Improving OCC Performance Through Transaction Batching and Operation Reordering

Bailu Ding
Cornell University, Microsoft Research
badin@microsoft.com

Lucja Kot
Cornell University
lucja@cs.cornell.edu

Magdalena Balazinska
University of Washington
Seattle, Washington, USA
magda@cs.washington.edu

Johannes Gehrke
Microsoft Corporation
johannes@microsoft.com

ABSTRACT

OLTP systems can often improve throughput by *batching* transactions and processing them as a group. Batching has been used for optimizations such as message packing and group commits; however, there is little research on the benefits of a holistic approach to batching across a transaction's entire life cycle.

In this paper, we present an OLTP system based on OCC that incorporates batching at multiple stages of transaction execution. Storage batching enables reordering of transaction reads and writes at the storage layer, and validator batching enables reordering of transactions before validation. We formalize the problem of validator transaction reordering, and we propose several algorithms and policies to solve this problem. We explain how validation can be parallelized for better performance. We carry out an in-depth experimental evaluation of the impact of storage and validator batching, and we show that our techniques can significantly increase throughput and reduce latency. We show how different batching algorithms perform, how parallelism in validator reordering helps, how storage and validator batching interact with each other, and how they improve system-wide and individual transaction performance.

1. INTRODUCTION

Transaction processing is a fundamental aspect of database functionality, and improving OLTP system performance has long been a key research goal in our community. It is well-known that the throughput of OLTP systems can be increased through *batching*-based optimizations, whereby some component buffers a number of transactions or requests as they arrive and processes them as a group.

Batching can improve system performance for several reasons. First, it increases the efficiency of communication by packing messages [15, 22]. Second, it amortizes the cost of system calls by condensing multiple requests into a single one, as in group commit [14, 25]. Third, it reduces the number of requests by discarding duplicate or stale requests, such as writes to the same record [19]. However, all of those are local optimizations based on low-level techniques.

We propose an OLTP system design that embraces batching as a core design principle throughout transaction execution. In particular, we explore the benefits of incorporating reordering into batching in optimistic concurrency control (OCC) to reduce conflicts [32]. OCC is a popular concurrency control protocol due to its low overhead in low-contention settings [2, 5, 7, 8, 9, 12, 17, 38, 40]. However, it wastes resources when conflicts are frequent [3]. We show batching and reordering reduce the number of conflicts, improve throughput and latency, and allow us to use OCC with higher-contention workloads.

Figure 1 shows a loosely coupled OCC-based transaction processing system with centralized validation. The system consists of three components: processors, storage nodes, and a single validator. External clients issue transactions to the system. On arrival into the system, each transaction is assigned to a processor and enters its *read* phase. The processor sends read requests to the storage, executes the transaction, and performs writes to a local workspace. After it has processed the transaction, it sends information about the transaction's reads and writes to the validator.

The transaction now enters the *validation* phase. In OCC with *backward validation*, the validator checks if a particular transaction conflicts with any previously committed ones. One example of a conflict that would fail validation is a *stale read*. Suppose a transaction t reads an object x, and a second transaction t' writes to the same object after t's read. If t' commits before t, t has a conflict, since it should have read the update t' made to x. Hence, t must fail validation.

If a transaction passes validation, the processor sends its writes to the storage; this is the *write* phase. Otherwise, the processor aborts and restarts the transaction.

The architecture of OCC with backward validation presents unique opportunities for batching because transactions are only serialized prior to commit. There are three obvious places to apply batching. The first is the processor in the transactions' read phase, where transaction requests can be batched before execution. Recent works in the context of locking-based protocols batch transactions and serialize

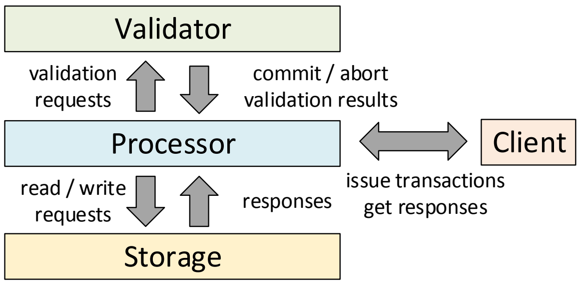


Figure 1: OCC system architecture

them before execution to reduce overhead [18, 36, 45]; these techniques could be adapted and applied in OCC as well.

The second possible place is the validator. If the validator buffers the requests, it can choose a validation order that reduces the number of conflicts and aborts. In our previous example, t reads the object x, and t' writes to x after t's read. Without batching, if t' arrives at the validator before t and commits, t will fail. With batching, we can serialize t before t' if they are in the same validation batch, and commit both transactions without any aborts.

Third, batching can be done at the storage level. This affects already-validated transactions in their write phase as well as transactions still in their read phase. The storage can buffer read and write requests into batches as they arrive. If a batch contains multiple read and write requests for the same object, the system can apply all the writes first, in the serialization order. Next, it can process the reads. Prioritizing writes over reads is always optimal as this reduces the number of aborts as much as possible. This is because OCC reads come from uncommitted transactions, while writes come from validated transactions that will commit soon. Thus, if the storage schedules a pending read before a pending write on the same object, the reading transaction will see a stale value and is guaranteed to fail validation.

Contributions of this Paper. In this paper, we explore in-depth the benefits of transaction batching and operation reordering in OCC with backward validation, with a focus on storage and validator batching. Our system design is best suited for integration with an in-memory versioned key-value store, as we require the ability to customize the validation logic. Our previous work on the Centiman system[15] proposed a loosely coupled architecture for OCC on top of key-value stores. The system we present here follows Centiman's architecture and enhances its design with batching and reordering.

Our first contribution is an enhanced OCC system architecture that utilizes batching and reordering at the storage and the validator levels. We analyze the reasons for conflicts and aborts at each stage of a transaction's life cycle, and develop techniques to reduce these aborts through batching.

Our second contribution is an optimal algorithm for storage reordering and approximate algorithms for validator reordering. We show that finding the optimal transaction ordering within a validator batch is NP-hard. We then describe two classes of greedy algorithms that balance abort rates and reordering overheads. These algorithms produce transaction orderings that are close to the optimal solution, and are sufficiently fast for practical use. We also extend these algorithms to weighted versions that can incorporate features such as transaction priorities.

Our third contribution is a design that reduces the amortized overhead of validation through parallelism. The parallelization is achieved with a four-stage pipeline, where the stages are batch preparation, transaction reordering, transaction validation, and cache update. Each of these stages can be further parallelized to allow concurrent processing.

Our final contribution is a detailed experimental study of the impact of storage and validator batching in a prototype system. Our results show that batching and reordering always increases transaction throughput and, surprisingly, also reduces transaction latency, especially the tail latency. This is because batching and reordering reduces the chance of conflicts and the number of transaction restarts, thus improving the end-to-end transaction latency. For workloads with high data contention, batching and reordering can improve the throughput by up to a factor of 3.1x and can reduce the average transaction latency by up to 68% for both a micro benchmark and the Small Bank Benchmark [4].

The remainder of the paper is organized as follows. In Section 2 we review OCC with backward validation. In Section 3 we discuss challenges, opportunities and techniques for storage and validator batching. In Section 4, we introduce our algorithms for intra-batch transaction reordering at the validator. In Section 5, we present an extensive experimental study of batching in our system. We discuss related work in Section 6 and conclude in Section 7.

