IEEE DAG-DLT'2024 1571004760

A Study on Shared Objects in Sui Smart Contracts

***Abstract*—In many smart contract architectures, every contract or object is mutably shared by default. The Sui smart contract platform bears the unique feature of distinguishing between shared and owned objects. While transactions operating on shared objects require consensus to sequence reads and writes, those involving only owned objects are independent and may bypass consensus; thus, the latter are less prone to this throughput bottleneck. However, it may not always be possible or desirable to avoid using shared objects. This article aims at identifying and investigating decentralized applications that require shared objects. Utilizing the Sui Rust SDK to query programmable transaction blocks, we analyze the frequency of transactions involving shared objects, shared resource contention levels, and most "popular" applications that contain shared objects. The presented results are reproducible and show the extensive usage of shared objects in Sui, low contention levels, and moderate dependency among shared objects in atomic transactions. This novel study of shared object use cases in a relatively new smart contract platform is important for improving the efficiency of such object-based architectures. This work is relevant for smart contract platform designers and smart contract developers.**

***Index Terms*—shared objects, Sui, smart contracts, shared state, decentralized applications**

I. Introduction

As blockchain technologies thrive, choosing the right smart contract platform becomes supreme for building dependable and flexible decentralized applications. Ethereum Solidity, Cardano Plutus, Sui Move are examples of prominent smart contract frameworks, with Ethereum being the first and continuing to be the most popular platform for smart contracts [1].

In Ethereum, as well as in many other smart contract platforms, every contract or object is public and mutably shared by default, and it can be thought of as an open API. This means anyone can call other smart contracts in their own smart contracts [2]. Ethereum smart contracts are not controlled by a user. Instead, they are deployed to the network, run as programmed, and are controlled by the logic of the smart contract code. All variables/objects on Ethereum are accessible for reading by everyone in the contract storage. The only way to interact with a smart contract is through a function call. To prevent anyone from calling certain functions, the contract creator needs to implement additional access control rules. For example, the contract code may require the message sender to match the contract owner's address to allow writing to a variable, meaning that only the contract owner can call the function that mutates the variable. This technique is very similar to the embedded permission pattern that restricts the invocation of individual functions to a permissioned set of accounts by embedding permission controls into the contract [3].

Similarly to Ethereum, Cardano smart contracts are publicly shared and available to everyone, unless the contract code includes additional access control logic. However, compared to Ethereum, concurrent operations on smart contracts are currently not possible in Cardano. Specifically, Cardano's Extended UTXO (EUTXO) model is limited to one state change (transaction) per block per smart contract [4], [5]. In other words, a shared resource (i.e., script EUTXO) can only be accessed once per block. Whenever a user needs to interact with a Cardano smart contract, they need to lock that contract for one block, which means that if other users want to interact with the same contract, only one interaction (transaction) will succeed, as UXTOs can only be spent once. This bottleneck limits the throughput of DeFi applications and also complicates the development of dApps that require shared resources on Cardano, as smart contract developers have to address concurrency and contention issues. Various solutions to this concurrency problem in Cardano have been proposed, and usually involve an off-chain third party (e.g., bots, batchers) that collects concurrent transactions and executes them as a single transaction [4], [6].

The Sui smart contract platform finds a middle ground between the Ethereum and Cardano smart contract architectures by employing an object-based model and distinguishing between shared and owned (single-writer) objects [7]. In general, a smart contract developer should prefer owned objects to shared ones whenever it is reasonable or possible to do so: in contrast to transactions operating on shared objects (which require sequencing and, thus, lead to a throughput bottleneck), transactions involving only owned objects may bypass consensus and do not need to be sequenced. However, it may not always be possible to avoid shared objects. In many applications, multiple users can interact with the same contract and may want to do it at about the same time. A decentralized exchange (DEX) is an example of such an application, where multiple swap transactions want to operate on the same liquidity pool, and it is often necessary to have a global view of all liquidity or total token supply to determine actions. Moreover, some DeFi concepts like constant-product Automated Market Maker (AMM) are only possible with shared objects [8].

