshua Tobkin, and David Yang. Oracle agreement: From an honest super majority to simple majority. In *43rd IEEE International Conference on Distributed Computing Systems, ICDCS 2023, Hong Kong, July 18-21, 2023*, pages 714–725. IEEE, 2023. doi:10.1109/ICDCS57875.2023.00025.

6 Defi kingdoms. https://defikingdoms.com/. [Online: accessed 3 September 2024].

7 Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 303–312, New York, NY, USA, 2017. PODC '17, ACM. doi:10.1145/3087801.3087835.

8 dydx v4. https://dydx.exchange/blog/dydx-chain. [Online: accessed 3 September 2024].

9 Ethereum (ETH): open-source blockchain-based distributed computing platform. https://www.ethereum.org/. [Online: accessed 15 January 2024].

10 Mohamed Fouda. The case for parallel processing chains. https://medium.com/alliancedao/the-case-for-parallel-processing-chains-90bac38a6ba4, September 2022. [Online: accessed 10 January 2024].

11 Sui Foundation. All about parallelization. https://blog.sui.io/parallelization-explained/, January 2024. [Online: accessed 23 January 2024].

12 Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 232–244, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3572848.3577524.

13 Aniket Kate, Easwar Vivek Mangipudi, Siva Maradana, and Pratyay Mukherjee. Flexirand: Output private (distributed) vrfs and application to blockchains. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1776–1790, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3576915.3616601.

14 Log-normal distribution. https://en.wikipedia.org/wiki/Log-normal_distribution. [Online: accessed 13 Jun 2024].

15 Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 17–30, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976749.2978389.

16 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://www.bibsonomy.org/bibtex/23db66df0fc9fa2b5033f096a901f1c36/ngnn, 2009.

17 Quicknode: Solana rpc. https://www.quicknode.com/docs/solana. [Online: accessed 10 January 2024].

18 Supra Research. Hypernova: Efficient, trustless cross-chain solution. Technical report, https://supra.com, 08 2023. URL: https://supra.com/docs/Supra-HyperNova-Whitepaper.pdf.

19 Architecture documentaion v1.2. https://sawtooth.hyperledger.org/docs/1.2/architecture/transaction_scheduling.html. [Online: accessed 10 January 2024].

20 Sei- the layer 1 for trading. https://docs.sei.io/advanced/parallelism. [Online: accessed 22 Sep 2023].

21 Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, page 204–213, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/224964.224987.

22 Solana documentation. https://docs.solana.com/. [Online: accessed 10 January 2024].

23 Sui documentation: Discover the power of sui through examples, guides, and concepts. https://docs.sui.io. [Online: accessed 10 January 2024].

24 Move on sui. https://docs.sui.io/concepts/sui-move-concepts. [Online: accessed 22 Sep 2023].

25 Umbraresearch. Lifecycle of a solana transaction. https://www.umbraresearch.xyz/writings/lifecycle-of-a-solana-transaction. [Online: accessed 15 January 2024].

26 Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper*, 2018.

27 Anatoly Yakovenko. Sealevel - parallel processing thousands of smart contracts. https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192, September 2019. [Online: accessed 23 January 2024].