2. BACKGROUND

We use the term optimistic concurrency control (OCC) to refer to the classical backward validation based OCC protocol introduced by Kung [32]. As explained in the introduction, every transaction goes through three phases. First comes a *read* phase, where the transaction reads data from the storage and executes while making writes to a private "scratch workspace". Then the transaction enters a *validation* phase. If validation is successful, the transaction enters the *write* phase, when its writes are installed in the storage. The validator assigns each transaction a timestamp i, and it examines the transaction's *read set* $RS(i)$ and its *write set* $WS(i)$, and it compares them to the writes of previously committed transactions. When validating a transaction $T(j)$ with timestamp j, the validator needs to check for conflicts with all transactions $T(i)$ with timestamp $i < j$ that have already committed and that overlapped temporally with $T(j)$, i.e., $T(i)$ had not committed when $T(j)$ started. $T(j)$ can be serialized after such a transaction $T(i)$ if one of the following conditions holds:

- $T(i)$ completed its write phase before $T(j)$ started its write phase, and $WS(i) \cap RS(j) = \emptyset$, or
- $T(i)$ completed its read phase before $T(j)$ started its write phase, and $WS(i) \cap RS(j) = \emptyset$ as well as $WS(i) \cap WS(j) = \emptyset$

If $T(i)$ and $T(j)$ overlap temporally, and $WS(i) \cap RS(j) \neq \emptyset$, we say there is a *read-write dependency* from $T(j)$ to $T(i)$. Intuitively, if there is such a dependency, $T(j)$ cannot be serialized after $T(i)$. In addition, we must ensure the writes of the two transactions are installed in the correct order to maintain consistency in the storage. If they write to the same object, the updates must be applied in their serialization order. The original OCC algorithms achieve this by putting the validation and write phases in a critical section [32], but there has been much progress on OCC over the last decades that allows system designers to relax most of these assumptions. The above discussion implies that validation only needs to check, for every transaction $T(j)$ and all transactions $T(i)$ overlapping temporally with $T(j)$, where $i < j$, that $WS(i) \cap RS(j) = \emptyset$. Data versioning provides an easy way to determine whether $T(j)$ overlapped temporally with $T(i)$; every time a transaction performs a read, we tag the read with the version of the object that was read. If $T(j)$'s read set contains an object X and the read saw version k, the validator only needs to check the write sets of all $T(i)$ with $k < i < j$ to see whether they contain X [15]. For now, we assume the validation phase is sequential for the ease of understanding. We will extend it to a parallelized design in Section 4.4. We handle out-of-order write requests with a *versioned* datastore, where every object in the datastore is versioned and every write request is tagged with a version number equal to the updating transaction's timestamp. If the datastore receives a write request with version (timestamp) i and a higher-numbered version $j > i$ already exists for the object, the write request is ignored.[1]

3. BATCHING

Batching involves buffering a number of operations as they arrive at some component of the system and processing them as a group. Given a batch, we run a lightweight algorithm that analyzes the batch and then reorders the operations in the batch in order to reduce aborts. We will introduce two types of batching: Validator batching and storage batching.

3.1 Validator Batching

In validator batching, we buffer and batch transaction validation requests at the validator as they arrive. We can first make a conceptual distinction between two types of transaction aborts: intra-batch and inter-batch aborts. Assume $T(j)$ abort due to its conflict with $T(i)$. If $T(i)$ and $T(j)$ are in the same batch, we call the resulting abort of $T(j)$ an *intra-batch abort*; otherwise, we call it an *inter-batch abort*. The optimal strategy for reducing inter-batch aborts is to ensure that transactions accessing the same objects are always batched together. That is, to cluster them based on access patterns, since the validator cannot do anything about conflicting inter-batch transactions. There has been a lot of research on data clustering either online or offline [16, 39], especially for fine-grained data partitioning. Any of these techniques can be used for validator batch creation.

Once a batch is collected, the validator can reduce intra-batch aborts by choosing a good validation (and thus resulting serialization) order. Such intra-batch transaction reordering can be done with several goals in mind. We can simply minimize intra-batch aborts, i.e. we maximize the number of transactions in each batch that commit. However, we may also want to prioritize certain transactions to have a greater chance of committing. For example, if we want to reduce the transactions' tail latency, we can increase a transaction's priority every time it has to abort and restart. Priorities could also be related to external factors, e.g., a transaction's monetary value or an external, application-defined transaction priority.

[1]In case the system also provides snapshot isolation, which is easy to support with OCC, such earlier versions are written to storage. Techniques described in this paper can also be extended to snapshot isolation.

We now define the problem of *intra-batch validator reordering of transactions (IBVR)* more formally. A *batch* B is a set of transactions to be validated. We assume all transactions $t \in B$ are *viable*, that is, no $t \in B$ conflicts with a committed transaction. If there are non-viable transactions in B, they can be removed in preprocessing, as they must always abort. Given B, the goal of IBVR is to find a $B' \subseteq B$ of transactions that must abort due to intra-batch aborts. IBVR must also find a total order $\prec$ on $B \setminus B'$ that *respects all read-write dependencies*; that is, for $t, t' \in B \setminus B'$, if $t \prec t'$, then there is no read-write dependency from t' to t. The validator processes each batch by running IBVR to identify B' and $\prec$, aborting all the transactions in B' and validating the transaction in $B \setminus B'$ in the order $\prec$. By the constraint we gave on $\prec$ above, and from the discussion in Section 2, $\prec$ is guaranteed to be a valid serialization order that allows all transactions in $B \setminus B'$ to commit.

There is always a trivial solution to any IBVR instance: aborting every transaction but one. Such solutions are not useful; therefore, every instance of IBVR is associated with an *objective function* on B', and the goal is to find a B' that maximizes the objective function. An example objective function could be the size of B' (smaller is better), or a more complex function that takes into account transaction priorities.

3.2 Storage Batching and Reordering

If a transaction reads a stale version of the object at storage, it is bound to abort at the validator due to conflicting with the update from a committed transaction. Thus, allowing updates to be applied at the storage layer as early as possible can reduce the chance of aborts for incoming transactions.

The optimal batching strategy is simple. We buffer a certain number of read and write requests into batches. The only way to reduce inter-batch aborts is to increase the batch size. To reduce intra-batch aborts later at the validator, we propose the following simple algorithm, which is easy to prove to be optimal given a batching: When a batch is ready (either when there are enough requests or a timeout is reached), for each object we apply the highest-version write request on that object. It is safe to discard all other writes on that object as explained in Section 2. Next, we handle all the read requests for the same object. This strategy is optimal for reducing intra-batch aborts, as it ensures that all available writes by committed transactions are applied to all objects before we handle any read requests on these objects.

4. VALIDATOR REORDERING

In this section we present our algorithms for intra-batch validator reordering of transactions (IBVR). We first show that the problem is NP-hard, and we then give two novel practical greedy algorithms in Section 4.1 that we illustrate in Section 4.2. Both algorithms are parameterized on *policies* that we discuss in Section 4.3. Finally we describe how to parallelize the validator in Section 4.4.

4.1 Intra-Batch Validator Reordering (IBVR)

Every batch B of viable transactions induces a *dependency graph* G; this is a directed graph whose nodes are the transactions in B and whose edges are read-write dependencies.

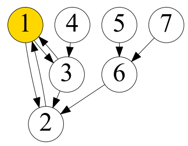


Figure 2: An example of a directed graph; node 1 forms a feedback vertex set.

If G is acyclic, then there exists a commit order Q on G that respects all read-write dependencies. We can construct Q by repeatedly committing a transaction whose corresponding node in G has no outgoing edge using topology sort. If G is not acyclic, we can show that the problem is NP-hard by reducing the NP-hard problem of finding a directed feedback vertex set to it [31].

A feedback vertex set (FVS) of a directed graph is a subset of vertices whose removal makes the graph acyclic. For example, consider the graph in Figure 2. The vertex 1 forms a FVS since the graph becomes acyclic after removing vertex 1 and its incoming and outgoing edges. Finding a minimal-size B' for IBVR is exactly the problem of finding the minimal (smallest-size) feedback vertex set on G. If we have a more complex objective function for IBVR, we can assign weights to the nodes to represent the desired transaction priorities, and look for a minimum-weight FVS. Once we find the FVS B', removing the nodes in B' from G yields an acyclic graph that determines the desired commit order Q.

The directed graph FVS (DFVS) problem is well-studied as it has many applications, including deadlock detection, program verification, and Bayesian inference. It is NP-hard and APX-hard [30, 31], and it is still an open problem whether there exists any constant-factor approximation. We propose two greedy algorithms for finding feedback vertex sets – one based on the graph's strongly connected components (SCCs) and the other based on sort. We also introduce a hybrid algorithm that makes strategic use of brute-force search, and consequently is slower but more precise.

All our IBVR algorithms begin by constructing the dependency graph G. We start with a set of transactions that have been batched at the validator and construct B by discarding all non-viable transactions. We can identify such transactions by validating each transaction against all the updates prior to this batch. Next, we create one node per transaction, and one edge per read-write dependency. To determine whether a read-write dependency holds from transaction t' to t, we check whether $WS(t) \cap RS(t') \neq \emptyset$. If so, we add an edge from t' to t. This can be implemented by creating a hash table from the write sets and probing it with the read sets. Since a write in t can potentially conflict with all the other transactions in the batch, the worst time complexity to probe hash table for a single write is $O(|B|)$, where $|B|$ is the size of the batch. The time complexity of building G is $O(|B|^2 + |R| + |W|)$, where $|R|$ is the total number of reads, and $|W|$ is the total number of writes.