This work aims at identifying and investigating decentralized applications that require shared objects. In particular, we analyze how often shared objects are involved in transactions on Sui and the level of contention and dependencies for shared objects. Our motivation for this analysis is that understanding shared object use cases is a crucial step in improving the efficiency of object-based smart contract platforms. Furthermore, this work may be helpful in designing a more flexible and fine-grained model for shared objects than Sui's one.

Some of the key findings and contributions of this work are as follows: (i) shared objects are extensively used in Sui; (ii) contention levels are relatively low; (iii) dependencies among

shared objects in an atomic transaction are moderate; and (iv) liquidity pools are a quite popular use case of shared objects.

The structure of this paper is as follows. In the next section, we provide a detailed description of the question and describe the methodology used in this work. Section III is devoted to the presentation and interpretation of the results. In the last section, we conclude our findings and discuss future research.

II. Methods

Sui is a layer-1 smart contract platform based on an object-centric model. The Sui ledger stores a collection of programmable objects, each with a globally unique identifier. The ability to distinguish between different kinds of objects (owned and shared objects) is a unique feature of Sui [7].

Owned objects are the most common type of object in Sui. Many transactions and operations with assets (such as asset transfers, NFT minting, smart contract publishing, etc.) can be designed using exclusively owned objects. An owned object is a *single-writer*, meaning that only the owner can access it via a read or write operation at a time. Since only a single owner can access objects of this type, transactions involving exclusively owned objects can be executed in parallel with other transactions that have no objects in common. In other words, transactions involving only owned objects do not need to be sequenced, and thus, they may bypass consensus in Sui.

Shared objects can be accessible to everyone for reading or writing on the Sui network. If needed, extended functionality and accessibility of shared objects require employing additional access logic. Some use cases (such as the pull model of price feed updates, an auction with open bidding, or a central limit order book) require shared objects. A shared object is a *multi-writer*, meaning it can be accessed by two or more users simultaneously. In contrast to transactions that involve only owned objects, transactions operating on shared objects require consensus to sequence (order) reads and writes.

Since shared objects may be required to implement the logic of accessibility by multiple users simultaneously, it is not always possible or desirable to use only owned objects. By analyzing Sui's object-based ledger, this work aims to address the following questions: *How often shared objects are used in Sui? What are the use cases for shared objects? What are the most popular applications utilizing shared objects on Sui? How high are the levels of contention and dependency among shared objects in Sui?* In the next section, we define some concepts and metrics used to address these questions and describe how data was collected for this study.

A. Terminology and Metrics

Definition 1 (Object)**.** An object is the basic unit of storage.

Definition 2 (Owned objects)**.** An owned object is an object owned by an address (or another object) and can only be used in transactions signed by the owner address at a time.

Definition 3 (Shared objects)**.** A shared object is an object that does not have a specific owner. Anyone can read or write (if additional access control rules are not set) shared objects.

Definition 4 (Epoch)**.** In Sui, each epoch takes ≈ 24 hours.

Definition 5 (Checkpoint)**.** A checkpoint (also called sequence number) in Sui changes approximately every $2-3$ seconds.

To estimate how often Sui transactions operate on shared objects and evaluate shared resource contention levels, new concepts and metrics are introduced and defined as follows.

Definition 6 (Shared-object transaction)**.** A shared-object transaction has at least one shared object in its inputs.

Definition 7 (Density)**.** The density is the ratio of the number of shared-object transactions to the number of all transactions.

Definition 8 (Interval)**.** An interval is a period of time expressed in the number of checkpoints.

Definition 9 (Contention)**.** Contention is a situation when multiple transactions touch the same shared object at the same time, i.e., concurrently access that shared object.

Definition 10 (Contention degree)**.** Contention degree is the ratio of the number of shared-object transactions (within some interval) to the number of shared objects touched by those transactions (within the same interval).

Definition 11 (Contended fraction)**.** Contended fraction is the ratio of the number of shared objects (within some interval) touched by more than one transaction to the total number of shared objects (within the same interval).

B. Data collection

For this analysis, we used the Sui Rust SDK [9] to query all Sui programmable transaction blocks starting from epoch 0 (April 12, 2023), i.e., the genesis, until epoch 315 (February 22, 2024), inclusively, which resulted in a total of $1,103,018,902$ transactions on the Sui mainnet. The source code for this analysis is publicly available on GitHub [10].

