Oze: Decentralized Graph-based Concurrency Control for Long-running Update Transactions

Jun Nemoto
Scalar, Inc.
jun@scalar-labs.com

Takashi Kambayashi
Nautilus Technologies, Inc.
kambayashi@nautilus-technologies.com

Takashi Hoshino
Cybozu Labs, Inc.
hoshino@labs.cybozu.co.jp

Hideyuki Kawashima
Keio University
river@sfc.keio.ac.jp

ABSTRACT

This paper proposes Oze, a concurrency control protocol that handles heterogeneous workloads, including long-running update transactions. Oze explores a large scheduling space using a multi-version serialization graph to reduce false positives. Oze manages the graph in a decentralized manner to exploit many cores in modern servers. We further propose an OLTP benchmark, BoMB (Bill of Materials Benchmark), based on a use case in an actual manufacturing company. BoMB consists of one long-running update transaction and five short transactions that conflict with each other. Experiments using BoMB show that Oze can handle the long-running update transaction while achieving four orders of magnitude higher throughput than state-of-the-art optimistic and multi-version protocols and up to five times higher throughput than pessimistic protocols. We also show Oze performs comparably with existing techniques even in a typical OLTP workload, TPC-C, thanks to a protocol switching mechanism.

PVLDB Reference Format:
Jun Nemoto, Takashi Kambayashi, Takashi Hoshino, and Hideyuki Kawashima. Oze: Decentralized Graph-based Concurrency Control for Long-running Update Transactions. PVLDB, 18(8): 2321 - 2333, 2025.
doi:10.14778/3742728.3742730

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jnmt/ccbench/tree/vldb-paper.

1 INTRODUCTION

1.1 Motivation

Conventional serializable concurrency control protocols are not well-suited for certain real-world OLTP workloads, especially those involving long-running update transactions. A typical example can be found in product costing systems used in manufacturing industries. These systems perform long-running update transactions based on a bill of materials (BoM), and such transactions must be isolated from others in a serializable manner to ensure accurate and optimal resource planning.

To isolate the long transaction from other transactions, traditional systems usually run it where short online transactions are absent, e.g., at night [5] or use stale materialized views [27]. However, this approach is becoming impractical. Modern product costing must handle frequent updates to materials and their costs, especially in the face of supply chain disruptions. This trend necessitates on-demand costing, which requires the ability to execute long and short transactions concurrently while preserving serializability. Snapshot isolation [2] cannot guarantee the consistency of BoM trees used by the long transactions for product costing, and it produces incorrect results, as discussed in Section 2.4.

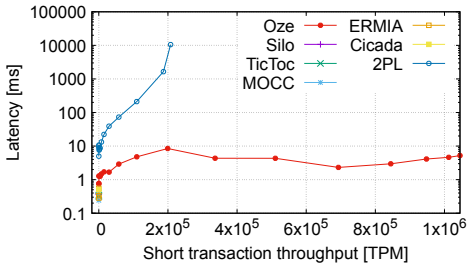


Figure 1: Throughput versus average latency of short transactions in BoM benchmark. Existing OCC and MVCC protocols cannot increase throughput any further because they will not be able to commit the long transaction (L1). 2PL shows lower throughput with higher latency than Oze because it has to wait for the L1 commit for a long time.

1.2 Challenges

None of the conventional protocols [24, 38–40, 43] can handle the new workload with BoM. As shown in Figure 1, the throughput and latency of the short transactions degrade when attempting to ensure the successful commit of a long transaction (referred to as L1) in our BoM-based benchmark. Existing OCC and MVCC protocols show lower throughput because they are more likely to abort the L1 transaction as the request rate increases. All of the L1 aborts in those protocols are false positives (i.e., false aborts), which result from protocol-specific constraints on transaction ordering. 2PL exhibits better throughput than OCC and MVCC; however, the L1 transactions significantly reduce the throughput of short transactions and increase the latency, as shown in the figure, because they must wait for a lock held by the L1 transaction for a long time. See Section 2.5 for details of these behaviors in each protocol.

Deterministic approaches [14, 37] can handle this workload under a certain condition, where BoMs do not change before and after the L1 transaction. We call it a static BoM. However, our target BoM is often updated to dynamically reflect the effect of supply chain disruption, and on-demand costing is required. We call such a

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:10.14778/3742728.3742730

BoM a dynamic BoM. Despite using reconnaissance queries [37] to identify BoM trees beforehand, deterministic approaches struggle to commit the L1 transaction without sacrificing the short transaction throughput in a dynamic BoM. Our deterministic extension of 2PL, called D2PL, showed only marginal performance, as depicted in Figure 9 in Section 5.2.

It is challenging for a serializable protocol to ensure committing long transactions without degrading the performance of short transactions running concurrently.

1.3 Contributions and paper organization

Oze Protocol: We propose Oze, a concurrency control protocol that can handle long-running update transactions while achieving high short transaction throughput with reasonable latency, as shown in Figure 1. The key ideas of Oze are twofold.

The first idea is to exploit the large scheduling space using a multi-version serialization graph (MVSG) [4] in a decentralized manner. Conventional MVSG-based protocols such as MVSGT [17] and MVSGA [16] assume a single centralized graph management and thus fail to exploit modern many-core architectures. Oze manages a record-local graph associated with each record, each of which represents only the dependencies among transactions accessing that specific record, and it partially synchronizes these graphs across records to check for acyclicity.

The second idea is dynamic protocol switching. To handle conventional workloads such as TPC-C [10], Oze provides the OCC mode in addition to the MVSG mode and switches them bidirectionally and dynamically depending on the workloads, as depicted in Figure 10 and in Figure 12.

BoMB Benchmark: We also present an OLTP benchmark, BoMB (Bill of Materials Benchmark), which emulates real-world on-demand product costing. Conventional OLTP benchmarks do not contain long transactions with consecutive anti-dependencies. TPC-C [10] and TPC-E [11] do not have any long transactions. TPC-EH [39] contains a long-running transaction that includes write operations, but the result of writes is never read by other transactions, and thus it does not produce consecutive anti-dependencies. HTAP benchmarks like OLxPBench [20] contain long *read-only* transactions. BoMB includes a long-running update transaction that will be falsely aborted with high probability in conventional serializable protocols due to two consecutive and long-term anti-dependency edges created by the other five short transactions. We design BoMB based on a workload in a real bread manufacturing company, Andersen [1], with diverse configurable parameters, allowing BoMB to represent a variety of BoM workloads.

We evaluate Oze with modern concurrency control protocols using BoMB on CCBench [36]. Experimental results show that only Oze can ensure the L1 transaction commit without sacrificing the short transaction throughput.

The rest of this paper is organized as follows. First, in Section 2, we describe the workload in BoMB and why it is challenging for existing protocols. Next, we describe the design and implementation of the Oze protocol in Section 3 and Section 4, respectively. In Section 5, we evaluate several protocols using BoMB and TPC-C. In Section 6, we describe related work. Finally, we conclude this paper in Section 7.

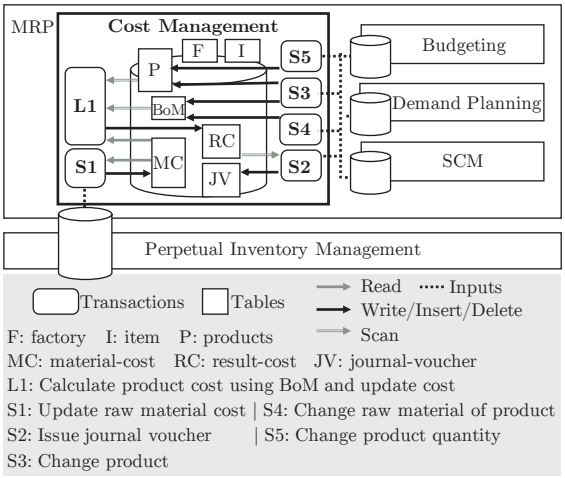


Figure 2: System and workload overview.

2 BOMB AND CHALLENGES

This section presents a new OLTP benchmark, BoMB (BoM Benchmark). First, we provide an overview of BoMB's database and workload. We then show how and why existing protocols cannot effectively handle the BoMB workload.

2.1 Overview

We model on-demand product costing based on a real bread manufacturing company, Andersen [1], extracting common components and parameters applicable to various industries. Currently, they completely isolate the product costing from the OLTP database. They export input data from the OLTP database to a Hadoop-based distributed system, calculate product costs, and write back the results into the OLTP database. To avoid any kind of inconsistency, they do not allow reading and writing the OLTP database during the cost calculation. Therefore, serializable isolation is required for on-demand costing as well.