We now process G to find a feedback vertex set. Both before and during the execution of our FVS algorithms, we *trim* the graph to remove all nodes which have no incoming edges and/or no outgoing edges; such nodes cannot participate in any cycles and are unnecessary to include in any FVS.

SCC-Based Greedy Algorithm. The intuition behind our first algorithm is that each cycle must be contained in a strongly connected component of the graph. After preprocessing, we partition the graph into SCCs. For a graph with V vertices and E edges, we can do this in time $O(|V| + |E|)$ using Tarjan's SCC algorithm [43].

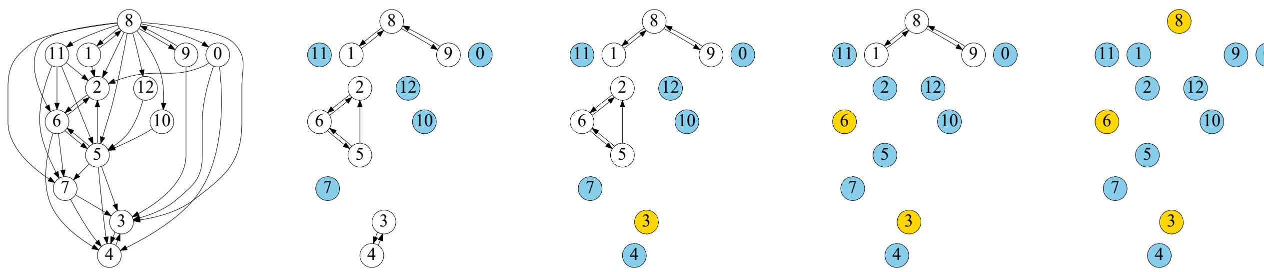
```
1  Algorithm GreedySccGraph(G, P)
     Input: Directed graph G, policy P
     Output: V, a feedback vertex set for G
2    V ← ∅
3    G′ ← trim(G)
4    SCC = StronglyConnectedComponents(G′)
5    for S ∈ SCC do
6      V ← V ∪ GreedyComponent(S, P)
7    end
8    return V
9  Algorithm GreedyComponent(S, P)
     Input: SCC S, policy P
     Output: V′, a feedback vertex set for S
10   if S.size == 1 then
11     return ∅
12   end
13   V′ ← ∅
14   v ← select(S, P)
15   V′ ← V′ ∪ v
16   S′ ← delete(S, v)
17   V′ ← V′ ∪ GreedySccGraph(S′, P)
18   return V′
```

Algorithm 1: SCC-based greedy algorithm

Any SCC that consists of a single node does not need to be considered further, as the single vertex involved cannot be on a cycle. For any SCC that consists of more than a single node, we choose a vertex to remove according to some *policy*. The policy is a ranking function over vertices, and we greedily choose the top-ranked vertex to remove. We then recurse on the remaining graph. Algorithm 1 shows the details of this procedure. We begin by trimming and partitioning the graph into SCCs (lines 3-4). We process each SCC S using $GreedyComponent(S, P)$ (lines 5-7). This subroutine starts by eliminating SCCs of size one (lines 10-12). Next, it chooses the top-ranked vertex v from S under Policy P (line 14). It includes v in the FVS of S (line 15), removes it from S (line 16), and it recursively calls $GreedySccGraph$ on the remaining graph (line 17). Finally, it returns the union of all the FVSs obtained in processing S (line 18). When the top-level procedure $GreedyComponent(G, P)$ has processed all the SCCs of G, it returns the union of the FVSs obtained (line 8).

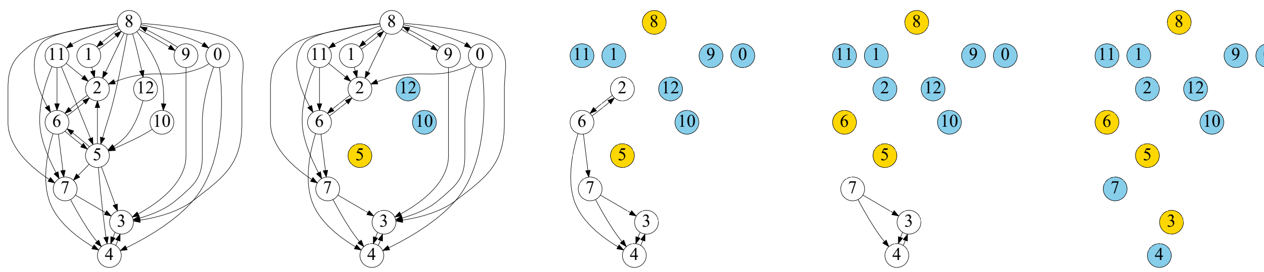
The policy P is at the heart of the algorithm, and it affects both its accuracy and runtime. Fortunately, there is no trade-off between accuracy and running time; a more accurate policy will lead to a smaller FVS and faster termination. We discuss possible policies in Section 4.3.

Sort-Based Greedy Algorithm. Our first algorithm relies on a SCC partitioning routine that takes linear time in the size of the graph. As this routine is called several times throughout the algorithm, it can cause a non-trivial overhead. Here we propose a faster greedy algorithm using a sort-based approach to remove nodes. Through extensive empirical tests of the SCC-based greedy algorithm we found that at each iteration, all the top ranked nodes in the graph are very likely to be included in the final FVS. Our second algorithm is based on this observation; it sorts the nodes according to their policy P, and includes the top-ranked k



(a) original (b) partition into SCCs (c) add 3 to FVS (d) add 6 to FVS (e) add 8 to FVS

Figure 3: An example of the SCC-based greedy algorithm using `prod-degree` policy



(a) original (b) add 5 to FVS (c) add 8 to FVS (d) add 6 to FVS (e) add 3 to FVS

Figure 4: An example of the sort-based greedy algorithm using `prod-degree` policy and multi factor 1

1 **Algorithm** *GreedySortGraph(G, P, k)*
  Input: Directed graph G, policy P, multi factor k
  Output: V, a feedback vertex set for G
2 $V \leftarrow \emptyset$
3 $G \leftarrow trim(G)$
4 **while** $G \neq \emptyset$ **do**
5   **if** $G.size < k$ **then**
6     $V \leftarrow V \cup GreedySortGraph(G, P, 1)$
7   break
8   **end**
9   $Q \leftarrow sort(G, P)$
10   **for** $i = 1; i \leq k; ++i$ **do**
11     $V \leftarrow V \cup Q[i]$
12     $G \leftarrow remove(G, Q[i])$
13   **end**
14   $G \leftarrow trim(G)$
15 **end**
16 **return** V

Algorithm 2: Sort-based greedy algorithm

nodes in the FVS. We call k the *multi factor* of the algorithm. The algorithm removes these nodes and iterates on the remaining graph.

Algorithm 2 shows this in more detail. We trim the graph (line 3); if the graph has no more than k nodes, we reduce the multi-factor to 1 (line 5-8). Otherwise, we sort the nodes into a queue Q using P, and include the top-ranked k nodes in V (line 9-13). After removing the selected nodes from G, we trim the remaining the graph again (line 14). We repeat this procedure until the graph is empty (line 4). The worst running time complexity of this algorithm is $O(C|B|)$, where $|B|$ is the size of the batch, and C is a constant for sorting, updating the graph, and the reduction of iterations using the multi-factor. Since the nodes are mostly sorted after updating the graph, the sorting in each iteration (except for the first one) is mostly linear. In practice, this algorithm is faster but less accurate than the SCC-based greedy algorithm. However, as we will see in Section 5, it produces results of comparable accuracy to the SCC-based algorithm in practice.

Hybrid Algorithm. We can combine the SCC-based greedy algorithm and the precise brute-force FVS search into a *hybrid algorithm*. The algorithm is similar to the SCC-based greedy algorithm but runs a precise, brute-force FVS search whenever it can afford to. Thus instead of processing all SCCs via the *GreedyComponent* subroutine (lines 5-7 of Algorithm 1), it runs the precise search when processing SCCs that are smaller than a certain *threshold*, and the greedy subroutine *GreedyComponent* on SCCs that are larger than the threshold. Adjusting the threshold allows us to trade off precision versus runtime.