Bullshark Quests: Before proceeding to the interpretation of the results in the next section, it is worth mentioning Sui's *Bullshark Quests* (*BQs*), which have been an ongoing initiative offering the opportunity to earn rewards by engaging with applications on Sui [11]. Each BQ is announced and launched by for a specific period of time. As we will see later in the next section, the metrics defined in Section II-A may be significantly different during the periods of BQs compared to those during periods when the quests did not take place. The time frames of BQs until epoch 315 are as follows:

- *BQ-1* started on July 6, 2023 (epoch 85) and ended on July 27, 2023 (epoch 106) [12].
- *BQ-2* started on July 28, 2023 (epoch 107) and ended on September 5, 2023 (epoch 146) [13].
- *BQ-3* started on October 12, 2023 (epoch 183) and ended on November 9, 2023 (epoch 211) [14].
- *Winter Quest* (*WQ*) started on December 18, 2023 (epoch 250) and ended on December 26, 2023 (epoch 258), when all rewards were claimed [15].

III. Results and Discussion

In this section, we estimate how often Sui transactions operate on shared objects and evaluate shared resource contention levels using metrics defined in Section II-A.

A. Number of transactions

Even though the number of transactions per epoch is not of a particular interest in this analysis, we begin by presenting this metric as it is used to calculate and interpret the density.

Figure 1 shows the total number of transactions (on a log scale graph) processed by the Sui mainnet per epoch.[1] As it can be seen, the number of transactions per epoch generally increased during the periods of BQs (depicted as vertical spans of different colors) compared to their number at epochs during which BQs did not take place. A significant increase in the number of transactions per epoch can be observed during the period of BQ-1 (vertical light red span). Overall, the number of transactions per epoch was slightly higher than one million during periods without quests for epochs after BQ-3 ended.

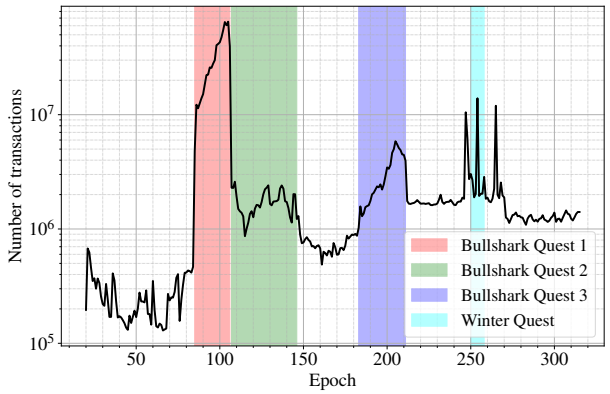

Fig. 1. Number of transactions per epoch in Sui. Note a log scale on y-axis.

B. Density of shared-object transactions

Recall from Definition 7 that the density is defined as the ratio of the number of shared-object transactions to the total number of transactions within some time interval. Figure 2 depicts the density of shared-object transactions per epoch.

As it can be seen in Figure 2, Sui transactions extensively operate on shared objects overall, especially starting from epoch 107 (BQ-2 start). At the beginning, starting from epoch 20, the density increased to approximately 0.5 and fluctuated around this value until epoch 35—during this period, many frequently used shared-object applications, such as `Sui Framework` [16], `Pyth Network` [17], `Cetus` [17], `Kriya DEX` [18], `Turbos Finance` [19], were deployed on Sui [10]. After this period, a trend of decreasing density to a value of 0.2 can be observed until epoch 80, after which it

[1]It is worth noting that in this and the following figures we plot the metrics starting from epoch 20 since the Sui network was in a bootstrapping phase during earlier epochs and there were almost no shared objects.

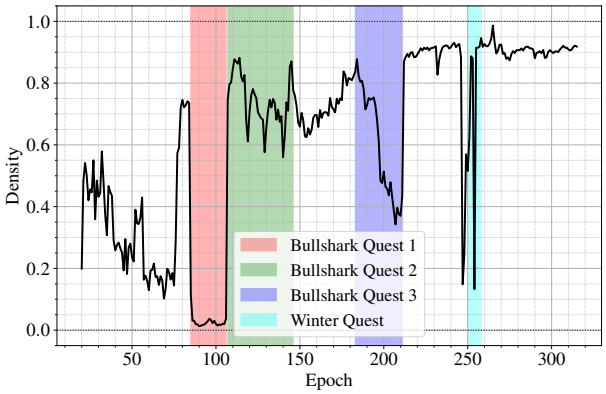

Fig. 2. Density of shared-object transactions per epoch in Sui.

peaked at ≈ 0.75. This peak can be explained by the deployment of new frequently used shared-object applications, such as `DeepBook` [20] and `DeSuiLabs Coin Flip` [21].