Figure 2 shows the modeled system that includes a manufacturing resource planning (MRP) system and an inventory management system, with modules for cost management, budgeting, demand planning, and supply chain management (SCM). The cost management module, involving long-running update transactions, produces the most complex and challenging workloads. Our model focuses on this module, identifying one long transaction (L1) and five major short transactions (S1-S5) that are common in manufacturing. L and S stand for "long" and "short," respectively. All these transactions commonly occur in manufacturing industries [31, 42] and can be widely applied beyond the bread company.

2.2 Tables

BoMB uses seven tables, as shown in Figure 2. The `factory` table manages a list of factories in the company. The `item` table manages the name of items with their type: product, material, or raw material. The `product` table manages which products are manufactured and their volumes in each factory. The `bom` table manages a hierarchical list of the materials and quantities needed to manufacture a product.

The material-cost table manages the costs of raw materials for each factory. The result-cost table stores the latest calculated costs for each product in each factory. The journal-voucher table manages journal vouchers, which are issued when manufacturing processes are executed using the calculated costs and utilized by each related module, e.g., SCM. For details of these tables, such as all attributes with data types and parameters for varying cardinality, see our specification document [28].

2.3 BoM Tree

The bom table represents an item along with its constituent child items and their quantities. Such a relationship can be logically expressed in a tree structure, which we call a BoM tree. A record in the bom table is identified by a parent item ID and the child item ID. To obtain the components of an item, the bom table is scanned by specifying the item ID as a parent. An example of a BoM tree is shown in Figure 3. The product consists of several major materials (hereafter, "root materials" for convenience). For example, in the production of sandwiches, the root materials correspond to bread and the ingredients inside (e.g., tuna salad). Each root material is made from multiple materials. For example, for bread, the material is dough, and the raw materials are flour, yeast, etc.

When starting the benchmark, the BoM trees are initialized based on several parameters (e.g., tree size), which can be configured to represent various shapes of BoM trees in manufacturing industries. See our extended paper for details of parameters, default values, and how the benchmark initializes the BoM trees [29].

The product cost is calculated using the BoM tree as follows: (1) Identify the products to be costed. (2) Construct a BoM tree by referring to the bom table and recursively acquiring the materials that comprise each product. Each tree node contains an item ID, a list of child item IDs, a unit price, and a required quantity. (3) Set the unit cost for each raw material, which is a leaf node of the BoM tree, by referring to the material-cost table. (4) Calculate the product cost in a depth-first manner, summing the child node costs multiplied by their required quantities.

2.4 Transactions

Long transaction: The L1 (update-product-cost) transaction is a long transaction that builds BoM trees described in Section 2.3 and calculates product costs. First, it randomly selects a factory and obtains all products manufactured at the factory and their quantity by referring to the product table. Next, it builds a BoM tree for the product and calculates the cost. When building the BoM tree, it refers to the material-cost table, updated by S1 transactions, in addition to the bom table. Then, it writes the result to the result-cost table, which is referred to by S2 transactions. These steps are repeated for all products. All products must be handled in a single transaction because optimal production planning must be performed based on a consistent view of the costing results throughout a factory. Since S2 transactions scan all product costs in a factory to guarantee such a consistent view, transaction chopping [35] cannot be applied without violating serializability. We use a parameter, target-products that represents how many products are currently manufactured at each factory (default: 100). This parameter is for adjusting the length of the transaction. When an

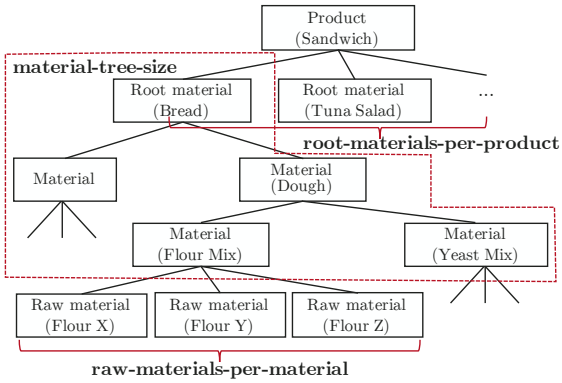


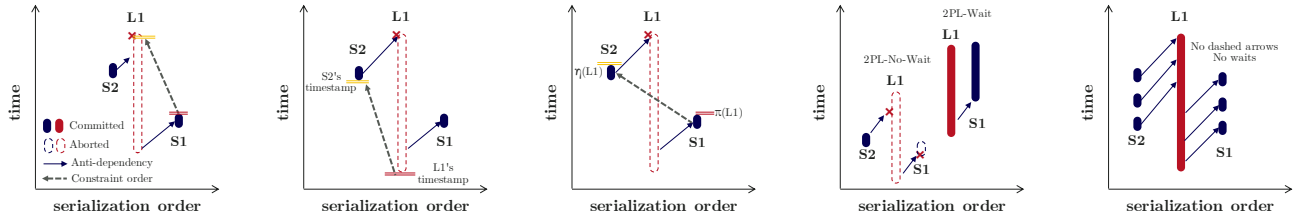
Figure 3: Example of BoM tree.

L1 transaction runs with the default setting, it reads/scans about 20,000 records in total for calculating costs and writes 100 records as the result.

Short transactions: The S1 (update-material-cost) transaction is a short transaction that changes the cost of raw materials. It performs read-modify-write operations on randomly selected records in the material-cost table. The S2 (issue-journal-voucher) transaction is a short transaction that creates a journal voucher based on the calculated product cost. It scans the result-cost table to obtain the costs of all products in a factory and inserts the voucher records into the journal-voucher table. The S3 (change-product) transaction is a short transaction that replaces an old product with a newly-developed product. It creates a new BoM tree for the product, inserts the records into the bom table, and replaces the product by deleting and inserting the record from/into the product table. The S4 (change-raw-material) transaction is a short transaction that replaces a raw material of a product with a different one (e.g., change a flour X to X'). It deletes and inserts a raw material record for a BoM tree in the bom table. The S5 (change-product quantity) transaction is a short transaction that updates a manufacturing quantity of a product in a factory. It updates a record in the product table.

Transaction mix: BoMB can run with two settings based on the characteristics of the target BoM: *static* and *dynamic* BoM. For static BoM, BoMB only runs L1, S1, and S2 transactions. For dynamic BoM, it additionally runs S3, S4, and S5 transactions, which change BoM structures dynamically. We set the default percentage of short transactions for static BoM to be 50% each for S1 and S2. Similarly, for dynamic BoM, we set the ratio at 45%, 45%, 1%, 1%, and 8% for S1, S2, S3, S4, and S5, respectively. This default transaction mix is configured based on a case for our industrial collaborators. Since it may vary depending on industries and companies, BoMB allows users to customize the transaction mix to meet their requirements.

Regulation: According to the discussion with our industrial collaborators, for the BoMB workload, the mandatory requirement is to ensure the commit of L1. Even if a protocol can accidentally commit L1 by repeating aborts and retries for a long time, this can delay the cost calculations and result in incorrect product pricing and profit calculations, affecting business decisions. To reflect this requirement, we define the BoMB benchmark score as the maximum short transaction throughput that can be achieved with less



(a) OCC [38, 43] (b) MVTO [24] (c) SSN [39] (d) 2PL [3] (e) Oze

Figure 4: Why existing protocols are not enough? (a)–(c) L1 aborts with false positives because the observed anti-dependency order (blue arrows) does not match the constraint order (dashed arrows, see Section 2.5 for details): (a) commit order, (b) begin timestamp order, and (c) commit-timestamp-based order. (d) L1 aborts, or S1 must wait for a long time. (e) Oze can commit L1 without false positives or S1's waits, thanks to precise order tracking.

than 1% abort rate of L1[1]. To always be running L1 transactions and short transactions concurrently, a new L1 transaction must start just after the previous L1 commit when measuring the short transaction throughput. Moreover, to ensure accurate product costing, all transactions must be executed with serializable isolation.

Why serializable: Product costing with lower isolation levels, e.g., snapshot isolation [2], easily causes anomalies. Suppose transactions that change raw materials, such as S4, and they must be executed with a certain condition regarding other intermediate materials; e.g., if a flour mix X consists of flour A and B, then yeast P in a yeast mix Y should be replaced with yeast P' or if a flour mix X consists of flour A and B', then yeast Q of Y should be replaced with yeast Q'. Such a transaction and another transaction that replaces B with B' may cause anomalies for L1 transactions, i.e., L1 observes an inconsistent combination of those materials as a BoM tree[2]. This can happen even if all transactions except for L1 run with serializable isolation. Even a slight change in the short transactions of BoMB can easily cause such a problem with the correctness of L1. Under lower isolation levels, verifying the correctness of schedules or their safety in application semantics requires significant effort. Thus, executing all transactions, including L1, with serializable isolation is the most practical solution to avoid incorrect results here.