4.2 An Example

We will illustrate how the two algorithms behave on a simple example. Figures 3 and 4 show executions of the two greedy algorithms on the same graph. Both algorithms use a policy where the ranking of a node is the product of its in-degree and its out-degree; we call this policy `prod-degree`.

Figure 3 shows the SCC-based greedy algorithm. The graph cannot be trimmed, so we partition it into SCCs. We remove all SCCs of size 1 – Nodes 0, 7, 10, 11, and 12 (Figure 3b). There are three remaining SCCs. We first look at the component containing Nodes 3 and 4. Since Nodes 3 and 4 have the same product of in-degree and out-degree, we can add either one of them to the FVS. We choose Node 3. Now Node 4 has neither incoming nor outgoing edges, so it is trimmed (Figure 3c). We repeat the process with the other components. For the SCC containing Nodes 2, 5, and 6, we add Node 6 to the FVS, as it has the largest product of in-degree and out-degree among the three nodes in this SCC. We now trim Nodes 2 and 5 (Figure 3d). Finally, we remove Node 8 from the last component, and trim Nodes 1 and 9 (Figure 3e). This leaves us with a final FVS consisting of Nodes 3, 6, and 8.

Figure 4 shows the same example using the sort-based greedy algorithm, with $k = 1$. After the first sort, we add Node 5 to the FVS since it has the highest product of in-degree and out-degree. After eliminating Node 5, Nodes 10 and 12 have only incoming edges, and get trimmed (Figure 4b). We sort the remaining nodes. This time, we add Node 8 to the FVS, and trim Nodes 0, 1, 9, 11 (Figure 4c). We repeat this process with the remaining nodes until the graph is empty. This yields a FVS consisting of Nodes 3, 5, 6, and 8 (Figure 4e), which contains one more vertex than the FVS we obtained with the SCC-based algorithm.

4.3 Incorporating Policies

Policies are of upmost importance for our algorithms. Recall that a policy is a ranking function on vertices of the graph, and a good policy ranks vertices which are likely to be in a desirable FVS highly.

We first discuss policies that aim at minimizing the number of conflicts, i.e., the size of the FVS. The simplest such policy is `random` that assigns all nodes random rankings. Alternatively, we can rank nodes using degree-based heuristics, which use the intuition that the removal of a node will break many cycles if the node is high in some measurement of its graph degree. Such heuristics have been shown to work well for FVS computation [13]. For example, the policy `max-degree` chooses the node with the largest degree (either in-degree or out-degree), `sum-degree` chooses the node with the largest total degree (in-degree plus out-degree), and `prod-degree` chooses the node with the largest product of in-degree and out-degree.

More sophisticated policies are possible if the system is optimizing a metric beyond maximizing the number of commits. For example, we may want to bound the transactions' tail latency; we can do that by incorporating latency information in our policies. We can rank transactions based on how many times they have been aborted and restarted; thus, transactions that have been restarted many times are less likely to enter the FVS and have a higher chance of committing. Alternatively, we can also devise policies that combine information about a transaction's number of restarts and its graph degree. For example, we can compute the ranking of a vertex as the product of its in-degree and out-degree divided by an exponential function of the number of restarts of the corresponding transaction.

4.4 Parallel Reordering and Validation

Since reordering occurs online in transaction execution, it can increase the latency of a transaction, resulting in a higher chance of conflicts. We explain how parallelization can alleviate this problem by reducing the amortized overhead of reordering.

In centralized reordering, the validator prepares a batch of transactions, reorders them, validates them and caches the updates of transactions that commit. Each of these steps corresponds to a subcomponent in the validator; we explain how we can parallelize execution across components through the use of pipelining, as well as within each component.

For pipelined parallelism, we can run each of the four validator subcomponents in a different thread. The batch preparation receives requests from the transaction coordinator, packages transaction into batches, and sends a reordering request to the transaction reordering subcomponent. Next, the transaction reordering first pre-validates a batch of transactions based on the latest database snapshot available, then reorders the remaining transactions, and sends a validation request consisting of the remaining transactions to the transaction validation subcomponent. The transaction validation takes a batch of ordered transactions, and validates them against updates from previously committed transactions. The cache update subcomponent finally applies the updates from transactions that pass the validation, serialized first by the order that the transaction validation subcomponent validates the batch, and then by the order that the reordering subcomponent computes within a batch.

In the single threaded centralized validator, a batch of transactions is pre-validated against the current snapshot of the database. Thus, the transactions that pass the pre-validation can only conflict with transactions within the batch, and will not conflict with previous committed transactions. So after reordering, the remaining transactions all commit.

In the pipelined reordering, the validation is decoupled into two phases. We denote a database snapshot as S_k, where transactions whose timestamps are less or equal to k have been committed as of this snapshot. In the reordering subcomponent, a batch of transactions B is validated against the latest database snapshot S_m. The remaining transactions after pre-filtering are reordered as a batch B' and sent to the validation. B' is again validated against the current database snapshot S_n. Since the validation subcomponent can be validating uncommitted transactions while B is being processed at the reordering subcomponent, when B' arrives at the validation, we may have $n \geq m$. Thus, transactions in B' can still abort due to conflicts with previously committed transactions in S_n.

We can further introduce parallelism into each of the subcomponents. In batch preparation, multiple threads can package transactions into batches. We can either assign each transaction coordinator to send its validation request to a specific batch preparation thread, or we can create a consumer-producer queue to connect the coordinators and the batch preparation threads. In transaction reordering, multiple threads can consume reordering requests from the batch preparation threads, and reorder batches of transactions concurrently. Since the transactions are not serialized yet in the reordering stage, the threads can send the processed batches to the validation subcomponent in any order. At the validation subcomponent, transactions are serialized and validated against all previously committed transactions. We cannot simply validate each individual transaction on different threads since transactions come from different batches can conflict with each other. However, within an ordered batch, since the only source of conflicts is from all the transactions committed prior the batch, the validation of each transaction within the batch can be processed in parallel. Since conflicts can happen across batch boundaries, processing a new batch should only start after all the transactions are processed in the previous batch. The transactions are first serialized at the batch level. Within each ordered batch, the transactions are serialized by the ordering produced from the reorder subcomponent, after removing the transactions that conflict against the current database snapshot at the validation subcomponent. The cache update subcomponent applies the updates in serialized order. While

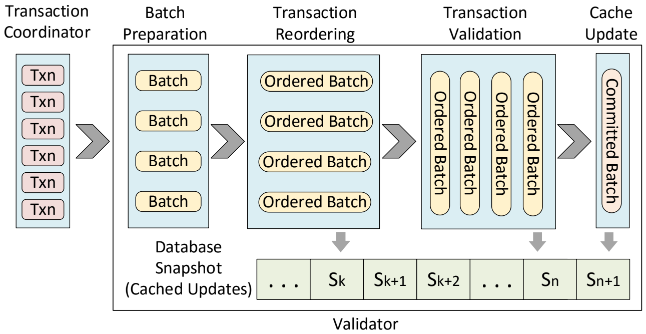


Figure 5: The architecture of parallel validator. It is decoupled into four subcomponents for pipeline parallelism: batch preparation, transaction reordering, transaction validation, and cache update. Each subcomponent can be further parallelized.

its operation is light-weight, if needed, we can introduce partitioned parallelism to update the cache concurrently. The idea is to partition the key space and break a transaction into smaller pieces, and one thread is assigned to apply the updates to one database partition.

Figure 5 shows the architecture of our parallel validator.

Alternatively, the transaction validation can use partitioned parallelism as in the cache update subcomponent. More discussion on this design can be found in our previous Centiman system [15].

5. EVALUATION

In our experimental evaluation, we wanted to understand the effect of batching and reordering at storage and validator, the performance of our validator reordering algorithms and policies, parallelizing transaction reordering at validator, and the impact of system configuration parameters. In particular, we asked the following questions:

1. How well do our validator reordering algorithms from Section 4.1 perform? How does batching and reordering at validator using these algorithms affect the end-to-end system performance?
2. How does the batch size impact performance?
3. How does parallel reordering impact the system performance?
4. How does storage and validator batching and reordering affect the system performance?
5. How do the different policies presented in Section 4.3 impact the system performance?
6. How does batching and reordering perform on real workloads?