During BQ-1 (i.e., from epoch 85 until 106), the density dramatically decreased to very low values (< 0.05). To better understand this decline, we refer to Figure 1, which shows a significant rise in the number of transactions per epoch during the BQ-1 period. Taking this observation and low values of density into account, it can be concluded that transactions related to BQ-1 did not use shared objects extensively.

For BQ-2, it can be observed that the corresponding period is characterized by the number of transactions per epoch being smaller by one order of magnitude than that for the BQ-1 period (see Figure 1). Despite this, the density dramatically increased once BQ-2 started and remained high (fluctuating around 0.7) until BQ-3 (see Figure 2), which indicates that transactions in BQ-2 extensively operated on shared objects.

Starting from the BQ-3 start, the density gradually decreased to ≈ 0.4 over the corresponding period. After BQ-3 ended, the density of shared-object transactions reached extremely high values and fluctuated around 0.9, except for a few epochs before WQ and during its period when it dropped below 0.2. As can be concluded, shared objects are in general extensively used in Sui, especially when the network becomes more mature (from epoch 211 until epoch 315, see Figure 2).

C. Contention degree

The density is quite a simple metric that provides an overall picture of how often shared objects are involved in transactions. However, this metric does not capture information about how many transactions *contend for* (operate on) the same shared object within some time interval. Consider one scenario with many shared-object transactions, each operating on a different shared object, and another scenario when the same number of shared-object transactions operate on a small number of shared objects (assume the total number of transactions in both cases is equal). In both scenarios, the density of shared-object transactions would be the same, even though these two cases are moderately different. Since shared objects are usually

multi-writers, multiple transactions may contend for the same object (the second scenario), which requires sequencing for execution. However, in the first scenario, there is no contention as each transaction operates on a different shared object, and thus, they are independent and can be executed in parallel. To capture such important details in our analysis, we use another metric called *contention degree* as defined in Definition 10.

Figure 3 shows the contention degree averaged over time intervals of various lengths, per epoch. That is, for a given time interval (see Definition 8), we count the number of shared-object transactions in the interval and divide that number by the number of shared objects used in the same interval. Such ratios are then summed up, and the sum is divided by the number of intervals in an epoch (see Definition 4). Since the sequencing of shared-object transactions in Sui happens on a per-checkpoint basis, the shortest time interval in our analysis equals one checkpoint (see Definition 5). Intervals of various lengths are used to investigate how contention levels change with increasing the interval duration.

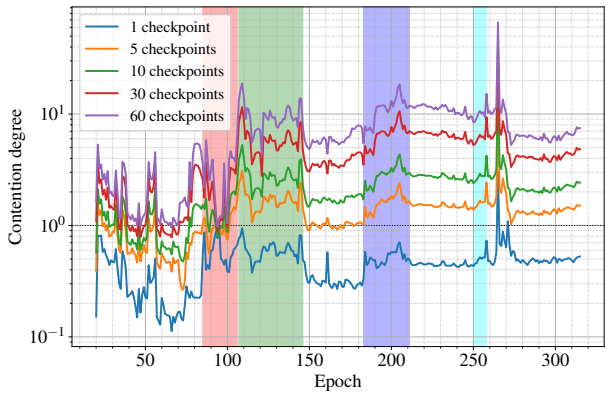

Fig. 3. Averaged contention degree per epoch for different time intervals in Sui. Note a log scale on y-axis.

From Figure 3, it can be seen that the average contention degree for the interval of one checkpoint is less than 1 almost across all epochs. A contention degree of 1 means that each shared-object transaction operates on a single different object on average, while higher values indicate multiple shared-object transactions contending for the same shared object. For intervals longer than one checkpoint, we can observe a significant rise in the average contention degree after BQ-1 ended, which indicates high involvement of shared objects in transactions when the network becomes mature. Finally, as expected, the longer the interval, the higher the contention degree, which implies that contention for shared objects will be less prominent in fast-committing consensus protocols.