2.5 BoMB with Conventional Protocols and Challenges

Figure 4 illustrates why existing concurrency control protocols fail to handle BoMB transactions. Inherently, transaction serializability can be determined solely by the dependency order, i.e., reads-from relation and version order, including anti-dependencies decided by version order [41]. To reduce the cost of tracking all of those dependencies, conventional protocols impose a certain constraint on the acceptable serialization order. This protocol-specific constraint is an order, and we call it constraint order. If the observed dependency order (including potential ones) of a transaction does not match the constraint order, the transaction is aborted. Most of the existing protocols abort the L1 transaction because the dependency order of BoMB transactions (S2<L1<S1) does not obey their own constraint orders, although this is a false positive.

For example, OCC protocols such as Silo [38] and TicToc [43] force transactions to obey a constraint order, which is an order in which they entered the serialization point by acquiring the necessary locks in the validation phase. The OCC-specific behavior, such as aborts due to finding an overwriter, can be explained as a mismatch between a potential anti-dependency order (L1<S1) and the constraint order (S1<L1), as shown in Figure 4(a). Due to the constraint order, the L1 transaction is aborted as a false positive even though they are serializable (S2<L1<S1).

MOCC [40] combines OCC with a pessimistic scheme using locks and a hotspot counter. Since not all records are treated pessimistically, the L1 transaction still fails read validation. Timestamp adjustments used in some OCC variants [6, 22, 23] do not contribute to the L1 completion with or without priority setting because the room for the adjustment is exhausted in a moment by several S1 transactions around the L1 transaction.

MVTO forces transactions to obey the begin timestamp order as the constraint order. Because the observed anti-dependency (S2<L1) of an L1 transaction is opposite to the begin timestamp order (L1<S2), the L1 transaction is aborted, as shown in Figure 4(b).

Serial safety net (SSN) [39] in ERMIA [21] uses a commit-time stamp-based order as the constraint order. This order is adjusted from the actual commit order based on the two watermarks of a transaction T: $\pi(T)$ and $\eta(T)$. Because the anti-dependency order (S2<L1<S1) observed by the L1 transaction using $\pi(L1)$ and $\eta(L1)$ is inconsistent with the commit-timestamp-based order (S1<S2), the L1 transaction is aborted, as shown in Figure 4(c).

2PL and its variants [3, 9, 15, 33, 41] are protocols that serialize transactions by using the locking order as the constraint order. As shown in Figure 4(d), in 2PL-based protocols, conflicting transactions cannot run concurrently because they need to obey the constraint order by waiting for locks. Specifically, in 2PL-Wait [3], S1 transactions must keep waiting for the L1 completion. Similarly, in 2PL-No-Wait [3], S1 transactions must stall because they continue to abort and retry. In addition, L1 transactions may abort when they access records already locked by S1 or S2 transactions.

Challenges: As stated above, OCC and MVCC protocols force a constraint order in addition to the dependency order. While these

[1]We choose 1% as a realistic compromise since it is theoretically impossible for protocols such as OCC to guarantee 0% abort rate under the condition where all transactions run concurrently.
[2]See the extended paper [29] for details.

protocols can efficiently check serializability, they limit the scheduling space to be explored by the constraint order and suffer from false aborts. 2PL-based approaches can commit L1 transactions since 2PL forces transactions to wait so that the transactions obey the constraint order decided by the locking order. However, this locking order limits the scheduling space and significantly reduces the short transaction performance if the preceding L1 transactions exist. Therefore, it is challenging to achieve no false aborts for the long transactions without reducing performance for the short transactions running in parallel.

3 OZE DESIGN

To avoid false positives and performance degradation due to the constraint order, we propose Oze, which is an MVSG-based protocol and does not depend on such a constraint at all. Oze also supports dynamic protocol switching between the MVSG mode and the OCC mode to efficiently handle conventional workloads.

3.1 Decentralizing MVSG

The literature [4] proposed the multiversion serialization graph (MVSG) and proved that a transaction schedule is serializable iff there exists an acyclic MVSG. An MVSG is formed by a multiversion schedule s and a version order $\ll$. A multiversion schedule contains two types of operations: a write operation $w_i(x_i)$ that denotes a transaction t_i writes a version of a record x and a read operation $r_j(x_i)$ that denotes t_j reads x_i written by t_i, which is called a reads-from relation. A version order $\ll$ for s is the union of all version orders of data items written by operations in s, where a version order for x is any non-reflexive and total ordering of all versions of x and $x_i \ll x_j$ denotes that a version x_i precedes another version x_j. For given s and $\ll$, $MVSG(s, \ll)$ has nodes for a set of transactions $t_0 \dots t_N$ and edges as follows: for distinct operations $w_j(x_j)$, $r_i(x_j)$, and $w_k(x_k)$ in s where $t_i \neq t_k$,

- $t_j \rightarrow t_i$ for a reads-from relation $r_i(x_j)$ (wr-dependency)
- $t_i \rightarrow t_k$ if $x_j \ll x_k$ (rw-dependency)
- $t_k \rightarrow t_j$ if $x_k \ll x_j$ (ww-dependency)

Conventional MVSG-based protocols [16, 17] are not scalable because they assume a single centralized graph structure for the MVSG. A giant lock is necessary to access the graph for each transaction operation. Oze avoids the giant lock by allowing both a transaction and a record to manage a sub-graph of the MVSG.

Let T_i be a set of transactions that depend on t_i directly or transitively, where for all t in T_i, there exists a path from t_i to t in $MVSG(s, \ll)$ through any types of edges, *wr*, *rw*, and *ww*. We refer to T_i as the *followers* of t_i. Next, let $SG(s, t_i)$ be the sub-graph of $MVSG(s, \ll)$, consisting of the nodes for t_i and for each transaction in T_i and their edges. We refer to $SG(s, t_i)$ as the *transaction-local* graph of t_i. Since T_i includes all transactions reachable from t_i, if $SG(s, t_i)$ has no cycle through t_i, then $MVSG(s, \ll)$ also has no cycle through t_i. Thus, t_i can be safely committed without violating serializability by checking $SG(s, t_i)$.

Let $SG(s, x)$ be the sub-graph of $MVSG(s, \ll)$, consisting of the nodes for each transaction that reads or writes a record x and the *wr*-, *rw*-, and *ww*- dependency edges regarding x. We refer to $SG(s, x)$ as the *record-local* graph of x. When X_i denotes the set of records read or written by t_i and T_i, it follows that

$$SG(s, t_i) = \bigcup_{x \in X_i} SG(s, x).$$

The essence of the Oze protocol is creating $SG(s, t_i)$ by merging a record-local graph $SG(s, x)$ for each x in X_i and checking for acyclicity. We refer to X_i as the *target record set* of t_i for the merging of record-local graphs.

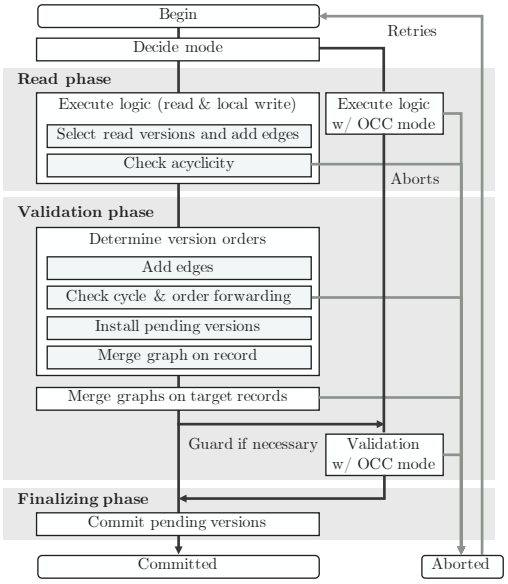


Figure 5: Overview of Oze protocol.

3.2 Protocol

Oze adopts a three-phase protocol, which consists of reading, validating, and finalizing, to smoothly switch to OCC [38], which has three similar phases.

3.2.1 Read phase. Figure 5 shows an overview of the Oze protocol. Oze executes a transaction while reading committed versions and writing new ones in the thread-local storage. When reading a record x, the transaction selects the latest version from the version list sorted by the serialization order, and a *wr*-dependency edge is added to the record-local graph of x. If the graph is acyclic (i.e., serializable), the transaction adds the version to the local read set. If the graph is cyclic, then it finds an older version that does not create a cycle. In such a case, the transaction adds *rw*-dependency edges, which are from the transaction to the writers of the versions newer than the selected one into the record-local graph.

When writing a record x, the transaction only adds the new version of the record in the local write set.