5.1 Implementation and Experimental Setup

Our system architecture has four components: transaction generation, a processor, storage, and validation. The components communicate through consumer-producer queues.

The transaction generator continuously produces new transactions until the system reaches the maximum permitted concurrency level. The processor multiplexes transactions as a transaction coordinator, receives transaction requests from the transaction generator, sends read/write requests to the storage, sends validation requests to the validator, and replies to the transaction generator. It also restarts aborted transactions; thus, it only communicates commit decisions to the transaction generator. The storage continuously processes read and write requests. When storage batching is enabled, it buffers requests into batches, and uses the optimal strategy described in Section 3: It first executes all the write requests in the batch and then all the read requests.

The validator performs backward validation. For every transaction, a validation request consists of the keys and versions of its reads and the keys of its writes. The validator caches the write keys of committed transactions in an in-memory hash table, until these writes are overwritten by later updates. When batching is enabled, the validator collects the requests into a batch as they arrive, and runs one of the algorithms from Section 4 to determine a serialization order. Every transaction that passes the validation is assigned an integer *commit timestamp*, which corresponds to the version number of the updates it will install in storage.

We have further decoupled the validator component into four subcomponents as described in Section 4.4. A batch preparation worker receives validation requests from the processor, packages transactions into batches, and sends them to transaction reordering workers. A transaction reordering worker pre-validates and filters the transactions in a batch against the validator cache, reorders the transactions, and sends the batch of the ordered transactions to the validation workers. A validation worker validates the transactions against the current validator cache, and sends committed transactions to the cache update worker. The cache update worker finally applies updates to the validator cache based on the transaction serialization order.

By default, the validator uses the sort-based greedy algorithm with the `prod-degree` policy and multi factor 2. To process a batch, we create a dependency graph as described in Section 4.1. We have empirically determined that validator reordering is not beneficial if the dependency graph is very dense, as there are fewer opportunities for conflict reduction. Therefore, we heuristically set a limit on the size of the dependency graph. Once we detect that the number of edges has hit this limit during the construction of the dependency graph, we discard the graph and validate the transactions without reordering.

We have parallelized the transaction generation, the storage, and the transaction reordering at the validator. By default, two transaction generators populate the transactions concurrently to supply sufficient load. Two storage workers concurrently process reads and writes, and the writes are applied based on its data versioning as described in Section 3. In the validator, we first introduced pipeline parallelism by processing the four subcomponents concurrently. Since we observed that transaction reordering is heavier weight as compared to the other subcomponents, we increased its capacity by multi-threading. We used four transaction reordering workers by default.

Our system is implemented in Java. All the experiments run on a machine with Intel Xeon E5-2630 CPU @2.20GHz and 16GB RAM. We use a key-value model for the storage, which we implement as an in-memory hash table. In our micro benchmark, we populate the database with 100K objects, each with an 8-byte key. The values are left null as they are not relevant to our evaluation. We generate a transactional workload where each transaction reads 5 objects and writes to 5 objects drawn from a Zipfian distribution [24], with one of the reads and one of the writes on the same object. We limit the concurrency level to 300, i.e., at any time there are at most 300 live transactions in the system. The default batch size is 40 for both storage and validator.

The baseline configuration (*baseline*) represents the system running with both storage and validator batching turned off. We add a batch mode (*batch*) to separate the effect of batching and reordering, where requests are batched at both storage and validator, but no reordering is performed. The batch mode can benefit from better caching with tighter loops in the processing.

All our experimental figures show the averages of 10 runs, each lasting for 60 seconds in between a 10-second warm-up and a 10-second cool-down time. The standard deviation was not significant in any of the experiments, so we omit the error bars for clarity of presentation. We report the good throughput (the number committed transactions per second), the average latency, and the percentile latency.

5.2 Validator Reordering Algorithms

We first investigate the performance of the feedback vertex set algorithms from Section 4.1 with a comparison of the raw performance of the algorithms, i.e., their accuracy and running time. We run the algorithms on graphs constructed as described in Section 4.1, using our micro benchmarks.

We test the SCC-based greedy algorithm with the `max-degree` (*greedy_max*), `sum-degree` (*greedy_sum*) and `prod-degree` policies (*greedy_prod*). We also test the sort-based greedy algorithm *greedy_sort* (using the `prod-degree` policy for sorting and multi factor 2), and the hybrid algorithm *hybrid_m*. The hybrid algorithm uses *greedy_prod* as a subroutine when the size of the SCC is larger than m, and switches to the brute force search otherwise. By increasing the threshold, we can progressively approximate the optimal solution.

We test these algorithms against several baselines: *search* is an accurate, brute force search algorithm; *random* is the SCC-based greedy algorithm which removes a vertex at random from each SCC to break the cycle. For each graph, *random_3* runs the *random* for 3 times and returns the smallest FVS, mitigating the worst-case impact of bad random choices.

Figure 6 shows the average size of the feedback vertex set found by each algorithm. The brute force search algorithm is so slow that it cannot produce results once the skew factor increases beyond 0.7 as the graphs become denser. The *random* baseline computes a FVS whose size is almost twice as large as the greedy and the hybrid algorithms. Running the random algorithm multiple times produces similar results. This confirms the theoretical results which show that finding a good FVS is hard. The greedy algorithms, on the other hand, produce very accurate results. The average size of the FVS is almost identical to that of the brute force search when the skew factor is no larger than 0.7, and is very close to the best hybrid algorithm (*hybrid_20*, i.e., one that uses the brute force search when the size of the SCC is no larger than 20). Among the greedy algorithms, *greedy_prod* is consistently the best, although the difference is small.

Figure 7 shows the running time of the algorithms. The running time of the hybrid algorithm depends on the threshold for switching to brute force search. Thus, *hybrid_20* and *hybrid_15* have a longer running time than other algorithms, while the running time of *hybrid_10* is comparable to the SCC-based algorithms. Each of the SCC-based algorithms (*greedy_max*, *greedy_sum*, *greedy_prod*, *random*) has a similar running time. The random algorithm takes slightly longer than the greedy algorithms because it removes more nodes and thus requires more computation. The running time of *random_3* is three times that of *random*, since it runs the random algorithm three times. The sort-based greedy algorithm (*greedy_sort*), while slightly less accurate than the SCC-based greedy algorithms, reduces 74% of the running time of these algorithms.

We compare the end-to-end performance of the best SCC-based algorithm (*greedy_prod*) against the sort-based greedy algorithm. Figure 8 and 9 show the throughput and the average latency of the system with *greedy_prod* (*srvc-g*) and *greedy_sort* (*srvc-gs*). In both cases, storage batching is enabled. The *baseline* line shows the throughput with both storage and validator batching disabled.

The two greedy algorithms have similar throughput when the skew is very low. However, *greedy_prod* degrades significantly when data skew increases. This is because while *greedy_prod* is slightly more accurate, it takes much longer to run. This increases transaction latency and leads to more conflicts, especially when the data contention is high. *greedy_sort* consistently gives the highest throughput over all the workloads for its high accuracy and low running time.

Figure 10 shows transaction latency by percentile, i.e., the latency threshold for up to 95% of the transactions. The tail latency of *greedy_sort* is much lower than that of the other two, which is consistent with the throughput data.

5.3 Parallel Validator Reordering

In this experiment, we study the benefit of introducing parallelism into validator. Since we have observed that the reordering of the FVS is the most time consuming subcomponent in the validator, we increase the number of threads to parallelize batch reordering as described in Section 4.4.

Figure 11 and 12 show the throughput and the average latency with the number of reordering workers from 1 to 4 ($rw-1$, $rw-2$, $rw-3$, $rw-4$). As we increased the number of threads in transaction reordering, both the throughput and latency improves significantly when the skew factor is medium to high.

Figure 13 shows the percentile transaction latency. With more reordering workers, more transactions are reordered concurrently, and the queuing time for each transaction at validator has reduced. Thus, the tail latency of a transaction has greatly improved.

Yet we didn't expect linear scalability in throughput or linear reduction in latency since the bottleneck of the system may well be shifted to other components as we increase the capacity of the transaction reordering subcomponent. The system can further scale up once we increase the concurrency of other components and carefully engineer the scaling parameters of each part, which is out of the scope of this paper.