D. Contended fraction

To further investigate contention for shared objects in Sui, we use a metric called *contended fraction* (see Definition 11), which gives the frequency of shared objects touched by more than one transaction within some time interval. Figure 4 depicts the contended fraction averaged over intervals of various lengths, per epoch. That is, for a given interval, we count the number of shared objects touched by more than one transaction in that interval and divide it by the number of shared objects. Such ratios are then summed up, and the sum is divided by the number of intervals in an epoch.

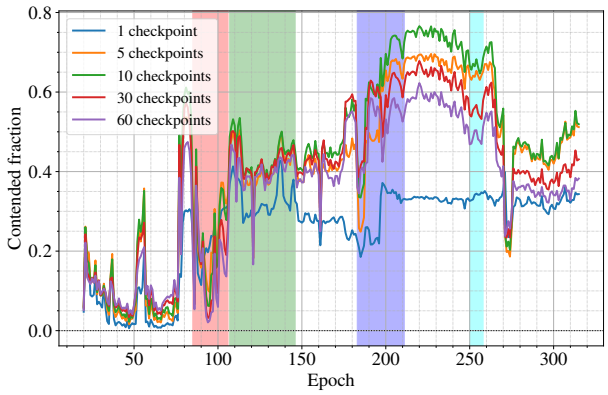

Fig. 4. Averaged contended fraction per epoch for different time intervals.

A few conclusions can be drawn from Figure 4. First, the averaged contended fraction does not vary much with increasing the duration of an interval at the early stages of the network. Second, more important, shared objects are more frequently touched by only one transaction rather than multiple transactions on average: for an interval of one checkpoint (the actual frequency of commitments in Sui), the average contended fraction fluctuates around 0.3, which can be easier observed for epochs when the network is more mature (e.g., after BQ-3 ended). Third, the average contended fraction greatly increases when the duration of the interval is lengthened for a more mature network, as can be seen in the figure after BQ-2 ended. Finally, longer intervals do not necessarily imply larger values of the average contended fraction.

E. Number of shared objects in transaction inputs

While the contended fraction (see Figure 4) shows how often shared objects are touched by more than one transaction, it does not capture information about how many shared objects are touched in an atomic transaction. Such information provides insights on composability and dependencies in smart contracts involving shared objects. Figure 5 depicts the average number of shared objects in transaction inputs, per epoch.

As can be observed in Figure 5, the average number of shared objects in transaction inputs fluctuated around 1 when the network was immature, until nearly epoch 70, when it dramatically increased to almost 7 shared objects per transaction on average right before BQ-1. After BQ-2 ended, it can be seen that the average number of shared objects in an atomic transaction fluctuates around a value of 5, which indicates a moderate degree of dependencies among shared objects in Sui.

Figures 2-5 illustrate metrics calculated based on the aggregation of all shared objects. That is, these metrics do not provide any information about which shared objects and

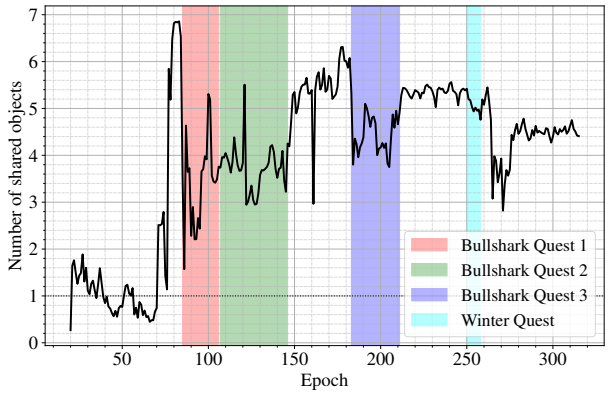

Fig. 5. Average number of shared objects touched by transactions.

applications are mostly used in Sui. Sections III-F and III-G address this.

F. Popular shared-object applications

Before we delve into the description of particular shared objects, we briefly describe some of the most used applications in Sui, as annotated in Figure 6.