3.2.2 Validation phase. On reaching the commit operation, a transaction t_i enters the validation phase, which consists of two parts: (1) choosing version orders based on the method in Section 3.3 and then (2) merging the record-local graph for each record in X_i to obtain $SG(s, t_i) = \bigcup_{x \in X_i} SG(s, x)$ while identifying followers and checking for acyclicity of the graph. For simplicity, we assume here

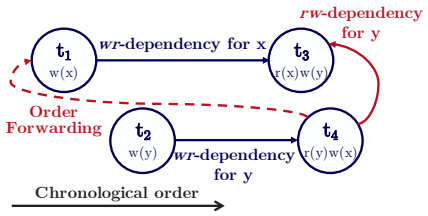


Figure 6: Example of order forwarding.

all concurrent transactions other than t_i are pending during the validation phase of t_i. We describe how to handle concurrent transactions without using a global giant lock in Sections 4.2 and 4.3.

(1) Choosing version orders: For each record in the write set, t_i performs the following three steps: (1a) t_i determines a version order and adds the corresponding edges to the record-local graph; (1b) after confirming acyclicity, t_i installs a pending version (which cannot be read by other transactions at this point) into the version list of the record; and (1c) t_i merges the record-local graph into its transaction-local graph.

(2) Merging record-local graphs: If the version order can be determined for all records in the write set while keeping the graph acyclic, t_i then merges the record-local graph for each record in the read set into its transaction-local graph. Each time t_i merges a record-local graph, it checks the acyclicity of the resulting graph and identifies all its followers. If a follower of t_i exists, t_i merges record-local graphs of the records read or written by that follower and repeats identifying its followers and merging the record-local graphs. If no cycle is found in $SG(s, t_i)$, which is obtained by merging all graphs on records read or written by all followers, t_i proceeds to the finalizing phase for committing.

3.2.3 Finalizing phase. For a transaction that passed the validation phase without aborting, Oze changes the status of versions written by the transaction to *committed*.

3.3 Dynamic Version Ordering

A notable feature of Oze is dynamic version ordering, which actively explores possible serializable schedules in the validation phase described in Section 3.2.2.

When choosing the version order, Oze first tries postposing the version so that the newer version in chronological order becomes the newer version in the serialization order. Given that there is a transaction t_i that reads x written by transaction t_j, postposing t_k means to select the order of $x_j \ll x_k$. If postposing breaks serializability (i.e., a cycle occurs in MVSG), then Oze tries preposing ($x_k \ll x_j$) to find an acyclic schedule. We call this technique order forwarding. With the order forwarding, Oze can change the version order while keeping the concurrent transactions' views, i.e., as long as the forwarding transaction does not interrupt the writer of the version and its readers. The order forwarding enables Oze to schedule transactions in the MVSR space, which is larger than the space generated by the MVSGT protocol [17].

Example: The following example illustrates the behavior of order forwarding. Consider the schedule s, which consists of transactions t_1 through t_4. Assume that all transactions are concurrent; here, c denotes an internal commit, which occurs before the response is returned to the client. All other notations follow the multi-version full schedule notation in the literature [17].

$$s = w_1(x_1)w_2(y_2)c_1c_2r_3(x_1)r_4(y_2)w_3(y_3)c_3w_4(x_4)c_4$$

After committing t_1 and t_2, t_3 and t_4 read x_1 and y_2, respectively. As a result, the corresponding *wr*-dependency edges (shown as blue edges in Figure 6) are added. When t_3 reaches $c3$, it checks if it can choose $y_2 \ll y_3$ (postposing). This is done by adding an *rw*-dependency edge from t_4 to t_3 (solid red edge in the figure) and checking for acyclicity. Since no cycle exists in this example, t_3 can safely commit.

In contrast, when t_4 enters its validation phase at c_4 (after c_3), t_4 creates a cycle when it tries to add a postposing edge from t_3 to t_4 (i.e., *rw*-dependency edge for the version order $x_1 \ll x_4$) due to t_3's read of x_1. Thus, it performs the order forwarding and tries to add a preposing edge from t_4 to t_1 (i.e., *ww*-dependency edge for the version order $x_4 \ll x_1$) as shown by the dashed red edge in the figure. The order forwarding makes the graph acyclic, thus t_4 can also be committed. The final serialization order is $t_2 < t_4 < t_1 < t_3$, which is different from the chronological order and cannot be obtained with MVSGT [17]. Note that Oze can still ensure *linearizability* [18] by introducing an epoch and restricting the forwarding within the epoch, as detailed in Section 4.3.1.

3.4 Dynamic Protocol Switching

Oze using MVSG is specially designed for complex workloads that consist of long-running update transactions and conflicting short transactions. Consequently, using MVSG for workloads without such long transactions is not economical due to its maintenance. In order to practically handle both types of workloads, Oze dynamically switches between the MVSG-based protocol and an OCC protocol, which is based on Silo [38], depending on the workload characteristics. Switching from the OCC mode to MVSG mode starts when a long transaction is aborted. Oze detects the long transaction when its read/write set is over a predefined size. Conversely, switching from the MVSG to OCC mode starts when workers do not observe long transactions for a certain period. Those switchings go through a transition mode so that each worker can continuously process transactions without quiescing. During the transition mode, the workers manage the graphs but validate transactions using the OCC protocol, as shown in the last part of the validation phase in Figure 5. This strategy preserves serializability because the OCC mode has a smaller scheduling space than that of the MVSG mode.

4 OZE IMPLEMENTATION

This section describes an implementation of Oze's MVSG mode.

4.1 Data Structure

Transaction: Each transaction has a transaction ID (*txid*) and read/write sets. The transaction ID is assigned at the beginning, and it consists of an epoch, a worker thread ID, and a local counter. We introduce the epoch to ensure linearizability and facilitate garbage collection. Like Silo [38], a dedicated thread increments the global epoch at regular intervals, and each worker thread refers to it. The

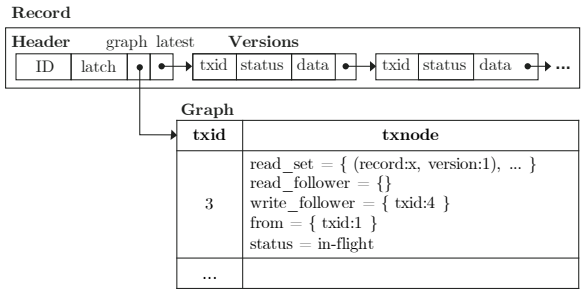


Figure 7: Data structures in Oze.

read/write sets manage entries with a primary key, a record pointer, and a version pointer.

Record: The structure of a record is shown in Figure 7. The record contains a record ID, a latch, and two pointers: one for a linked list of versions and one for a record-local graph. The record ID is a 64-bit integer used for OCC (i.e., TID in Silo [38]). The versions in the list are ordered in the serialization order. Each version has *txid* of the version creator, the pointer to the previous version, and the status.

Graph: The structure of a record-local graph is shown in Figure 7, and it is also used for a transaction-local graph. The graph is represented as a map whose key is *txid* and whose value is the node of the graph (*txnode*). Each node has three lists of *txids*. *read_follower* is a list of transactions that read the version, corresponding to the *wr*-dependency edges. *write_follower* is a list of transactions that write a version newer than the version read or written by the transaction, corresponding to the *rw*-dependency edges and *ww*-dependency edges. *from* is a list of transactions that point to the list owner, i.e., incoming edges. The node also has the transaction's read set and the status to find followers.

4.2 Read Phase

As described in Section 3.2, Oze executes the read phase, validation phase, and finalizing phase in that order. We describe the read and write protocols in the read phase below.

Lines 1–15 of Algorithm 1 show Oze's read protocol. This *read()* function is called if a transaction does not have the cached record and version in the local read and write sets. During *read()*, the transaction must hold a latch to access the record exclusively, including the versions and the record-local graph. First, it searches the version list of the record from the latest version to find a *readable version*, which is a committed version that does not create a cycle when reading. If it does not find a readable version even after reaching the oldest version, it aborts. If there is a readable version, it adds its *txid* to the *read_follower* in the *txnode* of the transaction that wrote the selected version and adds the writer's *txids* of the skipped versions to the *write_follower* in the own *txnode* (Lines 5–10). The record and the version are also added to the local read set and the read set on the record-local graph, respectively (Lines 12–13).

When writing, a transaction just creates a version and stores it in the local write set (Lines 17–18). The versions can be used for *"read-own-write"* [24, 32].