5.4 Batch Size

In this experiment, we explore how the batch size affects system performance. Smaller batch sizes should give lower latency but they offer fewer opportunities for reordering, leading to more aborts. We configure the system to perform both storage and validator batching with batch sizes from 40 to 240. Figure 14 and 15 shows the throughput and the average latency of the system with different batch sizes as data skew increases. The throughput first rises as we increase the size of the batch, and then degrades when the batch becomes too large. Batch size 40 gives the best

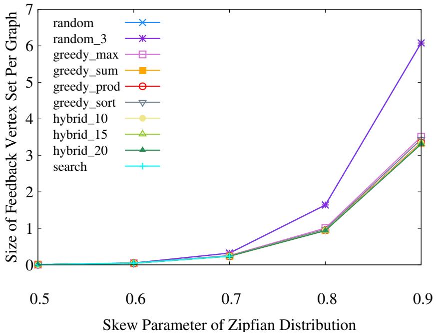


Figure 6: Size of FVS per graph with different algorithms

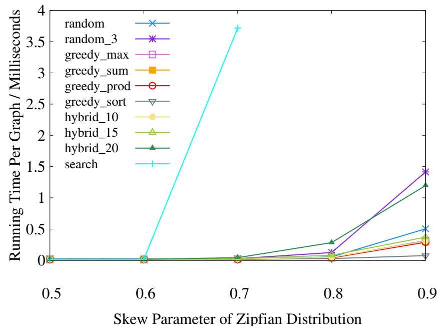


Figure 7: Running time of finding FVS with different algorithms

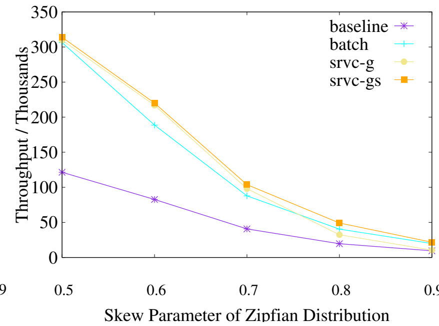


Figure 8: Throughput with SCC-based and sort-based greedy algorithms

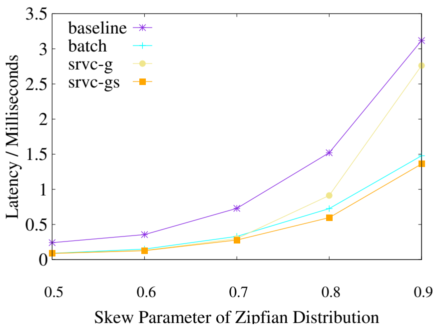


Figure 9: Average latency for greedy algorithms

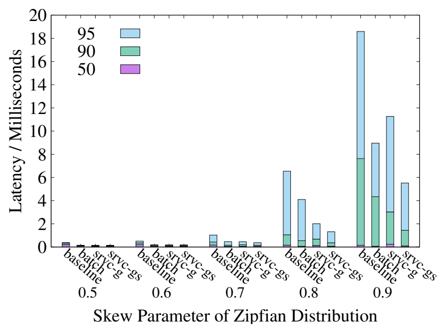


Figure 10: Percentile latency for greedy algorithms

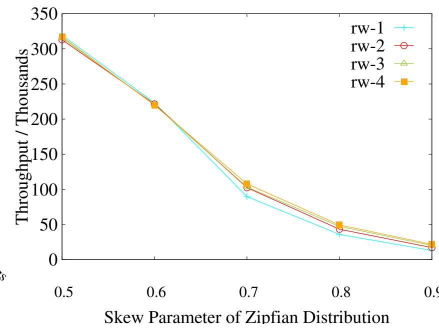


Figure 11: Throughput with different number of reorder workers

throughput and the latency. The percentile latency displays a similar pattern, as shown in Figure 16.

5.5 Storage and Validator Batching

Next, we perform a detailed analysis on the effects of storage and validator batching. We configure the system in several different modes: no batching (*baseline*), batching only (*batch*), storage batching (*sr*), validator batching only(*vc*), and both storage and validator batching (*srvc*).

Figure 17, 18, and 19 show the throughput, the average latency, and the percentile latency of different system modes under a variety of data skew parameters.

Overall, using batching at storage and/or validator consistently leads to significant improvements in the throughput and the latency profiles over the baseline. Batching alone (*batch*) improves the throughput significantly due to lower amortized overhead per transaction and better caching with tighter loops in processing. In addition, storage reordering and validator reordering consistently further improve the throughput. Moreover, validator reordering significantly reduces the average latency and the percentile latencies.

When the data contention is extremely high, the number of intra-batch conflicts that cannot be resolved by validator reordering increases. The validator reordering takes more time due to denser graphs, while bringing less benefit. Thus, the best throughput is achieved by using storage batching only (*sr*).

We conducted additional experiments with the batch size fixed to evaluate the system's peak performance with the load varied. Figure 20 shows the throughput of the system with batch size 20 and skew factor 0.7. The throughput increases with the concurrency level of the system, and then degrades as the system is overloaded. Enabling both storage and validator batching consistently outperforms the others. The figures on additional metrics and parameters are omitted due to the space limit.

5.6 Reduce Tail Latency

We explore validator reordering with a more complicated policy presented in Section 4.3; specifically, we look at policies that aim to reduce the transaction tail latency.

We explore the possibility of reducing transaction tail latency with latency-specific policies. Our baselines are the `prod-degree` policy that maximizes the number of commits (*max-c*) as well as *baseline* and *batch*.

Our first tail-latency aware policy (*rst-cnt*) favors transactions that have already been aborted and restarted. When choosing a node to include in the FVS, it chooses the one with the smallest number of restarts, breaking ties using `prod-degree`.

The second latency-aware policy considers both the number of restarts and a degree-based measurement of a transaction. It computes the weight of a node as the product of in-degree and out-degree over the exponential of the number of restarts with base 2. When choosing a node to include in the FVS, it chooses the node with the highest weight. Thus, a node with a high degree product can have its weight reduced if the corresponding transaction has restarted many times.

Figure 21 and 22 show the throughput and the average latency. The impact of tail-latency aware policies on overall transaction throughput and average latency are negligible as compared to when we maximize the number of commits (*max-c*).

Figure 23 shows the tail latency from 90% to 100%, i.e., the latency threshold for from up to 90% to up to 100% of the transactions. While our first latency-aware policy *rst-cnt* performs similar to *max-c*, the more sophisticated policy *rst-cnt* consistently performs significantly better than