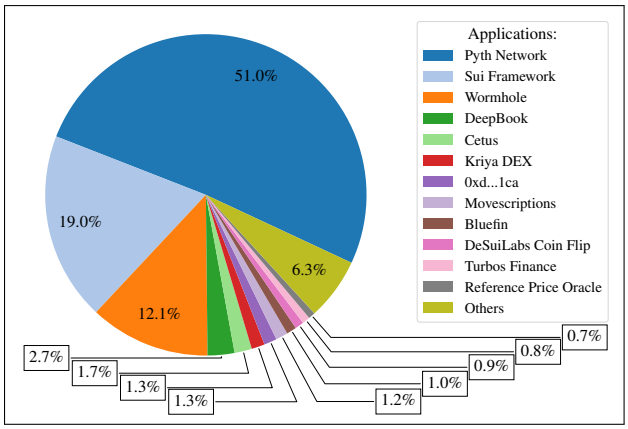

Fig. 6. Most frequently used applications involving shared objects in Sui.

`Pyth Network` is a data oracle that publishes financial market data to multiple blockchains [17]. Pyth price feeds employ a pull price update model, which requires shared objects. In this model, users are responsible for posting price updates on-chain. `Pyth Network` implements two shared objects, `PriceInfoObject` and `State` (see Figure 7), both extensively used in Sui: as can be seen in Figure 6, more than half (51%) of the total number of shared-object transactions is taken by those related to `Pyth Network`.

`Sui Framework` provides a collection of the core on-chain libraries for Move developers, and it is the second most frequently used (19% of all shared-object transactions) contract involving shared objects such as `Clock`, `Kiosk` (both extensively used; see Figure 7), `Table`, and others [16].

`Wormhole` is an interoperability protocol powering the seamless transfer of value and information across multiple blockchains [22]. It sends messages cross-chain using various verification methods to attest to the validity of a message. The `State` object is the only shared object (extensively used in Sui; see Figure 7) in `Wormhole`, and it is used (i) as a container for all state variables and (ii) to perform anything that requires access to data that defines the contract.

`DeepBook` is a decentralized central limit order book built for Sui to provide a one-stop shop for trading digital assets and to accelerate the development of financial and other apps on Sui [20]. `Pool` is the only shared object type in the contract.

`Cetus` is a DEX and concentrated liquidity protocol with two main shared objects: `Pool`, a concentrated liquidity market maker pool, and `GlobalConfig`, a container for global state variables in the contract (see Figure 7) [23].

`Kriya DEX` is a non-custodial order book-based DEX [18] with `Pool` being the key shared object in the contract.

`Movescriptions` is a semi-fungible assets protocol expanding the capabilities of *Inscriptions* using Move [24]. The contract implements two shared objects: `TickRecord` and `DeployRecord`.

`Bluefin` is a DEX for derivatives, offering perpetual swaps trading with up to 20x leverage [25]. The package includes `BankV2`, `Sequencer`, and other shared objects.

`DeSuiLabs Coin Flip` is a smart contract game for players to double their SUI by guessing heads or tails [21]. Each user's guess is represented by the `Game` shared object. If a guess is incorrect, the contract sends the player's bet into the house wallet, represented by the `HouseData` shared object.

`Turbos Finance` is a non-custodial and hyper-efficient DEX with two key shared objects: `Pool` (a concentrated liquidity pool) and `Versioned` (pool version "storage") [19].

`Reference Price Oracle` is a data oracle provided by Sui, and it is intended to serve as a general reference and for informational purposes only [26]. The contract implements only the `SimpleOracle` shared object.

G. Popular shared object types

We are now in a position to describe some of the most frequently used shared object types in Sui, as depicted in Figure 7, in which the percentages represent a relative number of shared-object transactions involving those types. Asterisks in the legend annotate *singletons*, i.e., only one instance of such object type can be created. As it can be seen, the only two shared objects of `Pyth Network`, `PriceInfoObject` and `State`, constitute more than half (51%) of all shared-object transactions. `PriceInfoObject` and some other shared objects of interest are described in detail below.