Algorithm 1: Read phase

```
1  Function read(txn, record)
2      latch(record)
3      graph = record.graph
4      version = record.latest
5      while version do
6          graph[version.txid].read_follower.add(txn.txid)
7          if is_acyclic(graph) then break ;
8          graph[version.txid].read_follower.remove(txn.txid)
9          graph[txn.txid].write_follower.add(version.txid)
10         version = version.next
11     if version then
12         txn.read_set.add((record, version))
13         graph[txn.txid].read_set.add(record, version)
14     else abort() ;
15     unlatch(record)
16 Function write(txn, record)
17     version = create_version(txn.txid)
18     txn.write_set.add((record, version))
```

4.3 Validation Phase

Algorithm 2 shows the entire flow of the validation phase. As described in Section 3.2, it consists of two parts: choosing a version order for each record in the write set (Lines 2–9) and obtaining a union of record-local graphs on records read or written by followers of the transaction in validation (Lines 10–20).

4.3.1 Choosing version order. For each record in the write set, a transaction first merges its transaction-local graph into the record-local graph to share it with concurrent transactions (Line 4). Sharing the transaction-local graph enables concurrent transactions to abort as early as possible before merging record-local graphs on many records and increases the chance of trying the order forwarding. After merging, the transaction chooses the version order and checks for acyclicity (Line 5; see also Lines 21–38 for details). Oze uses a per-record latch to access the version list and the record-local graph exclusively (Lines 3–9).

In *choose_version_order()*, the transaction gets *readers*, a list of transactions reading the record, based on the graph and adds *rw*-dependency edges; the transaction in validation is placed behind the readers in the serialization order (Lines 23–25). If there is no cycle, it inserts the version as the latest one (Lines 26–27).

If there is a cycle, Oze tries the order forwarding. The transaction removes the current *rw*-dependency edges, which are from *readers* to *txn* itself (Lines 29–30). To try another version order, the transaction finds *writers*, which are transactions that write versions read by *readers* so that *txn* can be ordered before *writers* (Line 31). Specifically, it finds such transactions by checking each *reader's* read set and the writers' transaction ID of the versions in the read sets. Then, the transaction adds new *ww*-dependency edges from *txn* to each of *writers* and checks for acyclicity (Lines 34 and 36). If there is no cycle, the new version is inserted into the version list, considering the order of the other versions written by *writers*.

The order forwarding can be performed across epochs. However, Oze limits it within the same epoch to guarantee linearizability

Algorithm 2: Validation phase

```
   // done: List of records already processed
   // target: List of target records X_i for merging graphs
   // Both lists are empty at beginning
1  Function validate(txn)
2      for (record, version) in write_set do
3          latch(record)
4          merge(record.graph, txn.graph) // record <- txn
5          choose_version_order(txn, record, version)
6          add_target_records(txn, record.graph, target, done)
7          merge(txn.graph, record.graph) // txn <- record
8          done.add(record)
9          unlatch(record)
10     target.add(records in read_set)
11     while ! target.is_empty() do
12         record = target.pop()
13         latch(record)
14         merge(record.graph, txn.graph) // record <- txn
15         if ! is_acyclic(record.graph) then
16             abort()
17         add_target_records(txn, record.graph, target, done)
18         merge(txn.graph, record.graph) // txn <- record
19         done.add(record)
20         unlatch(record)
21 Function choose_version_order(txn, record, version)
22     graph = record.graph
23     readers = find_readers(graph, record)
24     for r in readers do
25         graph[r.txid].write_follower.add(txn.txid)
26     if is_acyclic(graph) then
27         record.insert_version(version, [])
28     else
           // Order forwarding
29         for r in readers do
30             graph[r.txid].write_follower.remove(txn.txid)
31         writers = find_writers(graph, record, readers)
32         for w in writers do
33             if txn.txid.epoch == w.txid.epoch then
34                 graph[txn.txid].write_follower.add(w.txid)
35             else abort() ;
36         if is_acyclic(graph) then
37             record.insert_version(version, writers)
38         else abort() ;
39 Function add_target_records(txn, graph, target, done)
40     followers = get_followers(txn, graph)
41     for follower in followers do
42         for (rec, v) in graph[follower].read_set do
43             if rec not in done then target.add(rec) ;
```

and simplify graph cleaning (described in Section 4.6) and aborts transactions if it occurs across epochs (Lines 33–35).

For scalability, multiple transactions speculatively choose version orders and write the corresponding edges to each record-local graph in parallel. Thus, Oze's concurrent implementation may cause false-positive aborts, making it more restrictive than the centralized MVSG approach.

If the transaction successfully chooses the version order, it then merges the record-local graph into the transaction-local graph after checking the follower transactions to identify the target records (X_i defined in Section 3.1) to be merged later (Lines 6–7). The function *add_target_ records()* in Lines 39–43 illustrates how a transaction identifies the target records. It first retrieves all followers (Line 40) and adds each record in the follower's read set to the target list (Lines 41–43). Note that followers in the read phase can be ignored because they will identify their own followers during their validation phase. It is also unnecessary to inspect the followers' write sets. A follower merges record-local graphs on records in its write set into its transaction-local graph (Line 7) and then leaves it on records in its read set (Line 14). Thus, all *wr*-, *rw*-, and *ww*- dependency edges involving a follower are included in a record-local graph on one of the records in its read set. Therefore, a transaction in validation phase does not need to check the followers' write sets.

4.3.2 Merging record-local graphs on target records. Since record-local graphs for each record in the write set are merged when choosing version orders, the transaction then repeatedly merges graphs for each record in the read set while identifying followers and additional target records. Specifically, it merges its transaction-local graph into the record-local graph, checks the acyclicity of the merged graph, lists additional target records to be merged, and continues merging the record-local graph into the transaction-local graph until the target list becomes empty (Lines 11–20). The transaction leaves its footprints by merging the transaction-local graph into the records for the reason described in Section 4.3.1 (see the discussion on Lines 4 and 14 for more details). Note that the transaction must hold a per-record latch (Lines 13–20).

4.3.3 Correctness sketch. If a set of uncommitted transactions T_{cycle} forms a cycle, Oze ensures at least one transaction in the cycle will be aborted. Let t_i in T_{cycle} be a last transaction that determines its version orders and writes the corresponding edges to record-local graphs; i.e., t_i has reached Line 10 of Algorithm 2 lastly. Then, at this moment, t_i must exhaustively identify all followers in $T_{cycle} \subseteq T_i$ and their dependencies, which are already stored in the target record set X_i. Therefore, t_i will detect the cycle and abort.

4.3.4 Complexity. The dominant factor of the complexity in the validation phase is graph processing, such as merging and cycle-checking. The time complexities of both processes are $O(|V| + |E|)$ where $|V|$ and $|E|$ are the number of nodes and edges in a graph[3], respectively. The graph processing must be done for each record in the target record set X_i. Therefore, as a whole, the time complexity of the decentralized Oze protocol is $O(|X_i|(|V| + |E|))$.

[3]Precisely, the cycle check requires fewer nodes and edges since it is enough to only check the nodes that follow the transaction about to commit.

4.4 Parallel Validation

In Oze, merging record graphs and checking for acyclicity during the validation phase incurs high overhead, particularly for long transactions involving many read and write operations. Moreover, due to infrequent garbage collection during such transactions, the graph size may keep growing, potentially causing the validation process to run indefinitely. To mitigate this, Oze parallelizes the validation phase using multiple threads to accelerate validation.

A part that can be parallelized is merging record-local graphs on the target records (Lines 11–20 in Algorithm 2). Specifically, parallel validation works as follows. (1) A transaction worker thread invokes the specified number of validators and assigns merge targets to validators equally. (2) For each target record, each validator merges the validator-local graph into the record-local graph, checks for acyclicity of the graph, finds additional targets, and merges the graph into the validator-local graph again. (3) Once all validators finish, the transaction worker merges all the validator-local graphs into the transaction-local graph and checks for acyclicity. (4) If the graph remains acyclic, the transaction worker aggregates additional targets identified by each validator and reassigns them equally. Steps (1)–(4) are repeated until no additional targets remain.

4.5 Phantom Avoidance

Oze requires phantom avoidance that does not depend on an optimistic approach like index node validation [38] to keep precise tracking of dependencies. We use a variant of precision locking [19], which is also used in Hyper [30]. Although in Hyper, scan transactions validate if there are no conflicting insert transactions based on the predicates, in Oze, insert transactions validate scan predicates. This opposite way is suitable for Oze, because, in Oze, writer transactions decide version orders based on potential anti-dependencies, and insert transactions can be handled in the same manner.

Specifically, a transaction stores the *txid* and the query predicates[4] when scanning a table as a scan history. An insert transaction (inserter) checks the scan histories in the validation phase. If the inserter finds that its insertion matches the predicates of a scan history, it adds an edge from the transaction that created the scan history to the inserter on the transaction-local graph. Then, it checks for acyclicity and proceeds to validate other records in the write set if no cycle exists.