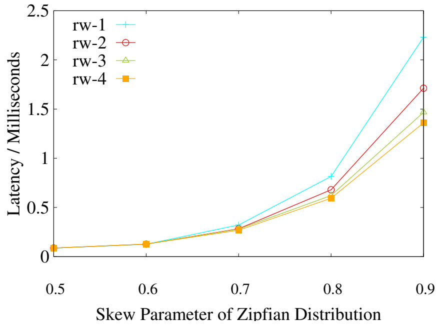


Figure 12: Average latency with different number of reorder workers

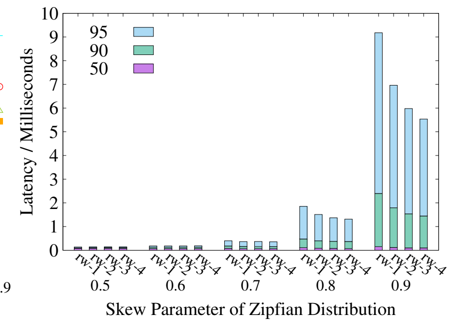


Figure 13: Percentile latency with different number of reorder workers

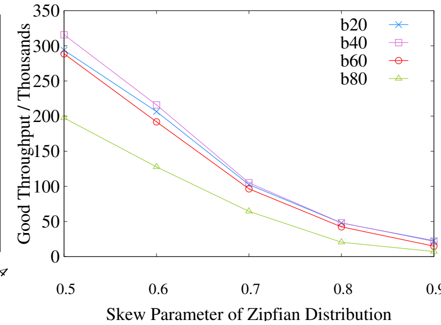


Figure 14: Throughput with various batch sizes

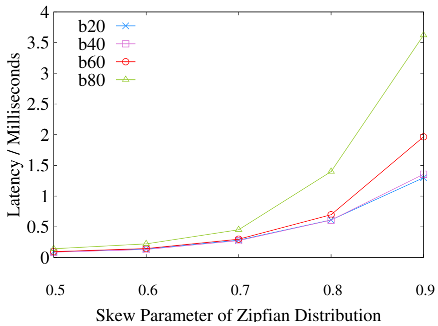


Figure 15: Average latency with various batch sizes

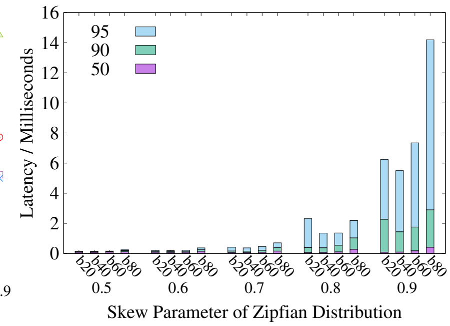


Figure 16: Percentile latency with various batch sizes

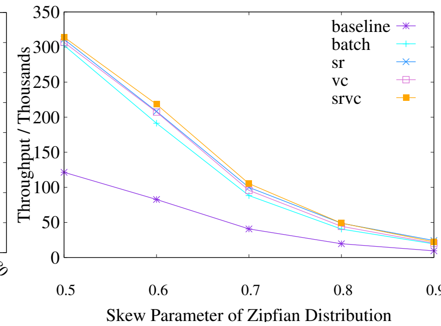


Figure 17: Throughput under workloads of Zipfian distribution

all the others.

5.7 Small Bank Benchmarks

The Small Bank benchmark contains transactions with a realistic and diverse combination of read and write conflicts. The transactions come from the financial domain: compute the balance of a customer's checking and savings accounts, deposit money to a checking account, transfer money from a checking account to a savings account, move all funds from one customer to another customer, and withdraw money from a customer's account. We use a Zipfian distribution to simulate skewed data accesses. We populate the database with 100K customers, i.e., 100K checking and 100K savings accounts.

Figure 24, 25, 26 show the throughput, the average latency, and the percentile latency of transactions. Overall, batching and reordering has improved the throughput up to 3.1x, while reduced up to 68% of the average latency. Storage and validator reordering always reduce latency on top of batching, again confirming our findings in Section 5.5.

5.8 Experiment Summary

We now summarize the results of our evaluation:

1. The simple sort-based FVS greedy algorithm strikes a balance between accuracy and time complexity, and leads to the best overall system performance.
2. There is a sweet spot for the batch size, where the system achieves its best combination of throughput, average latency, and percentile latency. We empirically selected the best batch size for our configuration and workload based on the experiment results.
3. As we increase the level of parallelism in validator reordering, the throughput, the average latency, and the percentile latency all improve, especially when the data contention is medium to high.
4. Batching by itself improves throughput significantly. Adding reordering consistently improves throughput, and significantly reduces the average and the percentile latency. It is always beneficial to use storage reordering. Validator reordering consistently improves the percentile transaction latency but can hurt the throughput and latency when the data contention is at the extremes.
5. For alternative reordering policies at the validator, privileging transactions with a metric that combines the degree of the transaction in the dependency graph and its restart time significantly reduces the tail latency.
6. Finally, in the Small Bank benchmark, batching and reordering provides significantly better performance with respect to the throughput, the average latency, and the percentile latency as compared to the baseline.

6. RELATED WORK

We discussed OCC and its applications in Sections 1. Here, we discuss related work on concurrency control under high data contention, transaction scheduling, and batching.

High contention concurrency control. The performance of concurrency control protocols suffers when concurrency level and data contention are high [20]; this has particular impacts on OCC [3]. Hybrid approaches combine OCC and locking to limit the number of transaction restarts [44, 48]. The problem can also be addressed by adjusting the concurrency level adaptively [28]. Transaction chopping reduces contention by partitioning transactions into smaller pieces and executing dependent pieces in a chained manner [36, 42, 47]. It is also possible to reduce conflicts by executing transactions at heterogeneous isolation levels [46, 47]. While we also address the problem of reducing conflicts under data contention, our batching and reordering techniques are different from previous work.

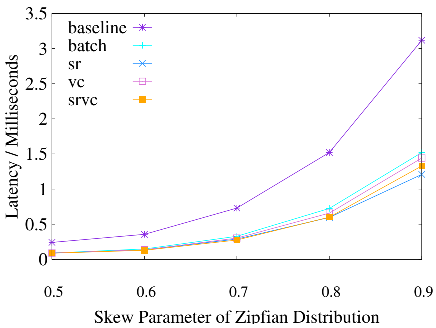


Figure 18: Average latency under workloads of Zipfian distribution

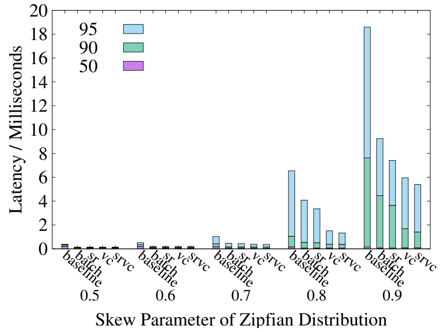


Figure 19: Percentile latency under workloads of Zipfian distribution

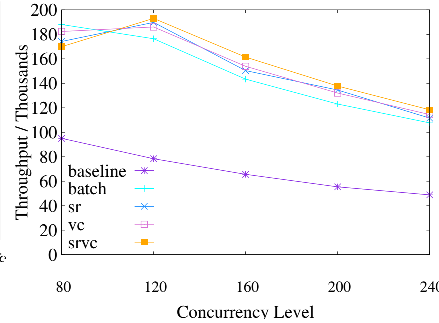


Figure 20: Throughput with micro benchmark (skew factor 0.7)

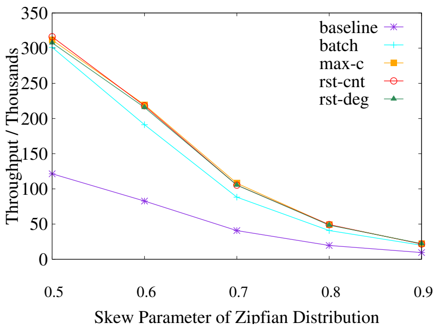


Figure 21: Throughput with tail latency optimized policies

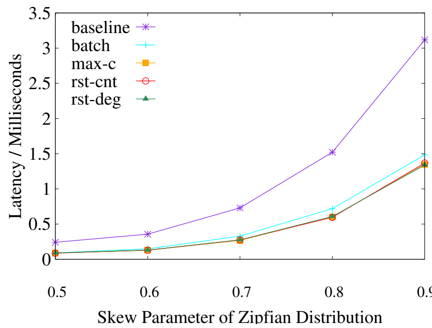


Figure 22: Average latency with tail latency optimized policies

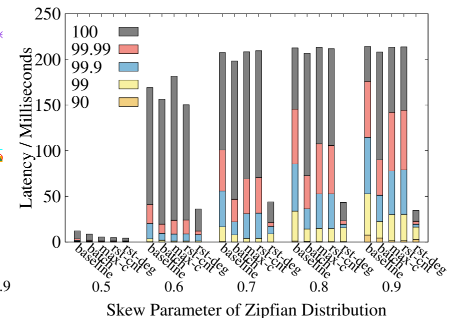


Figure 23: Percentile latency with tail latency optimized policies

Transaction scheduling. The dynamic timestamp assignment technique assigns each transaction a timestamp interval and flexibly picks the commit timestamp from the interval [6]. A similar technique can be used to optimize read-only transactions in distributed asynchronous OCC [15]. This approach can be extended to dynamically update the timestamp intervals of live transactions while committing a different transaction [10]. Dynamic timestamp assignment is compatible with our batching.

Transaction scheduling has also been studied in the real-time databases, where urgent or high value transactions are prioritized [26]. OCC with forward validation enables the validator to choose what to abort or defer a transaction if it would cause live transactions with higher priority to abort [27, 33, 34]. There are also systems that use locking and preemption [1], as well as hybrid optimistic/pessimistic methods [29, 35]. These approaches can be viewed as a simplified version of our validator reordering; moreover, none of the systems uses batching.

Transactions can be batched and serialized before execution [45, 36, 18]. These are complementary to our work. Our approach is more flexible as it allows reordering at multiple stages in a transaction's life cycle.