`PriceInfoObject` represents Pyth price feeds in Sui. There are 83 instances of the `PriceInfoObject` type, each corresponding to a single Pyth price feed in the global storage. Transactions involving `PriceInfoObject` shared objects (i) constitute 38.9% of all shared-object transactions (see Figure 7), and (ii) usually take such objects by a mutable reference, which likely indicates high contention for these shared objects [27]. Both observations can be explained by the underlying price update model employed by Pyth, as described

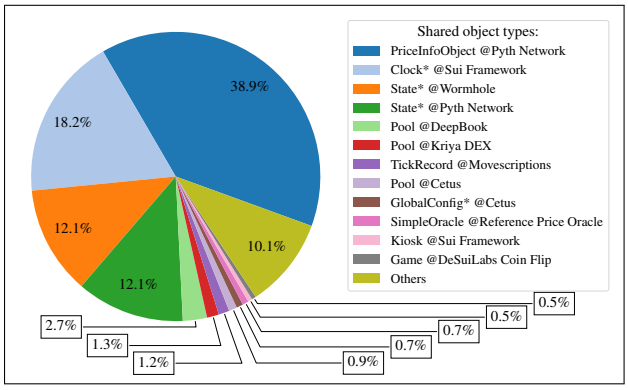

Fig. 7. Most frequently used shared object types in Sui. Asterisks in the legend annotate singleton shared objects.

in Section III-F. Typically, users of Pyth price feeds submit a single transaction that simultaneously updates the price and uses it in a downstream application. In other words, updating the on-chain price in Pyth is a permissionless operation, which requires employing shared objects.

`Clock` is a singleton shared object created during genesis, and it is used for accessing time from Move calls via APIs provided in the `Sui Framework` contract. Anyone can access `Clock` in transactions but only via an immutable reference: this, however, still requires making `Clock` a shared object. The `Clock`'s timestamp is set automatically by a system transaction every time consensus commits a checkpoint. Therefore, transactions that read `Clock` do not need to be sequenced relative to each other. However, any transaction that requires access to `Clock` must go through consensus in Sui because the only available instance is a shared object. Transactions involving `Clock` constitute 18.2% of all shared-object transactions (see Figure 7).

The discussed above `PriceInfoObject` does not have any access control logic: anyone can read or write it. A striking example of a shared object with the ownership notion is the `Kiosk` [28], which allows storing and trading any type of asset as long as the creator of those assets implements a transfer policy. `Kiosk` provides guarantees of "true ownership": similarly to owned objects, assets stored in `Kiosk` can only be managed by the kiosk owner, who can place, take, list items, and perform any other actions on assets in the kiosk. Anyone can create `Kiosk`: there is a high number of shared objects $(428,804)$ of this type [27]. By default, a `Kiosk` instance is made shared, in which case, the owner can sell any asset that has shared `TransferPolicy` available, guaranteeing creators that every transfer must be approved. Anyone can purchase openly listed items in `Kiosk`—this is the only kind of write operation on `Kiosk` that can be performed by anyone. While purchasing the `Kiosk` items is available for everyone, it is possible for the owner to make a `Kiosk` instance an owned object. However, such a kiosk might not function as intended or be inaccessible to other users.

In addition to `Clock`, `Sui Framework` also provides other shared object types, such as `TransferPolicy`—a highly customizable primitive that provides an interface for the owner to set custom transfer rules, `TreasuryCap`—a capability that allows the bearer to mint and burn coins of some type, guaranteeing full ownership over the currency, and `CoinMetadata`—a container for the metadata of any coin type created in Sui. While instances of these three shared object types are not extensively used in Sui transactions, it is worth mentioning an important observation: owned and/or immutable objects of these three types also exist [27].

A quite "popular" use case of shared objects in Sui is liquidity pools used in various DEX applications and usually represented by a `Pool` type, as in the following contracts: `DeepBook` [20], `Cetus` [23], `Kriya DEX` [18], `Turbo Finance` [19], and others (see Figure 7). A striking common characteristic of all the aforementioned `Pool` shared objects is that they are (almost) always accessed via a write operation, which implies high contention for these objects in transactions.

Another interesting observation worth mentioning here is related to `ABEx Finance`, a revolutionary on-chain derivatives protocol built to seek alpha for traders while settling beta for liquidity providers [29]. Specifically, the contract provides some shared objects that are always accessed only via read-only operations, and there are no write methods implemented to modify these objects [27]. It is not fully clear why these objects were not made immutable instead of shared.