4.6 Garbage Collection

In SGT [7], once a transaction commits, no incoming edges are added to the node of the transaction. This property allows safe garbage collection of committed nodes. Similarly, we can delete MVSG nodes that will never make a future cycle if we ensure that no new incoming edges will be added. However, in Oze, incoming edges can be added to the committed transaction in two situations. The first case can occur in the read protocol. As mentioned in Section 4.2, when selecting a version to read, a reader transaction may add edges to the transactions (*write_follower*) that wrote the skipped version. The second case can occur in the order forwarding; when preposing a transaction, it adds edges to the *write_follower* transactions as mentioned in Section 4.3.

[4]In our current implementation, scan queries support range-based predicates for primary keys, but it can be extended for arbitrary predicates on any columns.

Oze uses epochs to guarantee that no new incoming edges will be added to a node, allowing it to be safely deleted. An epoch that satisfies this condition is referred to as the *reclamation epoch* (e_r), defined as the minimum local epoch among all worker threads minus one. To enforce this, Oze prevents transactions from adding incoming edges by prohibiting: (1) reading versions from e_r or earlier (except for the latest version), and (2) performing the order forwarding to versions in e_r or earlier.

For GC of versions, we remove versions in e_r or earlier, except for the latest (in the serialization order). We must check the MVSG and hold versions if concurrent transactions are reading them.

5 EVALUATION

This section shows (1) Oze can handle long-running update transactions while achieving better short-transaction performance and smoothly switching from/to OCC in our target workload, BoMB, (2) Oze achieves comparable performance to existing protocols even in TPC-C [10], (3) Oze's order forwarding works effectively in a specific workload, and (4) Oze has the expected computational complexity through a runtime analysis with YCSB [8].

5.1 Experimental Setup

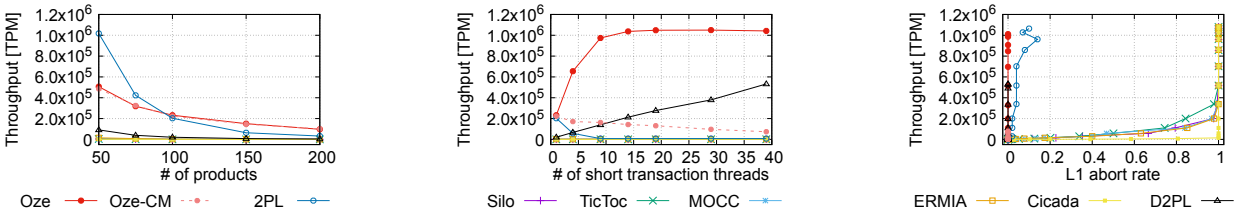
All experiments were performed using CCBench [36], a benchmark platform for various concurrency control protocols. We implemented two variations of Oze in CCBench: one used the proposed decentralized MVSG, and the other used a single Centralized MVSG (Oze-CM). We compared Oze with three OCC protocols: Silo [38], MOCC [40], TicToc [43] and two MVCC protocols: ERMIA [21] as SSN and Cicada [24]. We also used two pessimistic approaches: 2P-No-Wait [3] and D2PL. D2PL is a 2PL-based protocol that mimics deterministic behavior such as that found in Calvin [37]. D2PL first sorts all the accessing keys and then locks them in that order to avoid deadlocks.

The evaluation environment consisted of a single server with two Intel®Xeon®Gold 6138 CPUs with 2.00 GHz and twenty-four DDR4-2666 32 GB DIMMs (total 768 GB). Each CPU had 20 cores.

5.2 Experiments on BoMB

We first evaluated how each protocol could handle the BoMB workload using *static BoM*, i.e., L1, S1, and S2 transactions only; BoM trees never changed. Then, we ran all six transactions of *dynamic BoM*, including transactions that changed BoM trees.

5.2.1 Static BoM. We measured the maximum short transaction throughput based on the regulation we defined in Section 2.4. We used the default parameters of the BoMB [28]. We increased the request rate of short transactions (with a 50/50 ratio for S1/S2) as much as possible, keeping the L1 abort rate less than 1% based on the requirement. We stopped increasing the request rate when the average throughput did not change by more than 5% and used the effective throughput at that time as the score. Starting with one transaction per second, we increased the request rate by a factor of 2 and ran the execution for 60 seconds. Note that L1 transactions were always running during each execution on one thread. For Oze, we used a 40ms epoch length in all experiments because our preliminary experiments did not show significant differences between 40ms and 5ms. We also used 32 threads for parallel validation.

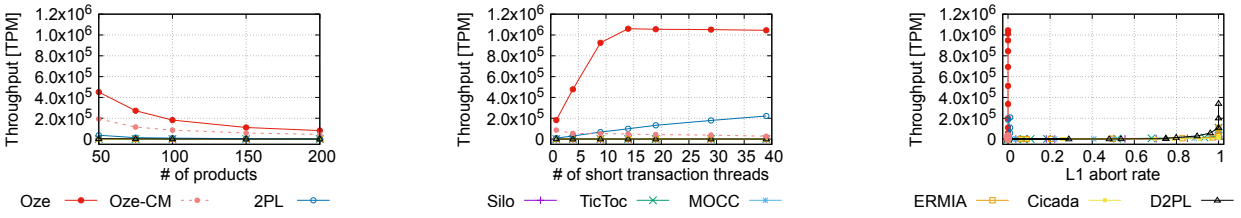


(a) L1 length scalability with L1 abort rate < 1% (1 thread for L1 and 1 thread for other short transactions)

(b) Thread scalability with 100 products and L1 abort rate < 1% (1 thread for L1 and varying # threads for short transactions)

(c) TPM vs. L1 abort rate with 100 products, 40 threads (1 thread for L1 and others for short transactions), and varying request rate

Figure 8: Short transaction throughput with static BoM.



(a) L1 length scalability with L1 abort rate < 1% (1 thread for L1 and 1 thread for other short transactions)

(b) Thread scalability with 100 products and L1 abort rate < 1% (1 thread for L1 and varying # threads for short transactions)

(c) TPM vs. L1 abort rate with 100 products, 40 threads (1 thread for L1 and others for short transactions), and varying request rate

Figure 9: Short transaction throughput with dynamic BoM.

L1 length scalability: Figure 8(a) shows the average throughput of short transactions (S1 and S2) while increasing the number of target products from 50 to 200. The number of worker threads was one for the L1 transactions and one for the short transactions. OCC protocols (i.e., Silo, TicToc, and MOCC) performed worse than others by several orders of magnitude since the L1 validation phase inevitably failed when increasing the S1 requests, which updated the cost of many raw materials. MVCC protocols (i.e., ERMIA and Cicada) also performed worse. L1 could proceed to build BoMs and calculate costs without being hindered by S1, thanks to the benefit of multiple versions. However, L1 frequently aborted with false positives when L1 tried to update the costing results that conflicted with the S2's reads. 2PL and D2PL worked better than OCC and MVCC when the L1 length was shorter; however, they reduced throughput as L1 became longer since short transactions that conflicted with L1 had to wait to acquire locks. When increasing products, Oze showed the highest throughput. Oze with a single centralized MVSG (Oze-CM) showed slightly lower throughput than decentralized Oze because the single-threaded short transaction did not cause much contention on the graph.

Thread scalability: Figure 8(b) shows the short transaction throughput with the fixed product size of 100 for the L1 transaction while increasing the number of worker threads for short transactions. The other measurement process was the same as the above. The protocols other than lock-based approaches and Oze could not benefit from parallelism since a single thread was enough to hinder L1. The throughput of Oze-CM gradually decreased due to contention on the graph as the number of worker threads increased. 2PL also reduced the throughput because the abort rate of the L1 transaction exceeded the requirement of 1% when increasing the threads. Oze and D2PL, where L1 did not abort at all, increased the short transaction throughput depending on the number of threads. D2PL was still inferior to Oze because short transactions with D2PL required a long time to acquire locks due to L1.

L1 abort rate: In Figure 8(c), we plotted the L1 abort rate (x-axis) against the short transaction throughput (y-axis). We fixed the number of products to 100 and the number of threads at 40, then increased the request rate until the throughput reached saturation, ignoring the 1% L1 abort rate regulation in BoMB. Although OCC and MVCC protocols increased throughput at the right end of the graph, the throughput reached saturation as the same as Oze. In contrast, Oze increased the throughput while committing the L1 perfectly, as all plots are on the left end.

Key finding with static BoM: The state-of-the-art optimistic protocols, including MVCC, hardly commit long-running update transactions in the BoM workload even if the BoM tree does not change. Oze and D2PL handle the static BoM workload well. 2PL can commit the L1 transactions in some cases.