Batching. Batching to amortize costs and condense work is a common optimization technique. One application is to pack networking and logging messages [11, 15, 22, 23]. Batching is also widely applied to application requests to improve performance, including group commits [14, 25], condensing IO requests [14, 19], and Paxos [41]. Since batching is often associated with a throughput/latency tradeoff, there is work on adaptive batching [21, 37]. Those uses of batching are low-level and are not aware of the overall system infrastructure or the application semantics. Our work embraces batching as a core design principle at multiple stages of transaction execution. In addition, unlike previous work, we focus on the use of batching for reordering.

7. CONCLUSIONS AND FUTURE WORK

We have shown how to improve transaction performance in an OCC system by integrating batching and reordering into the system architecture at storage and validator. Batching allows the reordering of requests and reduces the number of aborts. We have formulated validator reordering as the problem of finding the minimal feedback vertex set (FVS) in a directed graph, and we have proposed two greedy algorithms for finding a FVS that are flexible and that perform well in practice. We have carried out an extensive experimental study in a main memory transaction processing prototype. Our experiments show that there is a sweet spot for the batch size for the best balance between latency and the flexibility of reordering. We have also investigated the impact of storage and validator batching on system performance. While both storage and validator batching consistently improve the throughput, validator reordering significantly reduces the latency profiles. We further proposed a parallel validator design. Our experiments showed that parallel reordering has improved both throughput and latency.

In future work, we plan to explore more sophisticated batch creation techniques, as well as systematically investigating adaptive batching to intelligently enable batching and adjust batch sizes for best system performance.

8. REFERENCES

[1] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *TODS*, 17(3), 1992.

[2] A. Adya and B. Liskov. Lazy consistency using loosely synchronized clocks. In *PODC*, 1997.

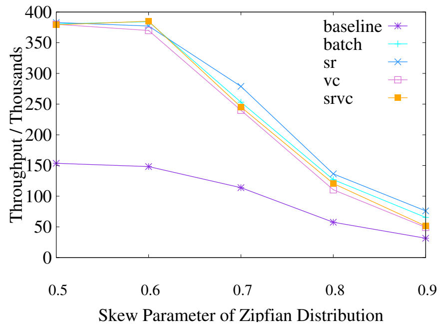


Figure 24: Throughput with Small Bank benchmark)

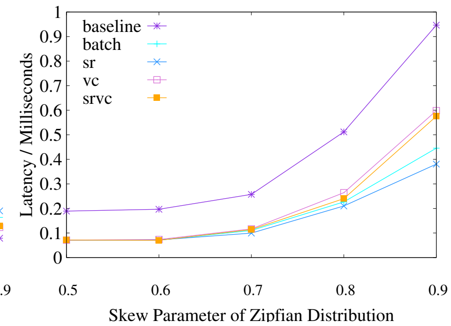


Figure 25: Average latency with Small Bank benchmark

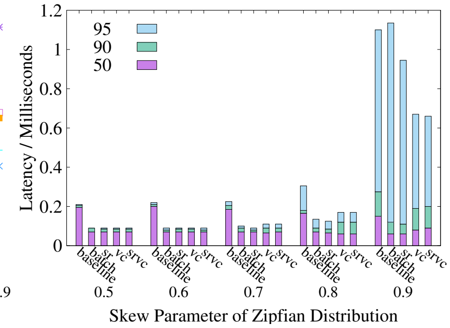


Figure 26: Percentile latency with Small Bank benchmark

[3] R. Agrawal, M. J. Carey, et al. Concurrency control performance modeling: alternatives and implications. *TODS*, 12(4), 1987.

[4] M. Alomari, M. Cahill, et al. The cost of serializability on platforms that use snapshot isolation. In *ICDE*, 2008.

[5] J. Baker, C. Bond, et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[6] R. Bayer, K. Elhardt, et al. Dynamic timestamp allocation for transactions in database systems. In *DDB*, 1982.

[7] P. A. Bernstein, S. Das, B. Ding, and M. Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *SIGMOD*, 2015.

[8] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - A transactional record manager for shared flash. In *CIDR*, 2011.

[9] P. A. Bernstein, C. W. Reid, et al. Optimistic concurrency control by melding trees. *PVLDB*, 4(11), 2011.

[10] C. Boksenbaum, M. Cart, et al. Concurrent certifications by intervals of timestamps in distributed database systems. *TSE*, SE-13(4), 1987.

[11] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *TODS*, 20(4), 2002.

[12] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[13] V. Cutello and F. Pappalardo. Targeting the minimum vertex set problem with an enhanced genetic algorithm improved with local search strategies. In *Intelligent Computing Theories and Methodologies*. 2015.

[14] J. DeBrabant, A. Pavlo, et al. Anti-caching: A new approach to database management system architecture. *VLDB*, 6(14), 2013.

[15] B. Ding, L. Kot, A. Demers, and J. Gehrke. Centiman: elastic, high performance optimistic concurrency control by watermarking. In *SOCC*, 2015.

[16] A. J. Elmore, V. Arora, et al. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *SIGMOD*, 2015.

[17] R. Escriva, B. Wong, and E. G. Sirer. Warp: Lightweight multi-key transactions for key-value stores. Technical report, Cornell University, Ithaca, NY, USA, 2013.

[18] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 2015.

[19] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD*, 2014.

[20] P. Franaszek and J. T. Robinson. Limitations of concurrency in transaction processing. *TODS*, 10(1), 1985.

[21] R. Friedman and E. Hadad. Adaptive batching for replicated servers. In *SRDS*, Oct 2006.

[22] R. Friedman and R. Van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *HPDC*, 1997.

[23] L. Glendenning, I. Beschastnikh, et al. Scalable consistency in Scatter. In *SOSP*, 2011.

[24] J. Gray, P. Sundaresan, et al. Quickly generating billion-record synthetic databases. In *SIGMOD Rec.*, volume 23, 1994.

[25] R. Hagmann. Reimplementing the cedar file system using logging and group commit. *SIGOPS Oper. Syst. Rev.*, 21(5), Nov. 1987.

[26] J. R. Haritsa, M. J. Canrey, and M. Livny. Value-based scheduling in real-time database systems. *The VLDB Journal*, 2(2), 1993.

[27] J. R. Haritsa, M. J. Carey, et al. Dynamic real-time optimistic concurrency control. In *RTSS*, 1990.

[28] A. Helal, T.-H. Ku, et al. Adaptive transaction scheduling. In *CIKM*, 1993.

[29] J. Huang, J. A. Stankovic, et al. Experimental evaluation of real-time optimistic concurrency control schemes. In *VLDB*, volume 91, 1991.

[30] V. Kann. *On the approximability of NP-complete optimization problems*. PhD thesis, Royal Institute of Technology Stockholm, 1992.

[31] R. M. Karp. *Reducibility among combinatorial problems*. 1972.

[32] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2), 1981.

[33] K.-W. Lam, K.-Y. Lam, et al. Real-time optimistic concurrency control protocol with dynamic adjustment of serialization order. In *RTSS*, 1995.

[34] J. Lee and S. H. Son. Using dynamic adjustment of serialization order for real-time database systems. In *RTSS*, 1993.

[35] Y. Lin and S. H. Son. Concurrency control in

real-time databases by dynamic adjustment of serialization order. In *RTSS*, 1990.

[36] S. Mu, Y. Cui, et al. Extracting more concurrency from distributed transactions. In *OSDI*, 2014.

[37] J. Nagle. Congestion control in IP/TCP internetworks. *SIGCOMM Comput. Commun. Rev.*, 14(4), Oct. 1984.

[38] S. Patterson, A. J. Elmore, et al. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *PVLDB*, 5(11), 2012.

[39] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *SIGMOD*, 2012.

[40] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[41] N. Santos and A. Schiper. Tuning paxos for high-throughput with batching and pipelining. In *Distributed Computing and Networking*. 2012.

[42] D. Shasha, F. Llirbat, et al. Transaction chopping: Algorithms and performance studies. *TODS*, 20(3), 1995.

[43] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2), 1972.

[44] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *TKDE*, 10(1), 1998.

[45] A. Thomson, T. Diamond, et al. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[46] C. Xie, C. Su, et al. Salt: Combining ACID and BASE in a distributed database. In *OSDI*, volume 14, 2014.

[47] C. Xie, C. Su, et al. High-performance ACID via modular concurrency control. In *SOSP*, 2015.

[48] P. S. Yu and D. M. Dias. Analysis of hybrid concurrency control schemes for a high data contention environment. *TSE*, 18(2), 1992.