Last but not least, such contracts as `DeSuiLabs Coin Flip` [21], `Sui Framework` [16], and `Scallop` [30] implement shared object types for which a large number of instances has been created: `Game` (2,702,164 instances), already mentioned `Kiosk` (428,804 instances), and `Obligation` (86,969 instances), respectively [27]. While shared `Kiosk` and `Obligation` resemble owned objects, but with additional access control logic, it is not fully clear why `Game` objects, each corresponding to a player, need to be shared objects.

IV. Conclusion

In this work, we analyzed shared objects on the Sui smart contract platform. The presented results show that shared objects are extensively used on Sui, especially when the network becomes more mature. Contention for shared objects was also investigated, and it appears to be relatively low in Sui. The most frequently used shared objects are those related to price feed updates and accessing time on chain. Liquidity pools are another popular use case for shared objects in Sui. The average number of shared objects in transaction inputs fluctuates around 5, which indicates a moderate degree of dependencies among shared objects. The question of composability and whether those shared objects come from the same or different contracts is not present in this study but is considered for future work, along with the evaluation of contention levels with decoupling read and write operations.

References

[1] W. Metcalfe *et al.*, "Ethereum, smart contracts, DApps," *Blockchain and Crypt Currency*, vol. 77, 2020.

[2] "Introduction to Ethereum smart contracts," hhttps://ethereum.org/developers/docs/smart-contracts.

[3] "Embedded Permission," https://research.csiro.au/blockchainpatterns/general-patterns/contract-structural-patterns/embedded-permission/.
[4] "Concurrency and Cardano: A problem, a challenge, or nothing to worry about?" https://builtoncardano.com/blog/concurrency-and-cardano-a-problem-a-challenge-or-nothing-to-worry-about.
[5] M. M. T. Chakravarty, J. Chapman, K. MacKenzie, M. Melkonian, Orestis Peyton Jones, and P. Wadler, "The Extended UTXO Model," in *Financial Cryptography and Data Security*. Springer International Publishing, 2020, pp. 525–539.
[6] "Transaction throughput scalability strategies for Plutus smart contracts," https://journal.platonic.systems/scaling-with-utxos/.
[7] "The Sui Smart Contracts Platform," https://docs.sui.io/paper/sui.pdf.
[8] J. Xu, K. Paruch, S. Cousaert, and Y. Feng, "Sok: Decentralized exchanges (DEX) with automated market maker (AMM) protocols," *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–50, 2023.
[9] "Sui Rust SDK," https://docs.sui.io/references/rust-sdk.
[10] "To maintain anonymity, the repository is currently private," https://github.com.
[11] "ACES Program: Introducing Bullshark," https://mystenlabs.com/blog/introducing-bullsharks.
[12] "Earn Sui via Bullshark Quests!" https://mystenlabs.com/blog/introducing-bullsharks-quests.
[13] "Announcing Quest 2 with Bullsharks and Capys," https://mystenlabs.com/blog/bullshark-quest-2.
[14] "Quest 3: Games," https://mystenlabs.com/blog/quest-3.
[15] "Winter Quest: Earn Presents!" https://mystenlabs.com/blog/winter-quest.
[16] "Sui Framework," https://docs.sui.io/sui-framework-reference.
[17] "Pyth Network," https://pyth.network.
[18] "Kriya DEX," https://www.kriya.finance.
[19] "Turbos Finance," https://turbos.finance.
[20] "DeepBook," https://docs.sui.io/standards/deepbook.
[21] "Cetus," https://www.desuilabs.io.
[22] "Wormhole," https://wormhole.com.
[23] "Cetus," https://www.cetus.zone.
[24] "Movescriptions," https://movescriptions.org.
[25] "Bluefin," https://bluefin.io.
[26] "Sui oracle," https://github.com/MystenLabs/sui/tree/main/crates/sui-oracle.
[27] "To maintain anonymity, the repository is currently private," https://github.com.
[28] "Sui Kiosk," https://docs.sui.io/standards/kiosk.
[29] "ABEx Finance," https://abex.fi/.
[30] "Scallop," https://www.scallop.io/.