5.2.2 Dynamic BoM. Using the same procedure as with the static BoM, we measured the short transaction throughput with the ratio defined in Section 2.4. For the dynamic BoM, we used different phantom protection techniques for each protocol. For 2PL, we used a variant of gap-locking [25]. For D2PL, we issued a reconnaissance query [37] and validated the results. For Oze, we used a variant of precision locking described in Section 4.5. For other protocols, we used index node validation [38]. Figure 9 shows the results of the same three experiments in Section 5.2.1 in the dynamic BoM

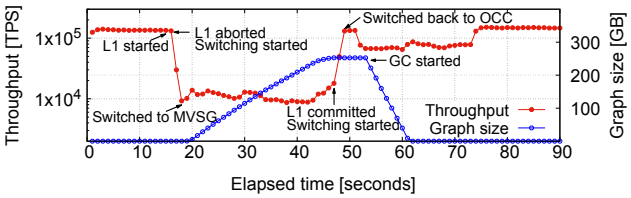


Figure 10: Protocol switching and memory consumption.

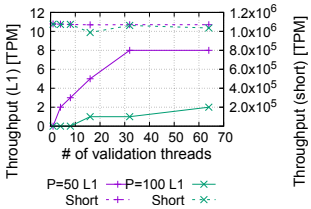


Figure 11: Validation scalability of Oze with static BoM.

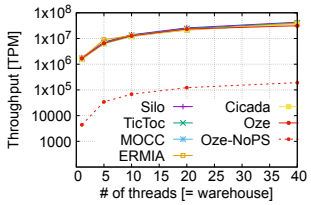


Figure 12: TPC-C throughput (full mix).

setting. Note that Figure 1 plots the throughput and latency using the results of Figure 9(c) as long as the L1 abort rate is less than 1%.

L1 length scalability: OCC and MVCC protocols performed worse, similar to the case in the static BoM. D2PL also could not handle the BoMB workload because the validation of the reconnaissance query almost always failed due to the S3 and S4 transactions. 2PL significantly reduced throughput due to the wait for the locks held by the L1 transaction since the S3 and S5 try to update the target products of the cost calculation. In contrast, Oze could handle L1 the same as in the static BoM.

Thread scalability: For protocols other than D2PL and 2PL, we observed similar results to the ones in the static BoM. For D2PL, additional threads no longer affected the throughput of D2PL due to the same reason in the single-thread case of Figure 9(a). Surprisingly, 2PL increased the throughput while committing almost all of the L1 transactions because S3 and S5 transactions reduce the chances of the S2 execution, which hinders L1. Nevertheless, Oze still achieved about five times higher throughput than 2PL.

L1 abort rate: As shown in the figure, existing OCC and MVCC protocols performed worse than those in the static BoM. Thus, we terminated increasing the request rate when the average L1 abort rate continued to be 100% three times for those protocols. For 2PL, as discussed in the thread scalability above, we saw a significant reduction of the L1 abort.

Key finding with dynamic BoM: Except for Oze, only 2PL can handle the dynamic BoM workload without aborting almost all L1 transactions. However, the short transaction throughput of 2PL is significantly lower than that of Oze because S3 and S5 transactions are stalled for a long time due to a lock held by L1.

5.2.3 Protocol switching. Figure 10 shows the behavior of protocol switching and memory consumption. The x-axis shows the elapsed time, and the y-axis shows the short transaction throughput (left) and the total size of MVSG graphs for all records (right).

We started ten threads with the OCC mode for only short transactions with a 50/50 ratio for S1/S2. We then submitted an L1 transaction after 15 seconds. The L1 transaction was aborted in the validation phase of the OCC mode about 1 second later, and the threads started switching to the MVSG mode. We confirmed that all the threads smoothly changed to the MVSG mode after a few seconds while reducing throughput, and the L1 was successfully committed after 30 seconds. While executing the L1, it spent almost all the time on the validation phase, and the graph size linearly increased due to merging the graphs on the target record set. Finally, all the threads switched back to the OCC mode a few seconds after the L1 commit because we configured the threads to switch to the OCC mode when no long transaction exists in a single epoch.

Once all the threads switch to the OCC mode, garbage collection for the graphs starts. We used 32 threads for garbage collection, and it was completed within less than 10 seconds. Note that the short transaction throughput decreased during garbage collection and page reclamation by the operating system.

5.2.4 Parallel Validation. Figure 11 shows the throughput of the L1 transaction with 50 and 100 products (P=50 and P=100, respectively) in the static BoM setting using 20 worker threads, as the number of validator threads increases from 1 to 64. The left y-axis shows the throughput for the L1 transaction, and the right y-axis shows the throughput for short transactions.

Oze could not commit the L1 transaction with a single validation thread due to timeouts (i.e., not abort). However, parallel validation enabled L1 commits, improving throughput with more threads. The benefit was reduced as the number of threads increased because the parallel validation presented an overhead when merging each validator's resulting graph and checking for acyclicity.

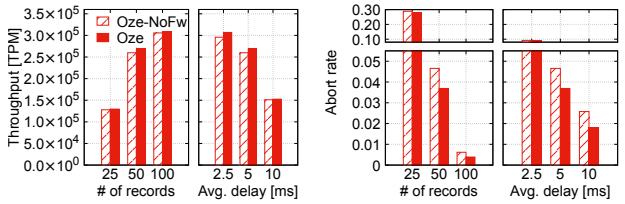
For short transactions, throughput remained stable even as the number of validator threads increased. This was due to the saturation of short transaction throughput at around ten threads, as shown in Figure 8(b), allowing L1 to utilize the remaining resources.

5.3 Experiments on TPC-C

We compared Oze with other protocols using the TPC-C benchmark, executing all five transactions in its specified ratio. Figure 12 shows the throughput of each protocol with varying numbers of threads and warehouses. Note that Oze started with the OCC mode, and Oze-NoPS disabled protocol switching. As shown in the figure, Oze-NoPS exhibited poor performance due to its heavy graph management. Oze showed reasonable performance with existing protocols thanks to protocol switching, although the peak throughput was about 27% lower than the highest one, Silo.

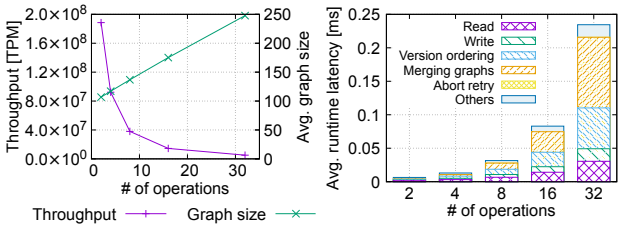
5.4 Experiments for Order Forwarding

In the experiments with BoMB and TPC-C, we could not observe the effects of the order forwarding because there is no such read and blind write combination that triggers it. To confirm the effects, we conducted experiments using synthetic workloads that included four types of transactions, as exemplified in Section 3.3. The target data was a table with 25, 50, or 100 records, each with a key of consecutive integers. T1 and T2 wrote one of the first and last half records, respectively. T3 read one of the first-half records and wrote one of the last-half records, and T4 vice versa. Each transaction



(a) Throughput. (b) Abort rate

Figure 13: Effects of order forwarding.



(a) Throughput and graph size. (b) Runtime analysis.

Figure 14: Protocol analysis of Oze with YCSB-A.

type was executed equally. We randomly inserted 0-5ms, 0-10ms, or 0-20ms delays (sleep) after each operation.

Figure 13(a)(b) show Oze's throughput and abort rate when running the workload with 40 threads in MVSG mode. In Oze-NoFw, the order forwarding was disabled. The left side of each graph shows results under a 5ms average delay while varying the number of records; the right side shows results with 50 records under varying delays. The order forwarding improved throughput by about 3% and reduced the abort rate by about 20%. The effect was negligible when the number of records was small (e.g., fewer than the number of threads), due to increased conflicts between transactions of the same type. Improvements were also limited with many records, as transactions committed without requiring the order forwarding.

5.5 Runtime Analysis with YCSB

Finally, we conducted an experiment using a YCSB variant for Oze protocol analysis. Like existing work [24, 36, 39], we modified YCSB-A so that each transaction included 50% reads and 50% pure writes and ran it on 100 million records with a uniform request distribution. We intentionally used small payloads (4 bytes) to observe pure overheads of concurrency control. The number of worker threads was 20, and all workers used the single-thread validation.

Figure 14(a) shows the throughput and the average graph size as the number of operations per transaction varies. The graph size is the average number of nodes involved in cycle checking. As shown in the analysis of computational complexity in Section 4.3.4, the throughput was nearly inversely proportional to the product of the number of operations and the graph size. Figure 14(b) shows the average latency required to process each major function of Oze. The percentage of time spent on merging orders became larger compared to other functions as the graph size grew.

6 RELATED WORK

Benchmark: TPC-C [10] and TPC-E [11] are benchmarks for OLTP systems with only short transactions. TPC-EH [39] has a read-mostly long transaction with write operations, but it lacks a transaction that reads the results written by the long transaction, unlike BoMB. YCSB [8], OLTP-Bench [12], and OLxPBench [20] provide synthetic and real-world workloads for benchmarking cloud services, OLTP, and HTAP systems, but none of them emulate BoMB's target characteristics. BoMB provides a long-running update transaction that uniquely stresses serializable protocols due to two consecutive and long-term anti-dependency edges created by the other five short transactions.

Lock-based protocols: 2PL variants using timestamp-based priority [9, 15, 33] can commit a long transaction like L1 when the long transaction gets the smallest timestamp. However, subsequent short transactions must either wait or abort until the long ones commit if they conflict. Altruistic locking [34] allows transactions to donate locked objects, which other transactions can access before they are unlocked. However, this can increase waiting times for short transactions that accept donations.

Graph-based protocols: Durner and Neumann proposed a graph-based concurrency control [13] for many-core systems using Serialization Graph Testing (SGT) [41]. Although they extended it as a multi-version concurrency control for read-only analytical queries, their SGT and the multi-version extension cannot handle long-running update transactions and conflicting short transactions concurrently because they always keep the commit order as the constraint order, which narrows the scheduling space.

Deterministic database systems: Calvin [37] executes transactions based on a pre-determined total order. Ocean Vista [14] and Aria [26] offer alternative approaches that do not require the read-set in advance. When using them for BoMB, however, L1 transactions deteriorate short transaction throughput as they must wait for resolving functors (Ocean Vista) or committing the previous batch (Aria). Oze is a non-deterministic protocol, and its scheduling space is wider than the deterministic ones [14, 26, 37].

7 CONCLUSION

We proposed Oze, a concurrency control protocol that exploits a large scheduling space using a multi-version serialization graph in a decentralized manner. We also proposed an OLTP benchmark, BoMB, based on a use case in an actual manufacturing company. Experiments using BoMB showed that Oze could handle the long-running update transaction while achieving four orders of magnitude higher throughput than optimistic and multi-version protocols and up to five times higher throughput than pessimistic protocols.

ACKNOWLEDGMENTS

This paper is based on results obtained from the project "Research and Development Project of the Enhanced Infrastructures for Post-5G Information and Communication Systems (JPNP20017)" and JPNP16007 commissioned by the New Energy and Industrial Technology Development Organization (NEDO), and from JSPS KAKENHI Grant Number 25H00446, and from JST CREST Grant Number JPMJCR24R4 and from SECOM Science and Technology Foundation and JST COI-NEXT SQAI (JPMJPF2221).

REFERENCES

[1] Andersen Institute of Bread & Life Co., Ltd. 2024. *Andersen Group.* Retrieved May 15, 2025 from https://www.andersen-group.jp/english/

[2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. *SIGMOD Rec.* 24, 2 (May 1995), 1–10.

[3] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (June 1981), 185–221.

[4] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control—Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (Dec. 1983), 465–483.

[5] Anja Bog. 2014. *Benchmarking transaction and analytical processing systems.* Springer.

[6] C. Boksenbaum, M. Cart, J. Ferrié, and J.-F. Pons. 1987. Concurrent Certifications by Intervals of Timestamps in Distributed Database Systems. *IEEE Trans. Softw. Eng.* 13, 4 (April 1987), 409–419.

[7] Marco A Casanova and Philip A Bernstein. 1980. General purpose schedulers for database systems. *Acta informatica* 14, 3 (1980), 195–220.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10).* 143–154.

[9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 22 pages.

[10] Transaction Processing Performance Council. 2010. *TPC Benchmark C - Standard Specification Revision 5.11.* Retrieved May 15, 2025 from https://www.tpc.org/tpcc/

[11] Transaction Processing Performance Council. 2015. *TPC Benchmark E - Standard Specification Version 1.14.0.* Retrieved May 15, 2025 from https://www.tpc.org/tpce/

[12] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (Dec. 2013), 277–288.

[13] Dominik Durner and Thomas Neumann. 2019. No False Negatives: Accepting All Useful Schedules in a Fast Serializable Many-Core System. In *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE '19).* 734–745.

[14] Hua Fan and Wojciech Golab. 2019. Ocean Vista: Gossip-Based Visibility Control for Speedy Geo-Distributed Transactions. *Proc. VLDB Endow.* 12, 11 (July 2019), 1471–1484.

[15] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD/PODS '21).* 658–670.

[16] Thanasis Hadzilacos. 1988. Serialization Graph Algorithms for Multiversion Concurrency Control. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '88).* 135–141.

[17] Thanasis Hadzilacos and Christos H. Papadimitriou. 1985. Algorithmic Aspects of Multiversion Concurrency Control. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS '85).* 96–104.

[18] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492.

[19] J. R. Jordan, J. Banerjee, and R. B. Batman. 1981. Precision Locks. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data (SIGMOD '81).* 143–147.

[20] Guoxin Kang, Lei Wang, Wanling Gao, Fei Tang, and Jianfeng Zhan. 2022. OLxP-Bench: Real-time, Semantically Consistent, and Domain-specific are Essential in Benchmarking, Designing, and Implementing HTAP Systems. In *Proceedings of the 38th IEEE International Conference on Data Engineering (ICDE '22).* 1822–1834.

[21] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16).* 1675–1687.

[22] Kwok-Wa Lam, V.C.S. Lee, Kam-Yiu Lam, and Sheung-Lun Hung. 1996. Distributed real-time optimistic concurrency control protocol. In *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems.* 122–125.

[23] J. Lee and S.H. Son. 1993. Using dynamic adjustment of serialization order for real-time database systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS '93).* 66–75.

[24] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17).* 21–35.

[25] David B. Lomet. 1993. Key Range Locking Strategies for Improved Concurrency. In *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB '93).* 655–664.

[26] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 12 (July 2020), 2047–2060.

[27] Stephan Müller, Lars Butzmann, Kai Höwelmeyer, Stefan Klauck, and Hasso Plattner. 2013. Efficient View Maintenance for Enterprise Applications in Columnar In-Memory Databases. In *Proceedings of the 17th IEEE International Enterprise Distributed Object Computing Conference (EDOC '13).* 249–258.

[28] Jun Nemoto. 2024. *BoMB on CCBench.* Retrieved May 15, 2025 from https://github.com/jnmt/ccbench/tree/vldb-paper

[29] Jun Nemoto, Takashi Kambayashi, Takashi Hoshino, and Hideyuki Kawashima. 2024. *Oze: Decentralized Graph-based Concurrency Control for Long-running Update Transactions (Extended Version).* Retrieved December 1, 2024 from https://arxiv.org/abs/2210.04179

[30] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15).* 677–689.

[31] Newman Cloud, Inc. 2022. *OpenBOM - User Stories.* Retrieved May 15, 2025 from https://www.openbom.com/user-stories

[32] C.U. Orji, L. Lilien, and J. Hyziak. 1988. A performance analysis of an optimistic and a basic timestamp-ordering concurrency control algorithms for centralized database systems. In *Proceedings of the 4th IEEE International Conference on Data Engineering (ICDE '88).* 64–71.

[33] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis. 1978. System Level Concurrency Control for Distributed Database Systems. *ACM Trans. Database Syst.* 3, 2 (June 1978), 178–198.

[34] Kenneth Salem, Héctor García-Molina, and Jeannie Shands. 1994. Altruistic Locking. *ACM Trans. Database Syst.* 19, 1 (March 1994), 117–165.

[35] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction chopping: algorithms and performance studies. *ACM Trans. Database Syst.* 20, 3 (Sept. 1995), 325–363.

[36] Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, and Osamu Tatebe. 2020. An Analysis of Concurrency Control Protocols for In-Memory Databases with CCBench. *Proc. VLDB Endow.* 13, 13 (Sept. 2020), 3531–3544.

[37] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12).* 1–12.

[38] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13).* 18–32.

[39] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. 2017. Efficiently Making (Almost) Any Concurrency Control Mechanism Serializable. *The VLDB Journal* 26, 4 (Aug. 2017), 537–562.

[40] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *Proc. VLDB Endow.* 10, 2 (Oct. 2016), 49–60.

[41] Gerhard Weikum and Gottfried Vossen. 2001. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Elsevier.

[42] Muhammad Younus, Tao Jian, and Fan Yuqing. 2010. Module-Based Parts Management System Using Single Source of Product Data. In *Proceedings of the 2010 Second International Conference on Computer Research and Development (ICCRD '10).* 151–155.

[43] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16).* 1629–1642.